Writing MightyFrame™ Device Drivers

WRITING MIGHTYFRAME DEVICE DRIVERS

Specifications Subject to Change.

First Edition (March 1986) 09-00619-01

CONTENTS

LIST OF FIGURES

1 HOW TO USE THIS MANUAL

You can use this manual in two different ways:

- As a **quick reference guide**, if you are a CTIX or UNIX systems programmer. First, read all of Chapter 1, *How to Use This Manual*, and Chapter 2, *Architectural Information*, followed by all of Chapter 3, *Differences from System V*, Chapter 9, *Integrating the Driver*, and Chapter 10, *Debugging the CTIX Kernel*. Finally, use Appendix A, *CTIX Interface Manual Pages*, the *Table of Contents*, and the *Index* to find answers to your specific questions.

- As a **tutorial introduction** to CTIX and its I/O system, if you are not familiar with UNIX-like operating systems. First, read all of Chapter 1, *How to Use This Manual*, through Chapter 8, *Block Device Example*, and Appendix A, *CTIX Interface Manual Pages*, for an understanding of the MightyFrame and its software. Then read Chapter 9, *Integrating the Driver*, and Chapter 10, *Debugging the CTIX Kernel*, for specific help in implementing your driver. Finally, use Appendix A, *CTIX Interface Manual Pages*, and the *Glossary* to answer any questions you may have.

WHAT YOU NEED TO KNOW

This manual imposes several requirements upon you. First and most important, you must be an expert C programmer who has written one or more device drivers for a multitasking operating system. That is, you must be familiar with **all** of the following concepts:

- Cooperating sequential processes.

- Mutual exclusion.

- Interrupt processing, including programmable interrupt priority levels.
- Direct memory access techniques.
- Buses and bus timing diagrams.
- Programmable hardware (I/O) devices, including their operational and timing characteristics.

If you do not understand these concepts, you will have difficulty writing a CTIX device driver. The section entitled *Related Documentation* in this chapter describes several excellent texts on operating system principles. **You should read at least one of these books before attempting to work through the material in this manual.**

In addition, you should be familiar with either UNIX or CTIX internals. If you are not, your task will be very difficult, even though this manual contains a substantial amount of tutorial information.

NOTE

If you are not a UNIX systems programmer, you should take the System V internals and device drivers courses, which are offered periodically by AT&T.

MANUAL CONVENTIONS

This manual consistently follows a few conventions throughout.

- New terms are underlined when they are defined in the text. You will find all of these underlined terms in the Glossary at the back of the book.

- Bits within bytes, words, and longwords are numbered starting with 00 on the low-order end: bytes contain bits 07 to 00, words contain bits 15 to 00, and longwords contain bits 31 to 00. Bits 31 to 24 represent the high-order byte within a longword.
- All references of the form **name(N)** can be found in the *CTIX Operating System Manual*: when **N** is 1, the documentation is in *Volume 1*; when **N** is greater than 1, the documentation is in *Volume 2*, **except** when **N** is **2K**. All of the **2K** (kernel) calls are documented in Appendix A, *CTIX Interface Manual Pages*, in this manual.

MANUAL ORGANIZATION

Chapter 1, *How to Use This Manual*, contains introductory information about the background you must have, the conventions this manual follows, and the manual's contents.

- *What You Need to Know* describes the background, knowledge, and experience that you must bring to your reading of this manual.
- *Manual Conventions* describes the standards and conventions to which this manual adheres.
- *Manual Organization* is the section you are reading now. It gives a section-by-section overview of the contents of this manual.
- *Related Documentation* contains the titles and, where applicable, the authors and publishers of other documents you may find useful in writing your device driver.
  - *Convergent Technologies Publications* describes the relevant manuals that are published by and available from Convergent Technologies, Inc.
  - *Other Reference Manuals* describes other reference works that are not published by Convergent Technologies.

- — *Tutorial Books and Articles* describes background material that you may find useful in understanding operating system principles in general and the internal design of the UNIX operating system in particular.

Chapter 2, *Architectural Information*, provides detailed information about the MightyFrame hardware and software environments.

- *MightyFrame Hardware* describes the MightyFrame hardware configuration.
  - — *MC68020 Microprocessor* describes the Motorola 68020, the central processing unit of the MightyFrame.
  - — *Hardware Interrupts* describes the various interrupts that can occur in the system and how they are handled by the hardware.
- *CTIX Software* briefly describes the MightyFrame operating system.
  - — *Interrupt Processing* describes how interrupts are handled by the operating system.
    - * *Facilities to Handle Interrupts* describes the mechanisms that CTIX provides to receive and process interrupts.
    - * *Facilities to Manage the Interrupt Mask* describes the mechanism that CTIX provides to alter the MC68020 interrupt mask register.
  - — *MightyFrame Address Map* consists of a table containing all of the addresses referenced within this chapter.
  - — *Address Translation* describes the algorithms used to translate
    - * Virtual to physical memory addresses,

  * MightyFrame memory addresses to VMEbus addresses, and
  * VMEbus addresses to MightyFrame memory addresses.
  * *DMA Considerations* describes some limitations and special problems imposed during DMA operations between the VMEbus and the MightyFrame system.

- *VMEbus Support* describes the industry-standard VMEbus and how it interfaces with the rest of the system.

  * *VMEbus Interface Board* describes the interface point between the main system bus and the VMEbus.
  * *VMEbus Map (Page) Register* describes the VMEbus Map register on the VMEbus Interface board.
  * *VMEbus Protection Register* describes the VMEbus Protection register on the VMEbus Interface board.
  * *VMEbus Interrupt Mask Register* describes the VMEbus Interrupt Mask register on the VMEbus Interface board.
  * *VMEbus EEPROM* describes the electrically eraseable PROM on the VMEbus Interface board.

Chapter 3, *Differences from System V*, explains concisely how CTIX differs from AT&T's UNIX System V at the device driver level. If you have written one or more drivers under System V, this chapter provides the key to your rapid and painless transition to CTIX.

- *Loadable Drivers* describes the loadable device driver facility under CTIX and what you must do to make your driver loadable.

- *User-Kernel Virtual Address Remapping* documents the CTIX facilities that support physical I/O between user virtual memory and VMEbus DMA devices.
- *SPL(2K) Macros* describes the enhancements that CTIX has made to allow you efficient control over the interrupt mask in the MC68020 processor status word.
- *Kernel Debugging* briefly describes CTIX's unique facilities for debugging the kernel.

Chapter 4, *CTIX Kernel Tutorial,* contains tutorial information about the CTIX kernel and how it operates.

- *The User Process* describes the execution environment of a user's program.
  - *The Process Table* describes the per-process information that is kept in the System Process Table.
  - *The User Area* describes the information that is kept in the User Area of each process.
  - *Kernel Memory Map* shows a detailed map of the user's virtual memory during execution.
- *System Call Processing* describes the system call and return mechanism.
  - *System Call Examples* presents two detailed system call examples.
    - *Synchronous System Call Processing - setuid(2)* describes the flow of control through CTIX during synchronous system calls.
    - *Asynchronous System Call Processing - read(2)* describes the flow of control through CTIX during asynchronous system calls.
- *The CTIX I/O System* describes the CTIX I/O system in general.

