AT&T 3B2 and 3B5 Computer Driver Design Guide

November 1984
305-495, Issue 1

AT&T

CONTENTS

Chapter 1

INTRODUCTION

Chapter 1

INTRODUCTION

OVERVIEW

AT&T 3B2 and 3B5 Computer Driver Design Guide is designed to provide information to users writing device drivers. It is assumed that persons writing device drivers know how the UNIX* Operating System works and know advanced C coding techniques. While users of the 3B5 Computer will have C software [including Software Generation System (SGS)] on the basic unit, 3B2 Computer users will have to purchase C software and SGS (including Extended SGS) to create device drivers.

Manuals for the 3B2 and 3B5 Computers that are referenced throughout this document include:

AT&T 3B2 Computer UNIX System V Release 2.0 System Administration Utilities Guide (for 3B2 Computers) and *UNIX System Administrator Reference Manual* (for 3B5 Computers) — which contain system maintenance commands and application programs (generally found in /etc directory), special files, and system maintenance procedures (such as crash recovery and boot procedures). In this manual, references to the *UNIX System Administrator's Guide* will mean either of these two books.

AT&T 3B2 Computer UNIX System V Release 2.0 User Reference Manual (for 3B2 Computers) and *UNIX System User Guide* (for 3B5 Computers) — which contain programs intended to be invoked directly by the user or by command language procedures (generally commands in /bin directory). These include general-purpose commands, communications commands and graphics commands. In this manual, references to the *UNIX System User's Guide* will mean either of these two books.

AT&T 3B2 Computer UNIX System V Release 2.0 Programmer Reference Manual (for 3B2 Computers) or *UNIX System Reference Manual* (for 3B5 Computers) — which contain system calls, subroutines, file formats, and miscellaneous facilities. The system calls section of the programmer reference manual describes the entries into the UNIX System kernel, including the C

* Trademark of AT&T Bell Laboratories

language interface. The subroutine section describes the subroutines available in the /lib and /usr/lib directories. In the file formats section, header files that contain layouts for system files and data structures generally found in /usr/include and /usr/include/sys directories are documented. Miscellaneous facilities describe, for example, characters sets and macro packages. In this manual, references to the the *UNIX System Programmer's Manual* will mean either of these two books.

3B2 Computer Feature Card Interface Design Manual (for 3B2 Computers) or *3B5 Computer I/O Board Interface Design Manual* (for 3B5 Computers) — which provide information Original Equipment Manufacturers (OEMs) need to design Feature Cards (for the 3B2 Computer) or I/O boards (for the 3B5 Computer).

DESCRIPTION OF CONTENTS

Firmware Implementation

Chapter 2, FIRMWARE IMPLEMENTATION, describes the firmware entries and explains how the system interfaces with the boards. Diagnostics for and initialization of Feature Cards (for the 3B2 Computer), I/O boards (for the 3B5 Computer), memory boards, and the input/output expansion board are introduced. A description of how the device driver schedules work for boards or Feature Cards is also included.

Kernel/Driver Interface

Chapter 3, KERNEL/DRIVER INTERFACE, describes the kernel interface to driver services such as open, close, read, write, strategy, and ioctl. It discusses the handling of interrupts, power failures (for the 3B5 Computers), and errors. Data types, system constants, user data access, and initialization are also explained.

Driver Configuration

Chapter 4, DRIVER CONFIGURATION, explains how the 3B2 and 3B5 Computers configure themselves when the system is booted. It describes how to write, add, and debug device drivers.

3B2 Computer Dependent Information

Chapter 5, 3B2 COMPUTER DEPENDENT INFORMATION, defines 3B2 Computer-specific information needed to write device drivers.

3B5 Computer Dependent Information

Chapter 6, 3B5 COMPUTER DEPENDENT INFORMATION, describes 3B5 Computer-specific information needed to write device drivers.

Diagnostics

Chapter 7, DIAGNOSTICS, defines diagnostics structures and explains the software and firmware interfaces to the diagnostics environment.

Appendices

Appendices at the end of the manual explain, in more detail, the memory driver, tty subsystem, block I/O subsystem, self-configuration commands, master file, system file, 3B2 Computer edit_tbl command, and registers, 3B2 Computer block device driver, 3B2 Computer character driver, 3B5 Computer block device driver, and 3B5 Computer character driver. Appendices are referenced throughout the manual.

Chapter 2

FIRMWARE IMPLEMENTATION

PAGE

Chapter 2

FIRMWARE IMPLEMENTATION

INTRODUCTION

To be effective, computers need to be able to interface easily with other computer equipment and other computers. A computer designed for expansion lets users add equipment as their needs grow. The 3B Computers are designed to facilitate this growth.

The firmware of the 3B Computers was created to let users tailor the system to their needs and to grow with their needs.

DESCRIPTION OF 3B2 COMPUTER FIRMWARE ENTRIES

The System Board is built into the computer and provides basic facilities. Feature Cards are circuit boards designed to connect peripherals to the 3B2 Computers. Memory cards add primary memory to the 3B2 Computer systems. An Input/Output (I/O) Expansion Board routes the I/O bus from the System Board to the Feature Cards. The 3B2/300 Computer is designed to house:

- One System Board (Slot 0)
- Up to four Feature Cards (Slot 1-4)
- Up to two memory cards
- Input/Output Expansion Board.

The 3B2/300 System Board contains:

- WE 32000 Series Processor Module
- Dynamic RAM Controller
- Feature Card Interface Support

- Direct Memory Access (DMA) Subsystem including
    - DMA Controller
    - Hard Disk Controller
    - Floppy Disk Controller
    - Universal Asynchronous Receiver Transmitter (UARTS)
- Interrupt Structure
- Timers
- Control and Status Register
- Backup Battery
- PROM
- Nonvolatile RAM

Dynamic RAM and associated drivers are on two boards that plug into the System Board. A third board, the 3B2 Computer Input/Output Expansion Board, is also provided. It, too, plugs into the System Board. The 3B2/300 Computer has four Feature Card port connectors on the Expansion Board.

The System Board, I/O Expansion Board, Memory Cards, and Feature Cards are in the computer cabinets of the 3B2 Computers. Cables carry peripheral control signals from each Feature Card to the correct connectors. The peripheral to be connected to the 3B Computers are plugged into its connector. This mates the peripheral to its Feature Card. All standard Feature Cards support automatic configuration and diagnostics.

Feature Cards may be programmable or intelligent. Programmable Feature Cards operate when they are programmed by the System Board CPU. These Feature Cards do not usually contain a microprocessor. Programmable Feature Cards communicate with the System Board through interrupts and on-board registers. Intelligent Feature Cards contain microprocessors or microcomputer-integrated circuits that can execute stored programs within their own memory. These Feature Cards operate autonomously. Intelligent Feature Cards use request and completion queues in primary memory to communicate with the System Board CPU.

Feature Cards can be 8- or 16-bits wide while primary memory is 32-bits wide. The I/O bus has a 16-bit data bus; so, up to two data bytes can be transferred in one transaction. The System Board houses a Byte Rotation Unit (BRU) that formats the data for the proper interface between primary memory and the Feature Cards.

Diagnostics for the 3B2 Computer

Routines are run when the system is powered up to ensure the integrity of the hardware and firmware. As soon as the system is powered up, the noninteractive mode of the Maintenance and Control Program (MCP) resets the system and performs basic sanity checks. Sanity checks include diagnosing the WE 32000 Series processor module, hard disk drives, dual ported dynamic RAM, and UART. When the sanity checks are finished, a self-configuration process takes place.

In self-configuration, the computer identifies each Feature Card and notes Feature Card type (identification code) and position on the bus. So, a hardware identification register is needed on each Feature Card. The identification code read and the Feature Card location on the I/O bus is stored in an Equipped Device Table (EDT) for later use by diagnostics and the UNIX Operating System.

The EDT is built in two stages. The first stage is finished in self-configuration. In self-configuration, a device code is read from a fixed location on each board. The second stage occurs in filledt when a table of devices is downloaded from the disk. Each device code is then associated with its device name, possible option, and so on.

A hardware timer on the System Board is provided to cause a fault on a " hung" bus. A nonexistent Feature Card (an empty I/O bus slot, for example) will be flagged as an I/O bus error.

When self-configuration terminates, more extensive diagnostics run causing the computer to enter either the interactive mode of MCP or, if no failures are found, an exit routine that boots the UNIX Operating System.

If a problem has been found, MCP prompts for a password. When the password is entered, diagnostics and utilities can be run to find the problem. MCP is in PROM with the boot firmware for diagnostics and utilities.

FIRMWARE IMPLEMENTATION

INIT File System

The INIT file system has been merged into / (root). Space is needed on disk in the root file system for the /dgn directory, which contains two diagnostic files per device type. As new Feature Cards add new device files to the system, the disk area may grow. So, a new Feature Card may be added without changing PROM. All that is required is that the file containing the correct phase is added to the disk under the root file system. (The edittbl command, described in an appendix, lets users change the file in the root file system that filledt reads during self-configuration to set the device and subdevice lookup tables.)

The diagnostics search the root system for diagnostic phases, and the firmware uses it for pulling in utilities. During initialization, the UNIX Operating System mounts the root system to pull in autoconfiguration information and the kernel. Software additions, such as adding new diagnostics and drivers, are done through installation scripts. Under the control of the UNIX System, this utilities installs the new files.

Boot Firmware

A large part of the boot program exists on the disk itself. The PROM resident firmware downloads the boot program from the disk to Dual Ported Dynamic RAM (DPDRAM) and calls it. A buffer defined in the transfer vector table passes the pathname of the program to be booted to/from these two modules.

The boot firmware is also used by the MCP to bring up FILLEDT and the diagnostic monitor when the system is powered up and the operating system is up.

Transfer Vector

All routines and global data must be accessed indirectly through a structure of pointers called vectors. In this way, routines in PROM may be accessed indirectly through the same address. So, recompilations will not affect addressing. This is important because it lets PROM loads and disks loads occur independently.

The transfer vector table can be found in /usr/include/sys/sbd.h. Additions may be made by adding to the end of the structure.

Scenario for the 3B2 Computer

When the system is powered on, the System Board resets all of the Feature Cards to their initialization routine address. The I/O Bus control signal resets line SYSRSTO, which starts the initialization. The Feature Card CPU writes its identification code to its identification/vector register in the Feature Card I/O bus interface circuit (PROM). The ID code must be written within 100 milliseconds so that the self-configure program on the System Board does not attempt to read the code before it is written. After the ID code is written, the Feature Card runs the basic sanity checks.

The ID code is read by the System Board, which causes INT0 (INT is short for interrupt). The ID code is stored in the EDT, which is maintained by the System Board firmware. The EDT contains information about the device such as the ID code, an ASCII name of the device, slot, board type and corresponding slot and sequence number, boot device, boot capability, number of subdevices, size of request and completion queue, width of data transfer (8 or 16 bits), console capability, and structure defining up to 15 subdevices. The System Board correlates the location of the Feature Card on the I/O bus with the ID code. In other words, the System Board knows which Feature Card is in which slot.

After INT0, the Feature Card knows that it has been identified by the System Board and expects the System Board to put it through the system generation (sysgen) sequence. The Feature Card might get more than one INT0 from the System Board when it reads the board ID/vector register. If the interrupt (INT0 and INT1) is not generated, the Feature Card never sysgens. When the Feature Card does not sysgen, system diagnostics fail and the operating system will not be loaded. After the INT0 sequence, firmware operations will stop and wait for a sysgen (INT1) interrupt. A sysgen INT1 interrupt occurs when the System Board reads or writes that control register.

Attention interrupts (INT1) happen when the System Board addresses an Intelligent Feature Card control register. An attention interrupt tells the Intelligent Feature Card to fetch the information in the request queue.

FIRMWARE IMPLEMENTATION

A data block for each Feature Card is created by the System Board firmware. The sysgen data block contains:

- Address of request queue
- Address of completion queue
- Request queue size
- Completion queue size
- Interrupt vector
- Number of request queues for a Feature Card.

A pointer to the first entry in the sysgen data block of the Feature Card is generated by the System Board firmware. It is stored in primary memory at location 2000000 (hexadecimal). All Feature Cards share that location so it must be used carefully. After an INT1 reset, the Intelligent Feature Card gets this pointer and uses it to reference the sysgen data block. The Feature Card firmware rewrites the ID/vector code register with the interrupt vector. Then, the System Board can read the ID/vector code register, if it wants to. Usually, the register is used at interrupt acknowledge time. This means the System Board firmware only has one chance to read a Feature Card ID code without resetting the Feature Card.

The sysgen data block also directs the Intelligent Feature Card to its queues in primary memory (DPDRAM). After the sysgen data block is created, the queues can be used by the Intelligent Feature Card. The structure of the sysgen follows:

```
struct sysgen {
       long request;         /*address of request queue*/
       long complt;          /*address of completion queue*/
       unsigned char req size;      /*number of entries in request queue*/
       unsigned char int_vect;      /*base interrupt vector*/
       unsigned char no_rque;       /*number of request queues*/
}
```

Queue Types

Feature Cards use two types of queues: request and completion. A request queue is used by the System Board CPU to hold jobs for a specific Intelligent Feature Card. When the Feature Card is done with a job, an entry is made in its completion queue.

The System Board CPU sets up a job request in the request queue and issues an attention interrupt (INT1) to the Intelligent Feature Card. The Intelligent Feature Card fetches the job from the request queue through attention interrupt INT1.

An Intelligent Feature Card may have as many job requests in its queue as are defined in its sysgen data block. The number of request queues are dependent on the number of subdevices that need a request queue on an Intelligent Feature Card.

Feature Cards handle two types of requests: normal and express. Normal requests are executed by the Feature Card on a first come, first serve basis. Express requests have the highest priority. Express requests are allocated in the request queue. The System Board CPU uses the express request when it wants the Intelligent Feature Card to finish the job being executed and then executes the express entry. The System Board uses INT0 to tell (signal) the Feature Card about the express job. (INT0 is used to read the Feature Card ID code when the system is turned on. Thereafter, INT0 is used to signal the Feature Card of an express entry.)

When the Intelligent Feature Card is done with its job, it makes an entry in its completion queue. The Intelligent Feature Card generates an interrupt request to the WE 32000 series CPU. The WE 32000 series CPU acknowledges this request, which lets the Feature Card gate the interrupt vector in the ID/vector register onto the I/O bus. The interrupt vector is used by the WE 32000 series to identify the interrupting Feature Card. When the interrupt Feature Card is identified, the appropriate completion queue can be read. The System Board CPU uses this information to assign new tasks.

Any Feature Card that can interrupt the System Board must respond with a vector at acknowledge time. These Feature Cards need to write its vector in the ID/vector register before it can generate an interrupt request. *A Feature Card must not access its ID/vector register or try to cause an interrupt while it has an interrupt pending.*

FIRMWARE IMPLEMENTATION

Queue Operation

The sysgen data block address-of-request queue points to the contiguous job request queue. The System Board loads these queues with jobs for the Feature Card. The address-of-completion queue points to the job completion queue. The Feature Card fills this queue with job completion reports and status information for the System Board.

Request queues are declared, one per subdevice, on the Feature Card. Since there is only one completion queue per Feature Card, the driver knows exactly where the completion report is and doesn't need to poll multiple queues. The structure is the same for request and completion queues. The only differences between queues are the number of queues, number of entries, and the size of an entry.

In this explanation, it is assumed that there is only one subdevice on the Feature Card and, therefore, only one request queue. If multiple request queues are used, they must be the same size and must be contiguous. The sysgen data block points to the first incidence of each queue.

Size members of the sysgen data structure give the number of possible entries in each request and completion queue. Each job request or completion report is an entry. The int_vect entry holds the interrupt vector the Feature Card is to use in subsequent operations that involve the System Board. This vector is a base number of which the upper four bits remain fixed. The lower four bits may change to serve up to 16 vectors per Feature Card. The final entry is the number of contiguous request queues declared for use by the subdevices of a Feature Card.

The Feature Card may now communicate with the System Board through these queue entries. The last thing the Feature Card does in the startup sequence is place a sysgen completion status report in the job completion queue. This starts normal operation mode.

Queue Structures

Normally, a Feature Card fetches jobs from the appropriate request queue, does the assigned work, and places status reports in the completion queue. The common firmware core maintains and provides access to these queues for the Feature Card. Two different data structures facilitate the queue communication protocol: the actual queue itself and the entries that contain information about the queue.

The job request and completion queues use the structures for sysgen (discussed earlier) and que_sets. The structure for que_sets follows:

```
typedef struct que_sets      {              /*entry for express requests*/
        ENTERY express;              /*RENTRY or CENTRY as per queue.h*/
        struct queue   {             /*Three ways to access load and*/
                union  {                    /*unload pointers*/
                        long all;           /* All pointers at once*/
                        struct  {    /*16-bit load pointer*/
                                     /*and 16-bit unload pointer*/
                                short load;
                                short unload;
                        } bit16;

                        struct  {           /*8-bit load pointer and 8-bit*/
                                     /*unload pointer*/
                                char pad1;
                                char load;
                                char pad2;
                                char unload;
                        } bit8;
                } p_queues;

                /*the queue entries  RQSIZE or CQSIZE*/
                ENTRY entry [QSIZE]:  /*RENTRY or CENTRY as per queue.h*/

        }queue[NUM_QUES];            /*multiple queue declaration*/
                                     /*for request queue  one*/
                                     /*for completion queue*/

}QUEUE;                              /*RQUEUE OR CQUEUE*/
```

Although there are different typedefs (type definitions) for request and completion queues and entries, they share a common structure. The only difference is the size of the members of the structures. For this reason, only one ENTRY typedef and one QUEUE typedef are shown. When declared, these should be separate typedefs referred to as RQUEUE and CQUEUE, for request and completion queues, respectively. When declared like this and in the following order, the different sizing option may be used.

```
       /*application defined area in each request.*/

typedef struct rapplic{
       long datal;
       long etc;
       }RAPP;

       /*application defined area in each completion.*/

typedef struct capplic{
       long datal;
       long etc;
       }CAPP;

       /*number of request queues defined.*/
       /*recommended one per subdevice*/

#define NUM_QUES 1

#define RQSIZE 8
#define CQSIZE 8

#define R_ADDR &request             /*address of request queue defined by driver*/
#define C_ARRD &complt             /*address of completion queue defined by driver*/

#include <queue.h>        /*now that constants are defined, include queue.h*/

RQUEUE request;
CQUEUE complt;
```

To declare a request and completion queue in the driver, several constants must be defined. The order of these declarations relative to the #include <queue.h> is important and should be as explained below.

The System Board enters a job request to the Feature Card by moving a job entry to the address pointed to by the request queue load pointer. The System Board increments the load pointer by the size of an entry. The Feature Card, then, unloads jobs from its unload pointer, increments the unload pointer by the size of an entry, and does the assigned task. The same protocol is used in the completion queue with the Feature Card loading and the System Board unloading.

The System Board or Feature Card must be able to access the queue pointers in three different ways: all pointers at the same time, as two 16-bit pointers, or as two 8-bit pointers. It is convenient to access the pointers as a long integer so they may all be written at the same time during initialization. This mode should not be used during normal operations. The other two modes of pointer access, described in the union portion of the data structure, are used in the operational mode.

If a 16-bit Feature Card is communicating with the 32-bit System Board, only a 16-bit pointer may be used. If 32-bit pointers are used, it is possible the Feature Card could write the lower 2 bytes of the pointer and the System Board read the old upper 2 bytes with the new lower 2 bytes. But, this operation would break down the entire protocol. The problem gets more complex with an 8-bit Feature Card that can only access memory a byte at a time.

To solve this problem, 16-bit pointers are used for 2-byte wide devices and 8-bit pointers are used for single byte devices. These pointers are not pointers in the true sense—they are offsets from the beginning of the entry structure.

An example of macros for accessing the queues are defined in the header file queue.h. These macro definitions, as well as the queue definitions, exist in a header file under the UNIX System header directory.

Queue Size

An 8-bit device may have 32 entries.

256 pointer address range / 8 bytes minimum per entry = 32 entries

The 16-bit device is limited by the queue size in the sysgen data block. Since this field is a byte, a 16-bit device may have 256 entries. These limits are maximums. Any queue may be declared to be smaller than the maximum value by changing its size in the sysgen data block.

A device may have one less job than the number of entries in its queue since the numbering may start at zero, instead of one.

Entry Structure

The entry list in the queue structure directs the Feature Card to the work to be done. Entry data structures are common between job request and job completion but the fields take on different meanings.

Byte Count 2 bytes	cmd_stat 1 bit	seqbit 1 bit	Subdev 6 bits	Opcode 1 byte
Data or Address 4 bytes				
Application Fields				

1 Byte = 8 Bits

The declaration of the completion entry in queue.h follows:

```
typedef struct entry {
        union{
                struct{
                        unsigned bytcnt:16;  /*offset of last byte to transfer*/
                                             /*0 transfers byte 0*/
                        unsigned cmd_stat:1; /*flag for command/status opcode*/
                        unsigned seqbit:1;   /*flag for block available*/
                        unsigned subdev:6;   /*subdevice being addressed*/
                        unsigned opcode:8;   /*command or status opcode*/
                        }bits;
                        struct {
                                unsigned short bytcnt;
                                char subdev;
                                char opcode;
                        }bytes;
                }codes;

                unsigned long addr;     /*data or memory address of data*/
                APP appl;               /*application of defined area*/
                                        /*CAPP or RAPP for CENTRY or RENTRTY*/
        }ENTRY                          /*CENTRY or RENTRY in req or cmplt queue*/
```

The byte count (bytcnt) field of the entry structure represents the offset of the last byte to be transferred on a request. So, a zero in this field means transfer one byte. If 255 appeared in this field, 256 bytes would be transferred. This lets a full 64-kilobyte transfer take place. An absolute count would not.

On a completion report, this field contains the byte number of the last byte transferred. For example, if an error occurred during a scheduled request, the bytcnt field of the completion report would contain the byte on which the error occurred.

The command/status (cmd_stat) field is usually zero. On a request, a zero in this field means that the operation code (opcode) field contains a command. A completion queue entry contains a zero in this location to indicate that status is contained in the opcode field.

The Feature Card may give the System Board commands by sending back a completion report with the cmd_stat bit equal to one. Requests for processor status, additional memory, or other commands may be passed to the System Board. The System Board passes back the completion reports to the Feature Card by setting the cmd_stat bit to one, placing a report in the request queue, and triggering the attention interrupt to the System Board. This gives the System Board and Feature Card great flexibility. In special applications, the Feature Card may use the System Board as a slave. The Feature Card may not give the System Board commands in the express queue because each Feature Card has only one interrupt.

The subdevice (subdev) field contains the subdevice being addressed. The driver fills this field in the request queue and the Feature Card returns it in the completion queue. Usually, commands are issued in the request queue or completion status is returned in the completion queue through the operation code (opcode) field. The System Board uses the sequence bit (seqbit) field to determine when the Feature Card has entered an express job completion report. The data/address (addr) field of the structure is a 32-bit/byte of data or the physical address of the data block to be accessed.

The application field of the structure is an application defined typedef. This field is allocated so that the entry size can change and be determined by the application. RAPP or CAPP must be defined by the application before the queue.h is included.

Scheduling Jobs

To explain job scheduling, a job will be discussed. Ten kilobytes of data in primary memory at address 2100000 will be moved to port zero of a 16-bit wide Feature Card.

To request work from the Feature Card, a sysgen sequence must have been successfully executed. This may be done by reading the sysgen completion report and by checking for a successful status opcode. This operation should be done before any other access is attempted.

The queues have been declared and passed to the Feature Card through the sysgen data block. The System Board fetches the unload pointer from the completion queue to determine the location of the sysgen completion report and reads the status:

```
if(TIMEOUT)
        Feature Card is dead, assert PFAIL, and halt.
else{
        Read SYSGEN status completion report at unload pointer in
                completion queue.
        Increment unload pointer by size of an entry
        Enter operational mode.
}
```

If the Feature Card does not respond with the sysgen status before the timeout interval, the System Board assumes the Feature Card failed its sanity checks and is dead. This interval should be controlled by a hardware timer, not software since a software timer changes with processor speed. If the Feature Card responds with the proper status, the System Board increments the unload pointer of the completion queue so that the sysgen report is erased. The Feature Card is ready to accept jobs.

Requesting a Job from the Feature Card

The System Board enters job requests by placing the job entries on the request queue of the subdevice and then alerting the Feature Card through an INT1. This interrupt, used in operational mode, is the attention interrupt. The System Board writes the job entry to the load pointer and increments the pointer by the size of the entry. It then sends the attention interrupt to the Feature Card. The Feature Card, in turn, begins processing the job at the unload pointer.

In this example, the ENTRY structure looks like this:

27FF (hex)	0	0	00	WRITE
2100000 (hex)				

The Feature Card firmware sees the WRITE opcode and calls the application write subroutine. It passes the subroutine a pointer to the ENTRY structure.

Completion Reports from the Feature Card

Completed jobs, with the exception of a reset type command, must register a completion report with the System Board. A completion report is another structure of type ENTRY. It contains a byte count—the byte being transferred when the job termination occurred. A normal termination places the offset of the last byte transferred in this field. If an error occurs, the byte number on which the error occurred is registered in this location.

The cmd_stat field is zero. This indicates that a status report is in the completion queue. The opcode is NORMAL unless a unique code appears in this field. If a code appears in the opcode field, an error is indicated. The subdevice and buffer address remain unchanged between the request and completion queue entries.

When the job is done, the Feature Card writes an interrupt request to the System Board. The System Board, then, removes the completion report from the queue by responding to the return status as defined by the driver. The System Board increments the unload pointer.

For this example, the Feature Card should leave a completion entry for a successful completion, which looks like this:

27FF (hex)	0	0	00	NORMAL
2100000 (hex)				

If an error occurred while transferring byte 20 (the 21st sequential byte), the completion report would look like this:

14 (hex)	0	0	00	Error Code
21000000 (hex)				

The System Board and the Feature Card both monitor the queues to make sure the limits of each queue are not overrun. If the System Board detects that either the load pointer on the request queue or the unload pointer on the completion queue are at the end of their allocated space, the System Board resets them to the beginning of the entry structure. The Feature Card maintains the unload pointer on the request queues and the load pointer on the completion queue.

As part of this pointer update, the Feature Card and System Board must ensure that the load pointer never catches the unload pointer. If this happens, it indicates that the queue is empty when it is really full. So, the load pointer must always point to an empty slot in the queue.

When a queue becomes full, the Feature Card waits for the System Board to clear the completion queue before accepting any more jobs for that subdevice. Before the System Board can add more jobs, it waits for the Feature Card to clear the request queue for that subdevice.

Express Work

In some situations, the System Board may need to send a job to the Feature Card for immediate processing. The express entries in the request and completion que_set structures are used to do this. The System Board builds its express request and places it in the express entry reserved at the head of the queues. An express interrupt (INT0) is sent to the Feature Card.

The Feature Card suspends the current operation, if possible, and downloads the request and any indirection, for servicing. If the current operation is uninterruptible, the Feature Card services the interrupt at the next break.

The Feature Card constructs a completion entry for the express jobs run. The express completion entry, however, is placed in the special space reserved for it at the head of the completion queue. A normal completion interrupt is sent to the System Board to indicate the job was done.

The System Board uses the seqbit to indicate that this is a new express completion entry and not a leftover from a previous express operation. This flag, stored in a local variable, is initialized to zero at sysgen time. Every time an express completion is placed in the queue, seqbit is complemented by both the System Board and the Feature Card and placed in the correct field in the completion entry. This lets a type of semaphore differentiate between old and new entries since only one attention interrupt is provided back to the System Board.

Error Handling

The application firmware must provide space in low core for error and interrupt vectors, which straps all exceptions and interrupts to handlers. The error handlers should mask all interrupts, write the error code into the interrupt vector register (all error codes have bits seven to four equal to one, allowing 16 possible error codes), and assert the Feature Card fail (PFAIL0) and hang.

When the System Board sees PFAIL0, it polls the Feature Cards by checking the most significant nibble of all ID/vector registers. This causes each Feature Card to get an INT0. The sane Feature Card can distinguish this from an express interrupt, which is also INT0, because of the order in which it receives

INT0s. Express interrupts occur first. Thereafter, INT0s are regular INT0 interrupts.

Common Status and Commands

Because of the common queue protocol, many opcodes are the same across Feature Cards. Common command opcodes include the downloading code, the code to examine memory, and the code to jump to a new address. Firmware makes extensive use of these common routines to download code to the board for remote execution. The first 32 eight-bit opcodes are reserved for common commands. Opcodes 20 through FE are application defined. FF is reserved for common I/O activities.

Determine Subdevices

The System Board uses the determine subdevices (DSD) command [opcode 05 (hexadecimal)] to determine the subunit equipage of the Feature Card. A buffer in primary memory is passed as the address. All other fields are " don't care" s. The information DSD uses is determined in usrinit( ) by the application by defining a subdevs structure and populating it with the needed data. The subdevs structure is type defined (typedefed) in the header file dsd.h as DSD_POLL. The DSD command then uploads this area to the System Board into the address specified.

The first DSD each Feature Card receives after a complete system reset comes from a System Board routine that doesn't know the queue sizes of each Feature Card. This routine has to use express interrupts (INT0) to do the DSD command, so all Feature Cards must be able to handle at least three INT0s: one or more for the read of the board ID register, and one for the DSD.

Common Status

Common status opcodes are useful for returning error status relating to the firmware code and queue entries that are common across all Feature Cards.

FIRMWARE IMPLEMENTATION

The first 32 eight-bit opcodes are reserved for common status.

- Normal Status—NORMAL: The status code (0, in hexadecimal) for a normal completion of a job is defined as NORMAL.
- Fault Status—FAULT: The fault status (01, in hex) must be returned if the hardware vectors to one of the default exception handlers during execution of a job.
- Invalid Queue Entry—QFAULT: The invalid queue entry status (02, hex) may be returned if the command opcode is undecodable or if requests are made to do impossible actions (such as abort a job that does not exist).
- Successful Sysgen—SYSGEN: This status opcode (03, hex) is returned after a successful sysgen.

Checking Configuration Information

The system definition (sysdef) command analyzes the file and extracts configuration information. This information covers all hardware devices as well as system devices and all tunable parameters. The sysdef program uses the master files in /etc/master.d to get information about devices. Hardware devices, software devices, and loadable modules are defined in separate files in master.d. Master.d files contain the element characteristic (like block or character device), number of interrupt vectors needed by a hardware device, handler prefix, software driver major number, number of subdevices, interrupt priority level, and, optionally, a dependency list. Master files are described in more detail in an appendix.

DESCRIPTION OF 3B5 COMPUTER FIRMWARE ENTRIES

The 3B5 Computer can come in one or two cabinets, depending on the selected configuration. All 3B5 Computers include eight standard RS-232-C asynchronous input/output (I/O) user terminal ports. Some of the features offered are: high-speed Central Control (CC) with cache, disk and tapes systems, RS-232-C and RS-449 peripheral interfaces, an uninterruptible power source, and peripheral interface growth capability. The 3B5 Computer is a 32-bit computer with eight megabytes of primary memory, expandable in one and two megabyte increments.

Three different hardware units are available for mounting circuit boards in the various 3B5 Computer models. They are:

- Basic Control Unit—Mounts in the basic computer cabinet of all 3B5 Computer models.
- Growth Control Unit—Mounts in the basic cabinet in Models 100B and D and Models 200E, F, and G. (Not included in Models 100A and C.)
- Extended Local Bus Unit—Mounts only in the second computer cabinet in Models 200E, F, and G.

The basic control unit is where the main circuit boards, DC power converters, and interconnections are mounted. Major circuit boards include:

- Central Control (CC)
- Main Store Controller
- Integrated Disk File Controller (IDFC)
- Power Control, Interface and Display.

The basic control unit also provides:

- Four slots for Main Store Array
- One slot for the cache unit
- Five I/O slots

- Two general-purpose slots
- One maintenance and test slot (used only by the factory for testing).

The I/O slots can mount the following types of boards:

- I/O Accelerator (IOA)
- Asynchronous Data Link Interface (ADLI)
- Synchronous Data Link Interface (SDLI)
- Synchronous/Asynchronous Data Link (SADL)
- Interface for TELETYPE* Terminals (TTI)
- Intelligent Tape Controller (ITC)
- All types of 3BNET and DATAKIT† Packet Switch interfaces
- Local Bus Extender (LBE).

General purpose slots are for:

- Storage Module Drive Circuit board(s)
- Certain 3BNET and DATAKIT Packet Switch circuit boards.

The growth control unit provides power converters and additional slots for I/O and memory circuit boards. If a growth unit is installed, no disk drives can be installed in the basic computer cabinet. Disk drivers would be in vertical or horizontal growth cabinets.

* Registered trademark of AT&T Teletype

† Trademark of AT&T

The only standard circuit boards that are provided in the growth control unit are the two power converters. The remaining slots in the growth control unit are empty slots and can be used for the following:

- Four slots for Main Store Array circuit boards.
- Ten Input/Output slots.
- Two general-purpose slots.

The extended local bus unit is in a horizontal growth cabinet next to the basic computer cabinets. It provides 14 more I/O slots. Only the I/OA and IOA-supported circuit boards are mounted in the extended local bus unit. All other 3B5 Computer circuit boards are mounted in the basic computer cabinet. The extended local bus unit also provides its own power converters.

All 3B5 Computer circuit boards are housed in slots in the basic control unit, the growth control unit, or in the extended local bus unit.

The CC circuit board is the main processing element of the 3B5 Computer. This microprocessor-controlled board contains:

- Read Only Memory (ROM)
- Random Access Memory (RAM)
- Memory management hardware
- Interrupt handling hardware
- Local bus arbiter
- Sanity and interval timer
- Two I/O interfaces
- Control and status register
- Diagnostic signature analysis circuit.

For more information about circuit boards, see the *3B5 Computer Owner/Operator Manual* and the *3B5 Computer I/O Board Interface Design Manual.*

Diagnostics

The diagnostic structure of the 3B5 Computer is based on self-diagnosing boards. It will support a variety of diagnostic information but the structure of diagnostic information must be uniform across all boards on the local bus. Diagnostic software runs in stand-alone mode, independent of the operating system and hardware configuration.

The boards of the 3B5 Computer have been designed to be self-diagnosing or at least containing their own tests in on-board PROM. So, the diagnostic code on these boards must be structured uniformly and each board must contain all of the tests needed for that board to be diagnosed. Each board must also have a ROM-resident file that contains diagnostic information about that board. The 3B5 Computer does not have an Equipment Configuration Database (ECD).

The 3B5 Computer is a simplex machine made up of a CC, Main Store (MAS), cache, and intelligent (microprocessor-based) peripheral controllers that communicate over a common bus, the local bus.

Input/Output Controller Boards

All boards on the local bus that are Common Interface Circuit (CIC)-based, or have some other diagnostic processing capability, are intelligent boards. Intelligent boards contain all of the diagnostic tests for that board in on-board PROM. The diagnostic control program that resides in PROM accepts diagnostic commands and runs its own tests. A special diagnostic control program, the diagnostic monitor, resides on the CC. The diagnostic monitor oversees the execution of all system diagnostics and can be thought of as a subset of the diagnostic monitor.

Interface Boards

All boards on the local bus that contain no processing capability are considered slave boards. The slave boards need to hold, in on-board PROM, the diagnostics tests necessary for them to be diagnosed properly. The diagnostic monitor on the CC is responsible for downloading and executing these tests, when required.

Board Complexes

Some boards on the local bus driver are board complexes. For example, 3BNET and Network Interface Controller (NIC) are on three boards but the CC sees only one board. Some bits (equipage) in the EDT tell the CC that other boards are included. In other words, the CC treats board complexes as a single board.

The diagnostic monitor only knows the local bus board that drives the board complex. The local bus board, therefore, must diagnose its complex by executing on-board tests, if it is an intelligent board, or supply on-board tests that can be downloaded and executed on the CC. For this reason, the diagnostic control software provides no special handling of subdevices. A subdevice failure message exists, which the controller board passes back to the diagnostics monitor.

The diagnostic software does not contain any provisions for direct terminal communications. It relies on a CC-resident Interactive Access Utility (IAU) program to provide two way terminal communications. The IAU accepts diagnostic input commands, parses them, and transfers control to the diagnostic software on the CC. It also provides a common output routine to be used by the diagnostic software to output diagnostic results.

Since a self-diagnostic philosophy implies that no Equipment Configuration Database (ECD) exists in the 3B5 Computer, some other way to identify board type and system configuration information is needed. To meet this need, every circuit pack connected to the local bus must have an on-board ROM file, an On-board Device Information Table (ODIT), that contains the boards' generic name, release and point of issue information for any ROM-resident firmware and the date (month/year) that the PROMs were installed. In this way, the information is distributed throughout all boards in the system and is automatically updated as boards are added to or removed from the system.

Scenario for the 3B5 Computer

When the system is powered on, the 3B5 Computer automatically resets all of the boards in the system. The CC polls each I/O address. It looks for an ODIT at 0x48F. If a board is in the polled slot, the IAU in the CC creates an Equipped Device Table (EDT), which contains board descriptor information. Every board has its ODIT at the same location so the IAU knows where to find it. Both the ODIT and the relative components must be consistent and uniform across all boards on the local buses.

The ODIT is the only information table that is required to be at a fixed address. All other diagnostic information on boards may “float” instead of being hard coded. This floating is accomplished by maintaining a transfer vector table in ROM on each board. The transfer vector table is pointed to by an entry in the ODIT. The vector table, in turn, contains loaded-dependent addresses of diagnostic structures on that board.

When the IAU finds the ODIT, it copies the information it needs from that ODIT into the EDT entry for that board. CC can tell from an entry in the ODIT if a board is an I/O interface board (dumb board) or an I/O controller board (intelligent board).

FIRMWARE IMPLEMENTATION

Intelligent boards run their own diagnostics and send a pass or fail message to the message queue. Dumb boards have their own diagnostics on-board, too, but the CC must pull the diagnostics into the CC and then execute that code.

The CC checks the message queue to make sure all intelligent boards have passed diagnostics. When all diagnostics pass, the CC resets the whole system again and then boots the UNIX System.

The IO Accelerator (IOA) is one device driver that uses the EDT. The EDT tells the IOA driver which ADLIs, SDLIs, and TTIs are connected to which IOA.

Firmware Header for ODIT

```
/* Fixed ROM address */
struct Odit{
       char dev_size;         /*see item a */
       char dev_type;         /*see item b*/
       char dev_name[9];      /*device name*/
       int:16;
       short phnum;           /*number of diagnostic phases*/
       long phtadr;           /*vector table pointer*/
       long romsz;            /*device ROM size*/
       long ramstrt;          /*RAM start address*/
       long ramsz;            /*RAM size*/
       long errmask;          /*control and status register (CSR) error mask for\
                              UNIX fault recovery*/

       short reldate;         /*release date*/
       short version;         /*version number*/
};
/*****item a******/
#define S8BIT 1                      /*defines 8-bit addressable unit*/
#define S16BIT 2              /*defines 16-bit addressable unit*/
#define S16HBIT 3             /*defines 16-bit with holes (bits 16-31)*/
#define S32BIT 4              /*defines 32-bit addressable unit*/

/****item b*****/
#define IOCONTRL 1            /*I/O controller board*/
$define IOINTERF 2            /*I/O interface board*/
```

The transfer vector table is maintained on each local bus board. The vector table contains addresses of diagnostic structures on that board that need to be accessed by the diagnostic control firmware on the CC.

Valid ODIT, Vector Table, and Phase Table

```
/*ODIT*/

struct Odit cic_odit = {
              S16BIT,
              IOCONTRL,
              "IDFC",
              20
              (unsigned long)&cicvects,
              0xc000,
              0x30000,
              0x2000,
              0x0000009F,
              0x0282,
              0x0001    };

/*Vector Table*/

extern struct Msg_req cic_req;
extern struct Msg_que cic_que;

struct vect cicvects = {
              0x03L,
              (unsigned long)tbl,
              (unsigned long)&cic_req,
              (unsigned long)&cic_que,
              0X00L,
              0x00L
              };
```

```
/*Phase Table*/

#define NRMLPH 0      /*non-interboard communication normal phase*/
#define LBTPH 1              /*local bus test phase*/
#define DEMAND 2      /*demand phase*/
#define DMNDSUB 3     /*demand phase with helper subdevice specified*/

       struct phtbl {
              char type;
              short phsz;
              int (*phaddr)();
       };

extern cpu68_1(),cpu68_2(),cpu68_3(),cpu68_4(), rom(),ram(),signnl(),
idfcsr(),intcntl(),intencr(),cicitst(),iencrtst(),sit(),lbcntl();

       struct phtbl tbl[] = {
              {NRMLPH,1500,cpu68_1},
              {NRMLPH,1500,cpu68_2},
              {NRMLPH,1500,cpu68_3},
              {NRMLPH,1500,cpu68_4},
              {NRMLPH,1500,rom},
              {NRMLPH,1500,ram},
              {NRMLPH,1500,signal},
              {NRMLPH,1500,idfcsr},
              {NRMLPH,1500,intcntl},
              {NRMLPH,1500,intencr},
              {NRMLPH,1500,cicitst},
              {NRMLPH,1500,iencrtst},
              {NRMLPH,1500,sit},
              {LBTPH,1500,lbcntl}
              };
       short int thlsz = sizeof(tbl)/sizeof(struct phtbl) +1;
```

Each board that has diagnostic firmware also contains descriptor information (size, address, etc.) on each diagnostic routine (phase). This descriptor information is described in a structure of a phase table. A phase table, simply put, is a linear list on n sequential entries, where n is the number of phases on that board, and each entry is a phase descriptor. The phase table pointer contained in the ODIT is the address of the first phase descriptor, n. The number of phases on a board (n) is also contained in its ODIT.

System Generation

During sysgen, the following happens:

- The device driver sets the request reset bit in the intelligent board.
- Once the board has initialized itself, it sets the reset complete bit in its CSR.
- While the board was being initialized, the driver has been polling the reset complete bit in the intelligent board's CSR and either recognizes that the reset is complete or recognizes that the board failed to reset. (A countdown timer displays an expiration message at the console if the board fails to reset.)
- If the board reset, a stand-alone command block (composed of a sysgen operation code and a pointer to the main store resident sysgen data block) is written to the board's RAM at the Offset Stand-Alone Command Block (OSACB) defined in /usr/include/sys/firmware.h. The sysgen data block is written in main store and describes where the work queues and queue pointers are located. For an example of a sysgen data block, see /usr/include/sys/sysgdb.g.
- The driver sets Program Interrupt Register 1 (PIR1) in the PIR on the intelligent board and goes into a countdown loop of sufficient length to allow sysgen to complete.
- On the intelligent board, the sysgen data block is copied from main store to the intelligent board. The queue handling initialization is completed. The intelligent board then issues a hard interrupt on vector zero back to the CC as an indication that sysgen is complete.
- If the countdown expires, the driver displays on the console that the device failed sysgen. Otherwise, the interrupt handler captures the interrupt and sets a flag that indicates that sysgen is complete. This flag is what the initialization routine is loop on so as soon as the interrupt handler relinquishes control, the initialization routine notes that sysgen is complete.

This procedure is followed for each intelligent device.

Queues

Communication between the driver and its peripheral is usually done through a set of queues manipulated by the driver and the device's firmware. All block devices are intelligent and, therefore, have operational firmware that knows how to manipulate these queues. Each piece of firmware in the 3B5 may use different queues to communicate with the device driver. For this reason, writers of device drivers for the 3B5 need to get specific information from the firmware designer. Basic philosophies, however, are similar. At least two queues, a job request queue and a job completion queue, are used. These queues may be replicated if the controller manages more than one device and jobs to each of those devices can be handled in parallel such as the Intelligent Disk File Controller (IDFC).

Along with each of these queues are associated queue pointers: a load pointer and an unload pointer. The load pointer always points to the next available slot in the queue to insert a job request. The unload pointer always points to the next job request to be processed.

The queues are circular so the unload pointer chases the load pointer. To guarantee consistency, the driver only alters the request queue load pointer and the completion queue unload pointer. The firmware manipulates the alternate two. The only exception to this is at initialization time when the driver sets both to the top entry in the queue.

Job Scheduling

Once again, firmware designers may implement job scheduling in different ways so device driver writers need to get specific information from the designers. Basically, a job is scheduled by following these steps:

1. The device driver checks the job request queue to see if a free slot exists. If there isn't a free slot, the process goes to sleep. When a job is completed, the firmware interrupts the driver and the driver wakes up sleeping processes waiting for empty slots. Sleeping processes will be assigned empty slots. Any processes that are not assigned empty slots will go to sleep again.

2. The driver creates a job in the request queue.

3. The driver increments the job queue load pointer and wraparound if its incremented value is greater than the end address of the queue.

4. The device driver interrupts the block device to call its attention to the new job in the queue.

5. The firmware fetches the jobs from the request queue, does the assigned work, and places status reports in the completion queue.

6. The firmware interrupts the driver to call attention to the completion report in the completion queue.

7. The driver removes the completion report, which wakes up sleeping processes. Job scheduling continues, starting again at step 1.

Chapter 3

KERNEL/DRIVER INTERFACE

Chapter 3

KERNEL/DRIVER INTERFACE

Accompanying each add-on peripheral device for 3B2 and 3B5 Computers is a loadable software module called the device driver. The device driver relates the add-on peripheral specific hardware interface to a standard UNIX System kernel interface.

This chapter describes major and minor device numbers and the standard kernel/driver interface supported on 3B2 and 3B5 Computers.