— *The Block I/O System* briefly describes the Block I/O system and provides guidelines to help you determine whether yours is a block device.

— *The Character I/O System* briefly describes the Character I/O system and provides guidelines to help you determine whether yours is a character device.

  * *Character Queue Processing* briefly discusses the basic queue data structures available to character devices.

  * *Terminal Devices* briefly describes the special case handling for terminal devices.

  * *Buffered Character I/O* briefly describes techniques available to buffer high-speed character devices.

  * *Physical (Raw) I/O* briefly describes CTIX support for direct memory access between user processes and very high-speed character devices.

Chapter 5, *Character I/O Tutorial*, describes character I/O in detail.

- *Overview* explains the flow of control through the character I/O system and helps you to understand what CTIX is doing before, while, and after your driver runs.

- *Character-at-a-Time I/O* describes the most commonly used interface for low-speed character devices, such as terminals and printers.

  — *The Network Interface Driver* contains pseudocode for a hypothetical low-speed character device.

- *Physical (Raw) I/O* describes the most common interface for high-speed character devices, such as raw disk and tape drives.

  — *The Speech Interface Driver* contains pseudocode for a hypothetical high-speed character device driver.

Chapter 6, *Character Device Example*, contains the complete source code for a functional MightyFrame character device driver. The driver is heavily commented and each page of code includes a page of detailed narration.

Chapter 7, *Block I/O Tutorial*, describes block I/O in detail.

- *Overview* explains the flow of control through the block I/O system.
- *System Buffer Cache* contains a detailed description of the associative cache maintained by the Block I/O system.
  - *Basic Structure* provides an overview of the buffer cache.
  - *Available (Free) List* contains a detailed description of the list of available system buffers.
  - *Hash Lists* contains a detailed description of the hash lists, which reduce the time required to search the buffer cache.
  - *I/O Queues* contains a detailed description of the I/O queues, which contain all of the buffers scheduled for I/O operations.
  - *General Disk I/O Queue Structures* describes the special, two-level I/O queues used for general disk-type devices.
  - *Summary* reiterates the information presented in the preceding subsections.
- *General Disk Driver* describes the device-independent portion of the system disk drivers.
- *An SMD Device Driver* contains a pseudocoded driver for a hypothetical disk controller.
  - *Device Architecture* explains the operation of the hypothetical controller.

— *The Pseudocode Driver* contains the annotated pseudocode for the driver.

Chapter 8, *Block Device Example,* contains the complete source code for a functional MightyFrame block device driver. The driver is heavily commented and each page of code includes a page of detailed narration.

Chapter 9, *Integrating the Driver,* describes the steps you must take to write your driver, integrate it into the kernel, and test it.

- *If You Have a Source Code License* describes the steps you must perform to build and link your driver if you have a CTIX source code license.
    - *Getting Started* describes the various header files, makefiles, and shell scripts you will use to write your driver.
    - *Integrating the Driver* describes how to integrate your driver into the kernel.
        - *Compiling the Driver* explains how to build your driver from source code.
        - *Linking the Driver* explains how to rebuild the kernel so that it contains your driver.
- *If You Have a Binary License* describes the steps you must perform to build and link your driver if you have a CTIX binary license.
    - *Getting Started* describes the various header files, makefiles, and shell scripts you will use to write your driver.
    - *Integrating the Driver* describes how to integrate your driver into the kernel.
        - *Compiling the Driver* explains how to build your driver from source code.
        - *Linking the Driver* explains how to rebuild the kernel so that it contains your driver.

- *Making the Special File(s)* explains how to create the special files that CTIX needs to grant users access to your device.

- *Some Example Master(4) File Entries* contains annotated **master(4)** file entries for each of the example drivers in this manual.

Chapter 10, *Debugging the CTIX Kernel,* describes several utilities that make it easier for you to get your device driver running.

- *The Kernel Debugger* describes the debugging monitor that is built into the CTIX kernel.

- *The Qprintf(2K) Macros* describes the queued **printf()** function, which you may find useful in debugging your driver.

- *Interactive Boot Loader* describes CTIX's interactive boot loader and how you can use it to speed up the debugging process.

- *Other Kernel Debugging Tools* briefly describes the **adb(1)**, **sdb(1)**, and **crash(1M)** utilities and how you can use them to debug a running system or a system crash dump.

Appendix A, *CTIX Interface Manual Pages,* contains detailed descriptions of the CTIX operating system calls you must use to implement your driver. It is written in the same format and style as Sections 2 and 3 of the *CTIX Operating System Manual.* The number and type of parameters, the operation, and exit conditions are given for each function. All references of the form **function(2K)** are found in Appendix A.

- *Introduction* contains background information that underlies the information in the manual pages.

  — *Kernel Interface to Device Drivers* documents the linkage mechanism between the CTIX kernel and the various character and block device drivers.

  — *General Disk-Type Devices* describes the special facilities that CTIX provides to support disk-like devices.

  - *Buffer Header Structure* documents the buffer header data structure and the fields and flags it contains.
  - *User Structure* documents the user data structure and the fields and flags it contains.

- The manual pages themselves follow the Introduction.

The *Glossary* contains concise definitions of the important terms and concepts introduced in the manual.

The *Index* includes chapter and page number references for each important term in the manual.

RELATED DOCUMENTATION

The following books, manuals, and papers contain information useful or necessary for your understanding of this manual.

CONVERGENT TECHNOLOGIES PUBLICATIONS

- *CTIX Operating System Manual, Volumes 1 and 2*
- *MightyFrame Hardware Manual*
- *MightyFrame Administrator's Reference Manual*
- *CTIX Programmer's Guide*

The *CTIX Operating System Manual, Volume 1,* describes the commands available to the user and/or administrator of a CTIX system. *Volume 2* describes the system calls, library functions, file formats, miscellaneous facilities, games, and special files available to a CTIX user.

The *MightyFrame Hardware Manual* contains the hardware description of the MightyFrame computer system. It augments the material in Chapter 2, *Architectural Information.*

The *MightyFrame Administrator's Reference Manual* contains detailed information useful to the system administrator. In particular, it explains how to add a device driver to CTIX.

The *CTIX Programmer's Guide* describes the CTIX programming environment. It also documents the CTIX C compiler and the MC68020 assembler.

OTHER REFERENCE MANUALS

- *Ikon 10084 DR11-W Emulator Hardware Manual*
- *Interphase V/SMD 3200 User's Guide*
- *MC68020 32-Bit Microprocessor User's Manual*
- *VMEbus Specification Manual*
- *UNIX System V Support Tools Guide*

The *Ikon 10084 DR11-W Emulator Hardware Manual* is the programmer's manual for the device whose driver is documented in Chapter 6, *Character Device Example.*

The *Interphase V/SMD 3200 User's Guide* is the programmer's manual for the device whose driver is documented in Chapter 8, *Block Device Example.*

The *MC68020 32-Bit Microprocessor User's Manual* is the programmer's manual for the MightyFrame central processing unit. It describes the Motorola 68020 in detail. The information it contains on exception processing and interrupt priority levels is invaluable to you as a device driver writer.

The *VMEbus Specification Manual* contains hardware, software, and timing information related to the VMEbus, and the rules that its devices must follow. The manual was written from the hardware perspective, and in some cases does not clarify software issues. Nevertheless, you must master this material before you can implement a driver for a VMEbus device.