DRIVER CONVENTIONS

Major/Minor Device Numbers and Translations

The major and minor numbers are the means by which the UNIX System associates a peripheral device with a file name. Each number is an 8-bit quantity, and both together are termed the device number. In systems not supporting a self-configuring UNIX System, the major number identifies a specific device driver and is assigned by the master file. The minor number is essentially a sequential number which identifies the *logical* device number for all devices controlled by a driver. It is determined by the order in which the devices are specified in the configuration file used as input by the **config**(1M) program. However, in a self-configuring system, major numbers are associated with physical hardware addresses. This ensures that the special files in the **/dev** directory remain constant and refer to the same physical device regardless of any configuration changes.

For actual hardware devices, the major number is the board hardware address code or board slot. Software drivers are assigned (by the *drvinstall* command, see Chapter 4) major numbers that do not conflict with the major numbers assigned to hardware devices. Major numbers for software drivers range from 16 to 127 for the 3B2 Computer and from 64 to 127 for the 3B5 Computer. The *external* major number (this is the major number that is visible in the special device file) must be translated to obtain the actual driver number, that is, the index in the block or character device switch table. This number (termed the *internal* major number) is assigned at boot time.

The external minor number will only identify the *logical* device on the individual *board* identified by the external major number. It too must be translated to obtain the actual *logical* device number used by device drivers.

Decoding the external device number into the internal major and minor numbers is accomplished by macros in the system header **/usr/include/sys/sysmacros.h**:

```
maj = major(dev);
min = minor(dev);
```

There are two cases where the driver should not perform the external to internal device number translation. The first is unbuffered read or write operations. Here the translation will be performed later when the *strategy* function is called by *physio* (see Appendix B). The second is the *driverprint* function which needs to identify specific hardware (that is, the external device number) in an output message.

KERNEL INTERFACE TO DRIVERS

This section defines the standard kernel/driver interface supported on 3B2 and 3B5 Computers. The driver/kernel interface may be divided into two parts:

- Services the driver provides for the kernel
- Services the kernel provides for the driver.

The interface to the function is defined by arguments passed to the function, operations performed by the function, and results returned by the function.

KERNEL INTERFACE TO DRIVER SERVICES

The services the driver provides for the kernel are generally accessed through the tables. Throughout this chapter, the prefix *driver* is used to replace the prefix which would actually be used. The actual prefix must be specified in the master file for that driver.

Bringing the Device Into Service

The open system call calls the driver *open* function when the special file associated with that device is opened.

```
driveropen(dev, flag)
dev_t dev;
int flag;
```

The parameters of the driver open function are the device number of the device file, a variable of type *dev_t*, and the flags described in the *oflag* field of the open system call [see the open(2)] manual page of the *UNIX System Programmer Reference Manual.* (Type *dev_t* is defined as a *short* in **/usr/include/sys/types.h**). These flags correspond to the flag values in a file descriptor data structure (field *f_flag* in the header file **/usr/include/sys/file.h**).

The open function must validate the minor portion of the device number (accessed via the *minor* macro). An incorrect special device file could cause the driver open function to be passed an incorrect device number. To perform this check, a driver will typically use a variable whose value is set up at " boot time." This is described in more detail in Chapter 4.

Additional work done by the open function is very much dependent upon the device being opened. For example, the open function for a removable media disk drive could lock the door to the disk drive and cause the disk controller to select the drive. The open function for a terminal interface controller would establish a connection between the user process issuing the open and the device being opened.

An open function should:

1. Adhere to the use of the flag parameter as specified in the open(2) manual page when applicable.
2. Set up the device for data transfer.

Taking a Device Out of Service

The kernel calls the driver *close* function when the last process that is using the device issues a close system call or exits.

```
driverclose(dev, flag)
dev_t dev;
int flag;
```

The parameters of the *close* function are the device number and the flags from the file descriptor associated with the open request (field *f_flag* in system header **/usr/include/sys/file.h**). The *close* function typically performs inverse operations to those of the open function. Note that a device may be opened simultaneously by multiple processes, and the *driveropen* function is called on each open. However, the *driverclose* function is only called once, when the last process that is using the device closes it.

A close function should:

1. Adhere to the use of the flag parameter as specified in the close(2) manual page when applicable

2. Make the device available for later use.

Character Device Data Transfer

When the user issues a read (write) system call from (to) a special file, the driver associated with the special file is called to initiate and supervise the data transfer from (to) the device to (from) the user data area (ublock).

```
driverread(dev)
dev_t dev;
```

and

```
driverwrite(dev)
dev_t dev;
```

The only argument passed to either the driver read or write function is the device number. If an error occurs the functions should set the *u_error* flag. Read and write parameters are defined in **/usr/include/sys/user.h**.

As was stated earlier, driver read and write functions are accessed through the character device switch table. When the device being accessed is truly a character device (not a block device being accessed through the character device switch table), the data being transferred must be buffered by the driver. That is, data is not transferred from the device directly to the user data area; instead, the driver transfers data between the user area and its own buffers and between its buffers and the device. When data is transferred to the device the reverse holds true.

Drivers for low-speed character devices like terminals and printers which are required to perform semantic processing of data typically use the *clist* data structure and the *tty* subsystem to perform buffering and transfer in and out of the user data area. The *clists* and the interface to the *tty* subsystem are described in detail in Appendix A. Drivers for high-speed character devices like network interface boards generally set up their own buffering schemes. The kernel provides several functions for copying data in and out of user memory.

When the device being accessed by the read or write system call is a block device, the data transfer is usually performed directly from user memory to the device. To facilitate this, the kernel provides the *physio* function. The device driver read and write functions call the physio function which in turn calls the driver strategy function. The physio function is discussed in Appendix B.

Block Device Data Transfer

Block device drivers must provide a *strategy* function to handle the data transfer to and from the device. A strategy function takes as its argument the address of an instance of the *buffer header* data structure defined in the system header file **/usr/include/sys/buf.h**.

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

A detailed description of the buffer header is presented in Appendix B. All information about the data transfer is contained in the buffer header. The buffer header is also used to return status and error information to the kernel, which conveys the information to the user.

The strategy function is responsible for initiating block data transfer. It validates the buffer header information and then uses this information to generate the appropriate device operations required to start the block data transfer.

Device Access Other Than Read/Write

The *ioctl* function is traditionally and most commonly provided by drivers for terminal interface devices. It controls device hardware parameters and establishes the protocol used by the driver in the semantic processing of data. However, the ioctl function has become the catch-all function for facilitating all device access that is not normal read/write access.

```
driverioctl(dev, cmd, arg, mode)
dev_t dev;
int cmd, arg, mode;
```

The driver ioctl function is accessed through the ioctl system call [see the *UNIX System Programmer Reference Manual* for the manual pages on ioctl(2)]. The function takes four arguments:

dev	The device number.
cmd	A command argument that the driver ioctl function interprets as the type of operation to be performed. The command types vary across the range of devices; the user manual specifies the command types that must work for terminals. Terminal interface drivers typically have a command to read the current ioctl settings and at least one other that sets new settings. The kernel does not interpret the command type; so, a driver is free to define its own commands.
arg	An arbitrary argument that can pass parameters between a user program and the driver. When used with terminals, the argument can be the address of a structure in the user program that contains settings for the driver or hardware. The driver reads the settings from the user program via the *copyin* function and does the appropriate operations. Similarly, the driver collects current settings and uses the *copyout* function to return settings to the user program structure. Alternatively, the argument may be an arbitrary integer that has some meaning to the driver. The interpretation of the argument is driver dependent and usually depends on the command type; the kernel does not interpret the argument.
mode	A mode argument that contains values set when the device was opened. The mode need not be used. However, the driver can use it to determine if the device was opened for reading or writing by checking the *FREAD* or *FWRITE* setting.

The use of the ioctl function by nonterminal drivers is open ended. For example, on 3B5 Computers, the ioctl function is used to rewind tapes. On 3B2 Computers, the ioctl function is used to format diskettes and implement bad block handling.

Block Device Print Warning Message

Block device drivers must provide a *print* function to output appropriate warning messages when various abnormal situations are detected by the kernel, for example, out of space on the device.

```
driverprint(dev, str)
dev_t dev;
char *str;
```

The nature of the problem is included in the character string *str*; this should be included in the driver output. Normally, the driver will also include information in the output to identify the specific hardware associated with *dev*. The kernel function *printf* is usually used by the driver to output a message to the system console.

Boot Time Initialization

Most drivers and peripheral devices require initialization before the kernel can use them. This initialization is typically performed during system boot. Driver functions to perform initialization generally initialize driver global variables and data structures and initialize peripheral hardware.

```
driverinit()
```

and

```
driverstart()
```

The *init* functions are used to initialize drivers and devices essential to initialization of the kernel. Drivers for the *root* file system device or the *system console* device would require *init* functions. The *start* functions perform initialization for devices and drivers which are not essential to initialization of the kernel. These functions are called immediately after kernel initialization.

Power Failure Functions

The 3B5 Computer supports an optional power holdover feature which allows the system to shutdown gracefully on power failure. Each driver may provide a function, *driverclr*, to facilitate this.

```
driverclr()
```

The *driverclr* function sets a flag prohibiting the initiation of any further I/O activity and notifies user processes awaiting I/O that the I/O has failed. Another function of *driverclr* is to purge all outstanding I/O requests that may be pending. The *driverclr* function accepts no arguments and returns no values.

Handling Interrupts

The driver interrupt handler is entered when a hardware interrupt is received from the device controlled by the driver. This function is passed a single argument: a number or " interrupt vector" corresponding to the interrupting device.

```
driverint(ivec)
int ivec;
```

The driver must translate the input argument into a *logical* controller number in order to be able to access the I/O hardware and to locate the status information associated with the interrupting device. This translation is dependent on the number of unique interrupt vectors per device controller and typically involves a shift operation. For example, suppose that for a given device, each controller has eight different interrupt vectors. Then to get the *logical* controller number corresponding to *ivec* shift right three times as follows.

```
ivec >> 3
```

The function of the device driver interrupt handler is very much dependent upon the device. In general, driver interrupt functions service the interrupting hardware or peripheral firmware, and notify the processes associated with the device that a transfer is complete. To service the peripheral hardware or firmware, an interrupt function might read data from firmware/driver buffers or hardware registers into driver buffer space. It might read transfer completion status information from driver/firmware buffers or hardware status registers and respond accordingly. When intelligent I/O drivers maintain their own work queues, the interrupt function also scans the work queue for new work and starts the next job if new work is found.

Since an interrupt is not associated with any user process, the driver interrupt handler must not attempt to access the user structure. Also, any previous local variables set up by the driver are not available. Each interrupt causes execution to begin on an empty stack.

Reporting Errors

Drivers must report I/O errors to the kernel. Character device drivers return errors in the *u_error* field of the *user* data structure unless the error comes from an interrupt handler. Block device drivers return errors in the *b_flags* and *b_error* fields of the *buffer header* data structure. These data structures and error codes are discussed later.

DRIVER INTERFACES TO KERNEL SERVICES

In order to ensure portability of drivers from one release of the UNIX System to the next, the UNIX System kernel provides a set of standard service functions and data structures for use by the drivers. Use of any other feature or data structure can result in a driver which will not port to a new software release. The discussion of the driver interface to kernel services is divided into three parts: general system constants and data types; kernel services which provide the driver access to user data; and services which allow the driver to control system execution flow.

General System Constants and Data Types

The kernel defines general system constants and data types for use by drivers in the system header files *types.h*, *param.h*, *sysmacros.h*, *signal.h*, and *errno.h*. These headers are found in the system directory **/usr/include/sys**. *Types.h*, *param.h*, and *sysmacros.h* should be included by all drivers, as they contain type definitions, macros, and parameters which are used in other system header files. If the driver sends signals to other processes [see psignal and signal in this chapter and the signal(2))manual page in the UNIX Programmer Reference Manual it must include the file *signal.h*. Standard error codes returned from drivers to the kernel in the *user* data structure and *buffer headers* are contained in the header file *errno.h*. The *intro(2)* manual page of the UNIX Programmer Reference Manual defines the error codes. To ensure portability of drivers, future changes to any of the header files mentioned in this paragraph will be upward compatible.

User Data Access

Before discussing the kernel services which allow the driver to control execution flow, we will discuss the services which allow the driver to control data flow. Included are descriptions of key data structures related to the user process and its state. These are also important to the execution flow services described later.

Defining the User Process and Its State

Each process has allocated to it one *user* and one *proc* data structure. Together these structures contain all the per-process information defining the process and its state to the UNIX System kernel. These structures are defined in **/usr/include/sys/user.h** and **/usr/include/sys/proc.h**, respectively. Since the *user* and *proc* data structures are basic to the UNIX System kernel, they are subject to change from one software release to another. However, some drivers require access to certain fields in these data structures. Therefore, the size and offset of a limited set of fields in these structures are guaranteed not to change. Elements of the *user* structure which will be frozen are:

u_error	This field is used to return errors information (see error.h) to the kernel which is then passed on to the user. The introduction to section two of the UNIX System manual describes the error codes.
u_base	This field specifies the base address for I/O to and from user data space.
u_offset	This field specifies the offset into the file from which or to which data is being transferred.
u_count	By convention, this field specifies the number of bytes which have not yet been transferred during an I/O transaction.
u_segflg	This field is an I/O flag.
u_ruid and **u_rgid**	These fields identify the real user and group ids, respectively.
u_uid and **u_gid**	These are the effective process user and group identification fields. They may be used to provide a process identified by the user and group identification fields (u_ruid and u_rgid) with the access permissions of another process or process group.
u_procp	This field is the address of the *proc* structure associated with this user structure.
u_r	This union is used to return values to system calls.

u_ttyp	This field is the address of the process group field (pgrp) of the tty structure for the terminal associated with this process.
u_qsav	This field is used as an argument to the kernel longjmp function.

The fields in the proc structure which will be frozen are:

p_pid	This is the unique process identification field.
p_uid	This is the user identification field.
p_pgrp	This field identifies a process group.

Translating Virtual Addresses to Physical Addresses

When a driver receives a memory address from the kernel, that address is virtual. The use of that address by the driver itself works correctly as memory management is performed by the CPU. However, some devices which access memory directly deal only with physical memory addresses. In such cases, the driver must provide the device with physical memory addresses. To perform the translation from virtual addresses to physical addresses, the kernel provides the function *vtop.* The 3B2 and 3B5 Computers provide slightly different *vtop* functions.

For the 3B2 Computer:

```
paddr_t
vtop(vaddr, p)
char *vaddr;
struct proc *p;
```

The *vtop* function accepts as arguments a virtual address and a pointer to a *proc* structure. (The *paddr_t* function is the type for a physical memory address. It is defined in the system header file **/usr/include/sys/types.h**.) The virtual address is the memory address being translated. The pointer to the *proc* structure is used by *vtop* to locate the information (tables) used for memory management. To indicate that the address is in kernel virtual space or in the driver itself, the second argument should be *NULL.* Block device drivers which can transfer data directly in and out of user memory space must use the *b_proc* element of the *buffer header* data structure as the second argument. The *vtop* function returns the translated address. If for some reason the virtual address cannot be translated correctly, vtop returns zero.

For the 3B5 Computer:

```
paddr_t
vtop(vaddr)
char *vaddr;
```

The *vtop* function accepts the virtual address to be translated as its argument. This can be an address in the current user virtual address space, in the kernel, or in the driver itself. The *vtop* function returns the translated address or a -1 if the virtual address cannot be translated correctly. Note that *vtop* can only be used to translate a user address in the currently active process. As a result, it cannot be used in an interrupt handler or a time-out entry.

Transferring Data from User Data Space to Driver Buffers and Back Again

Many drivers provide their own method of buffering data between the devices they control and the user data area. To assist the data transfer between driver buffers and the user data area, the kernel provides drivers with the *fubyte, fuword, copyout, subyte, suword,* and *copyin* functions. The *copyout, subyte,* and *suword* functions are used to copy data from driver buffers to user data space.

```
copyout(driverbuf, userbuf, n)
char *driverbuf, *userbuf;
int n;

subyte(userbuf, c)
char *userbuf, c;

suword(userbuf, i)
int *userbuf, i;
```

The *copyout* function is used to copy blocks of information from the driver buffer to user space. It accepts as arguments the address of the driver buffer, the address of the user buffer, and the number of bytes to be copied. The *subyte* function is used to copy a single character from the driver buffer to user space. It accepts two arguments: the address of the user buffer and the character to be copied. The *suword* function is used to copy a single integer from the driver buffer to user space. It accepts two arguments: the address of the user buffer and the integer to be copied.

By convention, within the UNIX System kernel, when a driver read or write function is entered, the *u_base* field of the *user* data structure contains the address of the buffer in the user address space, and the *u_count* field contains the number of bytes remaining to be transferred. After the *copyout*, *subyte* or *suword* function call completes, the driver should increment the value of the *u_base* field and decrement the value of the *u_count* field by the number of bytes transferred. These functions return 0 if the transfer is successful. If a nonzero value is returned, the *u_error* field of the *user* structure should be set to *EFAULT*.

The functions *copyin, fubyte* and *fuword* are used to copy data from a user buffer to a driver buffer. They are used in like manner to *copyout, subyte,* and *suword,* respectively.

```
copyin(userbuf, driverbuf, n)
char *driverbuf, *userbuf;
int n;

char
fubyte(userbuf)
char *userbuf;

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

Instead of using copyin in the preceding example, you could also use the *iomove(addr, n, flag)* to move the data.

The kernel is not responsible for a bad address in *u_base,* which is set as a result of a user system call. For example, if the user is reading 512 bytes from the device into a data structure that is only 256 bytes long, the system will not detect the error.

Verifying User Access Permissions

A device driver must sometimes verify that a user has access permission to the memory area specified in a read or write system call. The kernel function *useracc* performs this verification.

```
useracc(base, count, access)
caddr_t base;
int count;
int access;
```

The *useracc* function accepts three arguments. The first argument is the start address of the *user* data area (the *u_base* field of the user structure). The field *base* is of type *caddr_t.* (Type *caddr_t* is defined as a *character pointer* in **/usr/sys/include/types.h.**) The second argument is the size of the data transfer in bytes (the *u_count* field of the *user* structure). The third field specifies whether the access is a read or write. The defined constant B_READ specifies a write into memory (the user is performing a read operation). This requires that the user have write access permission for the specified data area. The defined constant B_WRITE specifies a read from memory; that is, it requires read access permission for the data area. (B_READ and B_WRITE are defined in the system header file **/usr/include/buf.h**, which must be included.)

The *useracc* function returns 0 if the user does not have the proper access permission to the memory specified and returns 1 otherwise. If the user does not have the proper access permission, the driver should set the *u_error* field of the user structure to *EFAULT*.

The TTY/Character Subsystem

The UNIX System kernel provides common functions and data structures which can be used by drivers for low-speed character devices like printers and terminals which are required to perform semantic processing of data. The *tty* subsystem and the *clist* and *tty* data structures provide both buffering and semantic processing of data. Special interface requirements are placed on device drivers which use the *tty* subsystem, which is discussed in Appendix A.

Block I/O

The UNIX System block I/O subsystem provides a common interface and buffering scheme for block device drivers. The block I/O subsystem is described in Appendix B.

Driver Execution Control

The UNIX System kernel provides functions which allow a driver to control system execution flow. Some functions allow the driver to block its execution for a specific period of time or until a certain event occurs. Others allow the driver to raise, lower, and restore the execution level of the CPU. Still others allow the driver to notify the system console when significant error conditions arise or to actually halt the system when the error condition mandates it. In this section we list these functions, define the interface between them and the driver, and discuss when and when not to call them.

Changing the CPU Execution Level

When a process is executing code in a driver, the system will not switch context from that process to another executing process unless it is explicitly told to do so by the driver. This protects the integrity of the kernel and driver data structures. However, the system does allow devices to interrupt the processor and handles interrupts immediately. The integrity of system data structures would be destroyed if an interrupt handler were to manipulate the same data structures as a process executing in the driver.

To prevent such problems, the kernel provides the *spl*n (set priority level) function where *n* is the priority level which the processor is set. These functions allow the driver to set processor execution levels and, thus, prohibit interrupts below the level set.

```
int s;
s = spln();
```

The *spl*n functions take no arguments since it sets the execution level to a specific value and returns the execution level of the processor at the time it was called.

The selection of the appropriate *spl*n function is quite important. The execution level to which the processor is set must be high enough to protect the region of code; but this level should not be so high that it unnecessarily locks out interrupts which need to be processed quickly. A hardware device is assigned to one of two interrupt priority levels depending on whether it is a character device or a block device. By using the appropriate *spl*n function, a driver can inhibit interrupts from its device or other devices at the same or lower interrupt priority levels. The following tables show the correspondence between interrupt levels (processor execution levels) and the *spl*n functions used by drivers.

INTERRUPT LEVEL		USE	spln FUNCTION TO BLOCK THIS AND LOWER INTERRUPTS
3B2	3B5		
0	0	Base level	spl0
10	6	Character devices	spl4 or spl5
12	10	Block devices	spl6
15	15	Inhibit all interrupts	splhi

The *spl0* function is used by drivers to restore base execution level. A driver function may use *spl0* when the function has been called through a system call; that is, it is known that the level being restored is indeed base level. Other *spl*n functions are used by drivers not to restore execution level but to raise execution level and, thus, protect critical regions of code.

Note that character device drivers may use either *spl4* or *spl5*; these have the same effect.

For regions of code that must be protected from all interrupts, the *splhi* function should be called. However, it should be noted that *splhi* locks out everything, including the clock, and should be used sparingly. (*spl6* on the 3B5 computer also inhibits clock interrupts and should, therefore, be used with care.)

A *spl*n function is usually called in conjunction with the *splx* function.

```
splx(n)
int n;
```

The *splx* function sets the processor level to that specified by its argument *n*. The argument to *splx* must be a value returned by a previous *spl*n function call.

Blocking Execution Awaiting an Event

Drivers must sometimes suspend or block execution to await certain events, such as reaching a known system state in hardware or software. For instance, when a process wants to read a device and no data is available, the semantics of the read may require the driver to wait for data to become available before returning to the kernel. The driver blocks by calling the *sleep* function. This causes the kernel to perform a context switch and schedule another process. The process which invoked the driver awaits the arrival of data for the device.

The *sleep* function takes two arguments: the address (signifying an event) upon which the process will sleep, and a priority value that is assigned to the process when it is awakened.

```
sleep(addr, pri)
caddr_t addr;
int pri;
```

The address used for sleeping may be the address of a kernel data structure or one of the device driver own data structures. The sleep address is an arbitrary address that has no meaning except to the corresponding *wakeup* function call. This does not mean that any arbitrary kernel address should be used for sleeping. Doing this could conflict with other, unrelated sleep/wakeup operations in the kernel. A kernel address used for sleeping should be the address of a kernel data structure directly associated with the driver I/O operation (for example, a buffer assigned to the driver). A driver should not sleep on the address of the user structure. When a process goes to sleep awaiting an event, the driver usually sets a flag in a driver data structure indicating the reason for the sleep.

When the event on which the process is sleeping occurs, either an interrupt handler or another process which knows that processes might be sleeping on the event calls the *wakeup* function to awaken the sleeping process. The *wakeup* function is called with one argument, the address upon which a process could be sleeping.

```
wakeup(addr)
caddr_t addr;
```

The *wakeup* function awakens all processes sleeping on the address, enabling them to execute when chosen by the scheduler. If no process is sleeping on the address when *wakeup* is called, *wakeup* silently returns with no adverse side effects. It is recommended for code readability and for efficiency to have a one-to-one correspondence between events and sleep addresses. Also, there is usually one bit in the driver flag field which corresponds to each reason for sleeping.

Interrupt handlers must not call *sleep*, since they cannot suspend execution. executing when the device interrupted. If the interrupt handler goes to sleep, the process that was interrupted is effectively put to sleep for reasons beyond its control and unpredictable results will occur.

The interrupt handler must, therefore not invoke other functions that could lead to a call to sleep.

Whenever a driver calls sleep, it should make a test after the call to sleep to ensure that the event on which it slept has really occurred. There is an interval between the time the sleeping process is awakened and the time it resumes execution during which the state forcing the sleep may have again been entered. This is due to the fact that all processes waiting for an event are awakened at the same time. The first process given control by the scheduler will ususally gain control of the event. All other processes that were awakened should recognize that they cannot continue and should reissue sleep.

An example demonstrating the use of sleep and wakeup is provided in Appendix C.

Sleep Priorities

The second argument to sleep is used for scheduling purposes when the process awakens. The parameter, called the *sleep priority,* has critical effects on the sleeping process reaction to signals. Sleep priorities range from 0 to 60, where higher numerical values indicate lower priority levels. If the numerical value of the sleep priority is less than or equal to the manifest constant PZERO (defined as 25 on both 3B2 and 3B5 Computers), then the system does not awaken sleeping processes on receipt of a signal. However, if it is greater than PZERO (values 26 to 60), the system will awaken the sleeping process prematurely (that is, before the event on which it was sleeping occurs) on receipt of a signal.

When a driver must call sleep, how can the driver developer determine the sleep priority? The first decision is whether the process should ignore the receipt of signals or not. If the driver puts the process to sleep for an event that is certain to happen, it should ignore receipt of signals and sleep at priority numerically less than PZERO.

If the driver puts a process to sleep while it awaits an event that may not happen, the process must sleep at a priority numerically greater than PZERO. An example of an event that may not happen is the arrival of data from a remote device. When the system tries to read data from a terminal, the terminal driver might sleep (that is, put the current user process to sleep) waiting for data to arrive from the terminal. If data never arrives, the driver will sleep indefinitely. When a user at the terminal hits the break key or even hangs up, the terminal driver interrupt handler sends a signal to the reading process, which is still asleep. The signal causes the reading process to finish the system call without having read any data. If the driver sleeps at a priority value that ignores signals, the process could have be awakened only by a specific wakeup call. If that wakeup call never happened (the user hung up the terminal), then the process sleeps forever.

Standard sleep priorities are defined in the system header file **/usr/include/sys/param.h**.

Getting to a Sane Point in the User Process

When the sleep process is terminated prematurely by a signal, it is necessary to abort the system call and return to a sane point in the user process. The kernel *longjmp* function provides this capability by transferring control any arbitrary depth in the kernel back to the user process. The effect seen by the user is the system call returning with an error (error code EINTR in *u.u_error*). Thus, when a sleeping process receives a signal, the *sleep* function does not normally return to the function which called it, but instead, executes *longjmp.*

However, drivers calling *sleep* must occasionally perform cleanup operations before *longjmp* is called. Typical items that need cleaning up are locked data structures that should be unlocked when the system call completes. If the *sleep priority* argument is or'ed with the defined constant *PCATCH*, the *sleep* function does not call *longjmp* on receipt of a signal; instead, it returns the value 1 to the calling function. If the sleeping process is awakened by an explicit *wakeup* call rather than by a signal, the *sleep* call returns 0. The code sequence

```
if (sleep(sleep_address, priority | PCATCH))
{
        /* driver code cleanup */
        .
        .
        .
        longjmp(u.u_qsav, 1);
}
```

allows the driver to clean up before performing the *longjmp*. The first argument to the *longjmp* function call must be the *u_qsav* field of the *user* structure (this is where the kernel saves the safe return state). No other argument should be used. Likewise, the second argument should always be 1.

Blocking Execution for a Specified Real Time Interval

In some cases, a driver will arrive at a state where there is no more work for it to do, but it wishes to reenter the driver after a given time interval at a specific function. The driver uses the *timeout* function for this purpose. Timeout takes three arguments: the function to be invoked when the time increment expires, an argument with which the function should be called, and the number of clock cycles to wait before the function is called. A sample timeout call is

id = timeout(driverscan, n, HZ/2);

or

id = delay(HZ/2);

where *n* is the parameter to the function *driverscan,* to be called after approximately 1/2 second. *HZ* is the number of clock cycles per second and is defined in **/usr/include/sys/param.h**. On both the 3B2 and 3B5 Computers, *HZ* is defined to be 100. The exact time interval over which the timeout takes effect cannot be guaranteed, but the value given is closely approximated. A function called by *timeout* must adhere to the same restrictions as a driver interrupt handler. It can neither access the user structure, nor can it use previously set up local variables.

The *timeout* function returns an identifier which may be passed to the *untimeout* function to cancel a pending request.

```
untimeout(id)
int id;
```

Recording Time Intervals

The kernel provides a variable which allows drivers to keep track of the time elapsed between events. The kernel variable *lbolt* (" lightning bolt") is incremented by the kernel *HZ* times a second. The variable *lbolt* is of type *time_t*. (Type *time_t* is defined as a *long* **/usr/include/sys/types.h**). A driver can record the value of *lbolt* as different events occur and use the differences of the recorded values to closely approximate the time elapsed between events.

System Error Messages and Halting the System

At times, a device driver may run across error conditions which require the attention of someone monitoring a system console. These conditions may even mandate halting the machine; however, this must be done only with great caution. Except during the debugging stage, a driver should only halt the system in the case of an error which affects the operation of the entire system.

On the 3B5 Computer the kernel function *printf* provides driver access to the system console. The kernel function *panic* is called to halt the machine. The kernel printf function is a scaled down version of the C-library printf. Only the %s, %u, %d, %o, %x, and %D option arguments are recognized. Kernel printf is used to print diagnostic information directly on the system console.

The panic function is called when unresolvable fatal errors are found. The panic function accepts as arguments a message (character string) to be printed on the system console. The panic function identifies the reason for panic, saves the state of the machine and exits the operating system by returning to firmware.

On the 3B2 Computer, drivers do not call panic and kernel printf directly. Instead, the UNIX System kernel provides the function *cmn_err* which in turn calls printf or panic. In the example below, *ARGS* represents a printf argument string where the maximum number of arguments is six.

```
#include <sys/cmn_err.h>
cmn_err(level, ARGS)
int level;
```

The cmn_err function is passed two arguments. The first argument is a defined constant, indicating the severity level of the error condition. The four severity levels are:

CE_CONT	A CE_CONT level message is used to specify that the error message is a continuation of the previous message. This is used when the error message is too long to be passed as one string.
CE_NOTE	A CE_NOTE level message is used to report system events which do not necessarily require user action, but may be of interest to the user. The fact that a sector on a disk needed to be accessed repeatedly before it could be accessed correctly might qualify as such an event.
CE_WARN	A CE_WARN level message is used to report system events which require immediate attention; that is, if no action is taken, the system may panic. For example, when a peripheral device does not initialize correctly, this level should be used.
CE_PANIC	A CE_PANIC level message results in a system panic. Drivers should specify the CE_PANIC level only under the most severe conditions: only when the error condition means that the system cannot continue to function. If the error is recoverable or not essential to continued system operation, the panic severity level should not be specified.

The second argument to cmn_err is basically the set of arguments which would be passed to printf. The cmn_err function with the CE_NOTE argument (on 3B2 Computers) and the kernel printf function (on 3B5 Computers) can be used by driver developers as a tool for debugging driver code but will likely change system timing characteristics.

Sending Signals to Processes

Some device drivers are required to signal processes of the occurrence of certain events. For example, when a user types a break character, the driver controlling the device which receives the character must signal all processes associated with the device that the break was received. The kernel provided functions *signal* and *psignal* are used by drivers for this purpose.

```
signal(pgrp, sig)
int pgrp;
int sig;
```

The *signal* function is called to send signals to all the processes associated with a certain process group. The identification number of the process group being signaled is the first argument to signal. The second argument to the *signal* function is the signal itself. All signals are defined in the system header file **/usr/include/sys/signal.h.**

A second function, psignal, is called by drivers that need to send a signal to a single process rather than to a process group.

```
psignal(p, sig)
struct proc *p;
int sig;
```

The first argument is a pointer to the proc structure of the process being signaled. The second argument is the signal itself.

Managing Driver Memory Allocation

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

Drivers call *malloc* using the following format:

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

where *mp* is the map from where the resource is drawn, and *size* is the number of units of the resource. The *malloc* function returns a zero if all map entries are already allocated. Drivers call *mfree* using the following format:

```
mfree(mp, size, i)
struct map *mp;
int size, i;
```

where *mp* is the map pointer described above, *i* is the index of the first unit of the allocated resource, and *size* is the number of units being freed.

The driver must initialize the map structure by calling *mapinit*, as in

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

where *mapsize* is the number of entries for the map table. Two map table entries are reserved for internal system use, and they are not available for map usage. The *mapinit* function does not cause the map entries to be marked as available. This must be done via *mfree* before objects can actually be allocated from the map.

Suppose a driver wishes to use buffers of size 256 bytes, and it wants to have 10 buffers configured in the system. A simple way to control buffer allocation using maps is to allocate memory for the buffers, as in

```
char mybufs[2560];
```

then to allocate the map.

```
struct map mymap[12];    /* number of buffers + 2 slots for
                                                     * map book-keeping
                                                     */
.
.
mapinit(mymap, 12);
mfree(mymap, 10, 1);     /* 10 buffers available for
                                                     * allocation beginning
                                                     * at buffer 1
                                                     */
```

Allocation of the buffers takes the form

```
if (i = malloc(mymap, 1)) {
         cp = &mybufs[256 * (i - 1)];
         .
         .
} else {
         /* no free memory */
         .
         .
}
```

and freeing the buffer is done by

```
mfree(mymap, 1, i);
```

Chapter 4

DRIVER CONFIGURATION

Chapter 4

DRIVER CONFIGURATION

This chapter describes automatically changing the configuration of a UNIX System kernel when a device driver or software module is added to or removed from a system. This feature allows boot time selection of drivers and modules to be included in the operating system without recompiling the UNIX System object. This feature also includes the utilities to simplify the installation of new UNIX System drivers, thereby, reducing human interaction. The traditional method of generating a conf.c file using *config* are eliminated. This chapter provides guidelines on how to write a driver to function in a self-configuring UNIX System.

The reconfiguration activity that takes place during the boot sequence is hidden from a user. Also, you have the capability of changing system parameters such as buffers and inodes. The schemes for installing and removing drivers and modules are also presented. The mechanism for reconfiguring the UNIX System is common for both the 3B5 and the 3B2 Computers.

This chapter also describes the self-configuration boot program and associated utilities.

DYNAMIC SELF-CONFIGURATION OVERVIEW

The following sections describe the self-configuration feature including the dependencies. A glossary of key terms is included in a glossary in the back of this document.

Self-Configuration Goals

The objectives are:

- Standardize the mechanism for installing drivers on 3B2 and 3B5 Computers. This does not imply that drivers developed for one computer can be easily ported to the other.

- Simplify the user interface for installing and removing drivers. This objective allows a non-UNIX System expert to install a new driver package.

Self-configuration provides:

- A simplified mechanism for including/excluding new drivers into an existing system.
- Interface definition for tools needed to set up files used in the self-configuring UNIX System.

Self-configuration does not include:

- Documentation of the acceptance criteria for drivers and other add-on packages. However, it does provide the foundation for adding new drivers via the self-configuration interface.
- Implementation of automatic tuning of parameters. Tuning of system parameters will continue to be done manually. Automatic tuning of parameters would allow the boot program to set parameter values based on the configuration of the system, possibly by looking at memory size, disk size, and the number of drivers in the system.
- Definition of how to write drivers for use in a UNIX System. It only deals with the requirements for self-configuration. An assumption made is that the detailed interface between the kernel and drivers is covered by the add-on developer's guide.
- Requirements for firmware and diagnostics for peripherals.

Self-Configuration Description

Self-configuration is a tool used to link the UNIX System kernel during the boot process. The boot program which loads the UNIX Operating System determines what hardware devices are actually in the system and generates a UNIX System kernel consisting of the core or basic UNIX Operating System as well as the drivers for the hardware devices. A system file defines any software drivers that are to be loaded.

Self-configuration provides the following features:

- The master file is broken into a set of individual files, one for each module, where a module represents a peripheral driver or an object that provides special software features such as the module supporting interprocess communication. Each individual file is named with its applicable module name. This collection of files is maintained in a new directory called **/etc/master.d**. Each individual file has the exact format of the "old" master file—but there is only a single driver or module entry per file. For convenience, this collection of files will be referred to as the **master** file, as though it were a single file. This allows a reference to the **master** file to mean the *individual file in the master.d directory that corresponds to the name of a device or module.*

- Detect hardware and software configuration changes and automatically relink objects used to build the operating system when required. This removes the need for reconfiguring the system on every boot.

- Provide administrative interfaces for installing and removing drivers. These interfaces are integrated into the simple administration menu driven interface. Part of this enhancement includes providing the tools needed to update system files, such as the **/etc/system** and the restructured **master** files.

- Place system description data, used in tuning parameters, in the UNIX System kernel object file as is done for drivers, such that data structures in the kernel may be adjusted without use of the **config** program or a software generation system (SGS).

Performance and Resource Requirements

Specific requirements have been placed on the implementation of a self-configuring UNIX System. They are:

- *Memory Usage* — (3B2 Computer Only) To reconfigure the UNIX System kernel, minimal memory must be used during normal system operation. Normal user level operations will not be disrupted because the reconfiguration is performed during the boot sequence. The reconfiguration procedure requires that a system have a minimum of 512 kilobytes of memory. During boot, the 3B5 and the 3B2 Computers require the upper 64 kilobytes of the equipped memory to be free for the boot program, requiring the total size of the kernel and the selected drivers to not exceed 350 kilobytes.

- *Disk Usage* — (3B2 Computer Only) The space required on disk for files needed for configuring a minimal UNIX System includes space needed for two copies of the UNIX System kernel, one configured and one nonconfigured; two system description files; and required drivers.

> ***Note:*** A minimal UNIX System is defined as a UNIX System generated by the source in all directories except those in the *io* directory but includes the driver supporting the boot and root devices, console driver, and the system interface to the block and character I/O subsystem.

The maximum disk allocation for a minimal system will not exceed 700 kilobytes.

- *Configuration Time* — The time required for the boot program to configure a UNIX System consisting of the core UNIX and the minimal set of drivers for the 3B5 or 3B2 Computers will not exceed 90 seconds. Configuration is expected to be done infrequently. If no configuration is needed, it should take at most 30 seconds to transfer control to the UNIX System kernel from the boot program.

INTERFACE REQUIREMENTS

Driver Development

This section discusses some of the administrative details involved in driver development. A brief discussion on driver debugging is also included. At the end of this section, a driver development "checklist" is provided. This checklist briefly summarizes the rules and recommendations that driver developers must consider.

Installer's Perspective

Hardware and software drivers can be added to a system, removed from the system or be replaced by a new version. Facilities allowing a user to make changes to the number of drivers in a system are provided through a simple administration interface on the 3B2 Computer and a driver install interface on the 3B5 Computer. Both interfaces serve as a buffer between the user and the system to reduce the number of errors that may occur when making changes to system files. Driver add-on packages require the following functions:

install—used

uninstall—used

The mechanism for adding drivers is included in the Driver Configuration Utilities. What is presented in the following sections are methods for introducing driver packages into a system.

Adding Drivers

Adding new drivers involves two distinct modes: installation of the software package and driver configuration during a reboot. The following steps are the procedures for adding drivers. Step 1 installs the software package while steps 2, 3 and 4 allow the driver configuration to take place. Step 4 occurs when hardware is associated with the driver being added. It is recommended that the state of the machine be single-user mode when adding drivers to the system.

1. Invoke the *install* command. The user is told to load the install medium.
2. Power down the system when hardware is being added, or reboot the system to configure software driver objects into the UNIX System kernel.
3. Install the hardware if a hardware driver is being added.
4. Power up the system.

When these steps have been completed, the new driver will be accessible through the UNIX System kernel. The installer is allowed to install several drivers before the system is powered down or rebooted. Although it is recommended that the system be checked after each driver installation.

Some hardware driver add-on packages will require a different method of installation. Some may require two steps for installing: the install method stated above and an install procedure after the system boots. Others may require that install be done after the hardware is in place. The install steps necessary may be decided by the driver developer.

Removing Drivers From the System

The removal of a UNIX System driver means removing from the system the driver object, associated files, and any hardware associated with the driver. Removal of a driver is done in single user mode. To remove a driver, the user must:

1. Invoke the uninstall command.
2. Power down the system if hardware is being removed, or reboot the system.

3. Remove the associated hardware if required.

4. Power up the computer and boot the system.

During the boot sequence, the UNIX System kernel is configured to exclude any removed drivers. Again, several drivers may be uninstalled before the system is rebooted.

Reconfiguring a UNIX System Kernel

During the configuration of the UNIX System, the position of hardware peripherals is embedded within the UNIX System. Subsequent booting of the UNIX System assumes that these positions remain unchanged. Rearranging peripherals may cause undesirable results and may result in system panics or failures to boot. This is especially a problem on the 3B2 Computer which requires that no slots be empty between boards. This implies that removing one driver may require a complete rearranging of other boards.

Manual Boot Procedure

While configuring the system, some error conditions may require user intervention to return the system to a sane state. Errors occurring on a power up boot will cause the system to go to the firmware level after an appropriate error message has been printed. A manual procedure to aid in error recovery will be available for the more experienced UNIX System user. Under most conditions, the knowledgeable installer may manually boot the system and be allowed to supply data that will allow the boot procedure to continue.

To boot the system manually, enter **boot** from the firmware level. On the 3B2 Computer, the user is prompted for the name of the boot device, which is the integral disk or the integral floppy. The boot program prompts for the name of the system file. The options available to the user for booting the UNIX System are:

— **/unix**

— **/etc/system**

— **/KERNEL**

Note: There are other files that may be given (for example, diagnostic programs), but here we deal only with those options used to boot the UNIX Operating System.

Entering **/unix** causes the boot program to load a fully configured kernel and print warning messages indicating discrepancies with the existing hardware configuration. The UNIX System will not boot if the /unix file contains drivers for hardware which has been removed.

Entering **/etc/system** causes the boot program to use data in that file to configure the system.

When **/KERNEL** or /boot/KERNEL on the 3B2 Computer is entered, the user will be prompted for software drivers to include and exclude. Hardware drivers are automatically included when the boot program detects that the hardware exists. The boot program prints

INCLUDE?

The response may be a carriage return (meaning no additional modules are to be loaded), or the response may be a comma or blank separated list of drivers to be included in the system. Next,

EXCLUDE?

is printed. Again, the installer may enter a carriage return (meaning no drivers are to be excluded), or the response may be a comma or blank separated list of drivers to exclude from the system. On the 3B5 Computer, you are prompted is done for the root, swap, pipe and dump devices. The response to these prompts must be that of the appropriate system file entry.

If the system still fails to boot, the user must call his/her service representative, or restore the system by using system restore procedures.

The Developer's Perspective

The following sections describe the interface that will be used by driver developers. It also presents a scenario for driver install and uninstall.

Driver Install and Uninstall Programs

This feature requires that there exist driver install and uninstall programs to update special files on the computer. The driver install program provides an interface for the driver developer install script to update **/etc/master.d** and **/etc/system** files. It will also invoke the **mkboot** command and place the module object in the boot directory. The driver uninstall program reverses these actions.

DRIVER DEVELOPMENT

This section discusses some of the administrative details involved in driver development. The most important of these is the *drvinstall* which is the developer interface for providing the information needed for self-configuration. A brief discussion on driver debugging is also included. At the end of this section, a driver development "checklist" is provided. This checklist briefly summarizes the rules and recommendations that driver developers must consider.

Driver Addition

The *drvinstall* program handles many of the steps needed to add a driver to a system. It is particularly well suited for use by driver developers during the debugging and testing phases of development. The *drvinstall* program provides the interface for updating the **/etc/master.d** and **/etc/system** files. On the 3B2 Computer, drvinstall also supplies the needed EDT information. It also invokes the *mkboot* command and places the driver object in the boot directory.

The normal syntax of the *drvinstall* command is:

```
drvinstall -n -b -d driver -m master -v version
```

The command arguments to *drvinstall* have the following meanings:

-n	Do not edit the system file.
-b	Do not run *mkboot* on the driver object.
-d driver	This is the path name of the driver or module. The last component of the path name must be the official name assigned to the module by AT&T Technologies. The names of hardware drivers must match the name placed in the EDT. See the section on the master file later in this document.
-m master	This is the path name of the master file entry. *drvinstall* uses this file to update **/etc/master.d**.
-v version	This is the release number of 3B2 or 3B5 Computer UNIX System used in generating the driver.

In addition to the normal parameters used with *drvinstall*, a number of debugging options are also provided.

-o bootdir	This option stores the driver objects in the directory specified by *bootdir* instead of in the normal boot directory.
-s sysfile	This option uses the file specified by *sysfile* as the system specification file instead of the normal **/etc/system** file.

A zero is returned if *drvinstall* completes successfully; a nonzero value is returned otherwise. For software drivers, *drvinstall* outputs the major device number assigned to the driver to the standard output device. For hardware drivers, the *getmajor* command is used to determine the assigned major number(s) as follows.

```
getmajor name
```

where *name* is the official device driver name in the 3B5 Computer or the official device identification code in the 3B2 Computer. There is a one-to-one correspondence between 3B2 Computer device identification codes and official device driver names; both are assigned by AT&T Technologies.

The *getmajor* command outputs (to the standard output device) the major numbers assigned to the specified device as a list of numbers separated by spaces. A zero is returned for success; nonzero for failure.

> ***Note:*** The *getmajor* command must be used after the new hardware is installed.

Included as part of each driver add-on package is an *INSTALL* script which simplifies installation of the driver in a system. Typically, the *INSTALL* script will use *drvinstall* to install the driver object in the **/boot** directory and to update the appropriate configuration files as described above. The *INSTALL* script must also create the necessary special device files by calling the *mknod* command using the major numbers obtained from *drvinstall* or *getmajor*.

On the 3B2 Computer it is also necessary to supply the information needed for the EDT via the **edittbl** command. Appendix G contains details on the use of the **edittbl** command.

Drivers are organized as "add-on" packages, and they are distributed on floppy diskettes for 3B2 Computer systems and either magnetic tape or removable disk cartridge for 3B5 Computer systems. The distribution media includes the driver object and other required files as well as installation procedures. This information is in the form of a mountable file system for disk or in *cpio* format for tape [see UNIX Manual CPIO(1)]. The **cpio -icBdu** command must be used to read the tape. The files in a driver package are organized in two main directories, *install* and *adm.*

The *install* directory contains subdirectories *root* and *usr*. Files in the package which must be moved to the system / or **/usr** file systems are placed in the corresponding subdirectories under *install.* The *adm* directory contains the files associated with installation of the driver. The *INSTALL* file is the actual script to be executed to install the package. The *adm* directory also contains any other scripts or commands needed that are involved in the installation process.

Different interfaces are used for installing drivers on 3B2 and 3B5 Computer systems. The 3B2 Computer uses the menu driven *sysadm* facility which includes an *install* procedure. The 3B5 Computer uses the *periphconfig* command. Although different interfaces are used, the functions and underlying structures are similar. Both copy files from the distribution medium, both execute the INSTALL script, etc. Thus, from the developers viewpoint, the INSTALL script and its associated use of the *drvinstall* is the focal point of the installation activity.

A separate document will be published which will describe the installation procedure as well as provides guidelines for developing *INSTALL* scripts.

Driver Debugging

An additional debugging mode (magic mode) is available. This mode allows a developer to gain control after the boot process has completed, but before transfer is made to the UNIX System kernel. It also produces a memory map which shows the location in memory of the UNIX System kernel and all modules loaded. The "Installer's Perspective" section described the manual boot procedure. This procedure is also used to initiate the debugging mode. In response to the prompt for the system file, a two-or three-word response is given. The first two words must be "*magic*mode." The third word is optional but, if present, it is used as the name of the directory containing the driver objects. The boot program again prompts for the file to boot. At this point, operation of the boot program proceeds, with two exceptions, as in the manual boot mode. A load map is printed which shows each object file loaded and the associated physical and virtual addresses. If an unconfigured kernel object file is being loaded, then the program prompts for all the information usually found in the **/etc/system** file. The response must be either a carriage return or the normal syntax expected for an entry in the system file. When the boot is complete, a return is made to firmware rather than transferring control to the booted object. On systems equipped with **DEMON**, the developer can enter the **DEMON** debug facility or transfer control to the UNIX Operating System.

The load map generated during boot may be used along with a disassembly listing of the driver for planting breakpoints and tracing execution flow through the driver. To use the *crash* debugger, it is necessary to generate a combined symbol table for the booted system using the *mkunix* command. The combined symbol table is also needed for determining the actual address of driver *bss* symbols. (A namelist generated from an unlinked driver does not contain the relative addresses for bss symbols). Of course, if a nonworking, driver prevents the system from booting then it is not possible to generate a namelist.

Normally, this is not a problem; but if it occurs, the developer should look carefully at the driver initialization code as the most likely problem area. It may even be necessary to temporarily remove the initialization calls in order to enable the system to boot successfully.

Another situation that may arise during debugging concerns the use of code optimizers on device drivers. Certain devices may require that hardware accesses be performed only in particular sequences. An optimizer can rearrange code sequences and can even eliminate unneeded instructions. These "unneeded" instructions may in fact be necessary because of hardware requirements. Thus, it is possible that optimized driver codes may not function properly. If this happens, then either the driver must be left unoptimized or the problem area must be moved into a separate, unoptimized file.

Crash is available when the UNIX System is running and will show you the data structures for the drivers.

OPERATIONAL OVERVIEW

A successful boot process requires close cooperation between the operational firmware, the bootstrap program (**mboot**), the main boot program (**lboot**), and the UNIX System kernel itself. The firmware locates **mboot** on the boot device, reads it into memory, and transfers control. It is the responsibility of **mboot** to locate and load **lboot**. **Lboot** will locate and load the UNIX System kernel. Finally, control is passed to the initialization routine located in the kernel.

Firmware Operation

The operational firmware loads the **mboot** program from a selected boot device into a fixed location in the lower portion of memory.

> ***Note:*** The *C* Language compiler does not generate position-independent code; consequently, all addresses within a program are fixed at the time the program is processed by the loader *ld*(1).

This location must be agreed upon by both the firmware and **mboot**. The firmware must provide **mboot** with the identity (and location if necessary) of the boot device. Finally, control is passed to the **mboot** program with a **call** instruction.

Mboot Operation

The **mboot** program occupies 1 disk block or 512 bytes. The **lboot** program is located on the boot device provided by the firmware. **Lboot** is then loaded at an origin of 64K below the top of the first half-megabyte of memory.

Lboot Operation

Lboot is the final boot program. It is responsible for loading the UNIX System kernel. The **lboot** program will perform the steps necessary to implement the hardware self-configuration feature. This step involves loading the object files of the kernel and any drivers or modules, resolving all references, building machine dependent structures such as process control blocks and interrupt routine linkages, and generating data structures that (in systems without self-configuring the UNIX System) were hard-coded in the kernel source.

Mkboot Command

The **mkboot**(1M) command is used to prepare object files for use by **lboot**. Each object file to be loaded by **lboot** must contain an optional header having the correct format. **Mkboot** will build an optional header using information in a **master** file, insert this header into an object file, and write the modified object file into the directory required by **lboot**. Only one a.out file should be supplied to **mkboot** at a time.

Mkunix Command

The self-configuration process results in a completely configured UNIX System kernel in memory. The **mkunix** program takes the memory resident kernel and creates a complete object file. This object file can then be booted directly (bypassing the self-configuration step) or may be used as the *namelist* file for commands such as **ps**(1), **crash**(1M), etc.

Newboot Command

The **newboot** command copies the boot programs to a disk. It is responsible for writing the boot programs in the format expected by the firmware.

DATA STRUCTURES

This section will describe various structures which are necessary for the self-configuration feature. Some existing structures were modified and others were added. The major and minor device numbers for special device files and the **master** file format were modified. Data structures include the equipped device table, the **system** file, and the directory structure expected to exist.

Major and Minor Numbers

The major and minor numbers are the means by which the UNIX Operating System associates a peripheral device with a file name. Each is an 8-bit quantity, and both together are termed the device number. In systems not supporting a self-configuring UNIX Operating System, the major number identifies a specific device driver and is assigned in the **master** file. Further, the minor number is essentially a sequential number which identifies the *logical* device number andfor all devices controlled by a driver. is determined by the order in which the devices are specified in the configuration file used as input by the **config**(1M) program. If the minor number was not changed, self-configuration would result in a complete remapping of minor device numbers whenever a new peripheral is installed. This remapping would critically affect the system device minor number assignments; it would open the possibility of using the wrong device for swapping, for instance, if the **/etc/system** file were not changed *before* the system was booted.

The encoding imposed by self-configuration for major and minor numbers insures that the special files in the **/dev** directory will, without change, refer to the same physical device regardless of any configuration changes. To support this, decoding of the device number into the major and minor numbers is always performed by macros defined in **sysmacros.h**. Therefore, there will be relatively little impact on most existing code. The macros are given below.

```
#define major(x) (int)(MAJOR[(unsigned)((x)>>8)&0x7F])
#define minor(x) (int)(MINOR[(unsigned)((x)>>8)&0x7F]+((x)&0xFF))
```

For drivers associated with hardware devices, the major number is the hardware board address. For software drivers, the major number is assigned by the **master** file. Integral device drivers, those drivers supporting devices that are part of the System Board, are treated as though they are software drivers. This implies that there must be some means of assigning the major number for software drivers at the time the driver is installed, and that this number will be unique for a particular machine. Since 127 is the largest major number, there can be no more than 128 different drivers configured within the UNIX System kernel at any given time. Likewise, since 255 is the largest minor number, there can be no more than 256 subdevices per controller.

3B2 Computer Conventions

For hardware devices, the major number is the board slot. Thus, only major numbers 1 through 15 can refer to hardware devices. Software drivers will be assigned (by the **master** file) major numbers 16 through 127. This number is called the *external* major number because it is visible in the special files for the device. In addition, an *internal* major number for the device is assigned by the boot program.

The external minor number will only identify the *logical* device on the individual *board* identified by the external major number. The major number will be used to index into a table, built by the boot program, to obtain a base number which is added to the external minor number to create the actual *logical* device number now used by device drivers.

3B5 Computer Conventions

For hardware devices, the external major number is the board code assigned by the physical setting of the DIP switch located on the backplane. This value is assigned at the time the peripheral is installed. The following diagram illustrates the bit assignments for bus addresses.

0		LL	BBBB
7	7	54	3210

Bit 7: always zero to prevent sign extension
Bit 6: no specific meaning
Bit 5-4: if 00, then BBBB is the local bus address
 if 01, then BBBB is the ELB address on the
  LBE at local bus address 14
 if 10, then BBBB is the ELB address on the
  LBE at local bus address 15
Bit 3-0: the local or extended bus address of a device

Devices located on the local bus extender (LBE) are identified by both the board code of the LBE and the address on the extended local bus (ELB), as described previously. The domain of major numbers is 0 through 127. External major numbers occur in the following ranges:

0-2:	Available for software drivers
3-15:	Devices on the local bus (not an LBE)
16:	Available for software drivers
17-31:	LBE 14 devices 1-15
32:	Available for software drivers
33-47:	LBE 15 devices 1-15
48-127:	Available for software drivers

External major numbers for software drivers are handled in the same manner as for the 3B2 Computer.

Equipped Device Table

The equipped device table (EDT) is a data structure built and maintained by the resident firmware. The EDT contains information for each peripheral installed on the system. This information is primarily for the use of diagnostic routines; however, **lboot** also makes use of the information. Two pieces of data are required by **lboot**: the identity and hardware address of each peripheral. The exact format of the EDT is machine dependent.

The 3B2 Computer firmware presently uses the same basic technique as the 3B5 Computer firmware; however, there are significant differences in the details of the implementation. The location of the EDT is specified by an address word which is at a fixed location in memory. The system header file defining the format of each element is also named **sys/edt.h**, but an additional header file (**sys/firmware.h**) is required to gain access to the EDT.

Master File

The **/etc/master** file has historically been a *database* of the device hardware characteristics required by the UNIX Operating System. This concept remains unchanged. Additional entries are supported for nondevice related configurable modules. The **lboot** program provides the services of a linking loader. This feature is used to support the inclusion of arbitrary software modules in addition to device drivers. Throughout the remainder of this chapter, the terms *module* and *driver* will be used interchangeably.

The **master** file is used by the **mkboot**(1M) program to obtain device information when generating the device driver files and by the **sysdef**(1M) program to obtain the names of supported devices. It also contains specifications for the generation and optional initialization of all memory resident data structures required by a module. The new format of the **master** file is shown on the manual page in Appendix E.

System File

The **/etc/system** file contains configuration information that cannot be obtained from the equipped device table (EDT) at system boot time. This file generally contains a list of software drivers to include in the load and the assignment of the system devices **dumpdev**, **rootdev**, **pipedev**, and **swapdev** (with **swplo** and **nswap**); as well as instructions for manually overriding the drivers selected by the **lboot** program.

The syntax of the system file is given below. This is essentially the format used on release 1.2 of the 3B5 Computer; however, changes were made to the precise meaning of the *INCLUDE* and *EXCLUDE* directives, and the *IGNORE* directive was removed. Lines may appear in any order. Comment lines must begin with an asterisk. Blank lines or comment lines may be inserted at any point. Entries for INCLUDE and EXCLUDE are cumulative. For all other entries, the last line to appear in the file is used—any earlier entries are ignored. The parser is case sensitive; therefore, all upper-case strings must be entered exactly as shown.

BOOT: path name

The path name specifies the object file to be booted; if the object file is fully resolved (such as that produced by the **mkunix**(1M) program), then no other line in the **system** file has any effect.

INCLUDE: name [(number)] ...

This line is used to identify software drivers or loadable modules from the **/boot** directory which are to be included in the load. It has no effect for hardware drivers. The optional "(number)" specifies the number (default of 1) of "devices" to be controlled by the driver. This number corresponds to the built-in variable *#C* which may be referred to by expressions in part one of the **master** file. The STUBS driver should never be included since it will cause the machine to panic when booted.

DUMPDEV: special-device-path name
DUMPDEV: DEV(major, minor)
ROOTDEV: special-device-path name
ROOTDEV: DEV(major, minor)
PIPEDEV:special-device-path name
PIPEDEV:DEV(major, minor)
SWAPDEV: special-device-path name swplo nswap
SWAPDEV: DEV(major, minor) swplo nswap

These lines are mandatory. They are used to identify the system device to be used for writing a crash dump, the device containing the root file system, the device to be used for pipe space, and the device to be used for swap space (with the beginning block number for swap space *swplo* and the number of swap blocks available *nswap*). The device may be specified in either of two ways. A path name of a special device file may be provided—the major and minor numbers are obtained from the *inode*. An alternative form is allowed in which the major and minor numbers are specified explicitly.

EXCLUDE: name ...

Identifies names in the EDT that are to be ignored. You may want to exclude a driver because no driver exists for the specific EDT entry, or the driver is simply not to be loaded.

File and Directory Structure

The **lboot** program must make some assumptions for the locations of various files and directories. This section describes those structures.

The file system accessed by **lboot** must contain the object files to be loaded and the **system** file. This file system is termed the *boot* file system. It must be located on the disk from which the **lboot** program was loaded. Presently, it is at a fixed location (the constant is compiled into the **lboot** program) on the disk. For the 3B2 Computer, the location of the boot file system is obtained from the VTOC. This file system contains everything necessary to complete the boot process. The following files must be present:

> ***Note:*** The term *must* is used with the understanding that a normal boot is to occur. It is possible to do a manual boot (for example, in a recovery situation) as long as either */unix* is present or */KERNEL* and the essential drivers in the */boot* directory are present.

/etc/system	The **system** file.
/boot	The directory containing the individual, configurable object files created as a result of executing the **mkboot**(1M) program.
/KERNEL	An object file containing the unconfigured UNIX System kernel. This name is the default chosen by the **mkboot**(1M) program; however, it can be thought of as a generic name since the actual name used is obtained from the *BOOT* line in the **system** file.
/unix	A fully configured UNIX System kernel object file. This file is the output of the **mkunix**(1M) program which is run following a self-configuring boot.

A boot may be done as long as either one of the **/KERNEL** or **/unix** files are present. The normal operation of **lboot** assumes both are present and, normally, will proceed by booting the **/unix** file unless there has been a configuration change. In that case, the **/etc/system** file is accessed (which will usually reference **/KERNEL**), and a full self-configuration is initiated.

The **lboot** program does not require anything further. However, additional files and directories are described here to document a standard location.

/etc/master.d	A directory containing the collection of individual files which have the format described by the previous section for the **master** file.
/etc/mkunix	The **mkunix** command resides in the **/etc** directory.
/etc/mkboot	The **mkboot** command resides in the **/etc** directory.
/etc/newboot	The **newboot** command resides in the **/etc** directory.
/etc/sysdef	The **sysdef** command resides in the **/etc** directory.

Driver Naming Convention

The names of drivers in the **/boot** directory are assigned by AT&T Technologies and will be unique. The names (for hardware device drivers) must correspond to entries in the EDT. A driver name can be up to 8 characters in length on the 3B5 Computer and 10 characters on the 3B2 Computer. All names must be in uppercase. Each driver also has an associated two- to four-character prefix that is used to identify driver functions such as open and close. Prefixes are also controlled by AT&T Technologies. To maintain consistency and to prevent duplicate names, the names of master files in the **/etc/master.d** directory must be equivalent to the driver name (lowercase).

DRIVER INTERFACES IN A SELF-CONFIGURING SYSTEM

This section discusses the aspects of kernel/driver interfaces in regard to the self-configuring the UNIX System on the 3B2 and 3B5 Computers.

Device Switch Tables

The device switch tables are generated dynamically at boot time. As driver modules are loaded appropriate device switch table entries are created and filled in with the standard entry points into the driver. Special names must be used for driver entry points. The names are composed of a prefix unique to a given driver and specified in the master file and one of the special suffixes.

Appropriate "stubs" are provided for unneeded entry points using information obtained from the master file (see Appendix E).

Driver Initialization and Power Fail Tables

Some drivers need additional entries for initialization and power fail recovery. These are provided by separate kernel to driver interface tables built dynamically in a manner similar to the device switch tables. The entry names must be the driver prefix concatenated with one of the special suffixes.

Interrupt Vectors

Interrupt vector assignment is performed dynamically by the boot process. Driver interrupt handlers are identified by naming convention. Normally, this is the driver prefix followed by "int". For devices which require paired transmit and receive interrupts, the suffixes "xint" and "rint", respectively, are used. The actual number of interrupt vectors needed for a particular hardware device is specified in the master file. Included in the interrupt vector assignment is the generation of appropriate assembly language code to interface to the driver C Language interrupt handlers.

Device Address Table

As part of the self-configuring boot, an array of controller addresses may be built for each hardware device type. This array must be used by the driver in order to access physical hardware locations. Each array is named by the driver prefix followed by "_addr". The contents of the device address tables differs between the 3B2 and 3B5 Computers. In the 3B2 Computer these tables contain actual addresses and may be used directly to access the hardware. A typical declaration used would be:

```
#include <sys/types.h>
extern paddr_t                      xx_addr[];
```

where "xx" is the appropriate driver prefix. The value of xx_addr[i] would be the base or starting virtual address of the "ith" controller of type "xx".

In the 3B5 Computer the tables contain address translation information which must be loaded into the memory management unit before the hardware may be accessed. A typical declaration used would be:

```
#include <sys/types.h>
#include <sys/mmu.h>
#include <sys/sysmacros.h>
extern struct mmuseg xx_addr[];
```

where "xx" is the driver prefix as above. The value of xx_addr[i] would be the address translation needed to access the base or starting address of the "ith" controller of type "xx". Macros must be used by drivers to load this information into the memory management unit and to reference the controller base address. When the driver is entered on base level from a system call, the *baseio* macro is used to load the memory management unit, and the *BIOADDR* gives the controller base address:

```
baseio(xx_addr[i]);
BIOADDR;
```

Corresponding macros are provided for use by drivers when entered on interrupt level:

```
intio(ipl, xx_addr[i]);
IIOADDR(ipl);
```

where *ipl* is the appropriate interrupt priority or execution level.

In both the 3B2 and 3B5 Computers, the controller address table is indexed by the *logical* controller number. This must be derived from either the interrupt number passed to the driver interrupt handler or the external device number passed to a driver function via the switch table. The translation of interrupt number to address table index depends on the number of interrupt vectors per controller. Typically, there is only one interrupt vector per controller, and the interrupt number passed to the driver is the *logical* controller number. When passed an external device number, the driver must convert this to (internal) minor device number via the *minor* macro. The *logical* controller number is derived from this based on the number of subdevices per controller. As an example, suppose that each controller supports eight minor devices. For a given device number, *dev*, the corresponding device table entry would be:

```
xx_addr[minor(dev) >> 3]
```

Data Structure Allocation

Most of the normal data storage needed by a driver is included in the driver data and bss sections. However, a driver often needs to associate some data structures with physical hardware devices. Since the number of equipped devices, and hence the amount of memory required, is not known until boot time, the self-configuration boot program provides a method for generating such hardware dependent structures. As mentioned in a previous section, the master file specifies the boot time generated symbols needed by the driver. Typically, arrays of structures are generated based on the number of device controllers equipped and the number of devices per controller as specified in the master file. Similar to the device address table, the *logical* controller number is used to index into this array in order to locate the structure associated with a particular hardware unit. Arrangements can also be made to initialize the boot time generated data structures. (Refer to Appendix E for details). It should be noted that there are two independent declarations of these data structures: one in the master file and one in the driver source or a system header file. It is important that consistency between these two declarations be maintained.

Driver Coding Restrictions Imposed by Self-Configuration

The self-configuring boot process imposes some restrictions on the coding techniques that are used in drivers. The device number passed to a driver is the external device number and must be converted to internal major and minor numbers via the *major/minor* macros.

Interdriver dependencies require special attention. If one driver depends on another (that is, requires the other driver to also be loaded), then that dependency must be specified in the master file. External symbols defined by a driver should begin with the driver unique prefix to avoid conflicts with other driver external symbols.

The need to produce an absolute boot file after a self-configuring boot operation requires that the use of initialized data be restricted. Initialized data should only be used for constants. The code should be included to explicitly initialize variables. (Uninitialized variables in the bss section are guaranteed to be initialized to zero.)

Driver Development Checklist

This section summarizes the various things that a developer must do to add a driver to a 3B2 or a 3B5 Computer system.

- Have a driver name and prefix assigned by the AT&T Technologies coordinator.

- Define the necessary driver entry points as per standards for the device switch tables and driver initialization tables.

- Use only the kernel services explicitly included in the kernel/driver interface specification.

- Use only the fields of system data structures explicitly included in the kernel/driver interface.

- External symbols defined by the driver should begin with the driver unique prefix.
- Use initialized data only for constants. Explicitly assign initial values to variables that are not initially zero.
- Define the driver data needed for the master file. (See Appendix E.)
- Check that data definitions in the master file entry are consistent with corresponding data declarations in the driver source or system header files. (See Appendix E.)
- Pay special attention to interrupt handling code - do not lower the execution level, do not try to sleep, do not access the user data structure, and do not assume that local variables are preserved for a subsequent entry (either another interrupt or timeout).
- Design an appropriate install script.
- Use *drvinstall* to install the master file information, update the *system* file, and create special device files.

The kernel/driver interface specifications of kernel services and system data structures are designed to insure object code compatibility with future releases of the system software. Failure to follow these guidelines may require driver recompilation and/or source code changes.

SUPPORT PROGRAMS

mkboot*(1M)*

The **mkboot**(1M) command prepares an object file for use by **lboot**. The object file is either a configurable module or an unresolved UNIX System kernel. Each module object file named must correspond to an entry in the **master** file. Correspondence is established by matching the object file name stripped of any optional path prefix or **.o** suffix. The resulting name is converted to lowercase before matching against the **master** file. A UNIX System kernel object file is identified with a command line option, and the **master** file entry is always *kernel*.

The **master** file is read and the configuration information associated with each object file is extracted. For each object file, a new file is created containing this configuration information. The new object files are written to the **/boot** directory (default) and are given the name (in capital letters) of the corresponding **master** file entry.

mkunix*(1M)*

The **mkunix** command creates a bootable kernel namelist file (also termed the *absolute* boot file) from the current contents of memory; this file will be named **a.out** and will be written to the current directory by default. This file contains the UNIX System kernel object file and all drivers and modules which were loaded by **lboot**. The **mkunix** program would be run following an auto-configuring boot with a new system configuration.

The resulting absolute boot file must be used as the namelist file for **ps, crash,** etc. In addition, this file may be booted directly, bypassing the self-configuration feature of **lboot**. This will probably save 30 to 90 seconds at boot time.

The unresolved kernel object file used by **lboot** must be available at the time **mkunix** is run. This is the path name specified as the **BOOT** program in the **/etc/system** file. This file is read to obtain the section names and the symbol table for the basic kernel.

sysdef*(1M)*

Historically, **sysdef** was the mirror image of **config**. You would prepare a configuration file to be used by the **config** program and build a new UNIX System kernel reflecting the contents of this configuration file. The **sysdef** program would recreate the configuration file given a kernel object file. The self-configuration feature breaks this circular chain.

The **sysdef** program will examine a kernel object file and extract configuration information. If the object file, is an unresolved kernel object file then there is no information available—it only makes sense to run **sysdef** on an absolute boot file. If the object file *is* an absolute boot file, then all hardware devices, their board slots and unit count as well as pseudo devices and system devices are listed. The values of all standard tunable parameters are also listed. It should be noted that the **sysdef** program requires the **master** file in order to determine the names of the devices that may be configured. **Sysdef** will indicate modules loaded in the UNIX System kernel to allow user programs to determine modules installed but not configured in the system.

newboot*(1M)*

The **newboot** program will write the *mboot* and *lboot* object files to the boot partition of a disk. The object files are written in the format expected by the firmware and the **mboot** program. This format is:

block 0:	The *mboot* object code
block 1:	A block containing the length and location at which to load the *lboot* object code
block 2-99:	The *lboot* object code.

On the 3B2 Computer system, block 1 will contain a volume table of contents which will contain the location of the **lboot** program.

EXTERNAL REQUIREMENTS

Unless otherwise indicated, all requirements apply equally to the 3B2 and 3B5 Computers.

Hardware

The minimum memory supported by **lboot** is 512K (one half megabyte). This requirement is based on:

- The C Language compiler not generating position-independent code; thus, **lboot** must be loaded at a fixed position
- There must be sufficient room remaining in memory to load the UNIX System kernel and support the temporary memory requirements necessary for symbol tables, relocation lists, etc.

3B2 Computer Requirements

The electrical design of the backplane of the 3B2 Computer requires that board slots be occupied sequentially; that is, there can be no unoccupied slots whose slot number is less than that of an occupied slot. The board slot is used as the major number for hardware devices. If a faulty board is removed, the remaining boards must be rearranged to satisfy the electrical design. However, the special device names in the **/dev** directory are now in error and must be changed.

Peripherals

Peripherals must interface with the hardware in a standard, predictable way. This insures that **lboot** can generate the proper linkages when the device driver is loaded. Each peripheral must interface with the hardware in the following ways:

- Each peripheral is allocated a group of 16 interrupt vectors due to the architecture of the 3B2 and 3B5 Computers. A peripheral designer may choose how many of the 16 interrupt vectors will actually be used. Regardless of the actual number of vectors used, they must be allocated sequentially beginning with the first vector.

- If the interrupt vectors assigned consist of paired receive-transmit vectors, then they must be allocated with all transmit vectors preceding all receive vectors. Furthermore, the pairing is be sequential; transmit vector one is paired with receive vector one, etc.

- The low-level interrupt routine is defined by the UNIX System kernel, and is the same for each peripheral.

The requirement that a driver be loaded is established by the presence of its name in the EDT. Therefore, each peripheral may be controlled by one and only one device driver. Another way to state this is that there can be one and only one device driver associated with a name in the EDT.

LBOOT-UNIX System Interface

To prevent the unnecessary creation of a **/unix** file on every boot, the **lboot** program passes the UNIX System an argument indicating a power-up boot or a manual boot. This allows the invocation of the **mkunix** delete - that is nothing program which will create **/unix** only on automatic or power-up boot if a configuration change occurred.

Drivers

The self-configuration feature imposes coding restrictions for the device drivers and configurable modules. These restrictions arise as a result of the dynamic linking of the kernel and configurable modules at boot time. These restrictions and requirements are:

- There may be no static variables whose initial contents are depended upon by code fragments. Such items as "first-time" switches, lock words, and initial pointers for linked lists are not allowed. The only initial value that may be assumed is zero for variables allocated in the bss section. Note that this restriction does not apply to statically allocated and initialized identifiers used as constants (that is, not a variable).

- There may be no references to routines or identifiers defined within other modules unless there is a strict dependency chain established by the dependency list in the **master** file. The single exception is a reference to a routine in another module which is defined in the routine definition lines of that module **master** file entry.

- Any necessary data areas must be definable using the capabilities of the variable definition lines in the **master** file. Furthermore, the sizes of all such data structures must be adjusted based upon the configuration that exists at configuration time using the capabilities allowed by the **master** file.

- Drivers must be written to expect the entire device number passed in their argument lists rather than just the minor number. This is a change to drivers written for non-self-configuration systems. A device number must, in general, be processed in three steps. First, the minor number must be inspected to determine that it refers only to devices on an individual controller. Second, the *minor* macro must be invoked to convert the device number into an internal minor number—a sequential number in the range zero to *n*, where *n* is total number of devices supported by the driver. Finally, this internal minor number must be verified to make sure that it only refers to an existing device.

- There may be one and only one driver controlling any peripheral device. Note that this applies only to drivers that control their hardware directly. Drivers that interface to hardware indirectly (such as those controlling devices through the *IOA* on the 3B5 Computer) do not violate this requirement. The justification for this requirement is due to the method used to associate a driver with a peripheral—the EDT entry for the peripheral is used to identify the single driver to be loaded. In addition, any interrupt routines required for a peripheral must interface to one and only one driver.

- Drivers for integral devices (such as the console or, on the 3B2 Computer, the integral hard/floppy disk) are treated as special cases of software drivers. The **master** file entry flags these drivers as *required* and *software*, but the number of interrupt vectors is not zero. The vector number is assigned by the **master** file rather than being computed by the peripherals hardware address. All other requirements for hardware drivers must be met.

- There are certain names which have special meaning to **lboot** when they are encountered within a module. Each such name is composed of the module prefix assigned by the **master** file and one of the following names.

init	An initialization routine called prior to kernel initialization
start	An initialization routine called immediately after kernel initialization
clr	A routine to be called at the time of a power fail interrupt

Hardware and software drivers have an additional list of special names. These are:

open	The device open routine
close	The device close routine
read	The UNIX System character device read routine
write	The UNIX System character device write routine
ioctl	The UNIX System character device ioctl routine
strategy	The UNIX System block device strategy routine
print	The UNIX System block device error message routine
[xr]int	The interrupt routines(s); the [xr] prefix used for paired transmit/receive interrupts

VTOC

The volume table of contents (VTOC) feature on the 3B2 Computer adds a data structure to each physical disk volume which identifies and describes the contents of the volume. The VTOC contains device information for firmware and device drivers. It also contains the partition mapping for the UNIX System kernel which previously was tabulated in the header file **sys/io.h**. The ability to describe the disk partitioning on the disk itself rather than compiled into the kernel or the boot program can be used to the advantage of **lboot**. Additional fields exist in the VTOC for the use of **lboot** and the self-configuration process. These fields are:

lboot address	The memory address at which to load the **lboot** program.
lboot length	The size of the **lboot** program in bytes.
Root file system flag	A pointer to the partition containing the root file system.
Swap partition flag	A pointer to the partition to be used for the swap area.

This additional information is used to allow **lboot** to make reasonable assumptions for the device information that is presently specified by the **/etc/system** file. In thoses cases where the assumptions would not be appropriate, the defaults can be overridden by the system file.

The following items are provided by VTOC.

- The boot file system is no longer hard-coded into the **lboot** program.
- There is no longer a size constraint on **lboot.** This is important since **lboot** is now very close to exceeding the 100 block boot partition.
- If the device information is omitted from the system file, the following assumptions are made.
  - *rootdev* is the boot device, partition is flagged
  - *pipedev* is rootdev
  - *dumpdev* is not supported since the crash dump code always prompts anyway
  - *swapdev* is *rootdev* and the VTOC identifies the swap partition from which *swplo* and *nswap* can be determined.

Relatively minor software changes are needed to accommodate the VTOC feature during the boot process. **Mboot** will assume that the VTOC occupies the second block on the disk device. **Mboot** presumes that **lboot** immediately follows the VTOC. The load address and the length of **lboot** are determined by reading the VTOC. The VTOC would again be read by **lboot** to determine the root file system.

TUNING SYSTEM PARAMETERS

One of the main impacts of this feature is the elimination of the **config** program. This program was used to change the values of system parameters. System parameters are now changed by modifying their values in the **master** file for the kernel. The **mkboot** program is then used to create a modified kernel. When the system is reconfigured with this kernel, the new parameter values will take effect.

Driver Development Checklist

This section summarizes the various things that a developer must do to add a driver to a 3B2 or a 3B5 Computer.

- Have a driver name and prefix assigned by the AT&T Technologies coordinator.
- Define the necessary driver entry points as per standards for the device switch tables and driver initialization tables.
- Use only the kernel services explicitly included in the kernel/driver interface specification.
- Use only the fields of system data structures explicitly included in the kernel/driver interface.
- External symbols defined by the driver should begin with the driver unique prefix.
- Use initialized data only for constants. Explicitly assign initial values to variables that are not initially zero.
- Define the driver data needed for the master file. (See Appendix E.)
- Check that data definitions in the master file entry are consistent with corresponding data declarations in the driver source or system header files. (See Appendix E.)
- Pay special attention to interrupt handling code — do not lower the execution level, do not try to sleep, do not access the user data structure, and do not assume that local variables are preserved for a subsequent entry (either another interrupt or time-out).
- Design an appropriate install script.
- Use *drvinstall* to install the master file information, update the *system* file, and create special device files.

The kernel/driver interface specifications of kernel services and system data structures are designed to insure object code compatibility with future releases of the system software. Failure to follow these guidelines may require driver recompilation and/or source code changes.

EXAMPLE DRIVERS

Appendix I contains the character driver for the ports card and a 3B2 Computer. Appendix J contains the block device disk driver for the 3B2 Computer.

Appendix K contains the character driver for the adli on a 3B5 Computer. Appendix L contains the block device disk driver for the 3B5 Computer.

Chapter 5

3B2 COMPUTER DEPENDENT INFORMATION

PAGE

Chapter 5

3B2 COMPUTER DEPENDENT INFORMATION

INTRODUCTION

This chapter contains 3B2 Computer specific information. The chapters indicated in parentheses are the chapters that contain more information on the subject mentioned in this chapter. The headings in Chapter 5 are the same as the heads in the referenced chapters. So, more detailed information should be easy to find.

3B2 COMPUTER FLOPPY RESTORE PROCEDURE

The 3B2 Computer floppy restore procedure changes slightly with self-configuration; however, this is transparent to the user. To boot the system from the floppy disk, an **lboot** not supporting self-configuration is maintained. This **lboot** is the same boot program existing in the 1.0 Release of the 3B2 Computer. Support of two lboots is needed because of the limited size of the floppy.

FIRMWARE (CHAPTER 2)

Firmware to device driver communication is different for 3B2 and 3B5 Computers. For this reason, the firmware chapter, Chapter 2, has been separated into 3B2 and 3B5 Computer information. See Chapter 2 for firmware information.

KERNEL/DRIVER INTERFACE (CHAPTER 3)

Driver Conventions

Major/Minor Device Numbers and Translations

For actual hardware devices, the major number is the board hardware address code or board slot. Software drivers are assigned (by the drvinstall command) major numbers that do not conflict with the major numbers assigned to hardware devices. Major numbers for software drivers range from 16 to 127 for the 3B2 Computers.

Kernel Interface to Driver Services

Device Access Other Than Read/Write

The *ioctl* function is traditionally and most commonly provided by drivers for terminal interface devices. It controls device hardware parameters and establishes the protocol used by the driver in the semantic processing of data. The ioctl function has become the catch-all function for facilitating all device access that is not normal read/write access.

The use of the ioctl function by nonterminal drivers is open ended. On the 3B2 Computer, the ioctl function is used to format diskettes and implement bad block handling.

Driver Interfaces to Kernel Services

Translating Virtual Addresses to Physical Addresses

The memory address a driver receives from the kernel is a virtual address. The use of the virtual address by the driver itself works correctly when memory management is done by the CPU. Some devices that access memory directly deal only with the physical memory addresses. In these cases, the driver must provide the device with physical memory addresses. To translate virtual addresses to physical addresses, the kernel provides the vtop function. The 3B2 and 3B5 Computers provide slightly different vtop functions.

For the 3B2 Computer:

```
paddr_t
  vtop(vaddr, p)
  char *vaddr;
  struct proc *p;
```

The vtop function accesses as arguments a virtual address and a pointer to a proc structure. (paddr_t is the type for a physical memory address. It is defined in the system header file /usr/include/sys/types.h.) The virtual address is the memory address being translated. The pointer to the proc structure is used by vtop to locate the information (tables) used for memory management. To indicate that the address is in kernel virtual space or in the driver itself, the second argument should be NULL. Block device drivers that can transfer data directly in and out of user memory must use the b_proc element of the buffer header data structure as the second argument. The vtop function returns the translated address. If, for some reason, the virtual address cannot be translated correctly, vtop returns zero.

System Error Messages and Halting the System: cmn_err/panic/printf

On the 3B2 Computer, drivers do not call panic and kernel printf directly. Instead, the UNIX System kernel provides the function cmn_err, which, in turn, calls printf or panic. The cmn_err function is passed two arguments. The first argument is a defined constant, indicating the severity level of the error condition. The second argument to cmn_err is basically the set of arguments that would be passed to printf (3B5 Computer). The cmn_err function with the CE_NOTE argument can be used by driver developers as a tool for debugging driver code, but this is likely to change system timing characteristics.

DRIVER CONFIGURATION (CHAPTER 4)

Performance and Resource Requirements

To reconfigure the UNIX System kernel, minimal memory must be used during normal system operation. Normal user level operations will not be disrupted because the reconfiguration is done during the boot sequence. The reconfiguration procedure requires that a system have a minimum of 512 kilobytes of memory.

3B2 COMPUTER DEPENDENT INFORMATION

The space required on disk for files needed for configuring a minimal UNIX System includes space needed for two copies of the UNIX System kernel (one configured and one nonconfigured), two system description files, and required drivers.

A minimal UNIX System is a UNIX System generated by the source in all directories except those in the io directory but includes the driver supporting the boot and root devices, console driver, and the system interface to the block and character I/O subsystems.

The maximum disk allocation for a minimal system will not exceed 700 kilobytes.

Interface Requirements

Installer's Perspective

Hardware and software drivers can be added to a system, removed from the system, or be replaced by a new version. Facilities that let users make changes to the number of drivers in a system are provided through a simple administration interface on the 3B2 Computer.

Reconfiguring a UNIX System Kernel

During the configuration of the UNIX System, the position of hardware peripherals is embedded within the UNIX System. Subsequent booting of the UNIX System assumes that these positions remain unchanged. Rearranging peripherals may cause undesirable results and may result in system panics or failures to boot. This is a problem on the 3B2 Computer because it requires that no slots be empty between boards. This implies that removing one driver may require a complete rearranging of other boards.

Manual Boot Procedure

To boot the system manually, enter **boot** from the firmware level. On the 3B2 Computer, the user is prompted for the name of the boot device, which is the integral disk or the integral floppy. The boot program prompts for the name of the system file.

When /KERNEL or /boot/KERNEL is used on the 3B2 Computer to boot the UNIX System, the user will be prompted for software drivers to include and exclude. Hardware drivers are automatically included when the boot program detects that the hardware exists. The boot program prints:

INCLUDE?

The response may be a carriage return (meaning no additional modules are to be loaded), or the response may be a comma or blank separated list of drivers to be included in the system. Next,

EXCLUDE?

is printed. Again, the installer may enter a carriage return (meaning no drivers are to be excluded), or the response may be a comma or blank separated list of drivers to exclude from the system.

Driver Development

Driver Addition

The drvinstall program handles many of the steps needed to add a driver to a system. It is particularly well suited for use by driver developers during the debugging and testing phases of development. The drvinstall program provides the interface for updating the /etc/master.d and /etc/system files. On the 3B2 Computer, drvinstall also supplies the needed Equipped Device Table (EDT) information. It also invokes the *mkboot* command and places the driver object in the boot directory.

For hardware drivers, the *getmajor* command is used to determine the assigned major numbers. The format is:

getmajor name

Name is the official device identification code in the 3B2 Computer. There is a one-to-one correspondence between 3B2 Computer device identification codes and office device driver names. Both are assigned by AT&T Technologies.

On the 3B2 Computer, the information needed for the EDT is supplied through the **edittble** command. Appendix G contains details on the use of this command.

Drivers are organized as add-on packages and are distributed on floppy diskettes for the 3B2 Computers.

3B2 COMPUTER DEPENDENT INFORMATION

Different interfaces are used to install drivers on the 3B2 and 3B5 Computers. The 3B2 Computer uses the menu-driven *sysadm* facility, which includes an *install* procedure. Although different interfaces are used, the functions and underlying structures are similar. For example, both copy files from the distribution medium, and both execute the INSTALL script.

Data Structures

3B2 Computer Conventions

For hardware devices, the major number is the board slot. So, only major numbers 1 through 15 can refer to hardware devices. Software drivers will be assigned (by the master file) major numbers 16 through 127. This number is called the external major number because it is visible in the special files for the device. An internal major number for the device is assigned by the boot program.

The external minor number only identifies the logical device on the individual board identified by the external major number. The major number is used to index into a table, built by the boot program, to obtain a base number that is added to the external minor number to create the actual logical device number now used by device drivers.

Equipped Device Table

The 3B2 Computer firmware uses the same basic technique as the 3B5 Computer firmware; however, there are significant differents in the details of the implementation. The location of the EDT is specified by an address word, which is at a fixed location in memory. The system header file defining the format of each element is also named sys/edt.h, but an additional header file (sys/firmware.h) is needed to access the EDT.

File and Directory Structure

For the 3B2 Computer, the location of the boot file system is obtained from the Volume Table Of Contents (VTOC). This file system contains everything needed to complete the boot process.

Driver Naming Convention

A driver name can be up to 10 characters long on the 3B2 Computer. All names must be in uppercase. Each driver also has an associated two- to four-character prefix that is used to identify driver functions such as open and close.

Driver Interfaces in a Self-Configuring System

Device Address Table

In the 3B2 Computer, device address tables contain actual addresses and may be used directly to access the hardware.

Support Programs

newboot

On the 3B2 Computer, block 1 of the object files written by the newboot program contains the VTOC, which contains the location of the lboot program.

External Requirements

3B2 Computer Requirements

The electrical design of the backplane of the 3B2 Computer requires that the board slots be occupied sequentially. That is, there can be no unoccupied slots whose slot number is less than that of an occupied slot. The board slot is used as the major number for hardware devices. If a faulty board is removed, the remaining boards must be rearranged to satisfy the electrical design. However, the special device names in the /dev directory are now in error and must be changed.

3B2 COMPUTER DEPENDENT INFORMATION

Peripherals must interface with the hardware in a standard, predictable way. This ensures that lboot can generate the proper linkages when the device driver is loaded. Each peripheral must interface with the hardware in the following ways.

- Each peripheral is allocated a group of 16 interrupt vectors due to the architecture of the computers. A peripheral designer may choose how many of the 16 interrupt vectors will actually be used. Regardless of the actual number of vectors used, they must be allocated sequentially, beginning with the first vector.

- If the interrupt vectors assigned consist of paired receive-transmit vectors, then they must be allocated with all transmit vectors preceding all receive vectors. Furthermore, the pairing is sequential: transmit vector one is paired with receive vector one, and etc.

- The low-level interrupt routine is defined by the UNIX System kernel, and is the same for each peripheral.

Sample Drivers

Appendix I contains the character driver for the ports card and a 3B2 Computer. Appendix J contains the block device disk driver for the 3B2 Computer.

Chapter 6

3B5 COMPUTER DEPENDENT INFORMATION

Chapter 6

3B5 COMPUTER DEPENDENT INFORMATION

INTRODUCTION

This chapter contains 3B5 Computer specific information. The chapters indicated in parentheses are the chapters that contain more information on the subject mentioned in this chapter. The heads in Chapter 6 are the same as the heads in the referenced chapters. So, more detailed information should be easy to find.

FIRMWARE (CHAPTER 2)

Firmware to device driver communication is different for the 3B2 and 3B5 Computers. For this reason, the firmware chapter, Chapter 2, has been separated into 3B2 and 3B5 Computer information. See Chapter 2 for firmware information.

KERNEL/DRIVER INTERFACE (CHAPTER 3)

Driver Conventions

Major/Minor Device Numbers and Translations

For actual hardware devices, the major number is the board hardware address code or board slot. Software drivers are assigned (by the drvinstall command) major numbers that do not confict with the major numbers assigned to hardware devices. Major numbers for software drivers range from 64 to 127 for the 3B5 Computer.

Kernel Interface to Driver Services

Device Access Other Than Read/Write

The ioctl function is traditionally and most commonly provided by drivers for terminal interface devices. It controls device hardware parameters and establishes the protocol used by the driver in the semantic processing of data. The ioctl function has become the catch-all function for facilitating all device access that is not normal read/write access.

The use of the ioctl function by nonterminal drivers is open ended. For example, on 3B5 Computers, the ioctl function is used to rewind tapes.

Power Failure Functions

The 3B5 Computer supports an optional power holdover feature that lets the system shutdown gracefully when the power fails. Each driver may provide a function, driverclr, to facilitate this.

The driverclr function sets a flag prohibiting the initiation of any further I/O activity and notifies user processes awaiting I/O that the I/O has failed. Another function of driveclr is to purge all outstanding I/O requests that may be pending. The driverclr function accepts no arguments and returns no values.

Driver Interfaces to Kernel Services

Translating Virtual Addresses to Physical Addresses

The memory address a driver receives from the kernel is a virtual address. The use of the virtual address by the driver itself works correctly when memory management is performed by the CPU. Some devices that access memory directly deal only with physical memory addresses. In such cases, the driver must provide the device with physical memory addresses. To translate virtual addresses to physical addresses, the kernel provides the vtop function.

For the 3B5 Computer:

```
paddr_t
vtop(vaddr)
char *vaddr;
```

The vtop function accepts the virtual address to be translated as its argument. This can be an address in the current user virtual address space, in the kernel, or in the driver itself. The vtop function returns the translated address or a -1 (if the virtual address cannot be translated correctly). Note that vtop can only be used to translate a user address in the currently active process. As a result, it cannot be used in an interrupt handler or a time-out entry.

Driver Execution Control

For regions of code that must be protected from all interrupts, the splhi function should be called. It should be noted, however, the splhi locks out everything including the close and should be used sparingly. The spl6 on the 3B5 Computer also inhibits clock interrupts and should be used very carefully.

System Error Messages and Halting the System

On the 3B5 Computer, the kernel function printf provides driver access to the system console. The kernel function panic is called to halt the computer. The kernel print function is a scaled down version of the C-library printf. Only the %s, %u, %d, %o, %x, and %D option arguments are recognized. Kernel printf is used to print diagnostic information directly on the system console.

DRIVER CONFIGURATION (CHAPTER 4)

Interface Requirements

Installer's Perspective

Hardware and software drivers can be added to a system, removed from the system, or be replaced by a new version. Facilities that let users make changes to the number of drivers in a system are provided through a driver install interface on the 3B5 Computer. The interface serves as a buffer between the user and the system to reduce the number of errors that may occur when making changes to system files.

Manual Boot Procedure

On the 3B5 Computer, you are prompted for the root, swap, pipe and dump devices. The response to these prompts must be that of the appropriate system file entry.

Driver Development

Driver Additions

The drvinstall program handles many of the steps needed to add a driver to a system. It is particularly well suited for use by driver developers during the debugging and testing phases of development. The drvinstall program provides the interface for updating the /etc/master.d and /etc/system files.

For hardware drivers, the *getmajor* command is used to determine the assigned major numbers:

```
getmajor name
```

Name is the office device driver on the 3B5 Computer.

Drivers are organized as either magnetic tape or removable disk cartridges for the 3B5 Computer.

Different interfaces are used for installing drivers on the 3B2 and 3B5 Computer systems. While the 3B2 Computer uses the menu-driven sysadm facility, the 3B5 Computer uses the periphconfig command. Although different interfaces are used, the functions and underlying structures are similar. For example, both copy files from the distribution medium and both execute the INSTALL scripts. So, from the viewpoint of the developer, the INSTALL script and its associated use of the drvinstall is the focal point of the installation activity.

Data Structures

3B5 Computer Conventions

For hardware devices, the external major number is the board code assigned by the physical setting of the DIP switch, on the backplane. This value is assigned when the peripheral is installed. A diagram in Chapter 4 illustrates the bit assignments for bus addresses.

Devices located on the Local Bus Extender (LBE) are identified by both the board code of the LBE and the address on the Extended Local Bus (ELB). The major number can be from 0 through 127.

Driver Naming Conventions

A device driver name can be up to eight characters long on the 3B5 Computer. All names must be in uppercase. Each driver also has an associated 2- to 4-character prefix that is used to identify driver functions such as open and close.

Device Address Table

In the 3B5 Computer, the device address tables contain address translation information that must be loaded into the memory management unit before the hardware may be accessed.

Sample Drivers

Appendix K contains the character driver for the ADLI on a 3B5 Computer. Appendix C contains the block device disk driver for the 3B5 Computer.

Chapter 7

DIAGNOSTICS

PAGE

Chapter 7

DIAGNOSTICS

This chapter describes how to structure diagnostics. There are two other documents that also discuss diagnostics which are:

- *3B2 Computer Model 300 Feature Card Interface Design Manual* - discusses the hardware and firmware required to design and setup a new feature card.
- *3B2 Computer Off-Line Diagnostics Manual* - discusses the diagnostics that are executable on the System Board and feature cards.

DEVICE NUMBERS

Major, minor, and subdevice device numbers are assigned by AT&T Technologies. However, these numbers will not be assigned until the feature card is almost ready for production. In the meantime, these numbers may be chosen by using the next highest number not in the EDT. For example, if the highest number in the EDT is seven, you can use eight. However, if the highest number is three you cannot use four since it is already reserved. To find out what is available in the EDT, use the command

```
edittbl -l -d  /edt_data
```

For initial setup and testing device numbers with FFXX and XXFF, hexadecimal (hex) should be avoided. This means any device number starting with FF or ending with FF hex.

DISK PARTITIONING

If you are working on block devices, the figure on the next page should help you restore the filesystem.

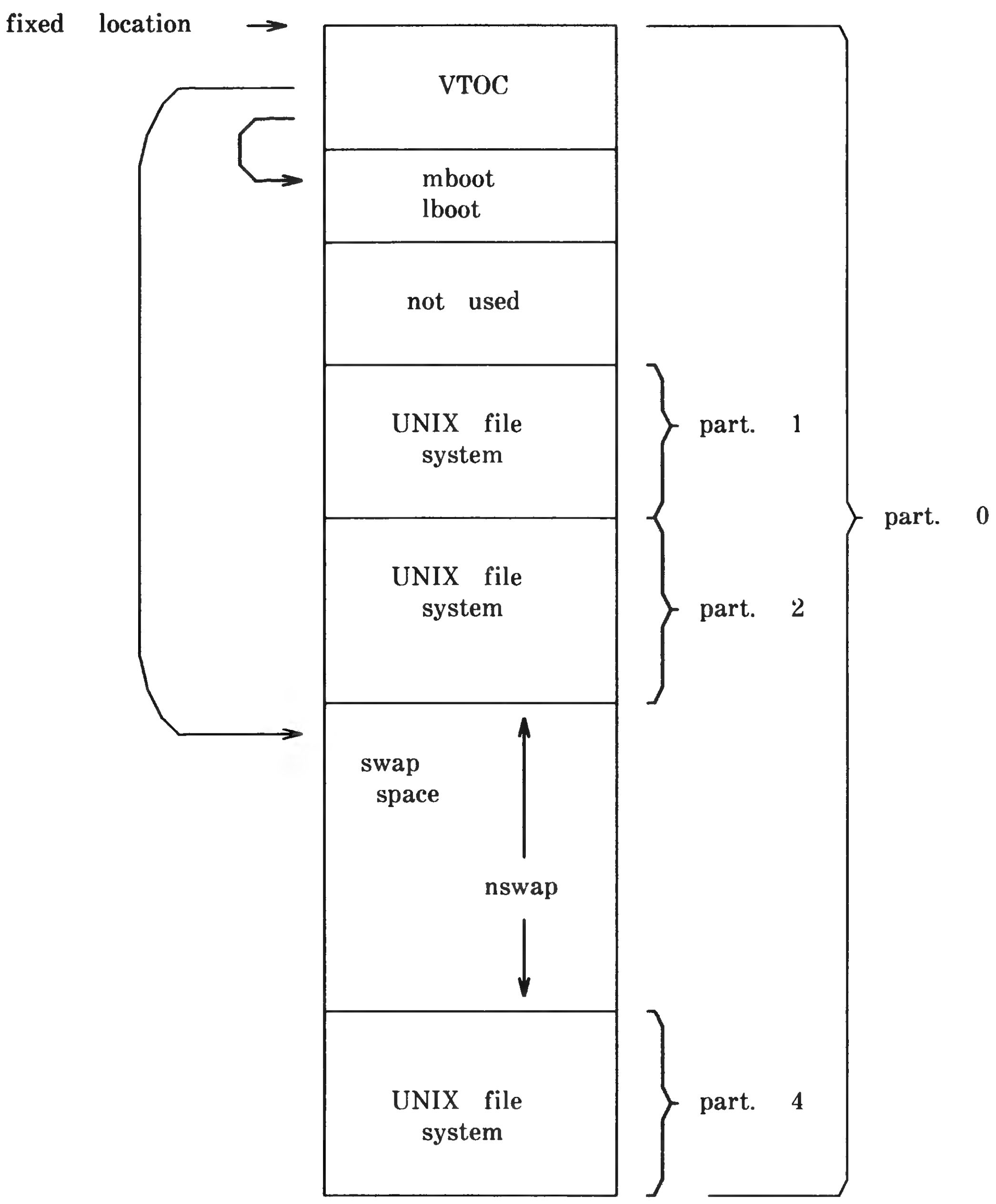

Figure 7-1. Disk Layout

APPENDIX A: TTY SUBSYSTEM

PAGE

APPENDIX A: TTY SUBSYSTEM

The UNIX System kernel provides common functions and data structures which can be used by drivers for low-speed character devices like printers and terminals: drivers which are required to perform semantic processing of data. The tty subsystem and the clist and tty data structures provide both buffering and semantic processing of data. Special interface requirements are placed on device drivers which use the tty subsystem. The services and data structures provided by the tty subsystem and requirements placed on drivers interfacing to the tty subsystem are defined in this appendix.

OVERVIEW OF THE TTY SUBSYSTEM

A tty structure exists for every possible terminal device in the system. This structure contains all the information needed to perform Input/Output (I/O) to a terminal. Each tty structure contains three clist structures. Therefore, there are three queues of cblocks associated with every terminal. A tty structure also contains a receive and a transmit control block; flags for input, output, and control modes. All storage for a tty structure must be declared in the device driver.

Clists and Cblocks

The tty subsystem processes characters by manipulating various queues and buffers. The data structures which are used to form these queues and buffers are found in the system header file *tty.h* and are described below.

The tty subsystem maintains character I/O queues with the clist data structure. The clist data structure is composed of a list head structure, clist, and member structures (cblocks). These structures are listed.

```
struct clist {
     int   c_cc;
     struct cblock *c_cf;
     struct cblock *c_cl;
};

struct cblock {
     struct cblock *c_next;
     char c_first;
     char c_last;
     char c_data[CLSIZE]; /* CLSIZE = 64 */
};
```

The diagram below illustrates how the clist queue is formed with these data structures.

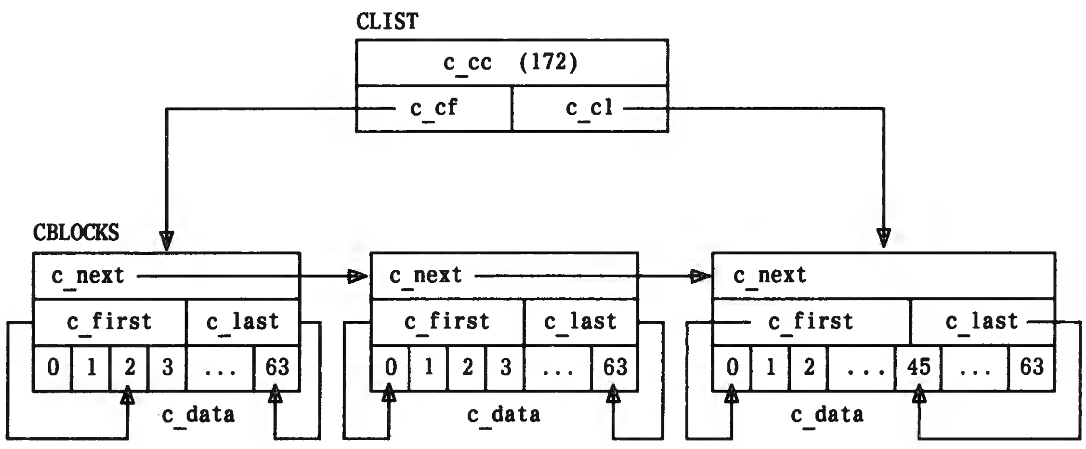

Figure A-1. Clist Queue

The clist head maintains a record of the number of characters in the clist (c_cc) with pointers to the first (c_cf) and last (c_cl) members of the clist. The cblocks form a singly linked list (c_next). Each cblock contains a buffer of up to 64 characters (c_data) and maintains indexes which point to the first (c_first) and last (c_last) character in the buffer.

The pool from which cblocks are drawn is the cfreelist. The cfreelist is headed by the chead data structure shown below.

```
struct chead {
     struct cblock *c_next;
     int   c_size;
     int   c_flag;
};
```

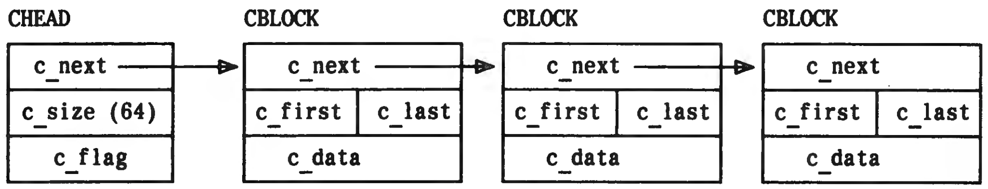

Figure A-2. Chead Data Structure

The cfreelist is a singly linked list (c_next). The *c_size* variable in the list head structure indicates the size of the cblock character buffer. Since the cfreelist is limited in size and shared by all tty devices, it is quite possible for the cfreelist to be empty when a cblock is needed by a tty device. When this occurs, the process that needs a cblock must sleep on the cfreelist. The *c_flag* variable is used to indicate that a process is waiting for a cblock.

Another data structure used by the character I/O subsystem is the character control block (ccblock). The character control block data structure is shown below.

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

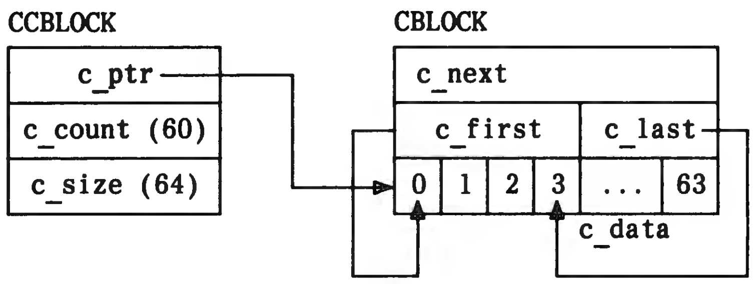

Figure A-3. Character Control Block

The ccblock structure is used as a temporary buffer for characters not in a queue. The ccblock *c_ptr* variable points to the character buffer (c_data) of a cblock. The *c_count* and *c_size* variables are initialized to the size of the cblock character array (64 characters). The *c_count* variable is then decreased by the number of characters in the cblock character buffer. The difference between the two variables is used to indicate the number of characters in the buffer.

Clist Routines

The tty subsystem provides several functions for manipulating clists and the cfreelist. These functions are described below.

```
getc(p)
struct clist *p;
```

The *getc* function receives as an argument a pointer to a clist. It retrieves the first character from the clist, decrements the clist character count, and returns the character to the calling function. If the character taken was the last in the cblock, the cblock is returned to the cfreelist. If processes were sleeping on the cfreelist (waiting for a free cblock), they are awakened after the cblock is returned.

```
putc(c, p)
struct clist *p;
```

The *putc* function is called to place the character pointed to by the first argument onto the clist pointed to by the second argument. If a new cblock is needed because there are none allocated for the clist or because the last one on the clist is full, putc retrieves a new cblock from the cfreelist. If the cfreelist is empty, putc returns a -1; this indicates to the calling process that it must sleep on the cfreelist. Otherwise, putc links the cblock to the clist, places the character in the cblock, and increments the clist character count.

```
struct cblock *
getcf()
```

The *getcf* function unlinks a cblock from the cfreelist and returns it to the calling function. *getcf* sets the cblock forward pointer to null and sets the c_first and c_last indexes to the front and back of the c_data array, respectively. If the cfreelist is empty, getcf returns null.

```
putcf(bp)
struct cblock *bp;
```

The *putcf* function is passed a pointer to a cblock. The *putc* function returns the cblock to the cfreelist and awakens any processes sleeping on the cfreelist.

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

The *getcb* function returns the first cblock on the clist specified by the argument *p*. It decrements the clist character count by the number of characters in the cblock and unlinks the cblock from the clist. If the specified clist is empty, a null is returned.

```
putcb(bp, p)
struct cblock *bp;
struct clist *p;
```

The *putcb* function is passed as arguments of a pointer to a cblock and a pointer to a clist. It links the cblock to the clist and increments the character count in the clist head.

The tty Data Structure

Character queues and buffers are associated with a given tty device through the tty data structure. The tty data structure follows.

```
#define   NCC  8
struct tty {
     struct    clist t_rawq;
     struct    clist t_canq;
     struct    clist t_outq;
     struct    ccblock   t_tbuf;
     struct    ccblock t_rbuf;
     int  (* t_proc)();
     ushort    t_iflag;
     ushort    t_oflag;
     ushort    t_cflag;
     ushort    t_lflag;
     shortt_state;
     shortt_pgrp;
     char t_line;
     char t_delct;
     char t_term;
     char t_tmflag;
     char t_col;
     char t_row;
     char t_vrow;
     char t_lrow;
     char t_hqcnt;
     char t_dstat;
     unsigned char   t_cc[NCC];
};
```

A character device driver using the tty subsystem must declare an instance of the tty data structure for each subdevice under its control. The tty data structure maintains all information relevant to the tty device. The elements of the tty data structure significant to driver developers are explained below.

- **t_rawq** — This field is the head for the devices raw input queue, a clist.

- **t_canq** — This field is the head for the devices canonical queue, a clist.

- **t_outq** — This field is the head for the devices output queue, a clist.

- **t_tbuf and t_rbuf** — These two fields are the devices transmit and receive buffers, respectively.

- **t_proc** — Each device driver for a tty device must provide a special hardware-specific access or *proc* function. This field holds the address of that driver function.

- **modes** — The next four fields of the tty structure specify modes defined in the *UNIX Operating System Administrator Manual* under *TERMIO(7)*. The *t_iflag* element holds the input modes specified in the *c_iflag* element of the *termio* structure. The *t_oflag*, *t_cflag*, and *c_lflag* elements hold output modes, control modes, and local modes as specified in the *c_oflag*, *c_cflag*, and *c_lflag* elements of the *termio* structure, respectively.

- **t_state** — This field maintains the internal state of the device and the driver. Each of the 16 bits of this field is assigned to one of the items in the following list. Thus, the state is a composite of one or more of the items below. Note that the *t_state* field is fully utilized and cannot be extended for additional state information that a particular driver may need.

TIMEOUT	Indicates that a delay timeout is in progress.
WOPEN	Indicates that the driver is waiting for an open to complete.
ISOPEN	Indicates that the device is open.
TBLOCK	Indicates that the driver has sent a control character to the terminal to block transmission from the terminal.
CARR_ON	This is a software image of the carrier-present signal.
BUSY	Indicates that output is in progress.
OASLP	Indicates that the processes associated with the device should be awakened when output completes.
IASLP	Indicates that the processes associated with the device should be awakened when input completes.
TTSTOP	Indicates that output has been stopped by a control-s character received from the terminal.
EXTPROC	Indicates that a peripheral device is performing semantic processing of data.
TACT	Indicates that a timeout is in progress for the device.

CLESC	Indicates that the last character processed was an escape character (\).
RTO	Indicates that a timeout is in progress for a device operating in raw mode; that is, no canonical processing is taking place.
TTIOW	Indicates that the process associated with the device is sleeping, awaiting the completion of output to the terminal.
TTXOFF	Indicates that transmission to the terminal is suspended; that is, a control-s character was received from the terminal.
TTXON	Indicates that transmission to the terminal is enabled; that is, a control-q character was received from the terminal.

- **t_pgrp** — This field identifies the process group associated with the device. It is needed to send signals to the process group.
- **t_line** — This field holds the line discipline type specified in the *c_line* element of the *termio* structure.
- **t_delct** — This field is used by the tty subsystem to keep track of the number of delimiters found while performing semantic processing of data.
- **t_col** — This field is used to record the current column position of the cursor on the terminal.
- **t_row** — This field is used to record the current row position of the cursor on the terminal.
- **t_vrow** — This field is reserved for system use.
- **t_lrow** — This field is reserved for system use.
- **t_dstat** — This field may be used by the driver to record driver defined states.
- **t_cc**[NCC] — This array holds the control characters specified in the *c_cc* member of the *termio* structure.

Opening a tty Device: ttinit and ttopen

The tty subsystem provides two functions for the driver *open* function: *ttinit* and *ttopen*. The *ttinit* function is called by the driver the first time a device is opened; that is, if the device was previously closed.

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

The tty subsystem accepts as an argument a pointer to the tty structure associated with the device being opened. It zeroes the *t_line*, *t_iflag*, *t_oflag*, and *t_lflag* elements of the tty data structure. It also sets default control modes (t_cflag) and control characters (t_cc).

The *ttopen* function is called each time the driver open function is called.

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

It accepts as an argument a pointer to a tty structure. It establishes the connection between the process opening the device and the device (t_pgrp). It also allocates and initializes a *cblock* for the receive buffer (t_rbuf) of tty structure. To take care of any initialization peculiar to the device hardware, *ttopen* calls the driver *proc* function specified in the *t_proc* element of the tty structure. This function is discussed later.

Reading a Character from a Terminal: ttread and ttin

The activity required to read a character from device hardware to user memory space through the tty subsystem is rather complex. It involves both operations initiated by the user at base level and operations initiated by the hardware at interrupt level. In the figure below, the large open arrows illustrate data flow and the small arrows illustrate control flow.

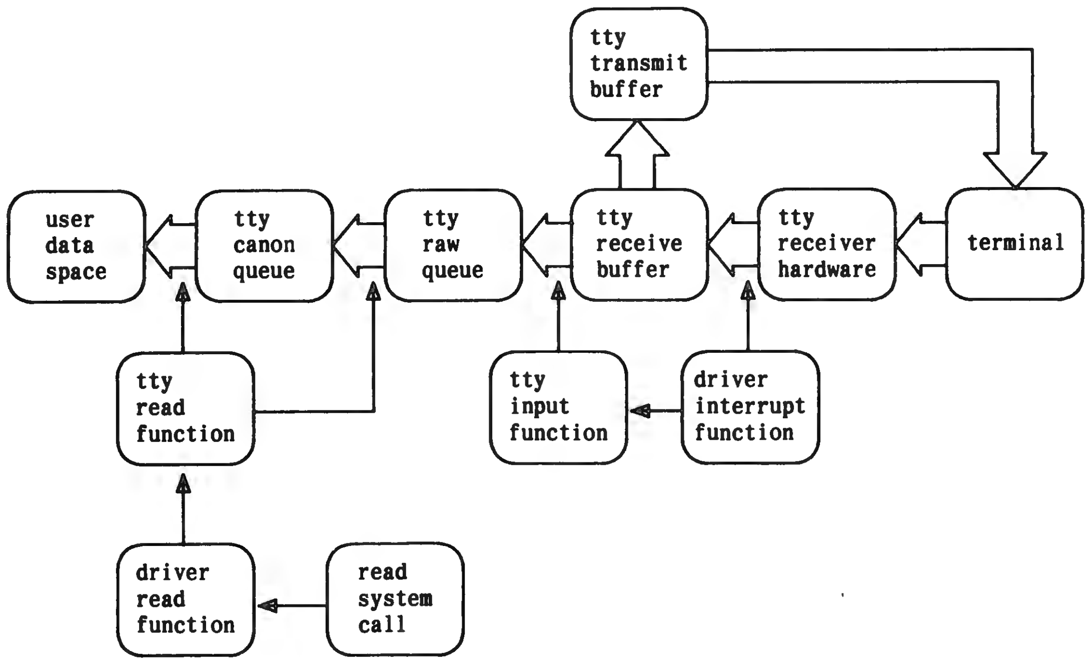

Figure A-4. Read Character

When the device hardware receives a character from a terminal, it interrupts the CPU, causing the device driver interrupt function to be entered. The interrupt function services the device hardware and transfers characters from the device to the receive buffer (t_rbuf) of the devices tty structure. It then calls the tty input function, *ttin,* which transfers characters from the receive buffer to the raw queue (t_rawq). *ttin* also copies characters from the receive buffer into the transmit buffer (t_tbuf) and echos them to the terminal.

A read of the device (initiated at base level by the user) causes the driver read function to be entered. The driver read function calls the tty read function, *ttread. ttread* transfers characters from the raw queue to the canon queue, and from the canon queue to user data space.

As seen in this very basic description, the tty subsystem functions needed by the driver to complete a read operation are *ttin* and *ttread.* A more detailed description of the operations performed by these two functions is provided below.

ttin(tp)
struct tty *tp;

ttin takes as an argument a pointer to the devices tty structure. It works through the tty receive buffer performing the conversion of newline, carriage return, and uppercase characters as specified in the mode fields of the tty structure, and places them in the raw queue.

If the number of characters in the raw queue exceeds a level called the *high water mark, ttin* calls the device driver *proc* function to send a stop character to the device. When the raw queue character count exceeds the *tty hog* level of 256 characters, *ttin* flushes the tty input queues. If the interrupt character (typically DEL) or the quit character (typically) is found, *ttin* sends the appropriate signal to the process group associated with the device. If processes associated with the device are sleeping (sleeping during a call to *ttread*) and *ttin* finds a delimiter character, *ttin* awakens the sleeping processes. The *ttin* function also takes care of echoing characters to the terminal.

When the terminal is operating in raw mode, the fifth and sixth elements of the tty structure control character array indicate the number of characters needed and the amount of time waited before processes associated with the device should be awakened. If the minimum character count has been met, *ttin* awakens processes associated with the terminal. If the character count has not been met *and* a time has been specified, *ttin* calls *timeout* to awaken the sleeping processes after the time period specified.

The device driver read function receives as an argument a device number. It uses this device number to determine the tty structure for the device being read. It then uses the address of the tty structure as an argument to *ttread.*

ttread(tp)
struct tty *tp;

ttread does all the work of read. It performs canonical (erase, kill, and escape) processing of data as it transfers characters from the raw queue to the canon queue. If no characters are available, it sleeps (on the address of the raw queue) until characters become available. After canonical processing has been performed, *ttread* transfers data from the canon queue to user data space. Finally, if transmission from the terminal had been blocked (t_state&TBLOCK) because the number of characters in the raw input queue was above the high water mark *and* if the read has caused that number to go below a safe level, *ttread* calls the device driver *proc* function to resume transmission from the terminal.

Writing a Character to a Terminal: ttwrite and ttout

The activity required to output, or write, a character to terminal parallels in many ways the character read functions. However, it is somewhat simpler than input since only one queue, the output queue (*t_outq*), is involved. Still, activities at both base and interrupt levels are involved. A transmit buffer provides for the buffering of characters between the base and interrupt portions.

A write to the device (initiated at base level by the user) causes the driver write function to be entered. This in turn calls the tty write function, *ttwrite*. *ttwrite* moves the characters to be output from the user data space to the output queue. It also calls the drivers access function (described later) to initiate actual output.

Once initiated, output is sustained by interrupts from the device. A transmit complete interrupt causes control to be passed to the driver transmit interrupt handler. The driver outputs the next character in the transmit buffer to the device. If the output buffer is empty, *ttout* is called to move characters from the output queue to the buffer. More detailed descriptions of *ttwrite* and *ttout* are provided in the following.

The device driver write function receives the device number as an argument. It uses this to determine the tty structure for the device being written. This is then passed to *ttwrite*.

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

The *ttwrite* routine transfers characters from user data space to the output queue as long as the output queue high water mark has not been exceeded. Processing is performed on the characters as they are put on the output queue to expand tabs and to add appropriate delays for newline, carriage return, and backspace characters. When the high water mark is reached, *ttwrite* sleeps (on the output queue). The *ttwrite* routine calls the driver *proc* function to initiate or resume output to the device.

The *ttout* routine is called by the driver transmit interrupt handler. It is passed the address of the tty structure associated with the device.

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

The *ttout* routine moves characters from the output queue to the transmit buffer in preparation for output by the device driver. The *ttout* routine implements the actual timing delays needed during output. When it detects a delay in the output queue, it uses the kernel *timeout* function to arrange for an entry after the appropriate amount of time has elapsed. This delayed entry invokes the driver *proc* function to resume output. The *ttout* routine is also responsible for awakening *ttwrite* when a sufficient number of characters have been transmitted; that is, when the number of characters in the output queue becomes less than the low water mark.

Changing Device Parameters: ttiocom

Changing the many parameters associated with terminal devices requires close cooperation between the driver and the tty subsystem. The *ttiocom* and *ttioctl* routines provide access to reading and changing the various tty parameters contained in the tty structure. Changing such parameters usually requires that device registers also be altered; the device driver is responsible for this.

A request to read or change terminal parameters is initiated by an *ioctl* system call from a user process. This causes the driver *driverioctl* function to be called. The driver locates the tty structure associated with the device and calls the common *ioctl* function *ttiocom.* It can be called in one of two ways:

```
ttiocom(tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;

ttiocom(tp, cmd, arg, mode)
struct tty *tp;
int *arg;
int cmd, mode;
```

cmd and *arg* are defined in the *UNIX System Administrator Manual* under *TERMIO(7). mode* contains the value of the *f_flag* field of the associated special device file (see **/usr/include/sys/file.h**).

Internally, *ttiocom* calls *ttioctl.* These two functions together affect the appropriate parameter settings and return to the driver. A nonzero value returned to the driver indicates that device registers must also be changed. Although *ttiocom* and *ttioctl* are together involved in parameter access, each has a different purpose. *ttiocom* is a general purpose routine that provides common parameter handling function. *ttioctl* is specialized in that it deal with parameters related to buffering and character processing. That is, it is associated with the terminal protocol or *line discipline.*

Closing a Device: ttclose

The line discipline close function, *ttclose,* is called by the device driver close function.

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

It accepts one argument, the address of the tty structure associated with the device being closed. The *ttclose* function dissociates the device from the process which opened it and resets the *ISOPEN* flag in the devices internal state register (t_state). It calls the driver *proc* function to transmit any characters in the devices transmit buffer (t_tbuf) out to the terminal, clears out all the tty buffers and queues, and returns to the *cfreelist* all *cblocks* allocated to the device.

After calling *ttclose,* the driver close function disconnects the link to the terminal and returns.

The Driver proc Function

The device driver must provide a function which can be called by the tty subsystem to process various device dependent operations, the *proc* function.

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

The *cmd* specifies the desired operation and can be one of the following:

T_OUTPUT	Initiates output to the device (unless the device is busy or output has been suspended).
T_TIME	Notifies the driver that delay timing for a break, carriage return, etc. has completed.
T_SUSPEND	Indicates that output to the terminal should be suspended; that is, a control-q character has been received. The **TTSTOP** bit in *t_state* should be set.

T_RESUME	Indicates that output to the terminal should be resumed; that is, a control-s character has been received. The **TTSTOP** bit in *t_state* should be cleared.
T_BLOCK	Blocks further input; that is, the input queue has reached the high water mark. Turn off **TTXON** and turn on **TTXOFF** and **TBLOCK** in *t_state.*
T_UNBLOCK	Allows further input; that is, the input queue has gone below the high water mark. Resets **TTXOFF** and **TBLOCK** in *t_state.*
T_RFLUSH	Same as **T_UNBLOCK** if **TBLOCK** is set; otherwise, does nothing.
T_WFLUSH	Clears all characters from the transmit buffer.
T_BREAK	Sends a break to the device. Sets **TIMEOUT** in *t_state* and initiates delay timing.
T_INPUT	Prepares device to receive input.

The commands above are defined in **/usr/include/sys/tty.h**

Accessing the tty Functions: The Line Discipline Switch Table

The protocols for processing and buffering characters are referred to as a line discipline. The tty functions described above comprise the default, system supplied line discipline, *line discipline 0.* In order to allow for other protocols, drivers must access the tty routines indirectly through the line discipline switch table. The *t_line* field of the tty structure contains the line discipline and is used to index into the line discipline switch table. The declaration of an entry in the line discipline switch table (from **/usr/include/sys/conf.h)** is shown below along with the *line discipline 0* values included as comments:

```
struct linesw {
     int  (*l_open)();    /* ttopen */
     int  (*l_close)();   /* ttclose */
     int  (*l_read)();    /* ttread */
     int  (*l_write)();   /* ttwrite */
     int  (*l_ioctl)();   /* ttioctl */
     int  (*l_input)();   /* ttin */
     int  (*l_output)();  /* ttout */
     int  (*l_mdmint)();  /* nulldev */
}
```

The *l_mdmint* field provides for a modem interrupt handler. It is presently not used and is " stubbed off" to the *nulldev* function.

APPENDIX B: BLOCK I/O SUBSYSTEM

APPENDIX B: BLOCK I/O SUBSYSTEM

The UNIX System kernel provides system buffers to be used as an intermediate holding area for data transfer between user data space and I/O devices. These buffers (1024 or 2048 bytes each on 3B2 and 3B5 Computers, respectively) are collectively referred to as the buffer cache. The buffer cache improves system performance and provides the user with single-byte access to block devices. System performance is improved because data which is in the cache can be reused; that is, the same block does not need to be read from a peripheral device multiple times when needed multiple times. Single-byte access is necessary because the user does not always need or want a full block of data from a block device; the buffer cache allows the system to read in a full block of data from a block device, and then, transfer only that data requested by the user to user data space. The use of the buffer cache and related data structures is elementary to the UNIX System block I/O and is described in this appendix.

BUFFER HEADERS: BUF.H

The basic data structure used in working with the buffer cache is the buffer header, *buf*. This structure is defined in the system header file **/usr/include/sys/buf.h**.

Each buffer in the buffer cache has an associated buffer header. The buffer header contains all the control/status information about the buffer. Most importantly to driver developers, the buffer header is the sole argument to a block device driver strategy function. It contains all the information needed to perform the data transfer. The buffer header data structure follows.

```
struct    buf
{
     int  b_flags;
     struct    buf *b_forw;
     struct    buf *b_back;
     struct    buf *av_forw;
     struct    buf *av_back;
     dev_tb_dev;
     unsigned b_bcount;
     union {
         caddr_t b_addr;
         int    *b_words;
         struct filsys *b_filsys;
         struct dinode *b_dino;
         daddr_t *b_daddr;
     } b_un;

     daddr_t    b_blkno;
     char b_error;
     unsigned int b_resid;
     time_t     b_start;
     struct  proc  *b_proc;
};
```

The fields of the buffer header which are available to the driver are:

- **b_flags** — This field maintains the status of the buffer and indicates to the driver whether the device is to be read or written. Valid flags are:

B_WRITE	Indicates that the data is to be transferred from main memory to the peripheral device.
B_READ	Indicates that data is to be read from the peripheral device into main memory.
B_DONE	Indicates that the transfer has completed.
B_ERROR	Indicates that an error occurred during the I/O transfer.
B_BUSY	Indicates that the buffer is in use.
B_WANTED	Indicates that the buffer is sought for allocation.
B_PHYS	Indicates that the buffer is being used for physical I/O.

B_STALE	Indicates that the buffer no longer contains valid information. This flag should be set when an error occurs during the I/O transfer.
B_AGE	Indicates that the buffer should be returned to the front of the buffer free list when released. This flag should be set when an error occurs during the I/O transfer.

- **av_forw and av_back** — These fields can be used by the driver to link the buffer into driver worklists.

- **b_forw and b_back** — These fields are also used to link the buffer header into lists. However, they should never be used by the device driver.

- **b_dev** — The major and minor device numbers of the device being accessed are contained in this field. The minor device number is contained in the low order eight bits and the major number in the high order eight bits.

- **b_bcount** — This field specifies the number of bytes to be transferred.

- **b_un.b_addr** — This field is the virtual address of the buffer controlled by the buffer header. Data is read (written) from (to) this address to (from) the device.

- **b_blkno** — This field identifies which block on the device (the device defined by the minor device number) is to be accessed.

- **b_error** — This field holds the error code which is eventually assigned by the kernel to the *u_error* field of the user data structure. It is set in conjunction with the B_ERROR flag.

- **b_resid** — This field indicates the number of bytes not transferred because of an error.

- **b_start** — This field holds the start time of the I/O; it is used to measure device response time.

- **b_proc** — This is the process table entry for the process requesting data transfer when the transfer is unbuffered (set to 0 when the transfer is buffered).

It is important to note that a buffer header may be linked in multiple lists simultaneously. Because of this, most of the fields in the buffer header cannot be changed by the driver, even when the buffer header is in one of the drivers worklists. The only fields that a driver can change are: *b_flags*, *av_forw*, *av_back*, *b_error*, *b_resid*, and *b_start*.

Buffer headers are also used by the system for unbuffered or physical I/O. In this case, the buffer describes a portion of user data space.

MANIPULATING BUFFERS WITH BUFFER ROUTINES

The UNIX System kernel provides the driver with several functions that can be used to manipulate the buffer cache. These functions are defined below.

Allocating, Clearing, and Releasing Buffers: geteblk, clrbuf, and brelse

Typically, block device drivers do not allocate buffers; the buffer is allocated by the kernel, and the associated buffer header is used as an argument to the driver strategy function. However, in order to implement some driver programs or ioctl functions, the driver may need its own buffer space. The driver developer has two choices:

1. Declare data space in the driver which can be used as a buffer

2. Borrow buffers from the buffer cache.

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

The function *geteblk* is used to allocate buffers.

```
struct buf*
geteblk()
```

The *geteblk* function accepts no arguments. It retrieves a buffer from the buffer cache and returns to the calling function the address of the buffer header. If no buffer headers are available, *geteblk* will sleep until one becomes available. Thus, *geteblk* should not be called at interrupt time.

When the device driver strategy function receives a buffer header from the kernel (that is, when the driver is entered through its strategy, read, or write functions), all the necessary fields are already initialized. However, when a device driver function allocates buffers for its own use, the function must set up some of the fields before calling the driver strategy function. The following list explains the state of these fields when the buffer header is received from *geteblk* and what must be done with them.

- **b_flags** — In this field B_BUSY flag is set to indicate that the buffer is in use. The driver must set the B_READ or B_WRITE flag, depending on the type of transfer.
- **b_dev** — This field is set to NULL and must be initialized by the driver.
- **b_bcount** — This field is set to the number of bytes in the buffer.
- **b_un.b_addr** — This field is set to the virtual address of the buffer.
- **b_blkno** — This field is not initialized by *geteblk* and, thus, must be initialized by the driver.
- **b_proc** — Since the buffers are in kernel data space, the driver should initialize this field to 0.

The remaining fields in the buffer header can be used as they are when the buffer header is received as an argument from the kernel.

The *clrbuf* function can be called to zero the buffer and set the *b_resid* field of the driver to 0.

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

The *clrbuf* function accepts as an argument the address of a buffer header. It returns no error/status values.

After the driver function is finished with the buffer, the *brelse* function is called to return the buffer to the kernel.

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

The *brelse* function accepts as an argument the address of the buffer header being returned to the kernel. It returns the buffer header to a list of free buffers and awakens any processes which might be sleeping on that list. The *brelse* function returns no error/status value.

Waiting on I/O: iowait and iodone

The kernel provides two functions used to suspend and continue execution during block I/O: *iowait* and *iodone*. The *iowait* function is called by driver functions which have allocated their own buffers and are awaiting data transfer completion.

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

iowait accepts a single argument: a pointer to a buffer header where the awaited data transfer is to take place. It sleeps on the address of the buffer header and is awakened by a corresponding call to *iodone* when the transfer completes.

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

iodone is called by the driver interrupt function, on the completion of any transfer. The only argument passed to *iodone* is the address of the buffer header associated with the buffer in which the I/O occurred. *iodone* awakens the process(es) sleeping on the buffer header.

Both *iowait* and *iodone* return no error/status values.

Unbuffered I/O: physck and physio

The block device driver read and write functions are called via the character device switch table to perform unbuffered I/O; that is, data is transferred directly to (from) the device from (to) user data space. The kernel provided functions *physck* and *physio* aid the driver in performing unbuffered I/O while maintaining the buffer header as the interface structure. These two functions are called by both the driver read function and the driver write function. Together they perform almost all the work to be done by a block device driver read and write functions.

```
physck(nblocks, rw)
daddr_t nblocks;
int rw;
```

The *physck* function accepts two arguments: the number of physical blocks on the device being accessed (the device determined by the minor device number) and a flag indicating whether the access is a read from (B_READ) or a write to (B_WRITE) the peripheral device. *physck* verifies that the user requested block exists on the requested device. If so, *physck* returns a 1. Otherwise, *physck* sets an error flag in the *u_error* field of the user structure and returns a 0.

The *physio* function is called by the driver if the *physck* function passes. It accepts four arguments.

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

The first argument is the address of the driver strategy function. The second is the address of a buffer header. When called from a driver read or write function, this argument is always 0. The third argument is a device number. The external device number received as an argument to the driver read or write function should be used here. The translation to an internal device number via the *minor* macro should be taken care of by the *strategy* routine when it is called later. The fourth argument to *physio* is a flag indicating whether the access is a read from (B_READ) or a write to (B_WRITE) the peripheral device.

The *physio* function sets up a buffer header describing the user data space. It then locks the user process in memory, calls the driver strategy function, and sleeps on the address of the buffer header. When the transfer completes, *physio* is awakened by the driver interrupt function via *iodone*. It then updates information on the user data structure and returns to the driver read or write function. *physio* returns no error/status value.

APPENDIX C: SLEEP AND WAKEUP EXAMPLE

The example sketches the use of sleep and wakeup in a driver that employs a limited pool of buffers for data transfer.

```
struct driverbuffer {
        int buf[256];
        unsigned int flag;
}

struct driverbufpool {
        struct driverbuffer pool[40];
        unsigned char empty;
        unsigned int flag;
} driverbufpool;

driverread(dev)
int dev;
{
        .
        .
        .
        while (spln(), driverbufpool.empty == TRUE) {
                driverbufpool.flag |= SLEEPING;
                sleep(&driverbufpool, PRIORITY);
        }
        spl0();
        .
        .
        .
}

driverint(dev)
int dev;
{
        .
        .
        .
        driverbufpool.empty = FALSE;
        if ((driverbufpool.flag & SLEEPING) == SLEEPING) {
                driverbufpool.flag &= ~SLEEPING;
                wakeup(&driverbufpool)
        }
        .
        .
        .
}
```

If the driver read function needed a buffer from the pool but the pool was temporarily empty, the read function would call sleep with the address of the pool as its first argument. The driver might also set a flag in one of its data structures to indicate to other driver functions that it is sleeping, waiting for a buffer. While the process is sleeping, other data transfer for other processes would take place. When the data transfer for one of the other processes completed, a buffer would become available and be returned to the pool: perhaps by the driver interrupt function. The interrupt function, after returning the buffer to the pool, would test the flag to see if processes were sleeping on the buffer pool. If so, the interrupt function would call wakeup with the address of the buffer pool as an argument and the process waiting for the buffer would be able to continue.

APPENDIX D: SELF-CONFIGURATION COMMANDS

APPENDIX D: SELF-CONFIGURATION COMMANDS

This appendix provides additional information on some of the self-configuration administrative commands useful during driver development. These commands should not be called directly as part of a driver installation procedure. Nevertheless, they are useful during development and the driver developer should understand their functions.

MKBOOT

The *mkboot* command prepares an object file for use by the boot program. The object file is either a configurable module or an unresolved UNIX System kernel. No more than one a.out file should be passed to *mkboot* at a time. Each module object file named must correspond to an entry in the master file. Correspondence is established by matching the object file name stripped of any optional path prefix or " .o" suffix. The alphabetic case of the resulting name is immaterial. A UNIX System kernel object file is identified with a command line option, and the master file entry is always *kernel*.

The master file is read and the configuration information associated with each object file is extracted. For each object file, a new file is created containing this configuration information. The new object files are written to the **/boot** directory and are given the name (in capital letters) of the corresponding master file entry.

MKUNIX

The *mkunix* command will create a bootable kernel namelist file (also termed the *absolute* boot file) from the current contents of memory; this file will be named **a.out** and will be written to the current directory by default. This file contains the UNIX System kernel object file and all drivers and modules which were loaded. Typically, *mkunix* would be run following an auto-configuring boot with a new system configuration.

The resulting absolute boot file must be used as the namelist file for *ps*, *crash*, etc. In addition, this file may be booted directly, bypassing the self-configuration feature process.

SELF-CONFIGURATION COMMANDS

The unresolved kernel object file used in the boot operation must be available at the time *mkunix* is run. This is the path name specified as the *BOOT* program in the **/etc/system** file. This file is read to obtain the section names and the symbol table for the basic kernel.

APPENDIX E: MASTER FILE

The *master* configuration database is a collection of files. Each file contains configuration information for a device or module that may be included in the system. A file is named with the module name to which it applies. This collection of files is maintained in a directory called **/etc/master.d**. Each individual file has an identical format. For convenience, this collection of files will be referred to as the *master* file, as though it were a single file. This will allow a reference to the *master* file to be understood to mean the *individual file in the master.d directory that corresponds to the name of a device or module.* This file is used by the *mkboot*(1M) program to obtain device information to generate the device driver and configurable module files. It is also used by the *sysdef*(1M) program to obtain the names of supported devices. *Master* consists of two parts; they are separated by a line with a dollar sign ($) in column 1. Part 1 contains device information for both hardware and software devices and loadable modules. Part 2 contains parameter declarations used in Part 1. Any line with an asterisk (*) in column 1 is treated as a comment.

Hardware devices, software drivers, and loadable modules are defined with a line containing the following information. Field 1 must begin in the left most position on the line. Fields are separated by white space (tab or blank).

Field 1: Element characteristics:

o	Specify only once
r	Required device
b	Block device
c	Character device
a	Generate segment descriptor array
t	Initialize cdevsw[].d_ttys

	s	Software driver
	x	Not a driver; a loadable module
	number	The first interrupt vector for an integral device
Field 2:	Number of interrupt vectors required by a hardware device; if none, " -"	
Field 3:	Handler prefix (4 characters maximum)	
Field 4:	Software driver external major number; " -" if not a software driver	
Field 5:	Number of subdevices per device; " -" if none	
Field 6:	Interrupt priority level of the device; " -" if none	
Field 7:	Dependency list (optional); this is a comma separated list of other drivers or modules that must be present in the configuration if this module is to be included.	

For each module, two classes of information are required by *mkboot (1M)*: external routine references and variable definitions. Routine and variable definition lines begin with white space and immediately follow the initial module specification line. These lines are free form; thus, they may be continued arbitrarily between nonblank tokens as long as the first character of a line is white space. No more than one a.out file should be given to mkboot at a time.

If the UNIX System kernel or other dependent module contains external references to a module, but the module is not configured, then these external references would be undefined. Therefore, the *routine reference* lines are used to provide the information necessary to generate appropriate dummy functions at boot time when the driver is not loaded.

Routine references are defined as follows:

Field 1:	Routine_name()
Field 2:	The routine type; one of

{ }	routine_name(){ }
{**nosys**}	routine_name(){return nosys();}
{**nodev**}	routine_name(){return nodev();}
{**false**}	routine_name(){return 0;}
{**true**}	routine_name(){return 1;}

Variable definition lines are used to generate all variables required by the module. The variable generated may be of arbitrary size, be initialized or not, or be arrays containing an arbitrary number of elements.

Variable references are defined as follows:

Field 1:	Variable_name
Field 2:	[expr] - Optional field used to indicate array size
Field 3:	(length) - Required field indicating the size of the variable
Field 4:	={ expr,... } - Optional field used to initialize individual elements of a variable

The *length* field is mandatory. It is an arbitrary sequence of length specifiers, each of which may be one of the following:

%i	An integer
%l	A long integer
%s	A short integer

%c	A single character
%number	A field which is *number* bytes long
%number c	A character string which is *number* bytes long

For example, the length field

```
( %8c %l %0x58 %l %c %c )
```

could be used to identify a variable consisting of a character string 8 bytes long, a long integer, a 0x58 byte structure of any type, another long integer, and two characters. Appropriate alignment of each % specification is performed (%number is word aligned) and the variable length is rounded up to the next word boundary during processing.

The expressions for the optional array size and initialization are infix expressions consisting of the usual operators for addition, subtraction, multiplication and division: +, -, *, and /. Multiplication and division have the higher precedence, but parentheses may be used to override the default order. The built-in functions *min* and *max* accept a pair of expressions and return the appropriate value. The operands of the expression may be any mixture of the following:

&name	Address of name where *name* is any symbol defined by the kernel, any module loaded, or any variable definition line of any module loaded
#name	Size of name where *name* is any variable name defined by a variable definition for any module loaded; the size is that of the individual variable—not the size of an entire array
#C	Number of controllers present; this number is determined by the EDT for hardware devices or by the number provided in the *system* file for non-hardware drivers or modules
#C(name)	Number of controllers present for the module *name*; this number is determined by the EDT for hardware devices, or by the number provided in the *system* file for nonhardware drivers or modules

#D	Number of devices per controller taken directly from the current *master* file entry
#D(name)	Number of devices per controller taken directly from the *master* file entry for the module *name*
#M	The internal major number assigned to the current module if it is a device driver; zero, if this module is not a device driver
#M(name)	The internal major number assigned to the module *name* if it is a device driver; zero, if that module is not a device driver
name	Value of a parameter as defined in the second part of *master*
number	Arbitrary number (octal, decimal, or hex allowed)
string	A character string enclosed within double quotes (all of the character string conventions supported by the C Language are allowed); this operand has a value which is the address of a character array containing the specified string.

When initializing a variable, one initialization expression should be provided for each %i, %l, %s, or %c of the length field. The only initializers allowed for a '%number c' are either a character string (the string may not be longer than *number*) or an explicit zero. Initialization expressions must be separated by commas, and variable initialization will proceed element by element. Note that %number specifications cannot be initialized—they are set to zero. Only the first element of an array can be initialized, the other elements are set to zero. If there are more initializers than size specifications, it is an error and execution of the *mkboot*(1M) program will be aborted. If there are fewer initializations than size specifications, zeros will be used to pad the variable. For example,

```
={ "V2.L1", #C*#D, max(10,#D), #C(OTHER), #M(OTHER) }
```

would be a possible initialization of the variable whose length field was given in the preceding example.

Parameter declarations may be used to refer to a value symbolically. Values can be associated with identifiers and these identifiers may be used in the *variable definition* lines.

Parameters are defined as follows:

Field 1:	Identifier (8 characters maximum)
Field 2:	=
Field 3:	The value may be a number (decimal, octal or hex allowed) or a string.

APPENDIX F: SYSTEM FILE

This Appendix includes a detailed description of the entries that make up the **/etc/system** file.

Lines may appear in any order. Comment lines must begin with an asterisk. Blank lines or comment lines may be inserted at any point. Entries for EXCLUDE and INCLUDE are cumulative. For all other entries, the last line to appear in the file is used—any earlier entries are ignored. Since the parser is case sensitive, all uppercase strings must be entered exactly as shown.

BOOT: path name

The path name specifies the object file to be booted; if the object file is fully resolved (such as that produced by the *mkunix (1M)* program), then no other lines in the *system* file have any effect.

EXCLUDE: name ...

This identifies names in the EDT that are to be ignored—this may either be because no driver exists for the specific EDT entry, or because the driver is not to be loaded for some other reason.

INCLUDE: name [(number)] ...

This line is necessary to identify software drivers or loadable modules from the **/boot** directory which are to be included in the load. It has no effect for hardware drivers. The optional " (number)" specifies the number (default of 1) of " devices" to be controlled by the driver. This number corresponds to the built-in variable *#C* which may be referred to by expressions in part one of the *master* file.