The *UNIX System V Support Tools Guide* contains a section that documents the link editor and the format of the system **ifile**.

TUTORIAL BOOKS AND ARTICLES

- Dijkstra, Edsger W. "The Structure of the 'THE' Multiprogramming Executive" in *Writings of the Revolution*, Yourdon, Inc., 1982.
- Deitel, Harvey M. *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, 1984.
- Kernighan, Brian W., and J. Mashey. "The UNIX Programming Environment" in *Software — Practice and Experience*, Vol. 9. Wiley and Sons, January, 1979.
- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., 1978.
- Ritchie, Dennis M. "A Retrospective" in *Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August 1978.
- Ritchie, Dennis M. "The Evolution of the UNIX Time-Sharing System" in *Bell System Technical Journal*, Vol. 63, No. 8, Part 2, October 1984.
- Thompson, Ken. "UNIX Implementation" in *Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July-August 1978.

Dijkstra's article, "The Structure of the 'THE' Multiprogramming Executive," introduced the formalism of semaphores to solve the mutual exclusion problem in operating systems. The idea was so revolutionary that the referees of the paper asked Dijkstra to write an Appendix justifying his exorbitant claims.

Deitel's book, *An Introduction to Operating Systems*, is exactly what its title implies. It covers all of the basic concepts required for operating system programming. It also has useful case studies in UNIX, VAX VMS, CP/M, OS/MVS, and OS/VM.

Kernighan and Mashey's article "The UNIX Programming Environment" provides a useful introduction to the UNIX operating system from the programmer's viewpoint.

Kernighan and Ritchie's book, *The C Programming Language*, is the definitive reference for the language, from the people who designed and implemented it.

Ritchie's articles, "A Retrospective" and "The Evolution of the UNIX Time-Sharing System," contain the musings of one of the original implementors of UNIX. The articles describe some of the advantages and disadvantages of the original design and how the system evolved into its present form.

Ken Thompson conceived, designed, and implemented the first UNIX system at Bell Labs in the early 1970's. His article, "UNIX Implementation," describes the internal architecture of UNIX. It is lucid in its explanation and, even now, is remarkably useful.

2 ARCHITECTURAL INFORMATION

This chapter contains specific information about the MightyFrame hardware and software environments. It describes the MC68020 CPU, the VMEbus, and the interface between them. It also discusses the CTIX operating system and the facilities it provides for interrupt handling and address translation.

> **NOTE**
>
> The hardware information presented in this chapter is specific to the MightyFrame I. Always refer to the *Mightyframe Hardware Manual* for the most current MightyFrame hardware information.

MIGHTYFRAME HARDWARE

The MightyFrame system is a multiuser, virtual memory computer based on the 32-bit MC68020 microprocessor. The heart of the system is the Main Processor board containing

- The CPU running at 12.5 MHz with one wait state.
- An optional 68881 floating point coprocessor.
- Memory mapping and protection circuitry.
- 1M byte of RAM.
- 32K bytes of ROM.
- One direct memory access hard disk controller supporting up to three ST506 Winchester disk drives.

- One DMA 1/4 inch tape controller.
- Two interrupt-driven RS-232-C serial channels.
- One Centronics-compatible parallel line printer port.
- One battery powered, real-time clock/calendar chip.
- One uninterruptible power supply connector.

Associated with the main processor board is a 10 slot memory and I/O expansion bus that can support

- 16M bytes of physical memory in 2 or 4M byte increments.
- Either 10, 20, 30, or 40 asynchronous RS-232-C serial ports running at up to 38,400 baud.
- One of the following:
  - An I/O Processor board (IOP) consisting of a 12 MHz 68000, an 8253 counter timer circuit, a parallel line printer port, and 64K bytes of memory.
  - An RS-422 board consisting of four RS-422 ports running at either 307K bits or 2M bits per second, and one parallel printer port.
- One VMEbus interface board that provides the electrical interface to a four-slot VMEbus expansion card cage. An additional 10-slot VMEbus expansion card cage can be added to the first.

See the *MightyFrame Hardware Manual* for a more detailed discussion of this material.

MC68020 MICROPROCESSOR

The MC68020 is a 32-bit microprocessor that supports demand-paged virtual memory. It is the first of the 68000 family to implement nonmultiplexed, 32-bit address and data buses both internally and externally. It is upwardly compatible (at the object code level) with the earlier members of the Motorola 68000 family, that is, with the 68000, 68008, 68010 (used in Convergent Technologies MegaFrame and MiniFrame Systems), and 68012 microprocessors.

The MC68020 has the following features:

- A three-stage instruction prefetch and decode queue.
- An on-chip cache memory containing 64 longwords.
- Separate user and supervisor states, each supported by its own 32-bit stack pointer.
- Separate master and interrupt states with a third 32-bit stack pointer reserved for the interrupt state.
- A 4 gigabyte direct addressing range.
- Instruction suspension and continuation to support demand-paged virtual memory.
- Dynamic bus sizing on a cycle-by-cycle basis to support 8, 16, and 32-bit wide memory and peripheral devices.
- Powerful exception processing controlled through a relocatable, 256 entry vector table. The first 64 exceptions are defined by and reserved for the CPU. The remaining 192 vectors are available to the system architect.
- Seven levels of interrupt prioritization, with full masking on the lowest six levels. The highest level is defined as the Non-Maskable Interrupt ($\overline{\text{NMI}}$) and is used to report impending power failure, memory not present, bus errors, and parity errors to the operating system.

See the *MC68020 32-Bit Microprocessor User's Manual* for further information about these and other features of the CPU.

HARDWARE INTERRUPTS

The MC68020 supports seven levels of prioritized interrupts. The Interrupt Priority Levels (IPLs) are numbered from 1 to 7, with 1 having the lowest priority and 7 the highest. IPL 0 indicates that the CPU is not processing (has not acknowledged) any interrupts currently. Using a 3-bit field in the status word, the programmer can mask out (disable) interrupts at any level except 7. IPL 7 is referred to as the NMI and is always acknowledged. Multiple devices may be daisy-chained together at the same IPL, effectively supporting an unlimited number of interrupt requesters.

The running priority level is determined by the contents of the interrupt mask in the processor status word. For interrupts at levels 1 through 6 to be acknowledged, the IPL must be greater than the current interrupt mask in the status word. If the IPL is less than or equal to this mask, the interrupt is denied. Interrupts at IPL 7 are recognized at all times (except for level 7 interrupts from the VMEbus).

MightyFrame Interrupt Priority Levels are assigned according to the following table:

IPL 7	Main System NMI.
	Bus Errors.
	Parity Errors.
	Memory Not Present.
IPL 6	Clock Tick (60 Hz).
	VME Subsystem Level 6.
IPL 5	On-board or Expansion RS-232.
	Either off-board RS-232 or an I/O Processor board.
	VME Subsystem Level 5.

IPL 4	VME Subsystem Level 4.
IPL 3	On-board 8259 Interrupt controller (manages disk and tape drives).
	VME Subsystem Level 3.
IPL 2	VME Subsystem Level 2.
IPL 1	On-board Printer Interface.
	Either RS-422 or an I/O Processor Board.
	VME Subsystem Level 1.