```
DUMPDEV:  {spec-dev-path name ¦ DEV(major, minor)}
ROOTDEV:  {spec-dev-path name ¦ DEV(major, minor)}
PIPEDEV:  {spec-dev-path name ¦ DEV(major, minor)}
SWAPDEV:  {spec-dev-path name ¦ DEV(major, minor)}  swplo  nswap
```

On the 3B2 Computer these items are taken care of by VTOC unless the system is booted manually and you are using system entries. However, these can be specified and put into the system file anytime.

These lines identify the system device to be used for writing a crash dump, the device containing the root file system, the device to be used for pipe space, and the device to be used for swap space (with the beginning block number for swap space *swplo* and the number of swap blocks available *nswap*). These are normally used only in the 3B5 Computer; on the 3B2 Computer, this information is derived from the disk volume table of contents (VTOC). The device may be specified in either of two ways. A path name of a special device file may be provided—the major and minor numbers are obtained from the *inode*. An alternative form is allowed in which the major and minor numbers are specified explicitly.

APPENDIX G: 3B2 COMPUTER EDITTBL COMMAND

APPENDIX G: 3B2 COMPUTER EDITTBL COMMAND

NAME

edittbl - edit edt_data file

SYNOPSIS

edittbl [-d] [-s] [-g] [-i] [-l] [-r] [-t] [file]

DESCRIPTION

Edittbl is a user-level command that permits changes to *edt_data,* the file in the root file system that the diagnostic monitor DGMON reads during self-configuration to get the device and subdevice look-up tables. This command permits independent selection of device or subdevice tables, generation of either base table, new entry installation for either table, entry removal for the device table, and entry listings for either or both tables.

Edittbl prints the option list if the command has no arguments. The arguments are:

-d This option selects the device look-up table for the utility operation(s).

−s This option selects the subdevice look-up table for the utility operation(s).

-g This option will generate the base look-up table entries for the selected look-up table(s). For the device table, these base entries are NULL, SBD, NI, and PORTS. For the subdevice table, they are NULL, IF, HD10, and HD30.

-i This option specifies that new entries are to be added to the selected table. The ID codes for table entries and the input are compared; only new codes are installed. The formats for entries are described below. An EOF or " ." end the data input.

-l	This option specifies that the selected table(s) are listed.
-r	This option specifies that entries are to be removed from the device look-up table. When removing subdevice look-up table entries from *init/dgn/edt_data* conjunction with removing a device entry, this command will check the Equipped Device Table (EDT) to verify that no subdevices specified for removal are present. The ID codes of the table are compared to the input and entries are removed for matches. The format is identical to that for the *-i* option and is listed below. An EOF or " ." end the data input.
-t	This option suppresses the program headings and user prompts; warnings and errors are not affected. This option is primarily useful in installation and removal scripts.
file	The user may specify a target path name for the utilities. If none is specified, *./edt_data* is the default.

INPUT FORMAT

Data for installation/removal are entered as hex format numbers or character strings, one line for each table entry. The data fields must be supplied in the sequence described.

Devices

ID code	This field is a number between 0x0 and 0xffff that a device uses to identify itself. ID codes are administered by AT&T Technologies.
name	This field is a character string (maximum of 9 characters) that holds the user-recognizable name for a device. Device names are administered by AT&T Technologies. This string is also the file name that DGMON loads to diagnose a device.
rq_size	This is a number between 0x0 and 0xff for the count of entries in a device job request queue.
cq_size	This is a number between 0x0 and 0xff for the count of entries in a device job completion queue.
boot device	This field determines whether a device may be used to boot programs. A " 1" means that it is bootable; a " 0" means that it is not.

word size	This field shows the word size of a device I/O bus. A " 1" is used for devices with a 16 bit bus word; a " 0" is used for devices with an 8 bit bus word.
brd size	This field specifies the I/O connector slots that a device requires. A " 1" indicates that two slots are needed; while a " 0" means that one is required.
smart board	This field determines whether a device is intelligent, that is, requires downloaded code for normal operation or supports subdevices. A " 1" indicates an intelligent device, while a " 0" specifies a " dumb" device.
cons_cap	This field shows whether a device can support the system console terminal. A " 1" is used for devices that can; a " 0" for those that cannot.
cons_file	This field shows whether a device requires pump code to provide a system console interface. A " 1" in this field means that the board cannot support the console interface without extra code. This field may have the " 1" value only when the *cons_cap* field does also. A " 0" in this field means that the device can support a system console terminal with PROM-based code when *cons_cap* has the value " 1" . This field must have a " 0" value when *cons_cap* is " 0" .

Subdevices

ID code	This is a number between 0x0 and 0xffff for the code that identifies a subdevice. Subdevice ID codes are administered by AT&T Technologies.
subdev name	This field is a string (maximum of 9 characters) for a subdevice name. Subdevice names are uppercase and are administered by AT&T Technologies.
dev name	This field is a string (maximum of 9 characters) for the device name to which a subdevice is associated. If a device table entry is to be removed, associated subdevice table entries may also be removed in a separate program call. The device name is necessary for an Equipped Device Table (EDT) check that will verify that a subdevice table entry is needed only for a device entry that is to be removed.

EXAMPLES

Generate and list the base entries for both the device and subdevice tables, saving the results in *./edt_data.*

```
edittbl -g -l -s -d
```

Install subdevice entries with new ID codes from the file *subdev.in* into the existing file *./edt_data.*

```
edittbl -i -s < subdev.in
```

List the device table entries found in an existing copy of the file that DGMON loads, the ROOT file system *edt_data* file.

```
edittbl -l -d /dgn/edt_data
```

APPENDIX H: REGISTERS

The WE 32000 Series has sixteen accessible 32-bit registers. Two additional registers, tempa and tempb, are reserved for the operating system and high-level support instructions. The accessible registers may be accessed in any addressing mode.

Registers r0 through r8 are general-purpose registers which may be used as accumulators or for addressing or temporary data storage. In assembly language, they are referenced as %rn, where n is the register number.

The first three of these registers, r0 through r2, are sometimes called scratch registers. In C Language, these three registers are used to return specific values during function calls. Registers r0 through r2 are also used by the data transfer instructions MOVBLW, STRCPY, and STREND.

Registers r3 through r8 may be used at any time by any program. These registers are saved whenever a new function or process is installed. They are most commonly used as register variables.

The remaining processor registers are special-purpose registers and are referenced by name. Three of these registers are pointers to data stored on an execution stack. The Frame Pointer (FP) register, register 9, is referenced as %fp; the Argument Pointer (AP), register 10, is called %ap; and the Stack Pointer (SP), register 12, is %sp. Function calls and returns affect the AP, FP, and SP implicitly. The FP identifies the starting location of local variables for the function. The AP identifies the beginning of the set of arguments passed to the function, while the SP always accesses the top of the execution stack.

The last set of registers have restrictions on how and when they may be used by instructions. Registers 11, 13, and 14 are privileged, which means that they may be read at any time but may be written only when the processor is in system mode; that is, when the operating system is in control. The other registers may be read or written in any execution level.

REGISTERS

The Process Status Word (PSW) register, register 11, contains status information about the current instruction and process. Register 13, the Process Control Block Pointer (PCBP), identifies a block of status information and pointers for a process. The Interrupt Stack Pointer (ISP) is held in register 14.

Register 15 contains the Program Counter. It is referenced as %pc. It may not be referenced in some address modes. In most cases, it is implicitly referenced by all program control instructions and is used by function calls and returns.

APPENDIX I: 3B2 COMPUTER BLOCK DEVICE DRIVER

The disk driver used on the 3B2 Computer consists of three parts:

- Internal hard disk driver
- Internal floppy disk driver
- Common file to check system board registers and contains the strategy routine.

```
/*
 *      Copyright 1984 AT&T
 *
 * Bell Laboratories
 * 3B2 UNIX Integral Winchester Disk Driver
 */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/sbd.h"
#include "sys/id.h"
#include "sys/if.h"
#include "sys/dma.h"
#include "sys/immu.h"
#include "sys/dir.h"
#include "sys/sysmacros.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/iobuf.h"
#include "sys/systm.h"
#include "sys/firmware.h"
#include "sys/cmn_err.h"
#include "sys/vtoc.h"
#include "sys/hdelog.h"
#include "sys/open.h"

/* pointer to disk controller */
extern int idisk;
#define ID ((struct iddev *) &idisk)
```

```
extern int dmac;
#define DMAC ((struct dma *) &dmac)

struct vtoc idvtoc[IDNDRV];

struct idsave idsvaddr[IDNDRV];

/* defect maps */
extern struct defstruct iddefect[];

/* bad block stuff */
struct hdedata idelog[IDNDRV];
extern hdelog ();

extern int idiskmaj;/* defined in master.d file for idisk */
/* physical description table */
struct pdsector idsect0[IDNDRV];

/* seek parameter structure */
struct idseekstruct idseekparam[IDNDRV];

/* controller initialization */
struct idspecparam idspec_s;
struct idspecparam idspec_f;

/* transfer parameter sturcture */
struct idxferstruct idxferparam[IDNDRV];


struct {
    unsigned char noparam;
} idnoparam;

/* drive status byte */
struct idstatstruct idstatus[IDNDRV];

/* temporary buffer for jobs which cross 64-Kbyte boundaries */
struct {
    unsigned int buf[128];
} idcache[IDNDRV+1];
unsigned int idpcacheaddr[IDNDRV];

struct  iobuf   idtab[IDNDRV];      /* drive information */
struct  iotime  idtime[IDNDRV];     /* drive status information */

int idspurintr;     /* spurious interrupt counter */
extern  paddr_t id_addr[];  /* local bus base address of disk controller */
extern unsigned ifstate;/* floppy driver state register */
```

```
/* rename buf structure variables */
#define actsio_s1
#define cylin   b_resid
#define ccyljrqsleep

idcopy (faddr, taddr, count)
unsigned int *faddr;
unsigned int *taddr;
unsigned int count;
{
    unsigned int *fptr;
    unsigned int *tptr;
    int i;

    tptr = taddr;
    fptr = faddr;
    for (i=0; i<(count/4); i++)
        *tptr++ = *fptr++;
}

unsigned char idscanflag;
unsigned char idnoscan;
idscan ()
{
    int s;
    s = spl6();
    if ((idtab[0].b_actf != IDNULL) || (idtab[1].b_actf != IDNULL)) {
        if (idscanflag == IDNULL)
            idrecal (IDNOUNIT);
        else
            idscanflag = IDNULL;
    }
    splx (s);
    timeout (idscan, 0, (10*HZ));
}

idsetblk (bufhead, cmd, blkno, dev)
struct buf *bufhead;
unsigned char cmd;
daddr_t blkno;
dev_t dev;
{
    clrbuf (bufhead);
    bufhead->b_flags |= cmd;
    bufhead->b_blkno = blkno;
    bufhead->b_dev = (dev|IDNODEV);
    bufhead->b_proc = 0x00;
    bufhead->b_flags &= ~B_DONE;
    if (cmd == B_WRITE)
```

```
        bufhead->b_bcount = idsect0[iddn(minor(dev))].pdinfo.bytes;
}

/* Set up the initial values for pdsector and clear defect map */
idsetdef(unit)
int unit;
{
    register int j;

    /* initialize sector 0 */
    idsect0[unit].pdinfo.driveid = 0x02;
    idsect0[unit].pdinfo.sanity = 0x00;
    idsect0[unit].pdinfo.version = 0x01;
    idsect0[unit].pdinfo.cyls = 1;
    idsect0[unit].pdinfo.tracks = 1;
    idsect0[unit].pdinfo.sectors = 18;
    idsect0[unit].pdinfo.bytes = 512;

    /* initialize mach defect management tables */
    for(j=0; j<(IDDEFSIZ/8); j++)    {
        iddefect[unit].map[j].bad.full = 0xffffffff;
        iddefect[unit].map[j].good.full = 0xffffffff;
    }
}

/* idopen - on first call reads in physical description, vtoc,
   and defect info */
idopen(dev,flag,otyp)
{
    struct buf *geteblk();
    struct buf *bufhead;
    register int unit, i, j;
    int defcnt, defaddr;

    if (idnoscan == IDSET) {
        idnoscan = IDNULL;
        idscan ();
    }
    unit = iddn(minor(dev));
    if (idstatus[unit].equipped == IDNULL)   {
        /* no disk out there */
        u.u_error = ENXIO;
        cmn_err(CE_NOTE,"\nhard disk:  drive %d not equipped\n",unit);
        return;
    }
    while (idstatus[unit].open == IDOPENING) {
        sleep(&idstatus[unit],PZERO);
    }
    if (idstatus[unit].open == IDNOTOPEN)    {
```

```
idstatus[unit].open = IDOPENING;

/* set up default values in the pdsector */
idsetdef(unit);

/* read physical description sector */
bufhead = geteblk();
idsetblk (bufhead, B_READ, IDPDBLKNO, dev);
idstrategy(bufhead);
iowait(bufhead);
if (bufhead->b_flags & B_ERROR) {
    cmn_err(CE_WARN,"\nhard disk:  cannot read sector 0 on drive %d\n",unit);
    goto badopen;
}
idcopy (bufhead->b_un.b_addr, &idsect0[unit], sizeof(struct pdsector));


if (idsect0[unit].pdinfo.sanity == VALIDINFO)   {
    idsetdef(unit);
    cmn_err(CE_WARN,"\nhard disk:  Drive %d is in the 1.0 layout.
             It can not be used until the conversion is made to the
             current layout.\n",unit);
    goto opendone;
}else if (idsect0[unit].pdinfo.sanity != VALID_PD)   {
    cmn_err(CE_WARN,"\nhard disk:  Bad sanity word on drive %d.\n",unit);
    goto badopen;
}

/* read the defect map */
if (idsect0[unit].pdinfo.defectsz > IDDEFSIZ) {
    cmn_err (CE_WARN, "\nhard disk: too little space allocated in driver
             for defect table on drive %d\n", unit);
    goto badopen;
}
for (defcnt = 0; defcnt <
  (idsect0[unit].pdinfo.defectsz/idsect0[unit].pdinfo.bytes);defcnt++) {
    idsetblk (bufhead, B_READ, idsect0[unit].pdinfo.defectst+defcnt, dev);
    idstrategy(bufhead);
    iowait(bufhead);
    if (bufhead->b_flags & B_ERROR) {
        cmn_err(CE_WARN,"\nhard disk:  Cannot read defect map on drive
                 %d\n",unit);
        goto badopen;
    }
    defaddr = ((int)&iddefect[unit])+(defcnt*idsect0[unit].pdinfo.bytes);
    idcopy (bufhead->b_un.b_addr, defaddr, idsect0[unit].pdinfo.bytes);
}

/* read in the vtoc */
```

```
        idsetblk (bufhead,B_READ, idsect0[unit].pdinfo.logicalst+IDVTOCBLK, dev);
        idstrategy(bufhead);
        iowait(bufhead);
        if (bufhead->b_flags & B_ERROR) {
            cmn_err(CE_WARN,"\nhard disk:  Cannot read the VTOC on drive
                    %d\n",unit);
            goto badopen;
        }
        idcopy (bufhead->b_un.b_addr, &idvtoc[unit], sizeof(struct vtoc));
        if (idvtoc[unit].v_sanity != VTOC_SANE) {
            cmn_err(CE_WARN,"\nhard disk: Bad sanity word in VTOC on drive %d.\n",unit);
            goto opendone;
        }

        /* open is complete - wakeup sleeping processes and return buffer */

        idstatus[unit].open = IDISOPEN;
        goto opendone;
badopen:
        u.u_error = ENXIO;
opendone:

        if(idstatus[unit].open != IDISOPEN)
            idstatus[unit].open = IDNOTOPEN;
        wakeup(&idstatus[unit]);
        brelse(bufhead);

    }
}

/* idclose is provided as a null function */
idclose(dev)
{
}

/* reset and initialize controller, initialize controller table */
/* check for drive ready and recalibrate drives */
idinit()
{
    register int i, j;
    int iplsave;
    dev_t ddev;
    extern int hdeeduc, hdeedct;

    /* ENTER CRITICAL REGION */
    iplsave = spl6();

    /*  controller initialization - specify parameter structure  */
    idspec_s.mode = 0x18;
    idspec_s.dtlh = 0xe2;
    idspec_s.dtll = 0x00;
    idspec_s.etn = 0xef;
    idspec_s.esn = 0x11;
    idspec_s.gpl2 = 0x0d;
    idspec_s.rwch = 0x00;
    idspec_s.rwcl = 0x80;

    idspec_f.mode = 0x1f;
    idspec_f.dtlh = 0xe2;
    idspec_f.dtll = 0x00;
    idspec_f.etn = 0xef;
    idspec_f.esn = 0x11;
    idspec_f.gpl2 = 0x0d;
    idspec_f.rwch = 0x00;
    idspec_f.rwcl = 0x80;

    for(i=0, j=0; i < IDNDRV; i++, j++) {
        /* initialize drive state */
        idstatus[i].open = IDNOTOPEN;
        idstatus[i].state = IDIDLE;
        idstatus[i].equipped = IDNULL;
        idpcacheaddr[i] = (unsigned int) vtop(&idcache[j],IDNULL);
        if(((idpcacheaddr[i] & MSK64K)+0x200) >BND64K) {
            j++;
            idpcacheaddr[i]=(unsigned int)vtop(&idcache[j],IDNULL);
        }
    }
    idrecal (IDNOUNIT);
    idnoscan = IDSET;
    idscanflag = IDSET;
    /* EXIT CRITICAL REGION */
    splx (iplsave);
    for (j = 0; j < 128; j++)
        if (MAJOR[j] == idiskmaj) break;
    for (i = 0; i < IDNDRV; i++)
        if (idstatus[i].equipped != IDNULL) {
            ddev = makedev(j, idmkmin(i));
            hdeeqd(ddev, IDPDBLKNO, EQD_ID);
        }
} /* end idinit */

idstrategy (bufhead)
register struct buf *bufhead;
{
    register struct iobuf *drvtab;  /* drive status pointer */
    daddr_t lastblk;    /* last block in partition */
    register int unit;       /* drive unit id */
    int partition;      /* drive partition number */
int iplsave;     /* saved interrupt level */
int sectoff;     /* start sector of this partition */

/* initialize local variables */
partition = idslice (minor (bufhead->b_dev));
unit = iddn (minor (bufhead->b_dev));
if(idstatus[unit].equipped == IDNULL)
    goto diskerr;

if (idnodev (minor (bufhead->b_dev)))    {
    lastblk = (idsect0[unit].pdinfo.sectors * idsect0[unit].pdinfo.tracks *
                idsect0[unit].pdinfo.cyls);
    sectoff = 0x00;
}else    {
    /* check for invalid VTOC */
    if (idvtoc[unit].v_sanity != VTOC_SANE) {
        goto diskerr;
    }
    /* check for read only partition */
    if (((idvtoc[unit].v_part[partition].p_flag & V_RONLY)==V_RONLY)
            &&((bufhead->b_flags & B_READ)!=B_READ)) {
        u.u_error = ENXIO;
        cmn_err (CE_WARN, "\nhard disk: partition %d on drive %d is marked
                  read only\n", partition, unit);
        goto diskerr;
    }
    lastblk = idvtoc[unit].v_part[partition].p_size;
    sectoff = (idvtoc[unit].v_part[partition].p_start
               + idsect0[unit].pdinfo.logicalst);
}
drvtab = &idtab[unit];

/* if the requested block does not exist within requested partition */
if ((bufhead->b_blkno >= lastblk) ||
    (bufhead->b_blkno < IDFRSTBLK)  ||
    (bufhead->b_blkno+((bufhead->b_bcount-1)/idsect0[unit].pdinfo.bytes) >=
     lastblk)) {
        if ((bufhead->b_blkno == lastblk) && (bufhead->b_flags&B_READ)){
            /* this case is here to help read ahead */
            bufhead->b_resid = bufhead->b_bcount;
            iodone(bufhead);
            return;
        }
        goto diskerr;
}


/* ENTER CRITICAL REGION */
iplsave = spl6();
```

```
    bufhead->cylin  = ((bufhead->b_blkno+sectoff)
            /(idsect0[unit].pdinfo.sectors
            *idsect0[unit].pdinfo.tracks));
    bufhead->b_start = lbolt; /* time stamp request */
    idtime[unit].io_cnt++;    /* inc operations count */
    idtime[unit].io_bcnt += (bufhead->b_bcount + BSIZE-1) >> BSHIFT; /* inc disk
                                                                  block count */
    drvtab->qcnt++;        /* inc drive current request count */


    /* link buffer header to drive worklist */
    bufhead->av_forw = IDNULL;
    if (drvtab->b_actf == IDNULL) {
        idscanflag = IDSET;
        drvtab->b_actf = bufhead;
        drvtab->b_actl = bufhead;
        drvtab->acts = (int)bufhead;
        idsetup (unit);
        idseek (unit);
    } else {
        register struct buf *ap, *cp;
        if (((int)idtime[unit].io_cnt&0x0f) == 0)
            drvtab->acts = (int)drvtab->b_actl;
        for (ap = (struct buf *)drvtab->acts; cp = ap->av_forw; ap = cp) {
            int s1, s2;
            if ((s1 = ap->cylin - bufhead->cylin)<0)
                s1 = -s1;
            if ((s2 = ap->cylin - cp->cylin)<0)
                s2 = -s2;
            if (s1 < s2)
                break;
        }
        ap->av_forw =  bufhead;
        if ((bufhead->av_forw = cp) == NULL)
            drvtab->b_actl = bufhead;
        bufhead->av_back = ap;
    }
    /* EXIT CRITICAL REGION */
    splx (iplsave);
    return;

diskerr:
    bufhead->b_flags |= B_ERROR;
    bufhead->b_error = ENXIO;
    iodone (bufhead);
    return;
}

idsetup (unit)  /* This routine fills the drive command buff from buff head */
```

```
unsigned int unit;
{
    register intblkcnt;     /* number of blocks this job */
    register struct buf *bufhead;   /* buffer header */
    register struct iobuf *drvtab;  /* head of drive worklist */
    register unsigned char *user, *driver;
    union diskaddr daddress;/* disk address for current job */
    int vaddress;           /* virtual memory address */
    int paddress;           /* physical memory address */
    int sectoff;    /* sector offset into drive */
    int sectno;         /* sector number on cylinder */
    int partition;      /* drive partition number */
    struct defect *deftab;      /* pointer to the defect table */
    union diskaddr lastsect;/* last sector for this job */
    int lsectoff;           /* offset of the last sector in job */
    int i;
    int defcnt, bytes;
    unsigned char partial;

    /* initialize local variables */
    drvtab = &idtab[unit];
    bufhead = drvtab->b_actf;
    deftab = iddefect[unit].map;
    partition = idslice (minor (bufhead->b_dev));
    if (idnodev (minor (bufhead->b_dev)))
        sectoff = 0x00;
    else
        sectoff = (idvtoc[unit].v_part[partition].p_start +
                    idsect0[unit].pdinfo.logicalst);

    /* if no work on worklist */
    if (bufhead == IDNULL)
        return(IDFAIL);

    /* if this is the first time this job has come through, time stamp it
       and save buffer header information */
    if (drvtab->b_active == 0) {
        idsvaddr[unit].b_addr = bufhead->b_un.b_addr;
        idsvaddr[unit].b_blkno = bufhead->b_blkno;
        idsvaddr[unit].b_bcount = bufhead->b_bcount;
        drvtab->io_start = lbolt;
    }

    /* increase activity count */
    drvtab->b_active++;

    /* clear result information */
    idstatus[unit].retries = IDRETRY;
    idstatus[unit].reseeks = IDRESEEK;
```

```
/* compute disk address */
bufhead->cylin = ((idsvaddr[unit].b_blkno+sectoff)/
                  (idsect0[unit].pdinfo.sectors*idsect0[unit].pdinfo.tracks));
    /* start cylinder */
sectno = (idsvaddr[unit].b_blkno+sectoff)%(idsect0[unit].pdinfo.sectors*
          idsect0[unit].pdinfo.tracks);
    /* offset into start cylinder */

/* load disk address */
daddress.part.pcnh = (bufhead->cylin>>8)&0xff;
daddress.part.pcnl = bufhead->cylin&0xff;
daddress.part.phn = sectno/idsect0[unit].pdinfo.sectors;
daddress.part.psn = sectno%idsect0[unit].pdinfo.sectors;

/* get physical address from buffer header */
vaddress = (int) idsvaddr[unit].b_addr;
paddress = vtop(vaddress, bufhead->b_proc);
if(paddress==IDNULL)
    cmn_err(CE_PANIC,"\nhard disk: Bad address returned by VTOP\n");
/* blocks to do this job */
blkcnt = ((idsvaddr[unit].b_bcount-1)/idsect0[unit].pdinfo.bytes) + 1;

/* chop the job up */
/* make sure we don't overrun track boundary */
if((daddress.part.psn+blkcnt) > idsect0[unit].pdinfo.sectors)
    blkcnt = idsect0[unit].pdinfo.sectors - daddress.part.psn;

/* check for 64K-byte boundary overrun  or partial sector r/w */
partial = 0;
if (idsvaddr[unit].b_bcount < idsect0[unit].pdinfo.bytes)
    partial = 1;
if ((((paddress & MSK64K)+(blkcnt*idsect0[unit].pdinfo.bytes)) > BND64K)
|| (partial)) {
    blkcnt = (BND64K - (paddress & MSK64K)) / idsect0[unit].pdinfo.bytes;
    /* if sector r/w crosses 64-Kbyte boundary or partial sector */
    if((blkcnt == 0) || partial) {
        blkcnt = 1;
        /* if its a write to disk, copy form user to driver */
        if((bufhead->b_flags&B_READ) != B_READ) {
            bytes = idsect0[unit].pdinfo.bytes;
            if (partial) {
                register unsigned int *zp;
                bytes = idsvaddr[unit].b_bcount;
                zp = (unsigned int *)idpcacheaddr[unit];
                for (i=0; i<128; i++)
                    *zp++ = 0x00000000;
            }
            user = (unsigned char *) paddress;
            driver = (unsigned char *) idpcacheaddr[unit];
```

```
                for (i=0; i<bytes; i++)
                    *driver++ = *user++;
            }
            paddress = idpcacheaddr[unit];
        }
    }

    /* look for any defective sectors in this job */
    for (defcnt=0;(defcnt<(IDDEFSIZ/8))&&(daddress.full>deftab->bad.full);
         defcnt++) {
            deftab++;
    }
    /* determine the address of the last sector for this job */
    lastsect.part.pcnh = daddress.part.pcnh;
    lastsect.part.pcnl = daddress.part.pcnl;
    lastsect.part.phn = (sectno+blkcnt-1)/(idsect0[unit].pdinfo.sectors);
    lastsect.part.psn = (sectno+blkcnt-1)%(idsect0[unit].pdinfo.sectors);

    if(lastsect.full >= deftab->bad.full) {
        if(daddress.full == deftab->bad.full) {
            daddress.full = deftab->good.full;
            blkcnt=1;
        } else {
            lsectoff = (deftab->bad.part.phn*
                        (idsect0[unit].pdinfo.sectors))+deftab->bad.part.psn;
            blkcnt = lsectoff-sectno;
        }
    }


    /* load seek parameters */
    idseekparam[unit].pcnh = daddress.part.pcnh;
    idseekparam[unit].pcnl = daddress.part.pcnl;

    /* load transfer parameters */
    idxferparam[unit].phn = daddress.part.phn;
    idxferparam[unit].lcnh = daddress.part.pcnh;
    idxferparam[unit].lcnl = daddress.part.pcnl;
    idxferparam[unit].lhn = daddress.part.phn;
    idxferparam[unit].lsn = daddress.part.psn;
    idxferparam[unit].scnt = blkcnt;
    idxferparam[unit].bcnt = blkcnt*idsect0[unit].pdinfo.bytes;
    idxferparam[unit].necop = (bufhead->b_flags&B_READ) ? IDREAD:IDWRITE;
    idxferparam[unit].dmacop = (bufhead->b_flags&B_READ) ? WDMA:RDMA;
    idxferparam[unit].b_addr = paddress;
    idxferparam[unit].unitno = unit;
    /* if the head number is greater than 7 */
    if (idxferparam[unit].phn >= IDMAXHD)
        idxferparam[unit].unitno += IDADDEV;
    /* adjust remaining byte count and start address */
    if(idxferparam[unit].b_addr != idpcacheaddr[unit]) {
        idsvaddr[unit].b_bcount -= (blkcnt*idsect0[unit].pdinfo.bytes);
        idsvaddr[unit].b_addr += (blkcnt*idsect0[unit].pdinfo.bytes);
    }
    idsvaddr[unit].b_blkno = idsvaddr[unit].b_blkno + blkcnt;
    return(IDPASS);
} /* end idsetup */

idrecal (unit)
register int unit;
{
    unsigned int i, j;
    unsigned char retval[2];
    register struct buf *bufhead;
    register struct iobuf *drvtab;
    register struct idstatstruct *stat;
    unsigned short cyl;
    int sects, tracks;

    if (unit != IDNOUNIT)
        idstatus[unit].reseeks--;
    idldcmd(IDRESET, &idnoparam, IDNOPARAMCNT, IDINTON);
    /* wait for controller to reset */
    for(i=0; i < 1000; i++)
    ;
    /* re-specify controller characteristics */
    idldcmd(IDSPECIFY, &idspec_s, IDSPECCNT, IDINTOFF);
    /* clear out not-ready interrupts from nonexisting drives */
    for(i=0; i<10000; i++)
    ;
    if (ID->statcmd & IDSINTRQ) {
        idldcmd(IDSENSEINT, &idnoparam, IDNOPARAMCNT, IDINTOFF);
        while ((ID->statcmd & IDSINTRQ) != IDSINTRQ)
        ;
        idldcmd(IDSENSEINT, &idnoparam, IDNOPARAMCNT, IDINTOFF);
    }
    /* init each drive attached to controller */
    for (i=0; i < IDNDRV; i++) {
        stat = &idstatus[i];
        /* check for drive ready */
        if (idldcmd(IDSENSEUS|i, &idnoparam, IDNOPARAMCNT,IDINTOFF)==IDFAIL) {
            if (stat->equipped == IDSET) {
                stat->equipped = IDNULL;
                idflush (i);
            }
            continue;
        }
        stat->ustbyte = ID->fifo;
        if((stat->ustbyte & IDREADY) != IDREADY) {
            if (stat->equipped == IDSET) {
                stat->equipped = IDNULL;
                idflush (i);
            }
            continue;
        }
        stat->equipped = IDSET;
        retval[i] = IDFAIL;
        for (j=0; ((retval[i] == IDFAIL) && (j<4)); j++) {
            if (idldcmd(IDRECAL|i|IDBUFFERED, &idnoparam,
                    IDNOPARAMCNT, IDINTOFF) == IDFAIL)
                continue;
            while((ID->statcmd & IDSINTRQ) != IDSINTRQ)
            ;
            if(idldcmd(IDSENSEINT,&idnoparam,IDNOPARAMCNT,IDINTOFF)==IDFAIL)
                continue;
            stat->istbyte = ID->fifo;
            if((stat->istbyte & (IDSEEKEND|IDSEEKERR)) != IDSEEKEND)
                continue;
            idtab[i].ccyl = 0;
            stat->state = IDIDLE;
            retval[i] = IDPASS;
        }
        if (retval[i] == IDFAIL) {
            stat->equipped = IDNULL;
            cmn_err(CE_WARN,"\nhard disk: cannot recal drive %d\n", i);
            idflush (i);
        }
    }
    idldcmd(IDSPECIFY, &idspec_f, IDSPECCNT, IDINTOFF);
    if (unit != IDNOUNIT) {
        stat = &idstatus[unit];
        drvtab = &idtab[unit];
        if ((stat->reseeks == 0) && (drvtab->b_actf != IDNULL)) {
            bufhead = drvtab->b_actf;
            drvtab->b_active = IDNULL;
            drvtab->b_actf = bufhead->av_forw;
            bufhead->b_flags |= B_ERROR;
            bufhead->b_error |= EIO;
            bufhead->b_resid = 0;
            drvtab->qcnt--;
            if (bufhead == (struct buf *)drvtab->acts)
                drvtab->acts = (int)drvtab->b_actf;
            /* update status information */
            idtime[unit].io_resp += lbolt - bufhead->b_start;
            idtime[unit].io_act += lbolt - drvtab->io_start;
```

```
                stat->state = IDIDLE;
                cyl=((idseekparam[unit].pcnh<<8)|idseekparam[unit].pcnl);
                idelog[unit].diskdev = bufhead->b_dev & ~(IDNODEV|idslice((-1)));
                sects = idsect0[unit].pdinfo.sectors;
                tracks = idsect0[unit].pdinfo.tracks;
                idelog[unit].blkaddr =
                    (cyl*sects*tracks)
                    + (stat->lhn*sects)
                    + stat->lsn;
                idelog[unit].readtype = HDECRC;
                idelog[unit].severity = HDEUNRD;
                idelog[unit].bitwidth = 0;
                idelog[unit].timestmp = lbolt;
                hdelog (&idelog[unit]);
                cmn_err (CE_WARN,"\nhard disk: cannot access sector %d, head %d,
                    cylinder %d, on drive %d\n", stat->lsn, stat->lhn, cyl, unit);
                /* return buffer header to UNIX */
                iodone (bufhead);

                if (drvtab->b_actf != IDNULL)
                    idsetup (unit);
            }
        }
        for(i=0; i < IDNDRV; i++)
            if (idtab[i].b_actf != IDNULL)
                idseek (i);
}

/* start seek for drive specified */
idseek (unit)
register int unit;
{
        unsigned int other;
        other = (unit^1);
        idstatus[unit].state = IDSEEK0;
        if ((idstatus[other].state & IDBUSY) == IDBUSY) {
            idstatus[unit].state |= IDWAITING; return;
        }
        idstatus[unit].state |= IDBUSY;
        if (idtab[unit].ccyl ==
            ((idseekparam[unit].pcnh<<8)|(idseekparam[unit].pcnl))) {
                idxfer (unit);
                return;
        }

        /* set drive current cylinder */
        idtab[unit].ccyl=((idseekparam[unit].pcnh<<8)|(idseekparam[unit].pcnl));
        idldcmd (IDSEEK|unit|IDBUFFERED, &idseekparam[unit], IDSEEKCNT, IDINTON);
}
```

```
idtimeout (unit)
register int unit;
{
    int iplsave;
    iplsave = spl6 ();
    dma_access (CH0IHD, idxferparam[unit].b_addr, idxferparam[unit].bcnt,
                DMNDMOD, idxferparam[unit].dmacop);
    idldcmd(idxferparam[unit].necop|idxferparam[unit].unitno,&idxferparam[unit],
            IDXFERCNT,IDINTON);
    splx (iplsave);
}

idxfer (unit)
register int unit;
{
    unsigned int other;
    int iplsave;
    unsigned ifcount;
    other = (unit^1);
    idstatus[unit].state = IDXFER;
    if ((idstatus[other].state & IDBUSY) == IDBUSY) {
        idstatus[unit].state |= IDWAITING;
        return;
    }
    idstatus[unit].state |= IDBUSY;
    idstatus[unit].retries--;
    if (idstatus[unit].retries == 0) {
        idstatus[unit].retries = IDRETRY;
        idrecal (unit);
        return;
    }
    if ((ifstate & IFFMAT1) == IFFMAT1) {
        timeout (idtimeout, unit, (2*HZ)/5);
        return (IDPASS);
    }
    if ((ifstate&IFBUSYF) == IFBUSYF) {
        iplsave = spltty ();
        DMAC->CBPFF = IDNULL;
        ifcount = 0;
        ifcount = DMAC->C1WC;
        ifcount |= (DMAC->C1WC<<8);
        if (ifcount != IFDMACNT) {
            timeout (idtimeout, unit, HZ/22);
            splx (iplsave);
            return (IDPASS);
        }
        splx (iplsave);
    }
    /* load the DMAC */
    dma_access (CH0IHD, idxferparam[unit].b_addr, idxferparam[unit].bcnt,
            DMNDMOD, idxferparam[unit].dmacop);
    if (idldcmd(idxferparam[unit].necop|idxferparam[unit].unitno, &idxferparam[unit],
            IDXFERCNT,IDINTON) == IDFAIL)
        return (IDFAIL);
    return (IDPASS);
} /* end idxfer */

idint (dev)
{
register struct buf *bufhead;
register unsigned char statreg;
register unsigned int unit;
register unsigned char *driver, *user;
struct iobuf *drvtab;
int vaddress;   /* virtual memory address of user space */
int istbyte;
unsigned int other;
int i, bytes;

statreg = ID->statcmd;
idscanflag = IDSET;

/* check spurious interrupt */
if ((statreg & (IDSINTRQ|IDENDMASK)) == 0) {
    /* increment spurious interrupt count */
    idspurintr++; return;
}

/* establish unit for command end interrupt */
if ((statreg & IDENDMASK) != 0) {
    if ((idstatus[0].state & IDBUSY) == IDBUSY)
        unit = 0;
    else if ((idstatus[1].state & IDBUSY) == IDBUSY)
        unit = 1;
    else {
        idspurintr++;
        ID->statcmd = IDCLCMNDEND;
        return;
    }
}
if ((statreg & IDSINTRQ) == IDSINTRQ) {
    /* if the controller is busy, mask the interrupt */
    if ((statreg & IDCBUSY) == IDCBUSY) {
        if ((ID->statcmd & IDCBUSY) == IDCBUSY)
            ID->statcmd = IDMASKSRQ;
        return;
    }
    if ((idstatus[0].state & IDBUSY) && (idstatus[1].state & IDSEEK1)) {
        idstatus[1].state |= IDWAITING;
        if ((statreg & IDENDMASK) == 0) {
            ID->statcmd = IDMASKSRQ; return;
        }
    }
    else if ((idstatus[1].state & IDBUSY) && (idstatus[0].state & IDSEEK1)) {
        idstatus[0].state |= IDWAITING;
        if ((statreg & IDENDMASK) == 0) {
            ID->statcmd = IDMASKSRQ; return;
        }
    }
    else if ((idstatus[0].state & IDBUSY) || (idstatus[1].state & IDBUSY)) {
        if ((statreg & IDENDMASK) == 0) {
            ID->statcmd = IDMASKSRQ; return;
        }
    }
    else {
        if(idldcmd(IDSENSEINT,&idnoparam,IDNOPARAMCNT,
            IDINTOFF)==IDFAIL) {
            idrecal(IDNOUNIT); return;
        }
        istbyte = ID->fifo;
        unit = istbyte & IDUNITADD;
        if (unit >= IDNDRV) {
            idspurintr++; return;
        }
        idstatus[unit].istbyte = istbyte;
        if ((idstatus[unit].istbyte & IDSEEKMSK) != IDSEEKEND) {
            idrecal (unit); return;
        }
        if ((idstatus[unit].state & IDSEEK1) != IDSEEK1) {
            idspurintr++; return;
        }
    }
}

    switch(idstatus[unit].state & (IDSEEK0|IDSEEK1|IDXFER)) {
    /* Driver expected seek complete? */
    case IDSEEK0:
        ID->statcmd = IDCLCMNDEND;
        other = (unit^1);
        if ((statreg & IDENDMASK) != IDCMDNRT) {
            idrecal (unit); return;
        }
        idstatus[unit].state = IDSEEK1;
        if ((idstatus[other].state & IDWAITING) == IDWAITING) {
            if ((idstatus[other].state & IDSEEK0) == IDSEEK0) {
                idseek (other); return;
                }
                if ((idstatus[other].state & IDSEEK1) == IDSEEK1) {
                    if (idldcmd(IDSENSEINT,&idnoparam,IDNOPARAMCNT,
                        IDINTOFF)==IDFAIL) {
                        idrecal(other); return;
                    }
                    istbyte = ID->fifo;
                    if ((istbyte & IDUNITADD) != other) {
                        idspurintr++; return;
                    }
                    idstatus[other].istbyte = istbyte;
                    if ((idstatus[other].istbyte & IDSEEKMSK) != IDSEEKEND) {
                        idrecal (other); return;
                    }
                    idxfer (other); return;
                }
                if ((idstatus[other].state & IDXFER) == IDXFER) {
                    idxfer (other); return;
                }
            }
            return;

        case IDSEEK1:
            if ((idstatus[unit].istbyte & IDSEEKMSK) != IDSEEKEND) {
                idrecal (unit); return;
            }
            idxfer (unit); return;

        case IDXFER:
            /* access extended access information */
            idstatus[unit].statreg = statreg;
            idstatus[unit].estbyte = ID->fifo;
            idstatus[unit].phn = ID->fifo;
            idstatus[unit].lcnh = ID->fifo;
            idstatus[unit].lcnl = ID->fifo;
            idstatus[unit].lhn = ID->fifo;
            idstatus[unit].lsn = ID->fifo;
            idstatus[unit].scnt = ID->fifo;