The interrupt control circuitry receives interrupt requests from local (non-VMEbus) devices at IPLs 1, 3, 5, 6, and 7; it receives VMEbus requests at priority levels 1 through 6. VMEbus interrupts at level 7 are ignored. When a local and a VMEbus device at the same IPL request interrupts simultaneously, the local device is serviced before the VMEbus device.

NOTE

VMEbus interrupt requests can be disabled separately from other interrupt sources. See *VMEbus Support* in this chapter for details.

See the *MightyFrame Hardware Manual* and the *MC68020 32-Bit Microprocessor User's Manual* for more information on hardware interrupts.

CTIX SOFTWARE

The CTIX operating system is derived from AT&T's UNIX System V, Release 2. That is, it is a virtual memory implementation of the System V user and programmer interfaces. In addition, certain features of 4.2 Berkeley Software Distribution (BSD) have been incorporated. In particular, the interprocess communication facility known as **sockets** has been partially implemented under CTIX.

The following sections discuss the features of CTIX software you must understand when writing your device driver.

INTERRUPT PROCESSING

Your driver must handle interrupts from the device and manage the processor interrupt mask. CTIX provides simple facilities to do this.

Facilities to Handle Interrupts

All interrupts from peripheral devices are vectored directly to a CTIX assembly language routine named **perint()**. **Perint()** saves the context and acquires the interrupt vector number from the stack. Using this number as an index into a kernel table named **Int_handle**, **perint()** retrieves the address of the appropriate interrupt handler and calls it with the interrupt vector number as a parameter. The interrupt handler runs to completion and then returns to **perint()**, which cleans up the interrupt stack and returns from the exception.

In order for **perint()** to pass control to your driver, you must place the address of your interrupt handler in the appropriate slot in the **Int_handle** table. The CTIX OS provides two routines to accomplish this: **get_vec(2K)** and **set_vec(2K)**.

- If your VMEbus device has software-programmable interrupt vector generation, you should call **get_vec(2K)** in your driver initialization code (see **devinit(2K)**). **Get_vec(2K)** takes as parameters your driver ID and the address of your interrupt handler. It returns the interrupt vector number corresponding to the first available slot in the **Int_handle** table. You must program your device to generate this vector number when its interrupt request is acknowledged.

- If your VMEbus device supports only hardware-strappable interrupt vector generation, you must use **set_vec(2K)**. **Set_vec(2K)** takes as parameters your driver ID, the address of your interrupt handler, and the interrupt vector number for which your device is strapped. The **set_vec(2K)** manual page contains a list of the (currently) available interrupt vectors.

  If the slot corresponding to the requested vector number in the **Int_handle** table is in use, **set_vec(2K)** returns a failure indication. This means that another device in the system is supplying the same interrupt vector number as your VMEbus device. When your device generates an interrupt, the interrupt handler for the other device will be called. Therefore, if **set_vec(2K)** fails, you should print a message on the system log to that effect (at the very least). In this case, you must take the machine down and restrap your device to generate an unused vector number.

For a more complete discussion of this topic, see the documentation for **devinit(2K), get_vec(2K)** and **set_vec(2K)** in Appendix A, *CTIX Interface Manual Pages*. Also see Chapter 9, *Integrating the Driver.*

Facilities to Manage the Interrupt Mask

In order to guarantee that each critical region of your code runs without interruption, you must raise and lower the processor priority level from within your driver. CTIX software provides various **SPL(2K)** (set priority level) requests to do this.

SPL0(2K) through **SPL7(2K)** set the interrupt mask explicitly. For instance, after a call to **SPL5()**, all interrupts at IPL 5 and below are disabled. After calling a function of the form **SPLn()**, you may call **SPLX()** to restore the interrupt mask to the value that it had before you changed it. Whenever you use the **SPLn** / **SPLX** pair, you must include a declaration of the form **SDEC;** within the local variables of your function. This macro declares local storage for a temporary copy of the status register.

> **NOTE**
>
> All of the uppercase SPL calls are in fact macros that generate in-line assembly language. The traditional UNIX OS functions named **spl0()** through **spl7()** also are supported. The macros are preferred for performance reasons.

VSPL0(2K) through **VSPL7(2K)** also set the interrupt mask explicitly, but they do not store the previous value of the processor status word. If you use a call of the form **VSPLn()**, you cannot later call **SPLX()**. Consequently, you should not include the **SDEC;** declaration.

In addition, CTIX provides the following mnemonic calls, making it unnecessary to hard-code explicit interrupt levels in your device driver. These macros are defined in **<sys/spl.h>**.

SPLDSK sets the interrupt mask to the appropriate level for all system disk drivers (currently, **SPL3**).

SPL422 sets the interrupt mask to the appropriate level for RS-422 devices (currently, **SPL3**).

SPLTAPE sets the interrupt mask to the appropriate level for the tape subsystem (currently, **SPL3**).

SPLSERIAL sets the interrupt mask to the appropriate level for all serial devices in the system (currently, **SPL5**).

SPLBLK sets the interrupt mask to a level guaranteed to be greater than or equal to the highest interrupt level of any block device in the system (currently, **SPL3**).

In general, you must raise the interrupt level whenever you are manipulating data structures that can also be changed by the interrupt handler: queue, c-list, and buffer manipulation are the most common examples of this. You must be careful not to raise the level too high, however, since this prevents CTIX from servicing devices that do not conflict with your driver. Such denial of service unnecessarily increases the interrupt response time for the affected device(s). This causes a needless degradation in system performance.

See the **SPL(2K)** manual page in Appendix A, *CTIX Interface Manual Pages*, for more information on managing interrupt priority levels.

MIGHTYFRAME ADDRESS MAP

The address space of the MightyFrame system is organized as shown in the following map.

Address Range	Contents
$00000000 - $017FFFFF	User virtual memory
$01800000 - $7F7FFFFF	Illegal
$7F800000 - $7FFFFFFF	Kernel virtual memory
$80000000 - $8FFFFFFF	Slow local I/O registers
$90000000 - $9FFFFFFF	Fast local I/O registers
$98000000	VMEbus Map (Page) register
$9A000000	VMEbus Protection register
$9C000000	VMEbus Interrupt Mask register
$9E000000	VMEbus EEPROM (8K bytes)
$A0000000 - $C1FFFFFF	VMEbus addresses
$A0000000 - $BFFFFFFF	A32 devices
$A0000000 - $AFFFFFFF	A32 - supervisor mode
$B0000000 - $B3FFFFFF	A32 - user mode
$B4000000 - $B7FFFFFF	A32 - user mode
$B8000000 - $BBFFFFFF	A32 - user mode
$BC000000 - $BFFFFFFF	A32 - user mode
$C0000000 - $C0FFFFFF	A24 devices
$C0000000 - $C0BFFFFF	A24 - supervisor mode
$C0C00000 - $C0DFFFFF	A24 - user mode
$C0E00000 - $C0FFFFFF	A24 - user mode
$C1000000 - $C100FFFF	A16 devices
$C1000000 - $C1007FFF	A16 - supervisor mode
$C1008000 - $C100BFFF	A16 - user mode
$C100C000 - $C100FFFF	A16 - user mode
$C1010000 - $C1FFFFFF	Unused
$C2000000 - $DFFFFFFF	Reserved
$E0000000 - $FFFFFFFF	Unused

MightyFrame System Address Map

Each of these addresses and address ranges will be explained in the following sections.