            ID->statcmd = IDCLCMNDEND;
            idstatus[unit].state &= ~IDBUSY;
            other = (unit^1);

            /* format controller has lost control of drive ? */
            if ((statreg & IDRESETRQ) || (statreg & IDERROR)) {
                idrecal (unit); return;
            }
            /* command terminated abnormally ? */
            if (statreg&IDCMDABT) {
```

```
            if (idstatus[unit].estbyte & (IDDMAOVR|IDEQUIPTC|IDDATAERR)) {
                idxfer (unit); return;
            }
            idrecal (unit); return;
        }
        if ((idstatus[other].state & IDWAITING) == IDWAITING) {
            if ((idstatus[other].state & IDSEEK0) == IDSEEK0) {
                idseek (other); goto wrapup;
            }
            if ((idstatus[other].state & IDSEEK1) == IDSEEK1) {
                if (idldcmd(IDSENSEINT,&idnoparam,IDNOPARAMCNT,IDINTOFF)==IDFAIL) {
                    idrecal(other); return;
                }
                istbyte = ID->fifo;
                if ((istbyte & IDUNITADD) != other) {
                    idspurintr++; goto wrapup;
                }
                idstatus[other].istbyte = istbyte;
                if ((idstatus[other].istbyte & IDSEEKMSK) != IDSEEKEND) {
                    idrecal (other); return;
                }
                idxfer (other);
                goto wrapup;
            }
            idxfer (other);
        }
    }

wrapup:
    drvtab = &idtab[unit];
    bufhead = drvtab->b_actf;

    /* if buffering for 64K-byte boundary crossing or partial sector r/w */
    if(idxferparam[unit].b_addr==idpcacheaddr[unit]) {
        bytes = idsect0[unit].pdinfo.bytes;
        if (idsvaddr[unit].b_bcount < bytes)
            bytes = idsvaddr[unit].b_bcount;
        /* if reading disk, copy out to user space */
        if(idxferparam[unit].necop==IDREAD) {
            vaddress = (int) idsvaddr[unit].b_addr;
            user = (unsigned char *)vtop(vaddress, bufhead->b_proc);
            driver = (unsigned char *)idpcacheaddr[unit];
            for (i=0; i<bytes; i++)
                *user++ = *driver++;
        }
        idsvaddr[unit].b_addr += bytes;
        idsvaddr[unit].b_bcount -= bytes;
    }
    /* if not done with multi-sector job */
    if(idsvaddr[unit].b_bcount != 0x00) {
        idsetup (unit);
        idseek(unit);
        return;
    }

    /* re-initialize drive worklist header information and unlink buffer header */
    idstatus[unit].state = IDIDLE;
    drvtab->b_active = IDNULL;
    drvtab->b_actf = bufhead->av_forw;
    bufhead->b_resid = 0;
    drvtab->qcnt--;
    if (bufhead == (struct buf *)drvtab->acts)
        drvtab->acts = (int)drvtab->b_actf;
    /* update status information */
    idtime[unit].io_resp += lbolt - bufhead->b_start;
    idtime[unit].io_act += lbolt - drvtab->io_start;

    /* return buffer header to UNIX */
    iodone (bufhead);

    /* if no active jobs for drive */
    if (drvtab->b_actf != IDNULL) {
        idsetup (unit); /* load job parameters in command buffer */
        idseek(unit);
    }
}

idflush (unit)
register unsigned unit;
{
    register struct buf *bufhead;
    register struct iobuf *drvtab;
    drvtab = &idtab[unit];
    while (drvtab->b_actf != IDNULL) {
        bufhead = drvtab->b_actf;
        drvtab->b_active = IDNULL;
        drvtab->b_actf = bufhead->av_forw;
        bufhead->b_resid = bufhead->b_bcount;
        bufhead->b_flags |= B_ERROR;
        bufhead->b_error |= EIO;
        drvtab->qcnt--;
        if (bufhead == (struct buf *)drvtab->acts)
            drvtab->acts = (int)drvtab->b_actf;
        idstatus[unit].state = IDIDLE;
        /* return buffer header to UNIX */
        iodone (bufhead);
    }
    cmn_err(CE_WARN,"\nhard disk: drive %d out of service\n", unit);
```

```
}
idread(dev)
{
    if (physck(idvtoc[iddn(minor (dev))].v_part[idslice(minor (dev))].p_size,
                B_READ))
        physio(idstrategy, 0, dev, B_READ);
}

idwrite(dev)
{
    if (physck(idvtoc[iddn(minor (dev))].v_part[idslice(minor (dev))].p_size,
                B_WRITE))
        physio(idstrategy, 0, dev, B_WRITE);
}

idprint (dev,str)
char *str;
{
    cmn_err(CE_NOTE,"%s on winchester drive, slice %d\n", str, dev&7);
}

/* routine to load hard disk controller */
idldcmd(command, params, paramcnt, intopt)

unsigned char command;  /* command opcode */
unsigned char *params;  /* pointer to first parameter in parameter list */
unsigned char intopt;   /* interrupt option */
short paramcnt; /* number of parameters for this command */
{
    while (ID->statcmd & IDCBUSY) /* wait for controller not busy */
    ;
    ID->statcmd = IDCLFIFO;      /* clear parameter fifo */
    while (paramcnt > 0) {       /* load parameters into controller */
        ID->fifo = *params++;
        paramcnt--;
    }
    ID->statcmd = command;  /* load command opcode into controller */
    if (intopt)
        return (IDPASS);
    /* wait for command end from controller */
    while (ID->statcmd & IDCBUSY)
    ;
    if ((ID->statcmd & IDENDMASK) != IDCMDNRT) {
        ID->statcmd = IDCLCMNDEND;
        return (IDFAIL);
    }
    ID->statcmd = IDCLCMNDEND;
    return (IDPASS);
}
```

```
ididle()
{
    register int i;
    for(i=0;i <IDNDRV;i++)
        if(idtab[i].b_actf != IDNULL)
            return(1);
    return(0);
}

idioctl(dev,cmd,args,flag)
struct io_arg *args;
{
    struct buf *geteblk();
    struct buf *bufhead;
    int errno, i, xfersz;
    register int unit;
    unsigned int block, mem, count, numbytes, defblock;


    unit = iddn (minor (dev));

    switch(cmd) {

    case V_PREAD:
        bufhead = geteblk();
        block = args->sectst;
        mem = args->memaddr;
        count = args->datasz;
        while ( count ) {
            idsetblk (bufhead, B_READ, block, dev);
            idstrategy(bufhead);
            iowait(bufhead);
            if (bufhead->b_flags & B_ERROR) {
                errno = V_BADREAD;
                suword (&args->retval,errno);
                goto ioctldone;
            }
            xfersz = min (count, bufhead->b_bcount);
            if (copyout(bufhead->b_un.b_addr, mem, xfersz) != 0) {
                errno = V_BADREAD;
                suword (&args->retval,errno);
                goto ioctldone;
            }
            block+=2;
            count -= xfersz;
            mem += xfersz;
        }
        break;
```

```
case V_PWRITE:
    bufhead = geteblk();
    block = args->sectst;
    mem = args->memaddr;
    count = args->datasz;
    defblock = idsect0[unit].pdinfo.defectst;
    numbytes = 0;
    while (count) {
        idsetblk (bufhead, B_WRITE, block, dev);
        xfersz = min (count, bufhead->b_bcount);
        if (copyin (mem, bufhead->b_un.b_addr, xfersz) != 0) {
            errno = V_BADWRITE;
            suword(&args->retval, errno);
            goto ioctldone;
        }
        idstrategy(bufhead);
        iowait(bufhead);
        if (bufhead->b_flags & B_ERROR) {
            errno = V_BADWRITE;
            suword(&args->retval, errno);
            goto ioctldone;
        }

        /* update memory image if special data */
        if (bufhead->b_blkno ==  IDPDBLKNO) {
            idcopy (bufhead->b_un.b_addr, &idsect0[unit], xfersz);
            defblock = idsect0[unit].pdinfo.defectst;
        }
        if (bufhead->b_blkno == defblock) {
            defblock++;
            idcopy (bufhead->b_un.b_addr, (((unsigned int)
                 &iddefect[unit]) + numbytes), xfersz);
            numbytes += xfersz;
        }
        if (bufhead->b_blkno==(idsect0[unit].pdinfo.logicalst+IDVTOCBLK))
            idcopy (bufhead->b_un.b_addr, &idvtoc[unit], xfersz);

        block+=1;
        count -= xfersz;
        mem += xfersz;
    }
    break;

case V_PDREAD:
    bufhead = geteblk();
    idsetblk (bufhead, B_READ, IDPDBLKNO, dev);
    idstrategy(bufhead);
    iowait(bufhead);
    if (bufhead->b_flags & B_ERROR) {
```

```
                errno = V_BADREAD;
                suword (&args->retval,errno);
                goto ioctldone;
            }
            if (copyout(bufhead->b_un.b_addr, args->memaddr,
                        idsect0[unit].pdinfo.bytes) != 0) {
                errno = V_BADREAD;
                suword (&args->retval,errno);
                goto ioctldone;
            }
            break;

        case V_PDWRITE:
            bufhead = geteblk();
            idsetblk (bufhead, B_WRITE, IDPDBLKNO, dev);
            if (copyin (args->memaddr, bufhead->b_un.b_addr,
                        idsect0[unit].pdinfo.bytes) != 0) {
                errno = V_BADWRITE;
                suword(&args->retval, errno);
                goto ioctldone;
            }
            idstrategy(bufhead);
            iowait(bufhead);
            if (bufhead->b_flags & B_ERROR) {
                errno = V_BADWRITE;
                suword(&args->retval, errno);
                goto ioctldone;
            }
            break;

        case V_GETSSZ:
            suword(args->memaddr, idsect0[unit].pdinfo.bytes);
            return;

        default:
            return;
        }

    ioctldone:
        bufhead->b_bcount = SBUFSIZE;
        brelse(bufhead);
    }
```

```
/*
 *      3B2 Computer UNIX Integral Floppy Disk Driver
 *
 */

#include "sys/types.h"
#include "sys/param.h"
#include "sys/sbd.h"
#include "sys/dma.h"
#include "sys/csr.h"
#include "sys/iu.h"
#include "sys/immu.h"
#include "sys/dir.h"
#include "sys/sysmacros.h"
#include "sys/conf.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/proc.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/iobuf.h"
#include "sys/systm.h"
#include "sys/if.h"
#include "sys/cmn_err.h"
#include "sys/vtoc.h"
#include "sys/open.h"


#define PASS 0
#define FAIL 1

struct  {
    daddr_t nblocks;/* number of blocks in disk partition */
    int cyloff;     /* starting cylinder # of partition */
} if_sizes[8] = {
    990,   24,  /* partition 0 -cyl 24-78 (root)      */
    810,   34,  /* partition 1 -cyl 34-78         */
    612,   45,  /* partition 2 -cyl 45-78         */
    414,   56,  /* partition 3 -cyl 56-78         */
    216,   67,  /* partition 4 -cyl 67-78         */
    1404,  1,   /* partition 5 -cyl 1-78  (init)      */
    1422,  0,   /* partition 6 -cyl 0-78  (full disk) */
    18,    0    /* partition 7 -cyl 0     (boot)      */
} ;

struct ifccmd ifccmd;

struct iobuf iftab;     /* drive and controller info */
struct iotime ifstat;       /* drive status info */

int ifcsrset;
extern int sbdwcsr;

int ifspurint;      /* counter for spurious interrupts */
extern paddr_t if_addr[];   /* local bus addr of disk controller */



#define acts io_s1      /* space for driver save info */
#define cylin b_resid       /* bytes not transferred on error */
#define ccyl jrqsleep       /* process sleep counter on jrq full */


#define ifslice(x) (x&7)
#define ifformatdev(x) ((x>>3)&1)
#define ifnodev(x) ((x>>4)&1)

#define IFNODEV     0x10
#define IFPDBLKNO   1422
#define IF_PREAD4
#define IF_PWRITE   5
#define IF_PDREAD   6
#define IF_PDWRITE  7

#define NULL 0
#define SET  1


extern int ifloppy;
#define IF ((struct ifdev *) &ifloppy)

extern int duart;
#define CONS ((struct duart *) &duart)


int iftoflag;       /* flag set when timeout() has been called */
int iflag;      /* FLAG FOR COMMAND INTERRUPT INTERPRETATION */
int ifisopen;       /* flag set by ifopen routine */
int ifotyp[OTYPCNT];
int ifisroot;       /* is this floppy the root device */
unsigned int ifstate;   /* used for formatting-I/O contention */


int ifskcnt;    /* JOB RETRY COUNTERS */
int ifxfercnt;
int iflstdcnt;
int ifwrtflag;      /* FLAG FOR DELAY AFTER WRITE */
int ifside;
int ifcpside;

struct ifsave {
    caddr_t b_addr;
    daddr_t b_blkno;
    unsigned int b_bcount;
}ifsvaddr;

unsigned char ifcache[512];

unsigned int ifcacheaddr;

ifcopy(faddr, taddr, count)
unsigned int *faddr;
unsigned int *taddr;
unsigned int count;
{
    unsigned int *fptr;
    unsigned int *tptr;
    int i;

    tptr = taddr;
    fptr = faddr;
    for (i=0; i<(count/4); i++)
        *tptr++ = *fptr++;
}

unsigned char ifscanflag;
unsigned char ifnoscan;
ifscan ()
{
    int s;
    s = spl6();
    if (iftab.b_actf != NULL) {
        if (ifscanflag == NULL)
            ifflush ();
        else
            ifscanflag = NULL;
    }
    splx (s);
    timeout (ifscan, 0, (10*HZ));
}

ifsetblk(bufhead, cmd, blkno, dev)
struct buf *bufhead;
unsigned char cmd;
daddr_t blkno;
```

```
dev_t dev;
{
    clrbuf(bufhead);
    bufhead->b_flags |= cmd;
    bufhead->b_blkno = blkno;
    bufhead->b_dev = (dev|IFNODEV);
    bufhead->b_proc = 0x00;
    bufhead->b_flags &= ~B_DONE;
    if(cmd == B_WRITE)
        bufhead->b_bcount = 512;
}

/* FLOPPY - UNIX COMPATIBLE OPEN ROUTINE */
ifopen(dev, flag, otyp)
{
    if (((minor(dev)) & 0x7f) >= 8) {
        u.u_error = ENXIO;
        return;
    }
    if (ifnoscan == SET) {
        ifnoscan = NULL;
        ifscan ();
    }
    CONS->scc_sopbc = F_LOCK;   /* DOOR LOCK */
    CONS->scc_sopbc = F_SEL;/* DRIVE SELECT */
    if (otyp < 0 || otyp >= OTYPCNT) {
    } else if (otyp == OTYP_LYR)
        ifotyp[OTYP_LYR]++;
    else ifotyp[otyp] |= 1 << (dev & 0x7f);
    ifisopen = SET;
}

/* FLOPPY - UNIX COMPATIBLE CLOSE ROUTINE */
ifclose(dev, flag, otyp)
{
    register int i, osum;

    if (otyp < 0 || otyp >= OTYPCNT) {
    } else if (otyp == OTYP_LYR)
        ifotyp[OTYP_LYR]--;
    else ifotyp[otyp] &= ~(1 << (dev & 0x7f));
    for (osum = i = 0; i < OTYPCNT; osum |= ifotyp[i++]);
    if (osum) return;
    ifisopen = NULL;
    ifdeselect();
}

/* FLOPPY INITIALIZATION OF POINTERS UPON KERNEL REQUEST */
ifinit()
```

```
{
    ifcpside=NULL;
    iftoflag=NULL;
    ifstate=IFIDLEF;

    iftab.io_addr = (paddr_t)&ifloppy;
    iftab.io_start = NULL;
    iftab.b_actf = NULL;
    iftab.b_actl = NULL;
    iftab.qcnt = NULL;
    iftab.b_forw = NULL;
    iftab.b_forw = NULL;
    ifisopen = NULL;


    ifnoscan = SET;
    ifscanflag = SET;
    /* assign physical address of temporary cache */
    ifcacheaddr = (unsigned int) vtop(ifcache, 0);

}

ifstrategy(bp)
register struct buf *bp;
{
    register struct iobuf *dp;

    daddr_t lastblk;    /* last block in partition */
    int part;       /* partition number */
    int iplsave;    /* save interrupt priority */
    int ifbytecnt;

    part = ifslice(minor(bp->b_dev));
    lastblk = if_sizes[part].nblocks;
    bp->cylin = bp->b_blkno/(IFNUMSECT*IFNTRAC)+if_sizes[part].cyloff;
    ifbytecnt = bp->b_bcount;

    if((ifformatdev(minor(bp->b_dev)) == SET) || (ifnodev(minor(bp->b_dev)) == SET)){
        lastblk = (IFNUMSECT * IFTRACKS);
        ifbytecnt = (IFBYTESCT*IFNUMSECT);
    }

    dp = &iftab;

    /* CHECK FOR PARTITION OVERRUN I.E. BLOCK OUT OF BOUNDS */
    if ((bp->b_blkno<0) ||
    (bp->b_blkno>=lastblk) ||
    (bp->b_blkno+(ifbytecnt/IFBYTESCT))>lastblk){
        if ((bp->b_blkno == lastblk) && (bp->b_flags & B_READ))
```

```
                bp->b_resid = bp->b_bcount;
            else{
                bp->b_flags |= B_ERROR;
                bp->b_error = ENXIO;
            }
            iodone(bp); /* JOB TERMINATION */
            return;
        }


        iplsave = spl6();   /* save previous IPL */

        bp->b_start = lbolt;
        ifstat.io_cnt++;
        ifstat.io_bcnt += (bp->b_bcount + BSIZE-1) >> BSHIFT; /* inc disk block count */
        dp->qcnt++;

        bp->av_forw = NULL;     /* mark request as last on list */
        if (dp->b_actf == NULL){    /* if no request for drive */
            dp->b_actf = bp;/* link to front of worklist */
            dp->b_actl = bp;
            dp->acts = (int)bp;
            ifscanflag = SET;
            if (ifcsrset){
                if(iftoflag!=NULL){
                    untimeout(iftoflag);
                    iftoflag=NULL;
                }
                ifsetup();
                ifseek();
            }
            else
                ifspinup();
        } else {            /* link to end of list */
            register struct buf *ap, *cp;
            if (((int)ifstat.io_cnt&0x0f) == 0)
                dp->acts = (int) dp->b_actl;
            for (ap = (struct buf*)dp->acts; cp = ap->av_forw; ap = cp) {
                int s1, s2;
                if ((s1 = ap->cylin - bp->cylin) < 0)
                    s1 = -s1;
                if ((s2 = ap->cylin - cp->cylin) < 0)
                    s2 = -s2;
                if (s1 < s2)
                    break;
            }
            ap->av_forw = bp;
            if ((bp->av_forw = cp) == NULL)
                dp->b_actl = bp;
        bp->av_back = ap;
    }
    /* RETURN TO ORIGINAL IPL */
    splx(iplsave);




}
/* ROUTINE FOR FIRST JOB AFTER MOTOR ON */
iffirstjob()
{
    int iplsave;

    iplsave = spl6();

    ifsetup();
    ifseek();
    splx(iplsave);

}

/* FLOPPY MOTOR ON ROUTINE */
ifspinup()
{
    ifcsrset = SET;
    Wcsr->s_flop = SET;
    timeout(iffirstjob,0,HZ*2);
}

/* FLOPPY MOTOR OFF ROUTINE */
ifspindn()
{
    register struct iobuf *dp;

    dp = &iftab;

    if (dp->b_actf != NULL) /* VERIFY THAT THERE IS NO NEW JOB */
        return;
    ifdeselect();
    ifcsrset = NULL;
    Wcsr->c_flop = NULL;/* MOTOR OFF CONTROL */
}

ifflush ()
{
    register struct buf *bp;
    register struct iobuf *dp;
    dp = &iftab;
    while (dp->b_actf != NULL) {
        bp = dp->b_actf;
        dp->b_active = NULL;
        dp->b_errcnt = NULL;
        dp->b_actf = bp->av_forw;
        bp->b_resid = bp->b_bcount;
        bp->b_flags |= B_ERROR;
        bp->b_error |= EIO;
        dp->qcnt--;
        if (bp == (struct buf *) dp->acts)
            dp->acts = (int) dp->b_actf;
        if(dp->b_actl == bp)
            dp->b_actl = NULL;
        /* return buffer header to UNIX */
        iodone (bp);
    }
    cmn_err(CE_WARN,"\nfloppy disk: cannot access disk, worklist flushed\n");
    ifwrtflag=NULL;
    iftoflag=(timeout(ifspindn,0,HZ*2));
}

ifdeselect()
{
    if((ifisopen==NULL) && (iftab.b_actf==NULL) && (ifisroot==NULL)) {
        CONS->scc_ropbc = F_LOCK;
        CONS->scc_ropbc = F_SEL;
    }
}

/* FILL COMMAND BUFFER WITH INFORMATION FROM THE BUFFER HEADER */
        /* AND COMPUTE DISK ACCESS ADDRESS */
ifsetup()
{
    register struct ifccmd *cp;
    register struct buf *bp;
    register struct iobuf *dp;
    register unsigned char *if_dmem, *if_umem;
    int bytes;
    int blkcnt;
    int side;
    int partit,lstblk;
    int cyl;
    int i;

    dp = &iftab;
    bp = dp->b_actf;
    cp = &ifccmd;

    ifskcnt = NULL; /* RESET FLAGS FOR RETRIES */
    ifxfercnt = NULL;
```

```
iflstdcnt = NULL;


if(bp == NULL)  /* VERIFY THERE IS A JOB TO DO */
    return;
if(dp->b_active == NULL){
    dp->io_start = lbolt;   /* TIME STAMP JOB */
    ifsvaddr.b_bcount = bp->b_bcount;   /* SAVE ADDRESS FOR 64K */
    ifsvaddr.b_blkno = bp->b_blkno;     /*      BOUNDS          */
    ifsvaddr.b_addr = bp->b_un.b_addr;
}

dp->b_active = SET;     /* MARK DRIVE AS ACTIVE */

if(ifformatdev(minor(bp->b_dev)) == SET){ /* FORMAT / VERIFY ? */
    cp->trknum = ifsvaddr.b_blkno/(IFNUMSECT*IFNTRAC);
    side = ((ifsvaddr.b_blkno%(IFNUMSECT*2))/IFNUMSECT)*2;
    cp->baddr = vtop(ifsvaddr.b_addr,bp->b_proc);
    if(cp->baddr == NULL)
        cmn_err(CE_PANIC,"\nfloppy disk Bad address returned from VTOP\n");
    if ((bp->b_flags & B_READ) == B_READ) {
        cp->c_opc = (IFRDS|IFSLENGRP1|IFMSDELAY|side);
        cp->sectnum = (ifsvaddr.b_blkno%IFNUMSECT)+1;
        cp->bcnt = IFBYTESCT;
        ifsvaddr.b_bcount -= IFBYTESCT;
        ifsvaddr.b_blkno++;
        ifsvaddr.b_addr += IFBYTESCT;
    }
    else {
        cp->c_opc = (IFWRTRK|IFMSDELAY|side);
        ifsvaddr.b_bcount -= bp->b_bcount;
        cp->bcnt = bp->b_bcount;
    }
} else {
    partit = ifslice(minor(bp->b_dev));
    lstblk = if_sizes[partit].nblocks;

    /* find start cylinder in partition */
    if(ifnodev(minor(bp->b_dev)))
        cyl = 0x00;
    else
        cyl = if_sizes[partit].cyloff;
    /* find cylinder offset in partition */
    bp->cylin = ifsvaddr.b_blkno/(IFNUMSECT*IFNTRAC)+cyl;
    /* compute sector offset into cylinder  */
    cp->sectnum = (ifsvaddr.b_blkno%IFNUMSECT)+1;
    /* compute side */
    side = ((ifsvaddr.b_blkno%(IFNUMSECT*2))/IFNUMSECT)*2;
```

```
        /* load command buffer */
        cp->trknum = bp->cylin;
        cp->c_bkcnt = blkcnt = 1;
        cp->bcnt = bytes = IFBYTESCT;
        cp->c_opc = ((bp->b_flags & B_READ) ? IFRDS : IFWTS)|IFSLENGRP1|IFMSDELAY;
        cp->c_opc |= side; /* FOR HEAD SWITCH */
        cp->baddr = vtop(ifsvaddr.b_addr,bp->b_proc);
        if(cp->baddr==NULL)
            cmn_err(CE_PANIC,"\nfloppy disk: Bad address returned by VTOP\n");
        /* crossing 64K-byte boundary or partial sector transfer */
        if ((((cp->baddr&MSK64K) + (blkcnt*IFBYTESCT))
            > BND64K) || (ifsvaddr.b_bcount < IFBYTESCT)) {
            if ((bp->b_flags&B_READ) != B_READ) {
                if (ifsvaddr.b_bcount<IFBYTESCT) {
                    register unsigned int *zp;
                    bytes = ifsvaddr.b_bcount;
                    zp = (unsigned int *) ifcache;
                    for(i=0; i<IFBYTESCT/4; i++)
                        *zp++ = 0x00000000;
                }
                if_dmem = (unsigned char *) ifcache;
                if_umem = (unsigned char *) cp->baddr;
                for(i=0; i<bytes; i++)
                    *if_dmem++ = *if_umem++;
            }
            cp->baddr = ifcacheaddr;
        }

        /* keep track of byte count, and blk and mem addresses  */
        ifsvaddr.b_blkno += blkcnt;
        if(cp->baddr != ifcacheaddr) {
            ifsvaddr.b_addr += bytes;
            ifsvaddr.b_bcount -= bytes;
        }
    }
}

/* ROUTINE FOR SEEKING TO DESIRED TRACK */
ifseek()
{
    register struct ifccmd *cp;
    int i;

    cp = &ifccmd;

    ifskcnt++;
    if(IF->track == cp->trknum){/* CHECK FOR BEING ON TRACK */
        ifxfer();
        return;
    }
    if(ifwrtflag == SET){
        ifwrtflag=NULL;
        for(i=0;i<50;i++);
    }
    iflag = IFSEEKATT;      /* INTERRUPT ON SEEK FLAG */
    IF->data = cp->trknum;      /* LOAD DESTINATION TRACK */
    if((cp->c_opc & IFWRTRK) == IFWRTRK){ /* USED FOR FORMAT */
        IF->statcmd = (IFSEEK|IFSTEPRATE);
    }
    else{
        IF->statcmd = (IFSEEK|IFSTEPRATE|IFLDHEAD|IFVERIFY);/* LOAD SEEK CMD */
    }
}
ifrest()/* THIS FUNCTION IS SIGNIFICANT ONLY TO SEEK ERRORS */
{

    iflag = IFRESTORE;      /* INTERRUPT ON RESTORE FLAG */
    IF->statcmd = (IFREST|IFLDHEAD|IFVERIFY|IFSTEPRATE);
        /* RESTORE HEADS TO TRACK 00 */
    return;
}

ifxfer()/* DATA TRANSFER IS IMPLEMENTED */
{

    register struct ifccmd *cp;
    unsigned char cmd;
    int i;

    ifxfercnt++;
    cp = &ifccmd;
    iflag = IFXFER;     /* INTERRUPT ON TRANSFER FLAG */
    ifstate |= IFBUSYF;
    if ((ifstate & IFFMAT0) == IFFMAT0)
        ifstate |= IFFMAT1;

    if((cp->c_opc & IFWTS)==IFWTS || (cp->c_opc & IFWRTRK) == IFWRTRK)
            /* IS CMD A WRITE */
        cmd = RDMA; /* SET DIRECTION FOR DMA */
    else
        cmd = WDMA;

    if(ifwrtflag==SET && (ifcpside != ifside)){
        ifwrtflag=NULL;
        ifcpside=ifside;
        for(i=0;i<50;i++);
    }
    /* INITIALIZE DMA FOR TRANSFER */
    dma_access(CH1IFL,cp->baddr,cp->bcnt,SNGLMOD,cmd);
    if((cp->c_opc & IFWRTRK) != IFWRTRK)
        IF->sector = cp->sectnum;   /* LOAD CONTROLLER REGS */
    IF->statcmd = cp->c_opc;/* WITH SECTOR NO. AND CMD */

}


/* INTERRUPT HANDLER - STATUS - COMMAND INFORMATION INTERPRETER */
ifint(dev)
{
    register struct buf *bp;
    register struct iobuf *dp;
    register struct ifccmd *cp;
    unsigned char dstat;
    register unsigned char *if_dmem, *if_umem;
    unsigned int i;
    int bytes;

    ifscanflag = SET;
    dp = &iftab;
    bp = dp->b_actf;
    cp = &ifccmd;
    dstat = IF->statcmd;


    if(dp->b_actf == NULL){ /* IF NO JOB ON LIST LOG SPURIOUS */
        ifspurint++;
        return;
    }
    if((dstat & IFNRDY) == IFNRDY){
        cmn_err(CE_NOTE,"\nFloppy Access Error:
                Consult the Error Message Section");
        cmn_err(CE_CONT,"of the System Administration Utilities Guide");
        cmn_err(CE_CONT,"\n");
        goto diskerr;
    }


    switch(iflag){      /* SWITCH FOR EXTRENUOUSLY USED COMMANDS */
        case IFRESTORE:
            iflag = IFNONACTIVE;
            if(((dstat &  IFSKERR)==IFSKERR)||(ifskcnt==IFMAXSEEK)){
                cmn_err(CE_NOTE,"\nFloppy Access Error:
                        Consult the Error Message Section");
                cmn_err(CE_CONT,"of the System Administration Utilities Guide");
                cmn_err(CE_CONT,"\n");
                goto diskerr;
            }
            /* KICKING OFF SEEK AFTER RESTORE */
            if(ifskcnt <= IFMAXSEEK){
                ifseek();
                return;
            }
            goto diskerr;

        case IFSEEKATT:
            iflag = IFNONACTIVE;
            if((dstat & IFSKERR)==IFSKERR){
                ifrest();
                return;
            }
            ifxfer();
            return;

        case IFXFER:
            iflag = IFNONACTIVE;
            ifstate &= ~(IFBUSYF|IFFMAT1);
            if((cp->c_opc & IFWTS) == IFWTS)
                ifwrtflag=SET;
            if((dstat & IFWRPT) == IFWRPT) {
                cmn_err(CE_NOTE,"\nFloppy Access Error:
                        Consult the Error Message Section");
                cmn_err(CE_CONT,"of the System Administration Utilities Guide");
                cmn_err(CE_CONT,"\n");
                goto diskerr;
            }
            if(dstat & (IFCRCERR|IFRECNF)) {
                if(ifxfercnt <= IFMAXXFER) {
                    ifxfer();
                    return;
                }
                ifxfercnt=NULL;
                ifrest();
                return;
            }

            if((dstat & IFLSTDATA) == IFLSTDATA) {
                iflstdcnt++;
                if(iflstdcnt <= IFMAXLSTD){
                    ifxfer();
                    return;
                }
                iflstdcnt = NULL;
                cmn_err(CE_NOTE,"\nFloppy Access Error:
                        Consult the Error Message Section");
                cmn_err(CE_CONT,"of the System Administration Utilities Guide");
                cmn_err(CE_CONT,"\n");
                goto diskerr;
            }
            goto goodend;
        }   /* end of switch */

diskerr:
    bp->b_flags |= B_ERROR;
    bp->b_error |= ENXIO;

goodend:
    /* if the data is in the temporary cache */
    if(cp->baddr == ifcacheaddr) {
        bytes = IFBYTESCT;
        if (ifsvaddr.b_bcount < IFBYTESCT)
            bytes = ifsvaddr.b_bcount;
        /* if read, copy out to user */
        if ((bp->b_flags&B_READ) == B_READ) {
            if_dmem = (unsigned char *) ifcache;
            if_umem = (unsigned char *) vtop(ifsvaddr.b_addr,bp->b_proc);
            for(i=0;i<bytes;i++)
                *if_umem++ = *if_dmem++;

        }
        /* update pointer to user address space */
        ifsvaddr.b_addr += bytes;
        ifsvaddr.b_bcount -= bytes;
    }

    /* if no errors and more to do, then go again */
    if(((bp->b_flags & B_ERROR)==0) && (ifsvaddr.b_bcount != 0)){
        ifsetup();
        ifseek();
        return;
    }

    dp->b_active = NULL;
    dp->b_errcnt = NULL;
    dp->b_actf = bp->av_forw;
    bp->b_resid = NULL;
    dp->qcnt--;

    if (bp == (struct buf *) dp->acts)
        dp->acts = (int) dp->b_actf;
    if(dp->b_actl == bp)
        dp->b_actl = NULL;

    ifstat.io_resp += lbolt - bp->b_start;
    ifstat.io_act += lbolt - dp->io_start;
    iodone(bp);
    if(dp->b_actf != NULL){
        ifsetup();
        ifseek();
        return;
    }
    ifwrtflag=NULL;
    iftoflag=(timeout(ifspindn,0,HZ*2));
}

/* READ DEVICE ROUTINE */

ifread(dev)
{
    if (physck(if_sizes[minor(dev)&07].nblocks, B_READ))
        physio(ifstrategy, 0, dev, B_READ);
}

/* WRITE DEVICE ROUTINE */

ifwrite(dev)
{
    if (physck(if_sizes[minor(dev)&07].nblocks, B_WRITE))
        physio(ifstrategy, 0, dev, B_WRITE);
}

/* LOCAL PRINT ROUTINE */

ifprint(dev,str)
char *str;
{
    cmn_err(CE_NOTE,"%s on floppy drive, slice %d\n", str, dev&7);
}

ifidle()
{
    if(iftab.b_actf != NULL)
        return(1);
    return(0);
}
ifioctl(dev,cmd,arg,mode)
unsigned int dev, cmd, arg, mode;
{


    switch(cmd){
        case IFBCHECK:{

            struct ifformat *ifmat;
            paddr_t ifbaddr;
        ifmat = (struct ifformat *)arg;
        ifbaddr = vtop(ifmat->data, u.u_procp);
        if(ifbaddr == 0){
            cmn_err(CE_WARN,"\nfloppy disk: Bad address returned from VTOP\n");
            u.u_error = EFAULT;
            return;
        }
        if(((ifbaddr & MSK64K)+ifmat->size) > BND64K){
            ifmat->retval = FAIL;
        }
        else {
            ifmat->retval = PASS;
        }
        break;
    }
    case IFFORMAT:{

        register struct buf *bp;
        struct iftrkfmat *trkpt;
        struct ifformat *ifmat;
        int bpbcount;
        caddr_t bpbaddr;

        trkpt = (struct iftrkfmat *)arg;
        if(useracc(trkpt,sizeof(struct iftrkfmat),B_READ)==NULL){
            u.u_error = EFAULT;
            return;
        }
        bp = geteblk();
        bp->b_error = 0;
        bp->b_dev = (dev | IFPTN);
        bpbcount = bp->b_bcount;
        bpbaddr= bp->b_un.b_addr;
        bp->b_bcount = sizeof(struct iftrkfmat);
        bp->b_proc = u.u_procp;
        bp->b_un.b_addr =  ((caddr_t)trkpt);
        bp->b_flags = (B_BUSY | B_WRITE);
        bp->b_blkno = ((trkpt->dsksct[0].TRACK*(IFNUMSECT*2))+(trkpt->dsksct[0].
            SIDE*IFNUMSECT));
        u.u_procp->p_flag |= SLOCK;
        ifstate |= IFFMAT0;
        ifstrategy(bp);
        iowait(bp);
        bp->b_bcount = bpbcount;
        bp->b_un.b_addr = bpbaddr;
        brelse(bp);
        ifstate &= ~IFFMAT0;
        u.u_procp->p_flag &= ~SLOCK;
        break;
        }
        case IFCONFIRM:{

            register struct buf *bp;
            struct iftrkfmat *trkpt;
            struct ifformat *ifmat;
            int bpbcount;
            paddr_t ifbaddr;

            caddr_t bpbaddr;

            ifmat = (struct ifformat *)arg;
            trkpt = (struct iftrkfmat *)ifmat->data;
            if(useracc(trkpt,(IFNUMSECT*IFBYTESCT),B_READ)==NULL){
                u.u_error = EFAULT;
                return;
            }
            bp = geteblk();
            bp->b_error = 0;
            bp->b_flags = (B_BUSY | B_READ);
            bp->b_dev = (dev | IFPTN);
            bpbcount = bp->b_bcount;
            bpbaddr = bp->b_un.b_addr;
            bp->b_bcount = (IFNUMSECT*IFBYTESCT);
            bp->b_proc = u.u_procp;
            bp->b_un.b_addr = ((caddr_t) trkpt);
            bp->b_blkno = ((ifmat->iftrack*(IFNUMSECT*2))+(ifmat->ifside*IFNUMSECT));
            u.u_procp->p_flag |= SLOCK;
            ifstrategy(bp);
            iowait(bp);
            bp->b_bcount = bpbcount;
            bp->b_un.b_addr = bpbaddr;
            brelse(bp);
            u.u_procp->p_flag &= ~SLOCK;
            break;

        }
    case V_PREAD:{
        struct io_arg *ifargs;
        struct buf *geteblk();
        struct buf *bufhead;
        int errno, xfersz;
        unsigned int block, mem, count;

        ifargs = (struct io_arg *)arg;
        bufhead = geteblk();
        block = ifargs->sectst;
        mem = ifargs->memaddr;
        count = ifargs->datasz;
```

```
            while (count)   {
                ifsetblk (bufhead, B_READ, block, dev);
                ifstrategy(bufhead);
                iowait(bufhead);
                if (bufhead->b_flags & B_ERROR) {
                    errno = V_BADREAD;
                    suword(ifargs->retval,errno);
                    goto preaddone;
                }
                xfersz = min(count,bufhead->b_bcount);
                if (copyout(bufhead->b_un.b_addr, mem, xfersz) != 0){
                    errno = V_BADREAD;
                    suword(ifargs->retval,errno);
                    goto preaddone;
                }
                block+=2;
                count -= xfersz;
                mem += xfersz;
            }
preaddone:
            bufhead->b_bcount = BSIZE;
            brelse(bufhead);
            break;
        }
        case V_PWRITE:{
            struct io_arg *ifargs;
            struct buf *geteblk();
            struct buf *bufhead;
            int errno, xfersz;
            unsigned int block, mem, count;

            ifargs = (struct io_arg *)arg;
            bufhead = geteblk();
            block = ifargs->sectst;
            mem = ifargs->memaddr;
            count = ifargs->datasz;
            while (count) {
                ifsetblk(bufhead,B_WRITE,block,dev);
                xfersz = min(count,bufhead->b_bcount);
                if (copyin(mem,bufhead->b_un.b_addr,xfersz) != 0) {
                    errno = V_BADWRITE;
                    suword(ifargs->retval,errno);
                    goto pwritedone;
                }
                ifstrategy(bufhead);
                iowait(bufhead);
                if(bufhead->b_flags & B_ERROR)  {
                    errno = V_BADWRITE;
                    suword(ifargs->retval,errno);
                    goto pwritedone;
                }
                block +=1;
                count -= xfersz;
                mem += xfersz;
            }
pwritedone:
            bufhead->b_bcount = BSIZE;
            brelse(bufhead);
            break;
        }
        case V_PDREAD:{
            struct io_arg *ifargs;
            struct buf *geteblk();
            struct buf *bufhead;
            int errno, xfersz;
            unsigned int block, mem, count;

            ifargs = (struct io_arg *)arg;
            bufhead = geteblk();
            ifsetblk (bufhead, B_READ, IFPDBLKNO, dev);
            ifstrategy(bufhead);
            iowait(bufhead);
            if (bufhead->b_flags & B_ERROR) {
                errno = V_BADREAD;
                suword(ifargs->retval,errno);
                goto pdrddone;
            }
            if (copyout(bufhead->b_un.b_addr, ifargs->memaddr, IFBYTESCT)  != 0){
                errno = V_BADREAD;
                suword(ifargs->retval,errno);
                goto pdrddone;
            }
pdrddone:
            bufhead->b_bcount = BSIZE;
            brelse(bufhead);
            break;
        }
        case V_PDWRITE:{
            struct io_arg *ifargs;
            struct buf *geteblk();
            struct buf *bufhead;
            int errno;

            ifargs = (struct io_arg *)arg;
            bufhead = geteblk();
            ifsetblk(bufhead,B_WRITE,IFPDBLKNO,dev);
            if (copyin(ifargs->memaddr,bufhead->b_un.b_addr,IFBYTESCT) != 0) {
                errno = V_BADWRITE;
                suword(ifargs->retval,errno);
                goto pdwrtdone;
            }
            ifstrategy(bufhead);
            iowait(bufhead);
            if(bufhead->b_flags & B_ERROR)  {
                errno = V_BADWRITE;
                suword(ifargs->retval,errno);
                goto pdwrtdone;
            }
pdwrtdone:
            bufhead->b_bcount = BSIZE;
            brelse(bufhead);
            break;
        }
        default:
            u.u_error = EIO;
            return;
        }
}
```

```
/*  This is the common part  */

/*
 *      Copyright 1984 AT&T
 */
#include "sys/types.h"
#include "sys/param.h"
#include "sys/sysmacros.h"
#include "sys/buf.h"
#include "sys/sbd.h"
#include "sys/csr.h"

extern struct r16 sbdrcsr;
sdinit()
{
    idinit();
    ifinit();
}
sdopen(dev,flag,otyp)
{
    if( dev & 0x80 )
        ifopen(dev,flag,otyp);
    else
        idopen(dev,flag,otyp);
}
sdclose(dev,flag,otyp)
{
    if( dev & 0x80 )
        ifclose(dev,flag,otyp);
    else
        idclose(dev,flag,otyp);
}
sdioctl(dev,cmd,arg,flag)
{
    if( dev & 0x80 )
        ifioctl(dev,cmd,arg,flag);
    else
        idioctl(dev,cmd,arg,flag);
}
sdstrategy(bp)
struct buf *bp;
{
    if( bp->b_dev & 0x80 )
        ifstrategy(bp);
    else
        idstrategy(bp);
}
sdprint(dev,str)
char *str;
```

```
{
    if( dev & 0x80 )
        ifprint(dev, str);
    else
        idprint(dev, str);
}
sdread(dev)
{
    if( dev & 0x80 )
        ifread(dev);
    else
        idread(dev);
}
sdwrite(dev)
{
    if ( dev & 0x80 )
        ifwrite(dev);
    else
        idwrite(dev);
}

extern void ifint();
extern idint();

sdint(dev)
{

    idint(dev);
    if( sbdrcsr.data & CSRDISK )
        ifint(dev);
}
```

APPENDIX J: 3B2 COMPUTER CHARACTER DRIVER

This is the character driver used on the ports card for the 3B2 Computer.

```
/*
 *  Copyright 1984 AT&T
 *
 *  SCCS_id:   "@(#)ppc.c   1.1.1.4 10/22/83 13:53:34"
 *
 *
 * PPC Peripheral (3B2) PORTS Controller Driver
 */
#include "sys/param.h"
#include "sys/types.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/sbd.h"
#include "sys/immu.h"
#include "sys/user.h"
#include "sys/iu.h"
#include "errno.h"
#include "sys/file.h"
#include "sys/tty.h"
#include "sys/termio.h"
#include "sys/conf.h"
#include "sys/sysinfo.h"
#include "sys/firmware.h"
#include "sys/devcode.h"
#include "sys/cmn_err.h"
#include "sys/pump.h"
#include "sys/sysmacros.h"
#include "sys/cio_defs.h"
#include "sys/pp_dep.h"
#include "sys/queue.h"
#include "sys/ppc.h"
#include "sys/lla_ppc.h"
#include "sys/ppc_lla.h"

int pp_bnbr;

/*
 * Xflag modes
 */
```

```
#define X_ITIME     0x004       /* enable ITIME and IFS */

/*
 * raw mode parameters (1200-19200 baud)
 */
#define ITIME   2   /* intercharacter timer: 25-50 msec */
#define IFS 64  /* input field size: 64 bytes */

#define SYSG_TIME 1500  /* timeout for sysgen  change*/
#define CIO_TIME 1500       /* timeout for LLC(CIO) commands to complete*/
#define CL_TIME 50000       /* Time to compelete output drain on port close*/
/*
 * intercharacter timer values
 * for low baud rates
 */
char pptime[] = {
    0,  /* 0 */
    18, /* 50 */
    12, /* 75 */
    9,  /* 110 */
    7,  /* 134.5 */
    6,  /* 150 */
    5,  /* 200 */
    3,  /* 300 */
};
#define PU_BLOCK256 /* size of  kernel scratch memory for pump */
#define PU_LBLOCK   PU_BLOCK/4
long    scratch[PU_LBLOCK]; /* scratch area for ppc download used by the pump
                               routines */

char    ppc_speeds[16] =
{
    0, 0, B75BPS, B110BPS, B134BPS, B150BPS,
    0, B300BPS, B600BPS, B1200BPS, B1800BPS,
    B2400BPS, B4800BPS, B9600BPS, B19200BPS, 0
};
SG_DBLK sg_dblk;

int pumpflag;

/*
 * Minor device definition
 */
#define TP(b,p) &pp_tty[b*5+p]  /* tp from board and port*/

#define CB_PER_PPC  4   /* desired number of cblock per port*/
#define INIT_CB     3   /* Additional cblocks given when first port a
                           board is opened */
```

```
ppopen(dev, flag)
register dev_t dev;
{
    register struct tty *tp;
    register struct ppcboard *pb;
    register char *cptr;
    register dev_t device;

    device = minor(dev);

    /* check if this board is to be pumped */
    if( flag&O_PUMP )
    {
        if(u.u_uid != 0)
        {
            u.u_error = EPERM;
            return;
        }
        pumpflag = 1;
        return;
    }
    if (pb->b_state&SYSGFAIL)
    {
        u.u_error = ENXIO;
        return;
    }
    pb = &pp_board[ppcbid[device]];
    tp = &pp_tty[device]; /* get tty structure */

    /* if this port is not open initialize its parameters */
    if(!(tp->t_state&ISOPEN))
    {
        splpp();
        /* get entry for device enable*/
        while((!(lla_xfree(ppcbid[device],ppcpid[device]))) ||(tp->t_dstat&CLDRAIN))
        {
            tp->t_dstat |= WENTRY;
            tp->t_dstat |= OPDRAIN;
            sleep(&tp->t_cc[1],TTOPRI);
            if (!(tp->t_dstat&OPDRAIN))
            {
                u.u_error = ENXIO;
                spl0();
                return;
            }


        }
        tp->t_dstat &= ~OPDRAIN;
```

```
                spl0();

            if (!(tp->t_state&ISOPEN))
            {
                splpp();
                ttinit(tp);
                ppc_conn(ppcbid[device],ppcpid[device]);
                spl0();
                ppparm(dev);
                if(u.u_error)
                    return;
            }
        }
        splpp();
        /* If delay on open is set wait unit carrier is on */
        if(!(flag&FNDELAY) && !(tp->t_state&CARR_ON))
            while(!(tp->t_state&CARR_ON)) {
                tp->t_state |= WOPEN; /* waiting for open to
                               complete */
                sleep((caddr_t)&tp->t_canq, TTIPRI);
            }
        if((!(tp->t_dstat&SUPBUF)) && (ppcpid[device] != CENTRONICS))
        {
            if (pb->dcb <= INIT_CB)
                pb->dcb = INIT_CB; /* give more cblocks for the
                    first port opened on a board */
            pb->dcb += CB_PER_PPC ; /* get more cblocks for the new port */
            tp->t_dstat |= SUPBUF;
        }

        /* call line discipline open */
        (*linesw[tp->t_line].l_open)(tp);
        spl0();
}

ppstart()
{
        int i, j;
        char *ptr;

        pp_bnbr = devcheck(D_PORTS, pp_addr);

        pumpflag = 0;

        for(i = 0; i < pp_bnbr; i++)
        {
            csbit[i] = 1;
            for(ptr = (char *)&pp_board[i]; ptr < ((char *)&pp_board[i] +
                      sizeof(pp_board[i])); ptr++)
        {
                *ptr = 0;
        }
        for(j = 0; j < 5; j++)
        {
            ppcbid[(i*5)+j] = i;
            ppcpid[(i*5)+j] = j;
        }
    }
}

nodrain(tp)
register struct tty *tp;
{
    tp->t_dstat &= ~CLDRAIN;
    wakeup((caddr_t)&tp->t_oflag);
}
ppclose(dev)
register dev_t dev;
{
    register struct tty *tp;
    int tout; /*time out parameter */
    extern nodrain();
    register dev_t device;


    device = minor(dev);

    /* check to see if this device was pump */

    if(pumpflag) {
        pumpflag = 0;
        return;
    }

    tp = &pp_tty[device];


    /* decrement counters for RECEIVE entry queue */
    splpp();
    if (ppcpid[device] != CENTRONICS)
    {
        if (tp->t_rbuf.c_ptr !=NULL)
        {
            pp_board[ppcbid[device]].qcb--;
            if (pp_board[ppcbid[device]].qcb < 0)
            {
                pp_board[ppcbid[device]].qcb=0;
            }
        }

        if(tp->t_dstat&SUPBUF)
        {
            pp_board[ppcbid[device]].dcb -= CB_PER_PPC; /* reduce cblocks */
            if (pp_board[ppcbid[device]].dcb <= INIT_CB)
                pp_board[ppcbid[device]].dcb -= INIT_CB;
            if (pp_board[ppcbid[device]].dcb <0)
            {
                pp_board[ppcbid[device]].dcb = 0;
            }

            tp->t_dstat &= ~SUPBUF;
        }
    }

    if ((tp->t_outq.c_cc) || (tp->t_state&BUSY))
    {
        if(tp->t_state&CARR_ON)
        {
                while ((tp->t_outq.c_cc) || (tp->t_state&BUSY))
            {
                tp->t_state |= TTIOW;
                tp->t_dstat |= CLDRAIN;
                tout=timeout(nodrain,tp,CL_TIME);
                sleep((caddr_t)&tp->t_oflag, PZERO);
                if (!(tp->t_dstat&CLDRAIN))
                {
                    tp->t_dstat |= CLDRAIN;
                    break;
                }
                else
                {
                    untimeout(tout);
                    tp->t_dstat &= ~CLDRAIN;
                }
            }
            if (tp->t_dstat&OPDRAIN)
            {
                tp->t_dstat &= ~OPDRAIN;
                wakeup((caddr_t)&tp->t_cc[1]);
            }
        }
        else
        {
            if(tp->t_state&BUSY)
                tp->t_state &= ~BUSY;
            lla_ldeuld(ppcbid[device], ppcpid[device]);
        }
    }

    spl0();
    (*linesw[tp->t_line].l_close)(tp);
    ppdis(tp,dev);
}

ppread(dev)
{
    register struct tty *tp;
    register dev_t device;

    device = minor(dev);
    tp = &pp_tty[device];
    (*linesw[tp->t_line].l_read)(tp);
}

ppwrite(dev)
{
    register struct tty *tp;
    register dev_t device;

    device = minor(dev);
    tp = &pp_tty[device];
    (*linesw[tp->t_line].l_write)(tp);
}

send_brk(tp,bid,pid,arg)
register struct tty *tp;
register short bid,pid;

{
    register sx;
    ttywait(tp);
    sx=splpp();
    while (!(lla_xfree(bid,pid)))
    {
        tp->t_dstat |= WENTRY;
        sleep((caddr_t)&tp->t_cc[1],TTOPRI);

    }
    tp->t_dstat |= WBREAK;
    ppc_break(bid,pid,arg);  /* This does not send a break
        to peripheral, it only drains the output */
    while (tp->t_dstat&WBREAK)
    {
        sleep((caddr_t)&tp->t_cc[2],TTOPRI);
    }
    splx(sx);
```

```
}
ppioctl(dev, cmd, arg, mode)
{
    register struct tty *tp;
    register dev_t device;

    device = minor(dev);

    switch (cmd)
    {
        case TCSBRK:
            tp = &pp_tty[device]; /* get tty structure */
            send_brk(tp,ppcbid[device],ppcpid[device],arg);
            break;
        case PUMP:
            if(pumpflag != 1)
            {
                u.u_error = EPERM;
                return;
            }
            pmctl(dev, cmd, arg, mode);
            break;
        default:
            if(ttiocom(&pp_tty[device], cmd, arg, mode))
                ppparm(dev);
            break;
    }
}

/*  device interrupt handler */

ppint(bid)
register short bid;
{
    register struct tty *tp;
    register short pid;
    CENTRY cqe;
    register char *cc_ptr;
    register struct ppcboard *pb;
        /*get completion queue entry */
        pb = &pp_board[bid];
    while (lla_cqueue(bid,&cqe) == PASS){
    if ((bid >= pp_bnbr) || (pb->b_state&SYSGFAIL))
        return;
        tp = TP(bid, cqe.common.codes.bits.subdev);
        /* determine the interrupt opcode */
        switch(cqe.common.codes.bytes.opcode) {
        case PPC_RECV:
            sysinfo.rcvint++;
        /* give returned cblock to t_buff */
        if
            ((tp->t_rbuf.c_ptr = (caddr_t)cqe.common.addr) ==
                NULL)
                break;

        if (cqe.appl.pc[0]&(RC_FLU|RC_DSR))
            /*c_count denotes number of unfilled characters
              in cblock */
            /* send an empty buffer to ttin */
            tp->t_rbuf.c_count = tp->t_rbuf.c_size;

        else
            tp->t_rbuf.c_count = tp->t_rbuf.c_size -
             cqe.common.codes.bytes.bytcnt
                 - 1;
        pb->qcb--; /* return cblock */
        if (cqe.appl.pc[0]&RC_BRK)
        {
            (*linesw[tp->t_line].l_input)(tp, L_BREAK);
            if (tp->t_state&CARR_ON)
                tp->t_state |= BUSY;
            ppc_device(bid, cqe.common.codes.bytes.subdev, DR_ABX);
            *tp->t_rbuf.c_ptr = 0;
            tp->t_rbuf.c_count = tp->t_rbuf.c_size - 1;
        }
        (*linesw[tp->t_line].l_input)(tp, L_BUF);
        /* supply buffers */
        ppproc(tp,T_INPUT);
        break;
    case PPC_XMIT:
        sysinfo.xmtint++;

        if ((cc_ptr= (char *)cqe.common.addr)!=NULL)
            putcf(CMATCH((struct cblock *)cc_ptr));

        if (tp->t_dstat&WENTRY)
        {
            tp->t_dstat &= ~(SETOPT|WENTRY);
            wakeup((caddr_t)&tp->t_cc[1]);
        }

        tp->t_state &= ~BUSY;
        ppproc(tp,T_OUTPUT);

        break;
    case PPC_ASYNC:
        switch(cqe.appl.pc[0]) {
        case AC_BRK:
                sysinfo.rcvint++;
                (*linesw[tp->t_line].l_input)(tp, L_BREAK);
                if (tp->t_state&CARR_ON)
                    tp->t_state |= BUSY;
                ppc_device(bid, cqe.common.codes.bytes.subdev, DR_ABX);
                break;
            case AC_DIS:
                sysinfo.mdmint++;
                tp->t_state &= ~CARR_ON;
                signal(tp->t_pgrp, SIGHUP);
                ttyflush(tp, (FREAD|FWRITE));
                break;
            case AC_CON:
                sysinfo.mdmint++;
                tp->t_state |= CARR_ON;
                if(tp->t_state&WOPEN) {
                    tp->t_state &= ~WOPEN;
                    wakeup((caddr_t)&tp->t_canq);
                }
                break;
            case AC_FLU:
                /* all output cblocks given to
                   request queue have been taken
                   by the ppc have been flushed */


                if (tp->t_dstat&WENTRY)
                {
                    tp->t_dstat &= ~(SETOPT|WENTRY);
                    wakeup((caddr_t)&tp->t_cc[1]);
                }
                tp->t_state &= ~BUSY;
                ppproc(tp,T_OUTPUT); /* resume output
                        processing */
                break;
            }
            break;
        case PPC_OPTIONS:
            /* return cblock to free list */
            if ((cc_ptr=(char *)cqe.common.addr) != NULL)
                putcf(CMATCH((struct cblock *)cc_ptr));
            if (tp->t_dstat&WENTRY)
            {
                tp->t_dstat &= ~(SETOPT|WENTRY);
                wakeup((caddr_t)&tp->t_cc[1]);
            }

            tp->t_state &= ~BUSY;
            ppproc(tp,T_OUTPUT);
```

```
    break;
case PPC_DISC:
case PPC_CONN:
    if (tp->t_dstat&WENTRY)
    {
        tp->t_dstat &= ~(SETOPT|WENTRY);
        wakeup((caddr_t)&tp->t_cc[1]);
    }

    tp->t_state &= ~BUSY;
    ppproc(tp,T_OUTPUT);
    break;

case PPC_DEVICE:
    break;
case PPC_BRK:
    tp->t_dstat &= ~WBREAK;
    wakeup((caddr_t)&tp->t_cc[2]);
    break;
case SYSGEN:

    pb->b_state |= ISSYSGEN;
    wakeup((caddr_t)pb);
    break;
case NORMAL:
case FAULT:
case QFAULT:
    if (E_OPCODE(cqe)==QFAULT)
        cmn_err(CE_WARN, "PORTS: QFAULT - opcode= %d, board = %d, \n,
                subdev = %d, bytecnt = %d, buff address = %x, \n\n",
                E_APPL(cqe,0), bid, cqe.common.codes.bytes.subdev,
                E_BYTCNT(cqe),E_ADDR(cqe));

    if (E_OPCODE(cqe)==FAULT)
        cmn_err(CE_WARN, "PORTS: FAULT - opcode= %d, board = %d, \n,
                subdev = %d, bytecnt = %d, buff address = %x, \n\n",
                E_APPL(cqe,0), bid, cqe.common.codes.bytes.subdev,
                E_BYTCNT(cqe),E_ADDR(cqe));
    if (!(pb->b_state&CIOTYPE))
        break;
    pb->b_state &= ~CIOTYPE;
    pb->retcode = cqe.common.codes.bytes.opcode;
    wakeup((caddr_t)&pb->qcb);
    break;
default:
    cmn_err(CE_WARN, "PORTS: unknown completion code: %d\n",
            cqe.common.codes.bytes.opcode);
    break;
}
```

```
    }
}

ppproc(tp, cmd)
register struct tty *tp;
{
    register short bid,pid;
    register sx;
    register struct cblock *cb_ptr;

    sx= tp - pp_tty; /* find index of pp_tty[] */
    bid = sx/5;
    pid = sx - bid*5;

    switch(cmd) {
    case T_WFLUSH:
        if (!(tp->t_state&CARR_ON))
        {
            sx = splpp();
            ppc_device(bid,pid,DR_ABX);
            splx(sx);
        }
        else
            break;
    case T_RESUME:
        ppc_device(bid,pid,DR_RES);
    case T_OUTPUT:
        sx = splpp();

        if(tp->t_state&BUSY)
        {
            splx(sx);
            break;
        }

        if(!(lla_xfree(bid,pid)))
        {
            splx(sx);
            break;
        }

        if(!(CPRES&(*linesw[tp->t_line].l_output)(tp))) {
            splx(sx);
            break;
        }
        else
        {
            ppc_xmit(bid,pid,tp->t_tbuf.c_ptr, tp->t_tbuf.c_count - 1);
            tp->t_tbuf.c_ptr = NULL;
        }

        tp->t_state |= BUSY;
        splx(sx);
        break;

    case T_RFLUSH:

        ppc_device(bid,pid,DR_ABR);
        if(!(tp->t_state&TBLOCK))
            break;
    case T_UNBLOCK:
        tp->t_state &= ~TBLOCK;
        ppc_device(bid,pid,DR_UNB);
        break;
    case T_INPUT:
        if (pid == CENTRONICS)
            break;
        sx=splpp();
        if(tp->t_rbuf.c_ptr != NULL)
        {
            register struct ppcboard *pb;
            register char *cptr;
            pb = &pp_board[bid];

            pb->qcb++; /* get cblock from ttopen or ttin*/
            if (pb->dcb >= pb->qcb)
            {
                ppc_recv(bid,tp->t_rbuf.c_ptr);
                tp->t_rbuf.c_ptr=NULL;
                    /* add more cblocks if you can get from
                            the free list */
                while (pb->dcb > pb->qcb)
                {
                        if ((cptr = getcf()->c_data)
                       == ((struct cblock *)NULL)->c_data)
                            break;
                            ppc_recv(bid,cptr);
                    pb->qcb++;
                }
            }
            else /* too many cblocks */
            {
                /* return cblock to free list */
                putcf(CMATCH((struct cblock *)tp->t_rbuf.c_ptr));
                tp->t_rbuf.c_ptr=NULL;
                lla_attn(bid);
                pb->qcb--;
            }
        }
        splx(sx);
        break;
    case T_SUSPEND:
        ppc_device(bid,pid,DR_SUS);
        break;
        case T_BREAK:

        /*send_brk(tp,bid,pid,0);*/
        break;
    case T_BLOCK:
        tp->t_state |= TBLOCK;

        ppc_device(bid,pid,DR_BLK);
        break;
        case T_PARM:
        ppparm((((bid<<4)|pid)&0xFF));
        break;
    }
}
ppparm(dev)
register dev;
{
    register struct tty *tp;
    register  xflag;
    register Options *opt;
    struct cblock *cp;
    register short bid,pid;
    register dev_t device;

    device = minor(dev);

/*
 * THIS IS a test to exclude the driver from setting the
 *  ppc parameters, these parameters are set by the ppc
 */

/*
return;
*/

    tp = TP(ppcbid[device],ppcpid[device]);
    if((tp->t_cflag&CBAUD) == 0) {
        ppc_device(ppcbid[device],ppcpid[device],DR_DIS);
        return;
    }
    splpp();
    while (!(lla_xfree(ppcbid[device],ppcpid[device])))
    {
            tp->t_dstat |= (SETOPT|WENTRY);
            sleep((caddr_t)&tp->t_cc[1],TTOPRI);
        }
        if((cp = getcf()) == NULL) {
            u.u_error = EIO;
            spl0();
            return;
        }
        opt = (Options *) cp->c_data;
        opt->line = 0; /* line discipline 0 */
        opt->ld.zero.iflag = tp->t_iflag;
        opt->ld.zero.oflag = tp->t_oflag;
        opt->ld.zero.cflag = tp->t_cflag;
        /* convert baud rate to duart register specification */
        opt->ld.zero.cflag &= ~CBAUD; /* zero baud bits */
        opt->ld.zero.cflag |= ppc_speeds[tp->t_cflag&CBAUD]&0xF;
        if (((opt->ld.zero.cflag)&0xF) == 0)
        {
            putcf(cp);
            u.u_error = EIO;
            spl0();
            return;
        }


        if (ppcpid[device] == CENTRONICS)
            opt->ld.zero.cflag &= ~CREAD;
        xflag = 0;

        xflag |= X_ITIME;
        if((tp->t_cflag&CBAUD) <= B300)
            opt->ld.zero.itime = pptime[tp->t_cflag&CBAUD];
        else
            opt->ld.zero.itime = ITIME;

        opt->ld.zero.lflag = xflag;

        /* send options to the ppc */

        ppc_option(ppcbid[device],ppcpid[device],(char *)opt,sizeof(Options));
        spl0();
    }

    badboard(bid)
    register bid; /* board number*/
    {
        /* sysgen did not work */
        wakeup((caddr_t)&pp_board[bid]);
    }
```

```
/* pprst is called by the pump routine to reset and sysgen the
   dumb firmware on the ports board */
int pprst(bid)
register int bid;
{
    register int errors;

    if( (pp_bnbr = devcheck(D_PORTS, pp_addr)) <= 0 ) {

        if (bid < pp_bnbr)
            pp_board[bid].b_state = SYSGFAIL;
        return( PU_DEVCH);
    }

    if ((bid + 1) > pp_bnbr)
        return( PU_DEVCH);



    errors = 0;
    lla_reset(bid);

    splpp();
    pp_board[bid].b_state &= ~(SYSGFAIL|ISSYSGEN);
    spl0();
    if( ppsysgen(bid) != PASS ) {
        cmn_err(CE_WARN, "PORTS: SYSGEN failure on board %d\n", bid);
        errors++;
    }

    return( (errors == 0) ? PASS : FAIL );
}


int ppsysgen(bid)
register short bid; /* board number */

{
    extern badboard();
    register struct tty *tp;
    register struct ppcboard *brd_ptr; /* ptr to board structure */
    register tid; /* timer id */
    register short pid; /* port id number */
    /* initialize qentries  !!!!!! */
    brd_ptr = &pp_board[bid];

    splpp();
    brd_ptr->b_state  = 0;
    sg_dblk.request   = (long)&R_QUEUE(bid);
    sg_dblk.complt    = (long)&C_QUEUE(bid);
    sg_dblk.req_size  = RQSIZE;
    sg_dblk.comp_size = CQSIZE;
    sg_dblk.no_rque   = NUM_QUEUES;


    /* try to sysgen the board */
    if (lla_sysgen(bid,&sg_dblk) == PASS)
    {
        tid=timeout(badboard,bid,SYSG_TIME);
        sleep(brd_ptr,PZERO);
        if (brd_ptr->b_state&ISSYSGEN)
        {
            untimeout(tid);
        }
        else
        {
            brd_ptr->b_state = SYSGFAIL;
            spl0();
            return(FAIL);
        }
    }
    else
    {
        brd_ptr->b_state = SYSGFAIL;
        spl0();
        return(FAIL);
    }
    /* set ppc proc routine */
    for (pid=0; pid<=4 ;pid++)
    {
        tp = TP(bid,pid);
        tp->t_proc=ppproc;
        tp->t_state = EXTPROC;
    }
    spl0();
    return(PASS);
}
/*wake up after timeout on disconnect */


ppdis(tp,dev)
register struct tty *tp;
register int dev;

{
    int eflush;
    char dcode; /*disconnect code */
    register dev_t device;
    device = minor(dev);

    splpp();
    while(!(lla_xfree(ppcbid[device],ppcpid[device])))
    {
        tp->t_dstat |= WENTRY;
        sleep(&tp->t_cc[1],TTOPRI);

    }

    /* calculate the number of CBLOCKS to be returned to thesystem */
    if ((eflush=pp_board[ppcbid[device]].qcb - pp_board[ppcbid[device]].dcb) < 0)
        eflush=0;
    tp->t_state &= ~CARR_ON;
    if(tp->t_cflag&HUPCL) /* hang up (disconnect) if HUPCL is set */
        dcode = (GR_DTR|GR_CREAD);
    else
        dcode=GR_CREAD;
    ppc_disc(ppcbid[device],ppcpid[device],(char)eflush,dcode); /* disconnect and
                                                                   flush ppc */
    spl0();
}
cio_time(tb)
register struct ppcboard *tb;
{
    wakeup((caddr_t)&tb->qcb);
}
ppdld(bid,mda,pda,mds)
short bid;          /* board id */
char *mda;          /* mainstore data address */
char *pda;          /* ppc ram address */
short mds;          /* mainstore byte count */
{
    extern cio_time();
    register struct ppcboard *tb;
    register tid;       /* timeout id */
    tb = &pp_board[bid];
    if (tb->b_state&CIOTYPE)
        return(PU_OTHER); /* there is a CIO type command already in process */
    splpp();
    tb->b_state |= CIOTYPE;
    if (lla_dlm(bid,mda,pda,mds) == FAIL)
    {
        tb->b_state &= ~ CIOTYPE;
        spl0();
        return (PU_OTHER);
    }
    tid= timeout(cio_time,tb,CIO_TIME);
    sleep((caddr_t)&tb->qcb,PZERO);
```

```
        if (!(tb->b_state&CIOTYPE))
        {
            untimeout(tid);
            spl0();
        }
        else
        {
            tb->b_state &= ~ CIOTYPE;
            spl0();
            return(PU_TIMEOUT);
        }
        return (tb->retcode);
    }
    ppfcf(bid,pda)
    short bid;          /* board id */
    char *pda;          /* ppc ram address */
    {
        extern cio_time();
        register struct ppcboard *tb;
        register tid;       /* timeout id */
        tb = &pp_board[bid];
        if (tb->b_state&CIOTYPE)
            return(PU_OTHER); /* there is a CIO type command already in process */
        splpp();
        tb->b_state |= CIOTYPE;
        if (lla_fcf(bid,pda) == FAIL)
        {
            tb->b_state &= ~ CIOTYPE;
            spl0();
            return (PU_OTHER);
        }
        tid= timeout(cio_time,tb,CIO_TIME);
        sleep((caddr_t)&tb->qcb,PZERO);
        if (!(tb->b_state&CIOTYPE))
        {
            untimeout(tid);
            spl0();
        }
        else
        {
            tb->b_state &= ~ CIOTYPE;
            spl0();
            return(PU_TIMEOUT);
        }
        return (tb->retcode);
    }
    ppdos(bid)
    short bid;      /* board id */
    {
```

```
    register struct ppcboard *tb;
    tb = &pp_board[bid];
    if (tb->b_state&CIOTYPE)
        return(-1); /* there is a CIO type command already in process */
    splpp();
    tb->b_state |= CIOTYPE;
    if (lla_dos(bid) == FAIL)
    {
        tb->b_state &= ~ CIOTYPE;
        spl0();
        return (-2);
    }
    while (tb->b_state&CIOTYPE)
        sleep((caddr_t)&tb->qcb,PZERO);
    spl0();
    return (tb->retcode);
}
ppdsd(bid,mda)
short bid;          /* board id */
char         *mda;      /* mainstore data address*/
{
    register struct ppcboard *tb;
    tb = &pp_board[bid];
    if (tb->b_state&CIOTYPE)
        return(-1); /* there is a CIO type command already in process */
    splpp();
    tb->b_state |= CIOTYPE;
    if (lla_dsd(bid,mda) == FAIL)
    {
        tb->b_state &= ~ CIOTYPE;
        spl0();
        return (-2);
    }
    /* wait for completion report */
    while (tb->b_state&CIOTYPE)
        sleep((caddr_t)&tb->qcb,PZERO);
    spl0();
    return (tb->retcode);
}
pppump(pmpr)
register struct pump_st *pmpr;
{
    register slices; /* number of of 256 (PU_BLOCK) byte memory slices */
    register  char *usr_addr; /* user address pointer pointing to the
        user memort to be  moved to kernel space*/

    register i;
    register char *ppc_addr; /* ports address for download */
    long bsize; /* block size to be given to the PPC */
long rem_size; /* (buffer size)% 256 */
long rtcod;
register dev_t device;

device = minor(pmpr->dev);

switch(pmpr->cmdcode)
{
case PU_DLD: /* download case */
    slices = pmpr->size/PU_BLOCK;
    if ((rem_size = pmpr->size%PU_BLOCK) != 0)
        slices++;
    if (ppcbid[device] >= pp_bnbr)
    {
        u.u_error = ENXIO;
        pmpr->retcode = PU_OTHER;
        return;
    }
    ppc_addr = (char *)pmpr->to_addr;
    usr_addr = (char *)pmpr->bufaddr;
    for (i = 1; i <= slices; i++)
    {
        if ((i == slices) && (rem_size != 0))
            bsize = rem_size;
        else
            bsize = PU_BLOCK;
        if (copyin(usr_addr,scratch,bsize))
        {
            u.u_error = EFAULT;
            pmpr->retcode = PU_OTHER;
            return;
        }
        if ((pmpr->retcode =
            ppdld(ppcbid[device],scratch,ppc_addr,bsize))
                != NORMAL)
        {
            return ;
        }

        usr_addr += PU_BLOCK;
        ppc_addr += PU_BLOCK;
    }
    break;

case PU_RST:
    if ((rtcod = pprst(ppcbid[device])) == PASS)
        break;
    else
    {
            if (rtcod == PU_DEVCH)
                pmpr->retcode = PU_DEVCH;
            else
                pmpr->retcode = PU_OTHER;
            return;
        }

    case PU_FCF:
        if ((pmpr->retcode = ppfcf(ppcbid[device],pmpr->to_addr))
            != NORMAL)
            return;
        break;
    case  PU_SYSGEN:
        splpp();
        pp_board[ppcbid[device]].b_state &= ~(SYSGFAIL|ISSYSGEN);
        spl0();

        for (i = 1; i <= 4000; i++);
        if (ppsysgen(ppcbid[device]) == PASS)
        {
            pmpr->retcode = SYSGEN;
            return;
        }
        else
        {
            pmpr->retcode = PU_OTHER;
            return;
        }

    default:
        pmpr->retcode = PU_OTHER;
        cmn_err(CE_WARN, "PORTS: Unknown pump command: %d\n", pmpr->cmdcode);
        return;
    }
    pmpr->retcode = NORMAL;
    return;
}


ppclr()
{
    register int bid, pid;
    register struct tty *tp;

/*
 * The systm has detected a power failure, and is about to go down:
 *
 *  1. Send a special notice to the firmware
 *  2. Mark all boards as down, so as to fail any further attempts
 *      to reference them
 *   3. Wake up any processes sleeping very deeply
 */

    for( bid = 0; bid < pp_bnbr; bid++ ) {
        ppc_clr(bid);

        pp_board[bid].b_state = SYSGFAIL;

        wakeup(&pp_board[bid]);
        for( pid = 0; pid < 5; pid++ ) {
            tp = TP(bid,pid);

            wakeup((caddr_t)&tp->t_dstat);
            wakeup((caddr_t)&pp_board[bid].qcb);
            }
        }

    return;
}

pmctl(dev, cmd, val, mode)
{
    struct pump_st pump;

    if(copyin((char *) val, (char *) &pump, sizeof(struct pump_st)))
    {
        u.u_error = EFAULT;
        return;
    }

    pppump(&pump);

    if(copyout((char *) &pump, (char *) val, sizeof(struct pump_st)))
    {
        u.u_error = EFAULT;
        return;
    }
    return;
}
```

APPENDIX K: 3B5 COMPUTER BLOCK DEVICE DRIVER

This is the disk driver used on the 3B5 Computer.

```
static char Sccsid[]="@(#)idfc.c3.1.1.5";

/*
 * The 3B-5 IDFC disk driver will control up to 8 IDFC controllers
 * simultaneously. The driver will maintain status on each controller
 * and the disk drives interfaced to the controller.
 */

#define GENERIC10   0x100   /* value in edt table for GENERIC 1.0 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/proc.h"
#include "sys/sysmacros.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/iobuf.h"
#include "sys/systm.h"
#include "sys/cc.h"

#include "sys/sysgdb.h"
#include "sys/idfc.h"
#include "sys/dfdrv.h"
#include "sys/firmware.h"
#include "sys/io.h"
#include "sys/edt.h"

extern struct mmutabccmmudesc;  /* MMU descriptor RAM */
extern struct edts  ccedts;     /* Equipped Device Table */
extern struct mmusegkmmudesc[]; /* Descriptors for kernel virtual address space */

extern paddr_t  vtop(); /* convert virtual address to physical */
long        gcksum();   /* gcksum to return a long value */

/*
```

```
 * Convert a virtual address WITHIN THE DF_DFC STRUCTURE to a physical address
 */
#define df_vtop(x)  ((long)x + df_offset)

/*
 * TEST = 1 for debugging printf's
 */
#define TEST    0
/*
 * PERFON = 1 for minimum requirements of 5.0
 */
#define PERFON      1
/*
 * FULLPERF = 1 for full performance monitoring
 */
#define FULLPERF0

#define SET     1   /* bit set state = 1 */
#define FAIL    -1  /* Error return code */
#define NJRQ    2   /* Number of Job Request Queues */
#define NJCQ    1   /* Number of Job Completion Queues */
#define DELAYMAX1000/* IDFC operation delay count */
#define DFC_IPL 10  /* Interrupt priority level of the disk */
#define ioipio_s1   /* IO in progress flag for stray interrupt check */



extern int      df_cnt;     /* number of IDFC controllers in system */
extern struct mmusegdf_addr[];  /* seg. descriptors for address xlate */
extern struct dfc   df_dfc[];   /* controller data area jrq,status etc */

static long df_offset; /* place to keep the difference between virt and phys
                          for df_dfc[] */
static paddr_t pkmmudesc; /* physical address of kmmudesc[] array */

#if TEST
int initpr, perfpr, intpr, stratpr; /* print flags */
#endif





/* SYSGEN data block for driver/controller communication protocol */

struct  sgdb
{
    struct  sgcom   sdbc;   /* sysgen db header */
    struct  sgjqd   jrqd0;  /* first request Q descriptor */
    struct  sgjqd   jrqd1;  /* second request Q descriptor */
    struct  sgjqd   jcqd0;  /* job completion queue desc. */
    longint sgcksum;/* sysgen db checksum */
};

struct  sgdbsdb;/* alloc for sysgen data block */
union   sa_cmd  lsacb;  /* LOCAL STAND ALONE COMMAND BLOCK FIX!! */



/*
 * The dfinit routine is called during OS initialization and is responsible
 * for performing SYSGEN operations on all IDFC controllers configured
 * on the system. This routine will also invoke periodic performance data
 * reporting from each of the initialized IDFC controllers.
 */
dfinit ()
{
    register struct dfc *dfcp;  /* IDFC controller pointer */
    register union sa_cmd *lsacbp;  /* IDFC SA cmd buf pointer */
    register union  jrqe *jrq0ldp;  /* load pointer to job request Q of SMD0 */
    register struct sgdb *sgdbp;/* sysgen data block pointer */

    unsigned short *tsacbp;     /* TEMP STAND ALONE COMMAND BUFFER */
    unsigned short *sacbp;      /* TEMP IDFC SACBP */

    struct idfc_wcsr *wcsrp;/* write access pointer to IDFC CSR */
    struct idfc_rcsr *rcsrp;/* read access pointer to IDFC CSR */
    struct pir16*pirp;      /* write access pointer to IDFC PIR */
    struct buf  jidb;       /* sysgen jid address */


    int delay;      /* operation delay counter */
    int delcnt;     /* intermediate delay counter */
    int i;      /* controller initialization counter */
    int cnt;    /* transfer counter */
    int j, k;       /* loop counters */
    longequip;      /* unit_equip from edt */
    longlb_baddr;   /* unit local bus base address */



    /* compute offset between virtual and physical address for df_dfc[] */
    df_offset = (long)vtop(df_dfc) - (long)df_dfc ;

    /* get physical address of kmmudesc[] array */
    pkmmudesc = vtop( kmmudesc );

    /* controller initialization */
    sgdbp = &sdb;

    for(i=0; i<df_cnt; i++) /* init all controllers spec'd in config */
    {
        /* controller initialization per IDFC */
        wcsrp = (struct idfc_wcsr *) (BIOADDR|OCSR);
        rcsrp = (struct idfc_rcsr *) wcsrp;
        lsacbp = (union sa_cmd *) &lsacb;
        tsacbp = (unsigned short *) &lsacb;
        sacbp = (unsigned short *) (BIOADDR|OSACB);
        pirp = (struct pir16 *) (BIOADDR|OPIR);
        /* Init IDFC register pointers */

        baseio(df_addr[i]);
        lb_baddr = (long)ctob(df_addr[i].base); /* compute local bus address */

        /* Initialize unit Queue and misc pointers */
        dfcp = &df_dfc[i];  /* init IDFC controller pointer */
#if PERFON
        for(k = 0;k < NDRV;k++){
            dfcp->df_stat[k].pttrack = &dfcp->df_stat[k].ptrackq[0];
            dfcp->df_stat[k].endptrack = &dfcp->df_stat[k].ptrackq[NTRACK];
        }
#endif
        /* Initialize pointer to sub unit equippage, */
        /* number of drives and list of drive numbers for
        performance gathering */
        for(k = 0,j = 0;j < ccedts.number;j++){
            if(strcmp(ccedts.edtx[j].dev_name,"IDFC") != 0) continue;
            if(ccedts.edtx[j].lb_baddr != lb_baddr) continue;
            equip = ccedts.edtx[j].unit_equip;
            if(ccedts.edtx[j].version == GENERIC10){
                /* This if is a temporary change for driver to work with
                   1.0 hardware */
                equip = D_DRV00 + D_DRV01 + D_DRV10 + D_DRV11 + D_SMD0 + D_SMD1;
            }

            if((equip & D_SMD0) && (equip & D_DRV00))
            {
                dfcp->df_part[0] = dskptbl[(equip>>4) & D_TYPE];
                dfcp->df_perf[k].smdnum = 0;
                dfcp->df_perf[k++].dsknum = 0;
                dfcp->numdrv++;
            }
            else dfcp->df_part[0] = NULL;
            if((equip & D_SMD0) && (equip & D_DRV01))
            {
                dfcp->df_part[1] = dskptbl[(equip>>6) & D_TYPE];
                dfcp->df_perf[k].smdnum = 0;
            dfcp->df_perf[k++].dsknum = 1;
            dfcp->numdrv++;
        }
        else dfcp->df_part[1] = NULL;
        if((equip & D_SMD1) && (equip & D_DRV10))
        {
            dfcp->df_part[2] = dskptbl[(equip>>12) & D_TYPE];
            dfcp->df_perf[k].smdnum = 1;
            dfcp->df_perf[k++].dsknum = 0;
            dfcp->numdrv++;
        }
        else dfcp->df_part[2] = NULL;
        if((equip & D_SMD1) && (equip & D_DRV11))
        {
            dfcp->df_part[3] = dskptbl[(equip>>14) & D_TYPE];
            dfcp->df_perf[k].smdnum = 1;
            dfcp->df_perf[k++].dsknum = 1;
            dfcp->numdrv++;
        }
        else dfcp->df_part[3] = NULL;
        break;
    }

    /* initialize the dfc structure for this idfc */

    dfcp->smd_jrqa[0].vrqldp = dfcp->smd_jrqa[0].df_jrq;
    dfcp->smd_jrqa[1].vrqldp = dfcp->smd_jrqa[1].df_jrq;
    dfcp->vcquldp = dfcp->df_jcq;
    dfcp->smd_jrqa[0].jrqldp = (union jrqe *)df_vtop(dfcp->smd_jrqa[0].df_jrq);
    dfcp->smd_jrqa[0].jrquldp = (union jrqe *)df_vtop(dfcp->smd_jrqa[0].df_jrq);
    dfcp->smd_jrqa[1].jrqldp = (union jrqe *)df_vtop(dfcp->smd_jrqa[1].df_jrq);
    dfcp->smd_jrqa[1].jrquldp = (union jrqe *)df_vtop(dfcp->smd_jrqa[1].df_jrq);
    dfcp->jcqldp = (struct df_jcqe *)df_vtop(dfcp->df_jcq);
    dfcp->jcquldp = (struct df_jcqe *)df_vtop(dfcp->df_jcq);
    dfcp->dfutab.io_nreg = 0; /* number of regs to log on error */
    dfcp->dfutab.jrqsleep = 0; /* init sleep counter on JRQ */
    dfcp->dfutab.io_addr = vtop(BIOADDR); /* local bus address */
    dfcp->dfutab.io_start = lbolt; /* time IDFC initialized */
    dfcp->dfutab.b_actf = NULL;
    dfcp->dfutab.b_actl = NULL;
    dfcp->dfutab.qcnt = NULL;
/*******dfcp->dfutab.io_stp->io_misc += 1; /* inc operations counter */



    /* initialize the content of the IDFC sysgen data block */

    sgdbp -> sdbc.sgjid = (long)&jidb;
    sgdbp -> sdbc.njcq = NJCQ;
```

```
        sgdbp -> sdbc.njrq = NJRQ;
        sgdbp -> sdbc.sgopc = D_VSYSG;
        sgdbp -> sdbc.sdbsize = (sizeof(sdb)/4);
        sgdbp -> jrqd0.jqsa = (paddr_t *) df_vtop(dfcp->smd_jrqa[0].df_jrq);
        sgdbp -> jrqd0.jqldp = (paddr_t *) df_vtop(&dfcp->smd_jrqa[0].jrqldp);
        sgdbp -> jrqd0.jquldp = (paddr_t *) df_vtop(&dfcp->smd_jrqa[0].jrquldp);
        sgdbp -> jrqd0.jqsize = (sizeof(dfcp->smd_jrqa[0].df_jrq)/4);
        sgdbp -> jrqd1.jqsa = (paddr_t *) df_vtop(dfcp->smd_jrqa[1].df_jrq);
        sgdbp -> jrqd1.jqldp = (paddr_t *) df_vtop(&dfcp->smd_jrqa[1].jrqldp);
        sgdbp -> jrqd1.jquldp = (paddr_t *) df_vtop(&dfcp->smd_jrqa[1].jrquldp);
        sgdbp -> jrqd1.jqsize = (sizeof(dfcp->smd_jrqa[1].df_jrq)/4);
        sgdbp -> jcqd0.jqsa = (paddr_t *) df_vtop(dfcp->df_jcq);
        sgdbp -> jcqd0.jqldp = (paddr_t *) df_vtop(&dfcp->jcqldp);
        sgdbp -> jcqd0.jquldp = (paddr_t *) df_vtop(&dfcp->jcquldp);
        sgdbp -> jcqd0.jqsize = (sizeof(dfcp->df_jcq)/4);
        sgdbp -> sgcksum = gcksum((long *) &sdb,sizeof(sdb)/sizeof(long));



#if TEST
        if(initpr) {
            /* print content of sysgen data block */
            printf("IDFC %x SYSGEN data block\n",i);
            printf("jid = %x\n",sgdbp->sdbc.sgjid);
            printf("njcq= %x\n",sgdbp->sdbc.njcq);
            printf("njrq= %x\n",sgdbp->sdbc.njrq);
            printf("sgopc   = %x\n",sgdbp->sdbc.sgopc);
            printf("sdbsize = %x\n",sgdbp->sdbc.sdbsize);
            printf("jrq0sa  = %x\n",sgdbp->jrqd0.jqsa);
            printf("jrq0ldp = %x\n",sgdbp->jrqd0.jqldp);
            printf("jrq0uldp= %x\n",sgdbp->jrqd0.jquldp);
            printf("jrq0size= %x\n",sgdbp->jrqd0.jqsize);
            printf("jrq1sa  = %x\n",sgdbp->jrqd1.jqsa);
            printf("jrq1ldp = %x\n",sgdbp->jrqd1.jqldp);
            printf("jrq1uldp= %x\n",sgdbp->jrqd1.jquldp);
            printf("jrq1size= %x\n",sgdbp->jrqd1.jqsize);
            printf("jcq0sa  = %x\n",sgdbp->jcqd0.jqsa);
            printf("jcq0ldp = %x\n",sgdbp->jcqd0.jqldp);
            printf("jcq0uldp= %x\n",sgdbp->jcqd0.jquldp);
            printf("jcq0size= %x\n",sgdbp->jcqd0.jqsize);
        }
#endif


        /* Start initialization of IDFC controller */

        wcsrp->req_reset = SET; /* send reset request to IDFC */
        for(delay = 0; delay < 1000; delay++)   /* allow CSR to be cleared */
            ;
```

```
delay = 0;
while ((rcsrp->rcsr3 & RESET_COMPL) != SET) /* wait for reset
                                complete in IDFC */
{
    if(delay < DELAYMAX)
    {
        for(delcnt = -512; delcnt != 0; delcnt++)
        {
            if((rcsrp->rcsr3 & RESET_COMPL) == SET)
                break;
        }
        delay++;
    }
    else
    {
        printf("IDFC %d FAILS RESET TIMEOUT\n", i );
        dfcp->dfutab.b_flags |= B_TIME; /* set timeout flag */
        break;
    }
}
if(dfcp->dfutab.b_flags & B_TIME) /* check for reset timeout */
{
    dfcp->dfutab.b_flags &= ~B_TIME; /* clear flag */
    break;
}



/* send sysgen command to IDFC */

lsacbp->sysgen.sysgdp = (char *)vtop(&sdb); /* load sysgen db pointer
                                                 in IDFC */
lsacbp->sysgen.cmdcode = D_VSYSG;   /* load sysgen opcode */

for(cnt = 0; cnt < 4; cnt++)/* copy command buffer to disk */
    *sacbp++ = *tsacbp++;

dfcp->dfutab.sgreq = SET;   /* set completion wait flag */
pirp->pir01 = SET;  /* set IDFC SYSGEN request pir */
delay = 0;      /* reset delay counter */
while(dfcp->dfutab.sgreq == SET)/* wait for sysgen to complete */
{
    if(delay < DELAYMAX)
    {
        for(delcnt = -512; delcnt != 0; delcnt++)
        {
            if(dfcp->dfutab.sgreq != SET)
                break;
        }
```

```
                delay++;
            }
            else
            {
                dfcp->dfutab.sgreq = FAIL; /* set SYSGEN fail flag */
                break;
            }
        }

        if(dfcp->dfutab.sgreq == FAIL) /* check for valid sysgen */
        {
            printf("IDFC %d FAILS SYSGEN!\n", i );
            break;
        }
#if PERFON
        /* Start performance reporting on current controller */

        /* init performance request in controller job request queue */
        jrq0ldp = dfcp->smd_jrqa[0].vrqldp; /* init driver queue pointer */
        jrq0ldp->preq.jid = (struct buf *)&jidb;
        jrq0ldp->preq.jcqid = 0;
        jrq0ldp->preq.opc = D_PRFON;
        jrq0ldp->preq.smdnum = 0;
        jrq0ldp->preq.dsknum = 0;
        jrq0ldp->preq.sp1 = 0;
        jrq0ldp->preq.pmta = pkmmudesc;
        jrq0ldp->preq.sma = (paddr_t)&dfcp->df_perf[0];
        jrq0ldp->preq.numdrv = dfcp->numdrv;     /* number of drives */
        jrq0ldp->preq.rptint = 0x00007530L; /* 30 second report interval */
        jrq0ldp->preq.cksum = gcksum((long *) jrq0ldp,
                              sizeof(struct df_jrqe)/sizeof(long));

#if TEST
        if(perfpr) {
            /* print content of performance request */
            printf("IDFC %x PERFORMANCE REQUEST ENTRY \n",i);
            printf("jid = %x\n",jrq0ldp->preq.jid);
            printf("jcqid   = %x\n",jrq0ldp->preq.jcqid);
            printf("opc = %x\n",jrq0ldp->preq.opc);
            printf("smdnum  = %x\n",jrq0ldp->preq.smdnum);
            printf("dsknum  = %x\n",jrq0ldp->preq.dsknum);
            printf("sp1 = %x\n",jrq0ldp->preq.sp1);
            printf("pmta    = %x\n",jrq0ldp->preq.pmta);
            printf("sma = %x\n",jrq0ldp->preq.sma);
            printf("numdrv  = %x\n",jrq0ldp->preq.numdrv);
            printf("rptint  = %x\n",jrq0ldp->preq.rptint);
            printf("cksum   = %x\n",jrq0ldp->preq.cksum);
        }
#endif
```

```
        jrq0ldp++;  /* update request queue load pointer */
        dfcp->smd_jrqa[0].vrqldp = jrq0ldp; /* save update value in unit area */
        dfcp->smd_jrqa[0].jrqldp = (union jrqe *)df_vtop(jrq0ldp);

        pirp->pir04 = SET;  /* send job pending PIR to IDFC */
#endif
    }
    clearbaseio;/* clear mmu */
}