ADDRESS TRANSLATION

The table in the preceding section shows that the MightyFrame System supports a total of 32M bytes of virtual memory: 24M bytes of user space and 8M bytes of kernel space. (Addresses above $80000000 are in I/O space and are not considered a part of virtual memory.) DMA addresses, however, are contiguous from $00000000 to $01FFFFFF: DMA hardware ignores the upper 7 bits of virtual addresses, effectively remapping the kernel space (addresses between $7F800000 and $7FFFFFFF) onto addresses between $01800000 and $01FFFFFF.

Virtual Memory Address Translation

Virtual memory is divided into 8,192 logical pages of 4,096 bytes per page (8,192 pages X 4,096 bytes per page = 32M bytes). Physical memory is divided into a maximum of 4,096 pages of 4,096 bytes per page (4,096 pages X 4,096 bytes per page = 16M bytes). Virtual to physical address translation takes place according to the following steps:

1. The high-order 7 bits (bits 31 to 25) of the 32-bit virtual address are the **K/U bits**. They determine whether the access is to kernel or user memory space.

2. The low-order 25 bits (bits 24 to 00) of the 32-bit virtual address are used to perform the translation.

   - The low-order 12 bits of this portion (bits 11 to 00) form the **byte offset** into the 4K byte physical page.

   - The high-order 13 bits (bits 24 to 12) contain the **virtual page #**. This is used as an index into an array of 8,192 **mapping registers**.

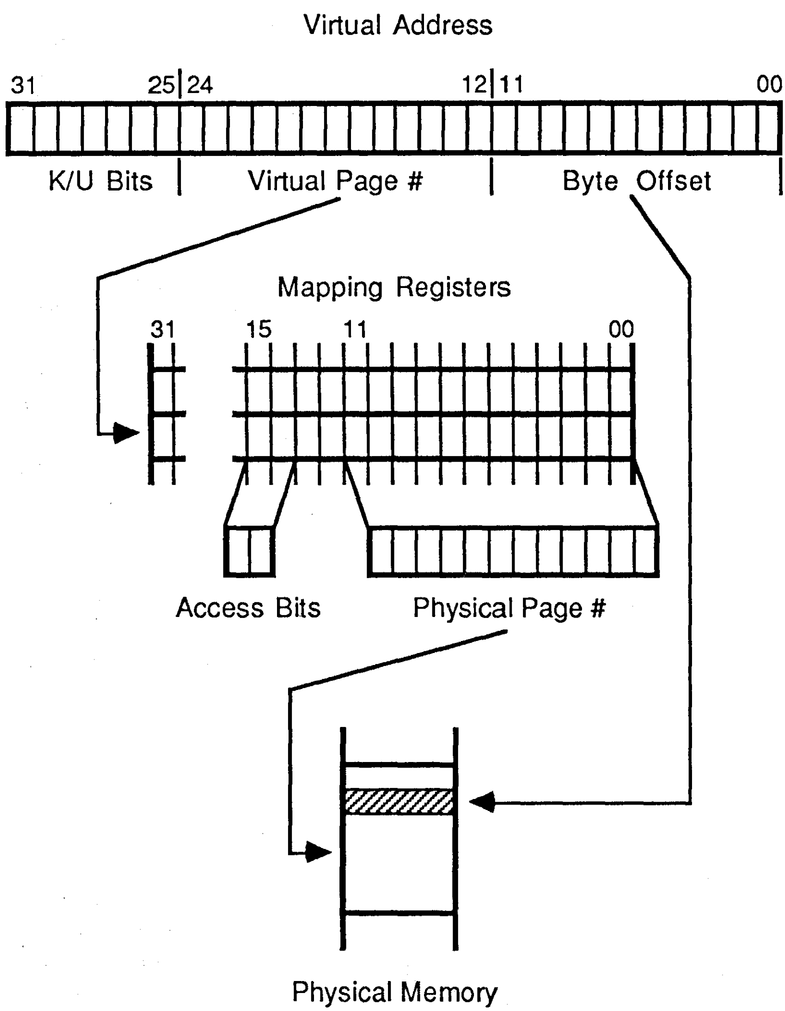
3. Two **access bits** are retrieved from the selected **mapping register**, and the access permissions are validated. These bits are used to differentiate among the following possible conditions and permissions:

- The page is absent (it may be on disk).
- The page is present. The supervisor can read and write the page; the user can neither read nor write it.
- The page is present. The supervisor can read and write the page; the user can read the page but cannot write it.
- The page is present. The supervisor can read and write the page; the user can read and write it.

If the page is absent or the process does not have the proper access permission, a page fault is generated, and appropriate processing is begun. If the page was absent, a read request is issued to the swapper process. If the user did not have proper access permission, the process is terminated.

4. If the page is present, a 12-bit **physical page #** is retrieved from the **mapping register**.
5. The 12-bit **physical page #** and the 12-bit **byte offset** (bits 11 to 00 of the **virtual address**) are presented to the memory address decoding circuitry.
6. The address decoding circuitry accesses one byte, word, or longword of **Physical Memory**.

The following diagram illustrates the translation process.



Virtual to Physical Address Translation

MightyFrame/VMEbus Address Translation

Address translation for VMEbus devices takes place in two different ways, depending upon whether the MightyFrame system is accessing the device, or the device is accessing the MightyFrame while performing DMA. The following sections consider address translation from the MightyFrame system to the VMEbus and from the VMEbus to the MightyFrame.

MightyFrame System to VMEbus Addressing

When you access a VMEbus device from the MightyFrame system, the device appears between virtual addresses $A0000000 and $C1FFFFFF. The MightyFrame System Address Map presented previously shows the VMEbus space. This address range provides the MightyFrame with a 544M byte ($22000000) "window" into the 4 gigabyte VMEbus address space. The window is further subdivided to provide support for the A16, A24, and A32 VMEbus device domains. The address map below describes the domains:

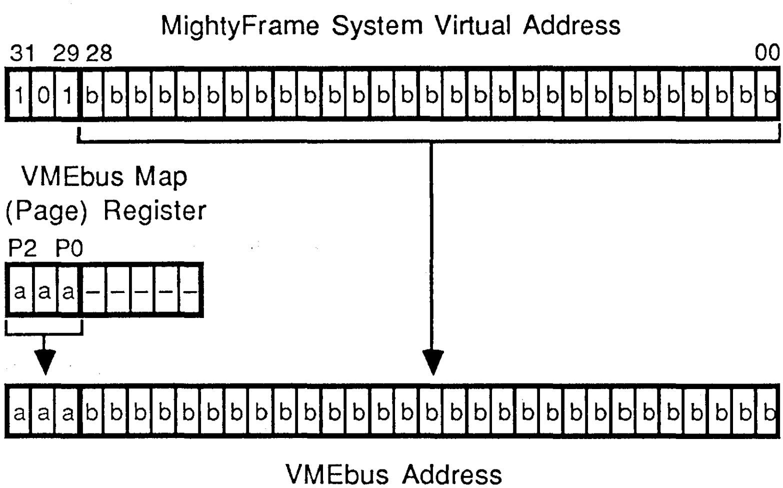
MightyFrame Addresses	VMEbus Domain
$A0000000 - $BFFFFFFF	A32 devices
$C0000000 - $C0FFFFFF	A24 devices
$C1000000 - $C100FFFF	A16 devices
$C1010000 - $C1FFFFFF	Unused

VMEbus Domain Map

Within the A32 domain, the hardware concatenates bits P2 to P0 from the VMEbus Map register (located at virtual address $98000000) onto the low-order 29 bits of the I/O address to provide a full, 32-bit VMEbus address. For example, if bits P2 to P0 of the VMEbus Map register are 001, and the I/O address is $A0000000, the VMEbus address is $20000000 within the A32 device domain. The complete contents of the VMEbus

Map register are defined in *VMEbus Map register,* later in this chapter.