/*
 * This routine will be entered on receiving
 * an operational interrupt from the IDFC. It will
 * check for proper job completion of the SYSGEN,
 * spurious interrups, failing job status, and copy
 * performance data into the drive status structures.
 */
dfint(unit)
int unit;   /* Unit ID of interrupting IDFC */
{

    register struct dfc *dfcp = &df_dfc[unit]; /* IDFC unit pointer */
    register struct df_jcqe *jcquldp;
    register struct buf *bp; /* pointer to user buffer header */
    register int drv;   /* drive number */
    register struct buf *fp; /* forward buffer pointer */
    register struct buf *rp; /* reverse buffer pointer */
    static struct mmuseg nullseg ={0,0,0,0,0,0}; /* Null segment descriptor word */

    int i;      /* loop counter */

#if TEST
    if(intpr) {
        printf("ENTRY TO dfint() pointer values\n");
        printf("jcqldp = %x jcquldp = %x\n",dfcp->jcqldp,dfcp->jcquldp);
    }
#endif

    if(dfcp->jcquldp == dfcp->jcqldp)/* check for spurious completion interrupt */
    {
        if(dfcp->dfutab.sgreq == SET)
        {
            dfcp->dfutab.sgreq = FAIL; /* fail SYSGEN flag */
            return;
        }
        else if(dfcp->dfutab.ioip == SET) /* check previous comp report handled */
            return;
        else
```

```
            {
                intio(DFC_IPL,df_addr[unit]);
                logstray(IIOADDR(DFC_IPL)); /* log interrupting controller address */
                intio(DFC_IPL,nullseg);
                return;
            }
        }

        while(dfcp->jcqldp != dfcp->jcquldp) /* unload all reports in queue */
        {

            jcquldp = (struct df_jcqe *)dfcp->vcquldp;  /* set to current cmp entry */
            bp = jcquldp->jid;  /* set bp to current comp */

            if(jcquldp->opc == D_VSYSG) /* check for SYSGEN completion */
            {
                if(jcquldp->jstat != 0) /* check for failing completion */
                {
                    printf( "SYSGEN ERROR: IDFC=%d\n", (df_addr[unit].base>>8)&0xff );
                    printf( "              jid=%x jstat=0x%x erstat=0x%x xerstat=0x%x\n",
                        jcquldp->jid, jcquldp->jstat, jcquldp->erstat, jcquldp->xerstat );
                    dfcp->dfutab.sgreq = FAIL;
                }
                else
                    dfcp->dfutab.sgreq = NULL; /* reset required flag */

                if(dfcp->dfutab.jrqsleep > 0) /* check for processes sleeping on JRQ */
                    wakeup((caddr_t)dfcp->smd_jrqa[jcquldp->smdnum].df_jrq);

                jcquldp++;
                if(jcquldp == &dfcp->df_jcq[DF_NJCQE]) /* check for bottom of queue */
                    jcquldp = dfcp->df_jcq; /* if yes - reset to top */
                dfcp->vcquldp = (struct df_jcqe *)jcquldp; /* save updated pointer value */
                dfcp->jcquldp = (struct df_jcqe *)df_vtop(jcquldp); /* save updated
                                                                      pointer value */

                return;
            }

#if PERFON
            if(jcquldp->opc == D_PRFON) /* is this perf report */
            {
                /* copy performance data to UNIT status */
#if TEST
                if(perfpr) {
                    printf("Copy performance data to unit status!!\n");
                }
#endif
```

```
                for(i=0; i<dfcp->numdrv; i++) {
                    drv = ((dfcp->df_perf[i].smdnum << 1) +
                        dfcp->df_perf[i].dsknum);

                    /* convert to HZ ( x * HZ / 1000.) from milliseconds */
                    dfcp->df_stat[drv].io_act += (dfcp->df_perf[i].cumutil * HZ) / 1000;
#if FULLPERF
                    dfcp->df_stat[drv].io_liact = dfcp->df_perf[i].cumutil;
                    dfcp->df_stat[drv].prfrpt++;
                    dfcp->df_stat[drv].io_intv = (time_t)dfcp->df_perf[i].sampint;
                    dfcp->df_stat[drv].tnrreq += dfcp->df_perf[i].tnrreq;
                    dfcp->df_stat[drv].tnwreq += dfcp->df_perf[i].tnwreq;
                    dfcp->df_stat[drv].cumseekd += dfcp->df_perf[i].cumseekd;
#endif /* FULLPERF */
                }

                if(dfcp->dfutab.jrqsleep > 0) /* check for processes sleeping on JRQ */
                    wakeup((caddr_t)dfcp->smd_jrqa[jcquldp->smdnum].df_jrq);

                jcquldp++;
                if(jcquldp == &dfcp->df_jcq[DF_NJCQE]) /* check for bottom of queue */
                    jcquldp = dfcp->df_jcq; /* if yes - reset to top */
                dfcp->vcquldp = (struct df_jcqe *)jcquldp; /* save updated pointer value */
                dfcp->jcquldp = (struct df_jcqe *)df_vtop(jcquldp); /* save updated
                                                                        pointer value */


                continue;
            }
#endif
            drv = (bp->b_dev & 030)>>3;
            if(jcquldp->jstat != 0) /* check for fail job status */
            {
                if(jcquldp->jstat != D_RETRY) {
                    printf( "I/O ERROR: IDFC=%d dsk=0%o block=%d count=%d\n",
                        (bp->b_dev>>8)&0xff, bp->b_dev&0xff, bp->b_blkno, bp->b_bcount );
                    printf( "           jid=%x jstat=0x%x erstat=0x%x xerstat=0x%x\n",
                        bp, jcquldp->jstat, jcquldp->erstat, jcquldp->xerstat );
                    bp->b_flags |= B_ERROR; /* set error flag in buffer header */
                }
                dfcp->dfutab.qcnt--; /* update jobs outstanding for controller */
                dfcp->df_stat[drv].io_qc--; /* update jobs outstanding for drive */
                logberr(bp,&dfcp->df_stat[drv],jcquldp); /* log error record */
            }
            else
            /* normal completion received */
            {
#if TEST
                if(intpr) {
```

```
                printf("jcqjid   = %x\n",jcquldp->jid);
                printf("jcqjstat= %x\n",jcquldp->jstat);
                printf("jcqopc   = %x\n",jcquldp->opc);
                printf("jcqsmdnum= %x\n",jcquldp->smdnum);
                printf("jcqdsknum= %x\n",jcquldp->dsknum);
                printf("jcqerstat= %x\n",jcquldp->erstat);
                printf("jcqxerstat= %x\n",jcquldp->xerstat);
                /* end of output */
            }
#endif


            dfcp->dfutab.qcnt--; /* update jobs outstanding for controller */
            dfcp->df_stat[drv].io_qc--; /* update jobs outstanding for drive */
            dfcp->df_stat[drv].io_resp += (lbolt - bp->b_start); /* update total block
                                                                     response time */
            bp->b_resid = 0; /* reset residual byte count */

#if PERFON
            /* gather performance data for a drive */
            /* wrap queue if needed */
            if(dfcp->df_stat[drv].pttrack >= dfcp->df_stat[drv].endptrack)
                dfcp->df_stat[drv].pttrack = &dfcp->df_stat[drv].ptrackq[0];
            (dfcp->df_stat[drv].pttrack)->b_blkno = bp->b_blkno;
            (dfcp->df_stat[drv].pttrack++)->bp = bp;

#endif

        }

        /* unlink job from work list */
        fp = (struct buf *) bp->av_forw; /* forward buffer pointer */
        rp = (struct buf *) bp->av_back; /* reverse buffer pointer */
        if(fp == NULL && rp == (struct buf *)&dfcp->dfutab) /* check for last
                                                                job in list */
        {
            dfcp->dfutab.ioip = NULL; /* take down IO in progress flag */
            dfcp->dfutab.b_actf = NULL; /* mark list head as empty */
            dfcp->dfutab.b_actl = NULL; /* clear tail pointer */
            if(dfcp->dfutab.qcnt != NULL) /* verify last job in list */
                printf("ERROR - IDFC driver queue count wrong!\n");
        }

        if(fp == NULL) /* check for job at end of work list */
        {
            rp->av_forw = NULL; /* mark next to last as last on list */
            dfcp->dfutab.b_actl = rp;
        }
        else
        {
            rp->av_forw = fp; /* link previous fwd pointer to next buf */
            fp->av_back = rp; /* link next buf back pntr to previous buf */
        }

        if(dfcp->dfutab.jrqsleep > 0) /* check for processes sleeping on JRQ */
            wakeup((caddr_t)dfcp->smd_jrqa[jcquldp->smdnum].df_jrq);

        jcquldp++;
        if(jcquldp == &dfcp->df_jcq[DF_NJCQE]) /* check for bottom of queue */
            jcquldp = dfcp->df_jcq; /* if yes - reset to top */
        dfcp->vcquldp = (struct df_jcqe *)jcquldp; /* save updated pointer value */
        dfcp->jcquldp = (struct df_jcqe *)df_vtop(jcquldp); /* save updated pointer
                                                                value */

        iodone(bp); /* report completion to user */

#if TEST
        if(intpr) {
            printf("EXIT OF dfint() queue pointers are\n");
            printf("jcqldp = %x jcquldp = %x\n",dfcp->jcqldp,dfcp->jcquldp);
        }
#endif
    }
}


/*
 * This routine will generate a vertical XOR
 * checksum over the specified buffer for the
 * size specified.
 */
long
gcksum(baddr,bsize)
register long *baddr;
register int bsize;
{
    register long cksum = 0;/* checksum value */

    while(--bsize > 0)
        cksum ^= *(baddr++);/* XOR next word */
    return(cksum);
}


/*
 * The dfstrategy routine is responsible for validating job
 * requests, placing the request in the proper request queue,
 * updating the appropriate controller and drive status,
 * and generating the work pending PIR for the correct IDFC.
 * All information required to generate the job request is
 * contained in a buffer header. The address of the buffer
 * header is passed to the routine as an input argument.
 */
dfstrategy(bp)
register struct buf *bp;/* buffer header pointer */
{
    struct  buf *dp;/* temp buffer pointer */



    register struct dfc *dfcp; /* IDFC controller data pointer */
    register union jrqe *vjrqldp; /* job request queue load pointer */
    register struct smd_jrq *smd_jrpt; /* job request queue struct pt */
    register union jrqe *pjrquldp;  /* job request queue unload pointer */
    register unit;  /* IDFC controller ID */
    union jrqe *pjrqldp; /* job request queue load pointer */
    union jrqe *tpntr; /* temp job request queue load pointer */
    union jrqe *jrqmax; /* end of job request queue */
    union jrqe *jrqstart;   /* start of job request queue */
    struct dskpart *partpt; /* pointer to partition info */
    daddr_t last;   /* drive last block address */
    paddr_t map;/* process map address */
    paddr_t maddr;  /* memory address of request */
    unsigned char smd; /* SMD subunit ID */
    unsigned char drv; /* IDFC drive ID */
    unsigned char part; /* drive partition */

    int sps;/* saved priority level */




    unit = minor(bp->b_dev);
    maddr = paddr(bp);
    part = unit&07;
    smd = (unit & 020) >> 4;
    drv = (unit & 030) >> 3;
    unit >>= 5;
    dfcp = &df_dfc[unit];

    smd_jrpt = &dfcp->smd_jrqa[smd];
    jrqstart = smd_jrpt->df_jrq;
    jrqmax = &smd_jrpt->df_jrq[DF_NJRQE];
    vjrqldp = smd_jrpt->vrqldp;
    pjrquldp = smd_jrpt->jrquldp;

    /* Check if drive equipped */
    if ((partpt=dfcp->df_part[drv]) == NULL) {
        bp->b_flags |= B_ERROR;
        bp->b_error = ENXIO;
        iodone(bp);
        return;
    }

    /* verify job request for valid size */
    last = partpt[part].nblocks;
    if (bp->b_blkno < 0 || bp->b_blkno >= last) {
        if (bp->b_blkno == last && bp->b_flags&B_READ)
            bp->b_resid = bp->b_bcount;
        else {
            bp->b_flags |= B_ERROR;
            bp->b_error = ENXIO;
        }
        iodone(bp);
        return;
    }

#if TEST
    if(stratpr) {
        /* output variables */
        pjrqldp = smd_jrpt->jrqldp;
        printf("unit = %x maddr = %x\n",unit,maddr);
        printf("smd = %x drv = %x jrqstart = %x jrqmax = %x\n",smd,drv,
                jrqstart,jrqmax);
        printf("jrqldp = %x jrquldp = %x\n",pjrqldp,pjrquldp);
    }
#endif

    /* make entry in job request queue */
    for(;;)
    {
        tpntr = vjrqldp; /* setup temp load pointer */
        if(++tpntr == jrqmax) /* check for end of queue */
            tpntr = jrqstart; /* reset to top */
        if((union jrqe *)df_vtop(tpntr) == pjrquldp) /* check for queue full */
        {
            dfcp->dfutab.jrqsleep++; /* inc process sleep count */
            sleep((caddr_t)smd_jrpt,PZERO); /* put process to sleep */
            dfcp->dfutab.jrqsleep--; /* decr process sleep count */

            pjrquldp = smd_jrpt->jrquldp;
            vjrqldp = smd_jrpt->vrqldp;
        }
        else break;
    }

    /* link buffer header into device work list */
    baseio(df_addr[unit]);  /* set up mmu */
    sps = spl6();   /* raise priority for critical code region */

    if(dfcp->dfutab.b_actf == NULL) /* check for first job on list */
    {
        dfcp->dfutab.b_actf = bp; /* link buf to fwd list pointer */
        dfcp->dfutab.b_actl = bp; /* link buf to back list pointer */
        bp->av_back = (struct buf *)&dfcp->dfutab; /* point buf back to list head */
    }
    else
    {
        dp = dfcp->dfutab.b_actl; /* get addr of last buf in list */
        dfcp->dfutab.b_actl = bp; /* link buf to dev back list pointer */
        dp->av_forw = bp; /* link buf in next to last buf fwd pntr */
        bp->av_back = dp; /* point to previous last buffer */
    }

    bp->av_forw = NULL; /* mark as last buf in device work list */


    /* update job request, controller and drive status data */

    dfcp->df_stat[drv].io_cnt++;/* inc operations count */
    dfcp->df_stat[drv].io_bcnt += btoc(bp->b_bcount); /* inc click count */
    dfcp->dfutab.qcnt++; /* update jobs outstanding for controller */
    dfcp->df_stat[drv].io_qc++; /* update jobs outstanding for drive */
    bp->b_start = lbolt; /* time stamp start of request process */

#if TEST
    if(stratpr) {
        /* print drive-controller status */
        printf("IDFC = %x SMD = %x DRIVE = %x\n",unit,smd,drv&1);
        printf("io_cnt = %x io_bcnt = %x\n",
                dfcp->df_stat[drv].io_cnt,dfcp->df_stat[drv].io_bcnt);
        printf("cntlqcnt = %x drvqcnt = %x\n",
                dfcp->dfutab.qcnt,dfcp->df_stat[drv].io_qc);
        printf("start time = %x\n",bp->b_start);
        /* end of status */
    }
#endif


    /*
     * is request kernel or user
     */
    if(maddr & VUSER)
        /* user map pointer */
        map = (paddr_t) &(((struct user *)
                            ctob((int)bp->b_proc->p_addr))->u_segdata);
    else
        /* kernel map pointer */
        map = pkmmudesc;

    /* enter job request in controller request queue */

    vjrqldp->req.jid = bp;  /* buffer header addr used as job id */
    vjrqldp->req.jcqid = 0; /* always 0 for single queue */
    vjrqldp->req.opc = (bp->b_flags & B_READ) ? D_READ : D_WRITE;
    vjrqldp->req.dsknum = (unsigned)(char) drv&0x01;
    vjrqldp->req.smdnum = smd;
    vjrqldp->req.sp1 = 0;
    vjrqldp->req.pmta = map;
    vjrqldp->req.sma = paddr(bp);
    vjrqldp->req.sdba = (bp->b_blkno+partpt[part].sblock) << 9;
    vjrqldp->req.bcnt = (unsigned)bp->b_bcount;
    vjrqldp->req.cksum = gcksum((long *) vjrqldp, sizeof(union jrqe)/sizeof(long));

#if TEST
    if(stratpr) {
        printf("jid = %x\n",vjrqldp->req.jid);
        printf("jcqid   = %x\n",vjrqldp->req.jcqid);
        printf("opc = %x\n",vjrqldp->req.opc);
        printf("smdnum  = %x\n",vjrqldp->req.smdnum);
        printf("dsknum  = %x\n",vjrqldp->req.dsknum);
        printf("sp1 = %x\n",vjrqldp->req.sp1);
        printf("pmta    = %x\n",vjrqldp->req.pmta);
        printf("sma = %x\n",vjrqldp->req.sma);
        printf("sdba    = %x\n",vjrqldp->req.sdba);
        printf("bcnt    = %x\n",vjrqldp->req.bcnt);
        printf("cksum   = %x\n",vjrqldp->req.cksum);
    }
#endif

    /* update controller request queue pointer */


    smd_jrpt->vrqldp = tpntr;
    smd_jrpt->jrqldp = (union jrqe *)df_vtop(tpntr);

#if FULLPERF
    {
        /* Calculate the queue length after it is added to the queue */
        int qsize;

        /* In the following if ... else ..., the shift is to divide by  */
        /* the sizeof(union jrqe) and thus will vary if this union */
        /* changes.  Currently this size is 32.             */
        pjrqldp = smd_jrpt->jrqldp;
        if ((unsigned)pjrquldp < (unsigned)pjrqldp)  /* check for wrap around */
            qsize = ((unsigned)pjrqldp - (unsigned)pjrquldp) >> 5;
        else
            qsize = 1 + (((unsigned)vtop(jrqmax) - (unsigned)pjrquldp) +
                        ((unsigned)pjrqldp - (unsigned)df_vtop(jrqstart))) >> 5;

        dfcp->df_stat[drv].cumqlen += qsize;
        if (dfcp->df_stat[drv].maxqlen < qsize)
            dfcp->df_stat[drv].maxqlen = qsize;
        if (dfcp->df_stat[drv].minqlen > qsize)
            dfcp->df_stat[drv].minqlen = qsize;
    }
#endif  /* FULLPERF */

    dfcp->dfutab.ioip = SET; /* set IO in progress flag */
    ((struct pir16 *)(BIOADDR|OPIR))->pir04 = SET; /* send job pending pir
                                                      to IDFC */

    splx(sps);  /* lower priority level */
    clearbaseio;/* clear mmu */
}




/*
 * The dfopen routine is called each time a raw disk is opened, or when
 * a file system is mounted
 */
dfopen(dev)
dev_t   dev;
{
    if(dev & 0xE0){
        /* minor number must be < 31 */
        u.u_error = ENXIO;
        return;
    }
    dev = minor(dev);
    if((dev >> 5) >= df_cnt){
        u.u_error = ENXIO;
        return;
    }
}


dfread(dev)
register dev_t dev;
{
    register unit;  /* IDFC controller ID */
    register unsigned char drv; /* IDFC drive ID */
    register struct dfc *dfcp;  /* IDFC controller pointer */
    register struct dskpart *partpt; /* pointer to partition info */
    register unsigned char part; /* drive partition */

    unit = minor(dev);
    dfcp = &df_dfc[unit>>5];
    part = unit&07;
    drv = (dev &030)>>3;
    if ((partpt=dfcp->df_part[drv]) == NULL)
        u.u_error = ENXIO;
    else if (physck(partpt[part].nblocks, B_READ))
        physio(dfstrategy, 0, dev, B_READ);
}


dfwrite(dev)
register dev_t dev;
{
    register unit;  /* IDFC controller ID */
    register unsigned char drv; /* IDFC drive ID */
    register struct dfc *dfcp;  /* IDFC controller pointer */
    register struct dskpart *partpt; /* pointer to partition info */
    register unsigned char part; /* drive partition */

    unit = minor(dev);
    dfcp = &df_dfc[unit>>5];
    part = unit&07;
    drv = (dev &030)>>3;
    if ((partpt=dfcp->df_part[drv]) == NULL)
        u.u_error = ENXIO;
    else if (physck(partpt[part].nblocks, B_WRITE))
        physio(dfstrategy, 0, dev, B_WRITE);
}


dfprint(dev, str)
register dev_t dev;
char *str;
{
    printf("%s on IDFC(%d) drive 0%o\n", str, (dev>>8) & 0x7f, dev&0xff);
}


/*
 *  return 1 if all disk work queue are empty, 0 otherwise
 */
```

```
dfidle()
{
    register i;         /* idfc index */

    for(i=0; i<df_cnt; i++)
        if (df_dfc[i].dfutab.b_actf != NULL) return 0;
    return 1;
}
```

APPENDIX L: 3B5 COMPUTER CHARACTER DRIVER

This is the character driver for the 3B5 Computer.

```
static char Sccsid[]="@(#)adli.c3.1.2.5";
/*
 *  ADLI driver
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <sys/signal.h>
#include <sys/user.h>
#include <sys/errno.h>
#include <sys/file.h>
#include <sys/tty.h>
#include <sys/termio.h>
#include <sys/conf.h>
#include <sys/sysinfo.h>
#include <sys/sysmacros.h>
#include <sys/adli.h>

extern struct mmutab ccmmudesc;

#define INTERUPT0x80000000

#define AD_ACU  0x80/* determine if adli or acu */

extern struct mmuseg ad_addr[];
extern int ad_cnt;
extern struct tty ad_tty[];

char    ad_speeds[16] = {
    0,B50BPS,B75BPS,B110BPS,B134BPS,B150BPS,0,B300BPS,B600BPS,B1200BPS,
    B1800BPS,B2400BPS,B4800BPS,B9600BPS,0,0
};

#define ON  1
#define OFF 0
#define ADLI_IPL6   /* Interrupt Priority Level of the ADLI */

/*
```

```
 * Minor number allocation for the ADLI.  This is the format
 * of the minor number that the kernel passes to adopen(), adclose(),
 * adread(), adwrite() and adioctl().  Before the minor number
 * is used, the minor() macro should be invoked to translate the
 * minor number to an index to a tty structure (however, the
 * ACU bit must be checked first).
 *
 *  1......XACU device; X is 0 or 1
 *
 *  0....XXXUART port; XXX is 0 to 7
 *
 * After translation by the minor macro, the minor number has the
 * values:
 *
 *  ACU device -0,1,8,9,16,17,...
 *  UART port - 0,1,2,...
 */

adopen(dev, flag)
register dev_t dev;
{
    register struct tty *tp;
    extern adproc();

    /*
     * determine if adli or acu
     */

    if (dev & AD_ACU)
        {
        acuopen( dev & ~AD_ACU );
        return;
        }

    /*
     * extract minor number
     */
    if ( dev & 0xF8 ) {
        /* minor number must be 0-7 only */
        u.u_error = ENXIO;
        return;
    }
    dev = minor(dev);

    baseio(ad_addr[(dev>>3)]);
    if (dev >= ad_cnt) {
        u.u_error = ENXIO;
        clearbaseio;
        return;
        }
        tp = &ad_tty[dev];
        if ((tp->t_state&(ISOPEN|WOPEN)) == 0) {
            ttinit(tp);
            tp->t_proc = adproc;
            adparam(BIOADDR,dev);
        }
        spl5();
        if (tp->t_cflag&CLOCAL || admodem(BIOADDR,dev, ON))
            tp->t_state |= CARR_ON;
        else
            tp->t_state &= ~CARR_ON;
        if (!(flag&FNDELAY))
        while ((tp->t_state&CARR_ON)==0) {
            tp->t_state |= WOPEN;
            clearbaseio;
            sleep((caddr_t)&tp->t_canq, TTIPRI);
            spl5();
        }
        (*linesw[tp->t_line].l_open)(tp);
        spl0();
        clearbaseio;
    }

    adclose(dev)
    register dev_t dev;
    {
        register struct tty *tp;

        /*
         * determine if adli or acu
         */

        if (dev & AD_ACU)
            {
            acuclose( dev & ~AD_ACU );
            return;
            }

        /*
         * extract minor number
         */
        dev = minor(dev);

        tp = &ad_tty[dev];
        (*linesw[tp->t_line].l_close)(tp);
        if (tp->t_cflag&HUPCL) {
            baseio(ad_addr[(dev>>3)]);
            spl5();
        admodem(BIOADDR,dev, OFF);
        spl0();
        clearbaseio;
    }
}

adread(dev)
register dev_t dev;
{
    register struct tty *tp;

    /*
     * determine if adli or acu
     */
    if (dev & AD_ACU)
        {
        nodev();
        return;
        }

    /*
     * extract minor number
     */
    dev = minor(dev);

    tp = &ad_tty[dev];
    (*linesw[tp->t_line].l_read)(tp);
}

adwrite(dev)
register dev_t dev;
{
    register struct tty *tp;

    /*
     * determine if adli or acu
     */
    if (dev & AD_ACU)
        {
        acuwrite( dev & ~AD_ACU );
        return;
        }

    /*
     * extract minor number
     */
    dev = minor(dev);

    tp = &ad_tty[dev];
    (*linesw[tp->t_line].l_write)(tp);
}

adioctl(dev, cmd, arg, mode)
register dev_t dev;
{
    int ttytype;

    /*
     * determine if adli or acu
     */
    if (dev & AD_ACU)
        {
        nodev();
        return;
        }

    /*
     * extract minor number
     */
    dev = minor(dev);
    baseio(ad_addr[(dev>>3)]);

    if (cmd == TTYTYPE) {
        ttytype = SATTY;
        if (copyout(&ttytype, arg, sizeof(int)))
            u.u_error = EFAULT;
    }
    else {
        if (ttiocom(&ad_tty[dev], cmd, arg, mode))
        {
            baseio(ad_addr[(dev>>3)]);
            adparam(BIOADDR,dev);
            clearbaseio;
        }
    }
}

adparam(baddr,dev)
registerchar *baddr;
{
    register struct tty *tp;
    register flags, mr1,mr2,cr;
    register struct uart *adaddr;

    tp = &ad_tty[dev]; /* get address of tty data structure */
    /* find uart address */
    adaddr = &(((struct adli_uart *)(baddr+OADLIUART))->port[dev&0x7].uart);
    ((struct adli_wcsr *)(baddr+OCSR))->inh_int = 0; /* clear int inhibit*/
    flags = tp->t_cflag;
    if ((flags&CBAUD) == 0) {
        /* hang up modem */
        admodem(BIOADDR,dev,OFF);
        return;
    }
    /*
     * construct mode register 1 from content of tty structure.
     */
    mr1 = ASYNCH1;
    if (flags & CS6)
        mr1 |= BITS6;
    if (flags & CS7)
        mr1 |= BITS7;
    if (flags & PARENB) {
        mr1 |= PENABLE;
        if ((flags & PARODD) == 0)
            mr1 |= EPAR;
    }
    if (flags & CSTOPB)
        mr1 |= TWOSB;
    else
        mr1 |= ONESB;

    /*
     * construct mode register 2
     */

    mr2 = ad_speeds[flags & CBAUD];
    mr2 |= XMITINT | RCVINT;

    /*
     * read the command register, thus setting the mode register
     * pointer to mr1.  Then, set up the command register.
     */

    cr = adaddr->command;
    cr |= (XMITENB|RESET);
    if (flags & CREAD)
        cr |= RCVENB;
    else
        cr &= ~RCVENB;
    adaddr->mode = mr1;
    adaddr->mode = mr2;
    adaddr->command = cr;
    adaddr->command &= ~RESET;
}

adrint(dev)
```

```
{
    register struct tty *tp;
    register char c;
    register char sr;
    register struct uart *adaddr;
    register char *baddr;

    sysinfo.rcvint++;
    if ((dev>>3) >= ad_cnt)
        return;

    intio(ADLI_IPL,ad_addr[(dev>>3)]);
    baddr = (char *)IIOADDR(ADLI_IPL);
    /* find uart address */
    adaddr = &(((struct adli_uart *)(baddr+OADLIUART))->port[dev&0x7].uart);
    tp = &ad_tty[dev]; /* get address of tty data structure for dev */
    while ((sr = adaddr->status) & RCVRDY) {
        c = adaddr->data;
        if (!(tp->t_state&(ISOPEN|WOPEN)))
            continue;
        if (tp->t_cflag & CLOCAL || sr & DCD) {
            if ((tp->t_state&CARR_ON) == 0) {
                wakeup(&tp->t_canq);
                tp->t_state |= CARR_ON;
            }
        } else {
            if (tp->t_state&CARR_ON) {
                signal(tp->t_pgrp, SIGHUP);
                tp->t_pgrp = 0;
                adaddr->command &= ~DTR;
                tp->t_state &= ~CARR_ON;
                ttyflush(tp, (FREAD|FWRITE));
            }
            continue;
        }
        if (tp->t_iflag&IXON) {
            register char ctmp;
            ctmp = c & 0177;
            if (tp->t_state&TTSTOP) {
                if (ctmp == CSTART || tp->t_iflag&IXANY)
                    (*tp->t_proc)(tp, INTERUPT | T_RESUME);
            } else {
                if (ctmp == CSTOP)
                    (*tp->t_proc)(tp, INTERUPT | T_SUSPEND);
            }
            if (ctmp == CSTART || ctmp == CSTOP)
                continue;
        }
                /* Check for errors */
```

```
{
register int flg;
char lbuf[3];        /* local character buffer */
short lcnt;     /* count of chars in lbuf */

lcnt = 1;
flg = tp->t_iflag;
if (sr&(FE|PARERR|OVRRUN)) {
    adaddr->command |= RESET;
    adaddr->command &= ~RESET;
}
if (sr&PARERR && !(flg&INPCK))
    sr &= ~PARERR;
if (sr&(FE|PARERR|OVRRUN)) {
    if ((c&0377) == 0) {
        if (flg&IGNBRK)
            continue;
        if (flg&BRKINT) {
        (*linesw[tp->t_line].l_input)(tp, L_BREAK);
            continue;
        }
    } else {
        if (flg&IGNPAR)
            continue;
    }
    if (flg&PARMRK) {
        lbuf[2] = 0377;
        lbuf[1] = 0;
        lcnt = 3;
        sysinfo.rawch += 2;
    } else
        c = 0;
} else {
    if (flg&ISTRIP)
        c &= 0177;
    else {
        c &= 0377;
        if (c == 0377 && flg&PARMRK) {
            lbuf[1] = 0377;
            lcnt = 2;
        }
    }
}
/* stash character in r_buf */
lbuf[0] = c;
if (tp->t_rbuf.c_ptr == NULL) {
    return;
}
while (lcnt) {
```

```
                *tp->t_rbuf.c_ptr++ = lbuf[--lcnt];
                tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size - (--tp->t_rbuf.c_count);
                (*linesw[tp->t_line].l_input)(tp, L_BUF);
            }
            }
        }
        /* turn off interrupt */
        ((struct adli_wcsr *)(baddr+OCSR))->req_int = 0;
}

adxint(dev)
register dev;
{
        register struct tty *tp;
        register struct uart *adaddr;
        register char *baddr;

        sysinfo.xmtint++;
        intio(ADLI_IPL,ad_addr[(dev>>3)]);
        baddr = (char *)IIOADDR(ADLI_IPL);  /* get address of adli board */
        /* find uart address */
        adaddr = &(((struct adli_uart *)(baddr+OADLIUART))->port[dev&0x7].uart);
        tp = &ad_tty[dev]; /* get address of tty data structure for dev */
        if (tp->t_cflag&CLOCAL || adaddr->status & DCD) {
            if ((tp->t_state & CARR_ON) == 0) {
                wakeup(&tp->t_canq);
                tp->t_state |= CARR_ON;
            }
        } else {
            if (tp->t_state & CARR_ON) {
                if (tp->t_state & ISOPEN) {
                    signal(tp->t_pgrp, SIGHUP);
                    adaddr->command &= ~DTR;
                    ttyflush(tp,(FREAD|FWRITE));
                }
                tp->t_state &= ~CARR_ON;
            }
        }
        while(adaddr->status & XMTRDY) {/* TX rdy */
            adaddr->command &= ~XMITENB; /* disable uart transmit, it will
                                be re-enabled if necessary */
            if (tp->t_state & TTXON) {
                adaddr->command |= XMITENB;
                adaddr->data = CSTART;
                tp->t_state &= ~TTXON;
                continue;
            }
            if (tp->t_state & TTXOFF) {
                adaddr->command |= XMITENB;
            adaddr->data = CSTOP;
            tp->t_state &= ~TTXOFF;
            continue;
        }
        if (tp->t_state & BUSY) {
            tp->t_state &= ~BUSY;
            adproc(tp, INTERUPT | T_OUTPUT);
            continue;
        }
        break;
    }
    /* turn off interrupt */
    ((struct adli_wcsr *)(baddr+OCSR))->req_int = 0;
}

adproc(tp, cmd)
register struct tty *tp;
{
    struct uart *adaddr;
    int dev;
    extern ttrstrt();
    int s;
    struct mmuseg save;

    s = spl5();
    save = savebaseio;
    dev = tp - ad_tty;
    /* find uart address */
    if(cmd & INTERUPT)
        adaddr = &(((struct adli_uart *)(IIOADDR(ADLI_IPL)+
                                    OADLIUART))->port[dev&0x7].uart);
    else {
        baseio(ad_addr[(dev>>3)]);
        adaddr = &(((struct adli_uart *)(BIOADDR+OADLIUART))->port[dev&0x7].uart);
    }

    switch(cmd & (~INTERUPT)) {

    case T_TIME:
        tp->t_state &= ~TIMEOUT;
        adaddr->command &= ~BREAK;
        adaddr->command &= ~XMITENB;
        goto start;

    case T_WFLUSH:
        tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
        tp->t_tbuf.c_count = 0;

    case T_RESUME:
        tp->t_state &= ~TTSTOP;
        goto start;

    case T_OUTPUT:
    start:
        {
        register struct ccblock *tbuf;

        if (tp->t_state & (BUSY|TTSTOP|TIMEOUT))
            break;
        tbuf = &tp->t_tbuf;
        if (tbuf->c_ptr == NULL || tbuf->c_count == 0) {
            if (tbuf->c_ptr)
                tbuf->c_ptr -= tbuf->c_size - tbuf->c_count;
            if (!(CPRES & (*linesw[tp->t_line].l_output)(tp))) {
                if (adaddr->status & XMTEMT)
                    adaddr->command &= ~XMITENB;
            break;
            }
        }
        tp->t_state |= BUSY;
        adaddr->command |= XMITENB;
        adaddr->data = *tbuf->c_ptr++;
        tbuf->c_count--;
        break;
        }

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

    case T_BLOCK:
        tp->t_state &= ~TTXON;
        tp->t_state |= TBLOCK;
        if (tp->t_state & BUSY) {
            tp->t_state |= TTXOFF;
        } else {
            tp->t_state |= BUSY;
            adaddr->command |= XMITENB;
            adaddr->data = CSTOP;
        }
        break;

    case T_RFLUSH:
        if (!(tp->t_state&TBLOCK))
            break;
    case T_UNBLOCK:
        tp->t_state &= ~(TTXOFF|TBLOCK);
        if (tp->t_state & BUSY)
```

```
                tp->t_state |= TTXON;
            else {
                tp->t_state |= BUSY;
                adaddr->command |= XMITENB;
                adaddr->data = CSTART;
            }
            break;

        case T_BREAK:
            adaddr->command |= BREAK|XMITENB;
            tp->t_state |= TIMEOUT;
            timeout(ttrstrt, tp, HZ/4);
            break;

        case T_PARM:
            adparam(BIOADDR, dev);
            break;

        }
        baseio(save);
        splx(s);
}

admodem(baddr, dev, flag)
register char *baddr;
{
        register struct uart *adaddr;

        /* get uart address */
        adaddr = &(((struct adli_uart *)(baddr+OADLIUART))->port[dev&0x7].uart);
        if (flag==OFF)
            adaddr->command &= ~DTR;
        else
            adaddr->command |= DTR;
        return(adaddr->status & DCD);
}

/*
 * ACU driver
 *
 * Minor number allocation for the ACU.  This is the format
 * of the minor number that the ADLI driver passes to acuopen(),
 * acuclose() and acuwrite().
 *
 *  .......XACU device; X is 0 or 1
 *
 * After translation by the minor macro, the minor number has the
 * values:
 *
 *  ACU device -0,1,8,9,16,17,...
 *
 * This sequence must be converted to the sequence 0,1,2,...
 */


#define ACUPRI  (PZERO + 5) /* sleep priority while polling */
#define ACUDELAY 3      /* wait 3 ticks while polling */
#define ACUTIME (300 * HZ)  /* software timer 5 minutes */

extern int acu_time[];      /* software timer */

/*
 * adstart()
 *
 * ADLIs require no initialization, but the ACUs do.  This routine is
 * called from main() during kernel initialization
 */
adstart()
    {
    register i, maxacu;

    maxacu = ad_cnt / 8;

    for (i=0; i < maxacu; ++i) {
        baseio(ad_addr[i]);

        /* init control, leave off */
        ((struct adli_acui *) (BIOADDR + OADLIACUI))->acui[0].ppi.control = A_AMODE;
        ((struct adli_acui *) (BIOADDR +
                              OADLIACUI))->acui[0].ppi.command = A_CRQ | A_DPR;

        ((struct adli_acui *) (BIOADDR + OADLIACUI))->acui[1].ppi.control = A_AMODE;
        ((struct adli_acui *) (BIOADDR +
                              OADLIACUI))->acui[1].ppi.command = A_CRQ | A_DPR;
    }
    clearbaseio;
}


/*
 * acuopen
 * if illegal device or if acu is not on, set a device error flag
 * if the acu is busy, set a busy error flag
 * if everything is ok, then start a call
 */

acuopen(dev)
register dev_t dev;
```

```
{
    register struct adppi *acuaddr;


    /*
     * extract minor number
     */
    if ( dev & 0xFE ) {
        /* minor number must be 0-1 only */
        u.u_error = ENXIO;
        return;
    }

    dev = minor(dev);

    if (dev >= ad_cnt) {
        u.u_error = ENXIO;
        return;
    }

    /* convert sequence 0,1,8,9,... to 0,1,2,3,... */
    dev = ((dev>>3)<<1) + (dev & 1);

    /* find address of acu(dev) */
    baseio(ad_addr[(dev>>1)]);

    acuaddr = &(((struct adli_acui *) (BIOADDR + OADLIACUI))->acui[dev&0x1].ppi);

    /* check for power on */
    if (acuaddr->status & A_PWI)
        u.u_error = ENXIO;

        /* check if device is busy */
    else if ( ! (acuaddr->status & A_DLO)  ||  ! (acuaddr->command & A_CRQ))
        u.u_error = EBUSY;

    else {  /* everything ok, start the call */
        acuaddr->command = A_DPR;   /* turn on CRQ, leave DPR off */
        acu_time[dev] = 0;      /* init timer */
    }
    clearbaseio;
}



/*
 * acuclose
 * all that has to be done is turn off CRQ and DPR
 */
```

```
acuclose(dev)
register dev_t dev;
{

    /*
     * extract minor number
     */
    dev = minor(dev);

    /* convert sequence 0,1,8,9,... to 0,1,2,3,... */
    dev = ((dev>>3)<<1) + (dev & 1);

    /* find address of acu(dev) */
    baseio(ad_addr[(dev>>1)]);

    /* turn off CRQ and DPR */
    ((struct adli_acui *) (BIOADDR+
                                OADLIACUI))->acui[dev&0x1].ppi.command = A_DPR | A_CRQ;
    clearbaseio;
}


/*
 * acuwrite
 * takes characters one at a time from users i/o space and dials them
 * quits when characters run out or call can't be completed
 */

acuwrite(dev)
register dev_t dev;
{
    register char ch, acustatus;
    register struct adppi *acuaddr;
    register quit;
    register struct mmuseg *segp;


    /*
     * extract minor number
     */
    dev = minor(dev);

    /* convert sequence 0,1,8,9,... to 0,1,2,3,... */
    dev = ((dev>>3)<<1) + (dev & 1);

    /* find address of acu(dev) */
    segp = &ad_addr[ dev>>1 ];
    baseio( *segp );
```

```
/* calculate address of acu(dev) */

acuaddr = &(((struct adli_acui *) (BIOADDR + OADLIACUI))->acui[dev&0x1].ppi);

/* dial character from user's i/o space */

quit = 0;
while ( ! ((acustatus = acuaddr->status) & A_PWI)
    &&  acustatus & A_ACR
    &&  acu_time[dev] < ACUTIME
    &&  !quit
    &&  (ch = cpass()) > 0 ) {

    switch (ch) {   /* dial next character */

    case '-':   /* delay for second dial tone */
        clearbaseio;
        delay(4 * HZ);
        baseio( *segp );
        break;

    case 'f':   /* flash off  hook for 1 second */
        acuaddr->control = A_SCRQ;  /* turn off CRQ*/
        clearbaseio;
        delay(HZ);          /* wait */
        baseio( *segp );
        acuaddr->control = A_RCRQ;  /* turn on CRQ */
        break;

    case '*':
    case ':':   /* dial a * */
        acudial(segp,acuaddr,0xA,dev);
        break;

    case '#':
    case ';':   /* dial a # */
        acudial(segp,acuaddr,0xB,dev);
        break;

    case 'e':
    case '<':   /* end of number */
        quit = 1;
        break;

    case 'w':
    case '=':   /* wait for second dial tone */
        acudial(segp,acuaddr,0xD,dev);
        break;
```

```
                default:/* dial a digit, ignore if non digit */
                    if (ch >= '0'  &&  ch <= '9')
                        acudial(segp,acuaddr,ch - '0',dev);
                    else
                        quit = 1;

                }
            }

            /*
             * if everything is still ok, send an end of number signal
             */

            if ( ! ((acustatus = acuaddr->status) & A_PWI)
                &&  acustatus & A_ACR
                &&  acu_time[dev] < ACUTIME
                &&  acustatus & A_COS)

                acudial(segp,acuaddr,0xC,dev);

            /*
             * wait for call to be connected or terminated
             */

            while ( ! ((acustatus = acuaddr->status) & A_PWI)
                &&  acustatus & A_ACR
                &&  acu_time[dev] < ACUTIME
                &&  acustatus & A_COS) {

                timeout(wakeup,acuaddr,ACUDELAY);
                acu_time[dev] += ACUDELAY;
                clearbaseio;
                sleep(acuaddr,ACUPRI);
                baseio( *segp );
            }

            /*
             * set error if call was not completed
             */

            if (acustatus & A_PWI  || ! (acustatus & A_ACR)  || acu_time[dev] >= ACUTIME)
                u.u_error = ENXIO;
            clearbaseio;
        }


        /*
         * acudial
         * sends a digit to the ACU for dialing
```

```
 * follows RS-366 protocol
 *  wait for PND to turn on
 *  put out the digit
 *  turn on DPR
 *  wait for PND to turn off
 *  turn off DPR
 * the sequence is terminated if the ACU loses power or if the call times out
 * since the ACU does not generate interrupts, the ACU has to be polled
 *
 * NOTE: This function can only be called from code executing at BASE Level
 *       Immediately following the call to sleep the macro to reload the DMAP
 *       register with the segment descriptor for kernel I/O will be called.
 */

acudial( basegp, acuaddr, digit, dev)
register struct mmuseg *basegp;
register struct adppi *acuaddr;
register dev;
{
    register char acustatus;



    /*
     * wait for PND to turn on
     */

    while ( ! ((acustatus = acuaddr->status) & A_PWI)
        &&  acustatus & A_ACR
        &&  acu_time[dev] < ACUTIME
        &&  acustatus & A_PND) {

        timeout(wakeup,acuaddr,ACUDELAY);
        acu_time[dev] += ACUDELAY;
        clearbaseio;
        sleep(acuaddr,ACUPRI);
        baseio( *basegp );
    }

    /*
     * put out the digit and turn on DPR
     */

    if ( ! (acustatus & A_PWI)  &&  acustatus & A_ACR  &&  acu_time[dev] < ACUTIME) {
        acuaddr->command = digit | A_DPR;
        acuaddr->control = A_RDPR;
    }

    /*
     * wait for PND to turn off, then turn off DPR
```

```
         */

        while ( ! ((acustatus = acuaddr->status) & A_PWI)
            &&  acustatus & A_ACR
            &&  acu_time[dev] < ACUTIME
            &&  ! (acustatus & A_PND)) {

            timeout(wakeup,acuaddr,ACUDELAY);
            acu_time[dev] += ACUDELAY;
            clearbaseio;
            sleep(acuaddr,ACUPRI);
            baseio( *basegp );
        }

        acuaddr->control = A_SDPR;
    }
```

GLOSSARY

EDT	The Equipped Device Table contains the characteristics of all peripherals currently in the system.
edt_data	A 3B2 Computer UNIX System file used to fill in the entries in the EDT. This file contains data for any driver that may be in the system.
config	Config is a command that has historically generated the kernel data structure from a configuration file.
/dev	The directory on UNIX Systems containing special device files controlled by hardware and software drivers.
/etc/master	A file describing all drivers that can be in the system.
mkboot	A command that updates object files to be used by the boot program.
mkunix	A command that creates a bootable UNIX System kernel from the current contents of memory.
newboot	A command that places the 2 boot programs on the boot partition to be accessible by the firmware.
sysdef	The command that lists the current system configuration, that is, the drivers in the system and system parameters.
/etc/system	A file containing the details on what drivers to configure in the system. This file is read by the boot program to determine how the system is to be configured.

INDEX