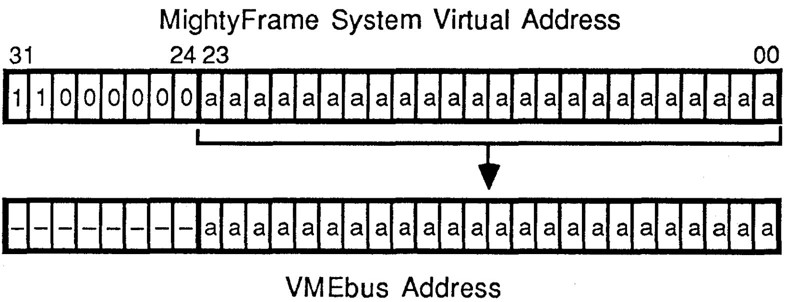
The following diagram illustrates MightyFrame to VMEbus address translation for A32 devices.



MightyFrame to VMEbus Address Translation for A32 Devices

For A24 devices, MightyFrame system addresses $C0000000 to C0FFFFFF are used. This block of virtual memory corresponds to VMEbus addresses $000000 to $FFFFFF.

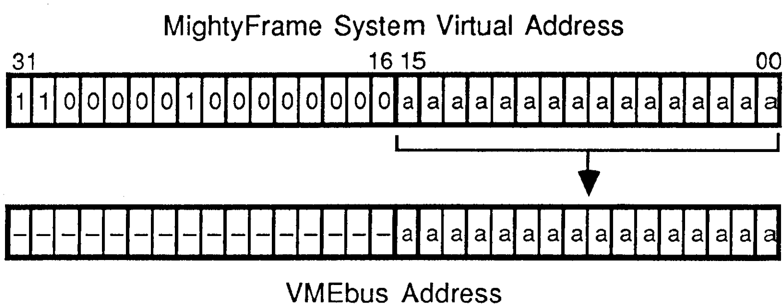
The following diagram illustrates MightyFrame to VMEbus address translation for A24 devices.



MightyFrame to VMEbus Address Translation for A24 Devices

For A16 devices, MightyFrame addresses $C1000000 to $C100FFFF are used. This block of virtual memory corresponds to VMEbus addresses $0000 to $FFFF.

The following diagram illustrates MightyFrame to VMEbus address translation for A16 devices.



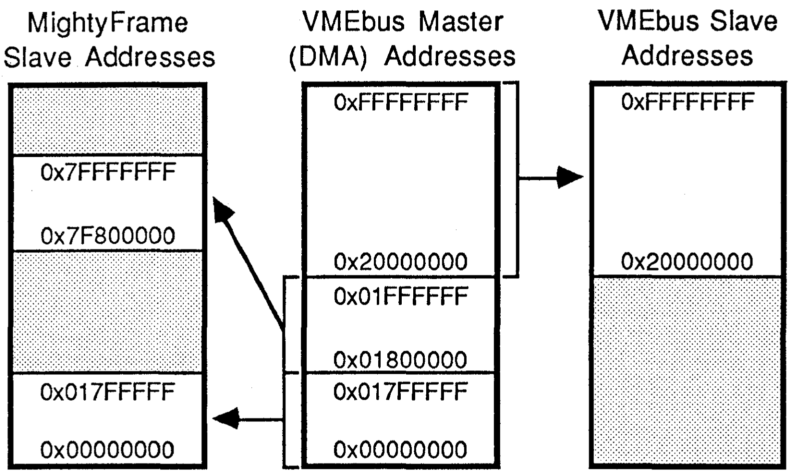
MightyFrame to VMEbus Address Translation for A16 Devices

See the *MightyFrame Hardware Manual, VMEbus Support* in this chapter, and the include files <**sys/hardware.h**> and <**sys/vme.h**> for more information about MightyFrame system to VMEbus address translation.

VMEbus to MightyFrame System Addressing

When the VMEbus device is acting as the master (that is, when it is performing DMA), DMA addresses are interpreted according to the address type of the VMEbus device. The following diagrams and tables show the address translations for A32, A24, and A16 VMEbus devices. The symbols **M** and **V** in the tables stand for MightyFrame system and VMEbus address spaces, respectively. The sections of the address space that are filled in halftone are unreachable from the device in question.

The following diagram and table illustrate that A32 devices can perform DMA transfers in MightyFrame user and kernel memory spaces, and also in most of the VMEbus address space.



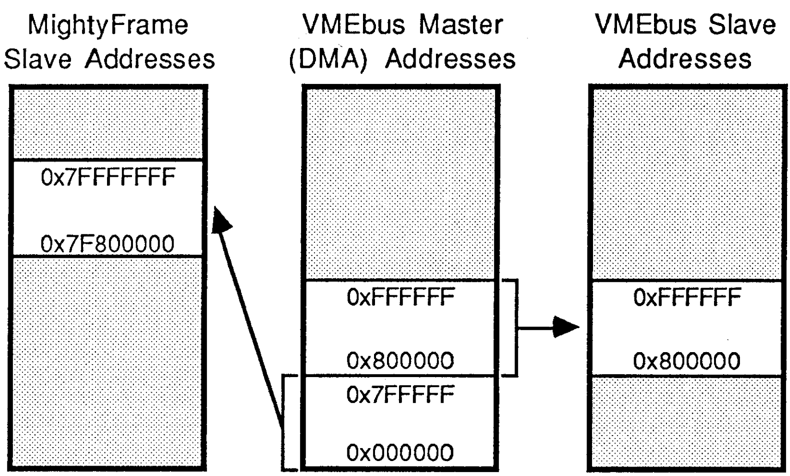
VMEbus Master (DMA) Address Translation for A32 Devices

Device Address	Translated Address	
\$00000000 - \$017FFFFF	\$00000000 - \$017FFFFF	M
\$01800000 - \$01FFFFFF	\$7F800000 - \$7FFFFFFF	M
\$02000000 - \$FFFFFFFF	\$02000000 - \$FFFFFFFF	V

NOTE

You are not allowed to perform DMA operations with user virtual memory addresses, even though the hardware supports it for A32 devices. When your driver performs DMA directly into or out of a user's buffer, you are responsible for remapping the buffer from user virtual space to kernel virtual space. For a complete discussion of the proper technique, see *Physical (Raw) I/O* in Chapter 5, *Character I/O Tutorial.* Also see **sptalloc(2K)**, **physio(2K)**, and **setmap(2K)** in Appendix A, *CTIX Interface Manual Pages.* Also, the driver documented in Chapter 6, *Character Device Example*, illustrates the use of physical I/O.

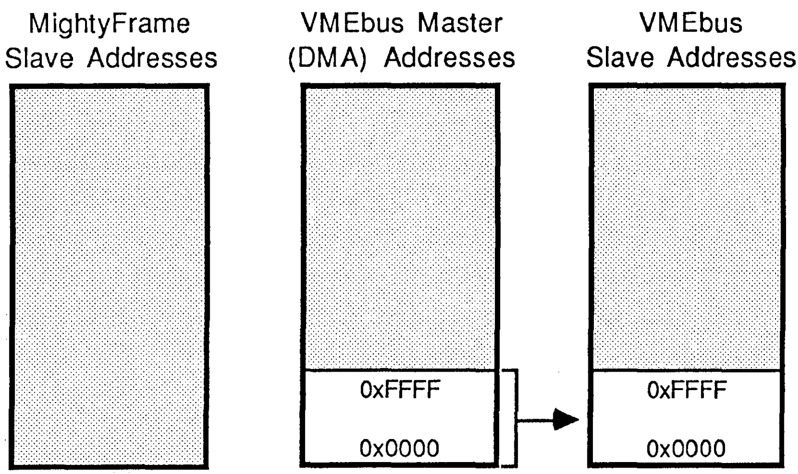
The following diagram and table illustrate that A24 devices can perform DMA transfers in MightyFrame kernel memory space, and also in some parts of the VMEbus address space. The hardware does not allow A24 devices to perform DMA transfers in MightyFrame user space, but, in practice, it is illegal to do so anyway. See the note above, under the discussion of A32 devices, for more information.



VMEbus Master (DMA) Address Translation for A24 Devices

Device Address	Translated Address	
\$000000 - \$7FFFFF	\$7F800000 - \$7FFFFFFF	M
\$800000 - \$FFFFFF	\$800000 - \$FFFFFF	V

The following diagram and table illustrate that A16 devices can perform DMA transfers in VMEbus address space only. A16 devices cannot access MightyFrame user or kernel memory spaces; therefore, their usefulness is extremely limited in the MightyFrame system.



VMEbus Master (DMA) Address Translation for A16 Devices

Device Address	Translated Address	
\$0000 - \$FFFF	\$0000 - \$FFFF	V

Whenever the VMEbus device is transferring data to MightyFrame system memory, the memory management circuitry on the Main Processor board is used. See *VMEbus Support* in this chapter for more information.

DMA Considerations

If a page fault occurs on a DMA transfer, an NMI is generated. This results in a **panic(2K)** call displaying the following message:

Page fault during DMA

Thus, page faults on DMA transfers always cause a system crash.

When performing physical I/O, your driver is responsible for bringing all of the user's buffer pages into physical memory and assigning kernel virtual addresses to them. The kernel routines **sptalloc(2K)**, **physio(2K)**, and **setmap(2K)** perform these services. See Appendix A, *CTIX Interface Manual Pages*, for a complete description of these functions.

CAUTION

The memory protection mechanism is disabled on DMA accesses; VMEbus DMA devices are allowed to read or write any page of MightyFrame system memory that they can address. You must thoroughly debug your DMA-based device drivers, since they can overwrite kernel data structures and cause catastrophic system failures.

VMEBUS SUPPORT

The VMEbus is an industry-standard, 32-bit bus that is available as an option on the MightyFrame system. The bus is connected to the processor through an interface board residing in the memory and I/O expansion chassis. This board provides all of the system control signals for the VMEbus and operates at a bandwidth of approximately 6M bytes per second with the 12.5

bandwidth of approximately 6M bytes per second with the 12.5 MHz CPU.

Local DMA devices in the MightyFrame system can read and write local MightyFrame system memory, but they cannot access VMEbus memory. On the other hand, A32 and A24 VMEbus DMA devices can read and write MightyFrame virtual memory. So DMA works in one direction only: from the VMEbus to the MightyFrame. You cannot use DMA to transfer data from the MightyFrame to the VMEbus.

> **NOTE**
>
> Virtual memory translation is performed for VMEbus DMA devices, but the normal memory protection scheme is disabled. See *Address Translation* in this chapter for more information.

VMEbus Interface Board

You can access three registers and an EEPROM (electrically eraseable PROM) on the VMEbus interface board. They are defined in the following table:

Virtual Address	Contents
$98000000	VMEbus Map (Page) register
$9A000000	VMEbus Protection register
$9C000000	VMEbus Interrupt Mask register
$9E000000	VMEbus EEPROM (8K bytes)

VMEbus Interface Board Registers

You can read the contents of any of these registers by opening one of the **/dev/vme/*** special files and issuing an **ioctl(2)** call. Only the super user can change the contents of the registers. The contents of the VMEbus Protection register (only) are saved and loaded at context switch time so that user access to VMEbus space is maintained on a per-process basis. See **VME(7)** in the *CTIX Operating System Manual, Volume 2,* for more information.

VMEbus Map (Page) Register

The VMEbus Map (Page) register is shown and described below:

07	06	05	04	03	02	01	00
P2	P1	P0	ACF	CT	CT	CT	CT

VMEbus Map (Page) Register

Bits	Function
07-05	**P2-P0** are concatenated onto the low-order 29 bits of the I/O address to form a 32-bit VMEbus address for A32 devices. You can think of this field as the VMEbus page number.
04-04	If the **ACF** bit is set (= 1), the AC (Power) Fail signal is enabled to the VMEbus. If the **ACF** bit is clear (= 0), the AC Fail signal is not delivered to the VMEbus.
03-00	Bits marked **CT** are reserved for Convergent Technologies.

VMEbus Protection Register

For some of the peripherals available on the VMEbus, it is advantageous to allow user processes more direct control than a device driver provides. The CTIX operating system has made provision for this by allowing up to one half of the VMEbus space to be accessible in user mode. This facility is implemented through the VMEbus Protection register.

The VMEbus Protection register is shown and described below:

07	06	05	04	03	02	01	00
C1	C0	B1	B0	A3	A2	A1	A0

VMEbus Protection Register

<u>Bits</u> <u>Function</u>

07-06 **C1-C0** are used with A16 VMEbus devices only. They define the areas of the VMEbus space that are accessible in user mode. The bits and the regions they control are defined below:

- **C0 = 1** $C1008000 to $C100BFFF is accessible in user mode.
- **C1 = 1** $C100C000 to $C100FFFF is accessible in user mode.

05-04 **B1-B0** are used with A24 VMEbus devices only. They define the areas of the VMEbus space that are accessible in user mode. The bits and the regions they control are defined below:

- **B0 = 1** $C0C00000 to $C0DFFFFF is accessible in user mode.
- **B1 = 1** $C0E00000 to $C0FFFFFF is accessible in user mode.

03-00 **A3-A0** are used with A32 VMEbus devices only. They define the areas of the VMEbus space that are accessible in user mode. The bits and the regions they control are defined below:

AO = 1 $B0000000 to $B3FFFFFF is accessible in user mode.

A1 = 1 $B4000000 to $B7FFFFFF is accessible in user mode.

A2 = 1 $B8000000 to $BBFFFFFF is accessible in user mode.

A3 = 1 $BC000000 to $BFFFFFFF is accessible in user mode.

The following memory map summarizes the information above.

Address Range	Contents
$A0000000 - $AFFFFFFF	A32 - supervisor
$B0000000 - $B3FFFFFF	A32 - user
$B4000000 - $B7FFFFFF	A32 - user
$B8000000 - $BBFFFFFF	A32 - user
$BC000000 - $BFFFFFFF	A32 - user
$C0000000 - $C0BFFFFF	A24 - supervisor
$C0C00000 - $C0DFFFFF	A24 - user
$C0E00000 - $C0FFFFFF	A24 - user
$C1000000 - $C1007FFF	A16 - supervisor
$C1008000 - $C100BFFF	A16 - user
$C100C000 - $C100FFFF	A16 - user
$C1010000 - $C1FFFFFF	Unused

VMEbus Access Permissions Map

> **NOTE**
>
> The hardware supports direct user access to VMEbus devices, but this facility imposes one severe restriction. Your process can read and write VMEbus device registers, but it cannot cause a VMEbus device to perform DMA transfers in your process memory space. DMA transfers into and out of user space must be performed by the kernel, usually by the **physio(2K)** routine. See *Physical (Raw) I/O* in Chapter 5, *Character I/O Tutorial,* for more information.

VMEbus Interrupt Mask Register

The VMEbus Interrupt Mask Register is described below:

07	06	05	04	03	02	01	00
00	M6	M5	M4	M3	M2	M1	CT

VMEbus Interrupt Mask Register

Bits	Function
07-07	VMEbus interrupts at level 7 are always disabled.
06-01	Setting any of the **Mn** bits (= 1) masks out VMEbus interrupts at the corresponding level. Unlike the interrupt mask in the CPU, each VMEbus interrupt level is independent of the others.
00-00	This bit is currently unused.

CTIX software initializes the VMEbus Interrupt Mask register such that VMEbus interrupt level 7 is masked off, and levels 1 through 6 are on.

VMEbus EEPROM

The VMEbus Interface board contains an EEPROM that is located at $9E000000 and is 8K bytes long. The following restrictions apply to this device:

- The MC68020 cache memory must be disabled in order to execute from the EEPROM.
- If you write to the EEPROM, you must allow a minimum of 10 milliseconds to elapse before the next read or write access to the device. Generally, you should use the **ldeeprom(1M)** command to alter the EEPROM.

For more information about the VMEbus interface, see *Address Translation* in this chapter, Appendix A, *CTIX Interface Manual Pages*, the *MightyFrame Hardware Manual*, the include files **<sys/hardware.h>** and **<sys/vme.h>**, and the *VMEbus Specification Manual.*

3 DIFFERENCES FROM SYSTEM V

From the point of view of the device driver writer, there is very little difference between AT&T's UNIX System V Release 2 and the CTIX operating system on the MightyFrame. CTIX is derived from and is a superset of the System V software, so wherever differences do exist, they are enhancements available only under CTIX. This chapter documents the CTIX enhancements that affect device drivers and explains what you must do to take advantage of them.

LOADABLE DRIVERS

The most fundamental change from UNIX System V is CTIX's provision for loadable device drivers. Under System V software, you **must** compile and link your driver with the kernel in order for it to execute. Under CTIX, you can still link your driver with the kernel; however, you can also install it into a running system using the **lddrv(1M)** utility.

Using options available with the **syslocal(2)** call, **lddrv(1M)** allocates kernel memory space to hold the driver, loads the code into the kernel, patches the kernel **bdevsw** or **cdevsw** tables with the driver's entry points, and then executes the driver's **devinit(2K)** routine. From this point until the driver is unbound, it runs exactly as though it had been linked with the kernel.

> **NOTE**
>
> Because **syslocal(2)** currently does not patch the **gdsw** table, drivers for general disk-type devices are not loadable.

Whether they are to be loaded with **lddrv(1M)** or linked into the kernel, all device drivers under CTIX must have a driver ID assigned. To accomplish this, include the following lines of code in your driver:

```
extern int DFLT_ID;
static int Drv_id = (int)&DFLT_ID;
```

The loader assigns a driver ID of 0 for all device drivers that are linked with the kernel. If you use **lddrv(1M)** to load your driver, **syslocal(2)** assigns a unique driver ID when it performs the BIND operation.

DRIVER RELEASE ROUTINE

In order to be unbound by **lddrv(2)**, a loadable driver must contain a **devrelease(2K)** routine. The primary responsibilities of this routine are

- To ensure that the device is not in use.
- To disable interrupts from the device.
- To cancel any **timeout(2K)** requests from the driver that are still active.

- To return to the system any resources that the driver acquired when it was bound.

If your driver does not have a **devrelease(2K)** routine, the **syslocal(2)** call to unbind the driver will fail with **EBUSY**. In this case, you must reboot CTIX to deallocate your driver.

USER-KERNEL VIRTUAL ADDRESS REMAPPING

Most transfers are done into or out of kernel memory; for instance, block reads from a file system are done into buffers in the system buffer pool. In some cases (for example, to achieve better performance) it is desirable to support transfers directly between the device and user memory. Under the CTIX operating system, such transfers are known as **physical** (or **raw**) I/O. These transfers are performed with DMA hardware.

If your driver supports physical I/O, you must make provision for the fact that DMA devices are not allowed to reference user virtual addresses, since they depend upon page table entries that change whenever a context switch occurs. Instead, your driver must acquire kernel virtual memory and "remap" it to point to the same physical memory referenced by the user's page table entries. In effect, this gives one buffer in physical memory two virtual addresses, a user virtual address (which is valid only when the original user process is running) and a kernel virtual address (which is always valid).

Both the UNIX System V and CTIX operating systems provide the **sptalloc(2K)** function to allocate kernel virtual memory (that is, page table entries). The CTIX operating system also provides the **setmap(2K)** function to copy the page frame numbers from the user's page table entries to the kernel's. This in effect makes two sets of page table entries point to the same buffer in physical memory. Both Chapter 6, *Character Device Example*, and Chapter 8, *Block Device Example*, contain examples of the use of **setmap()**.

SPL(2K) MACROS

UNIX System V provides several functions to control the current running priority level of the CPU. In order to eliminate the overhead of the subroutine call/return mechanism, CTIX software augments these functions with macros that generate in-line assembly language. See **SPL(2K)** in Appendix A, *CTIX Interface Manual Pages*, for more information.

KERNEL DEBUGGING

The CTIX OS provides a built-in debugger that runs as a device driver under the kernel. This utility makes it possible to set breakpoints in and single-step through the kernel. It provides sophisticated control over debugging output produced by the kernel **printf(2K)** function. Using the **qprintf(2K)** macros, you can implement multiple levels of output, and then enable and disable each level selectively through the debugger. See the **qprintf(2K)** documentation in Appendix A, *CTIX Interface Manual Pages*, and Chapter 10, *Debugging the CTIX Kernel*, for more information.

Unlike UNIX System V, CTIX includes an interactive bootstrap loader that allows you to boot from any file on any device in the system. This interactive boot loader can greatly reduce the time it takes to test and debug your device driver. See *Interactive Boot Loader* in Chapter 10, *Debugging the CTIX Kernel*, for a complete description.

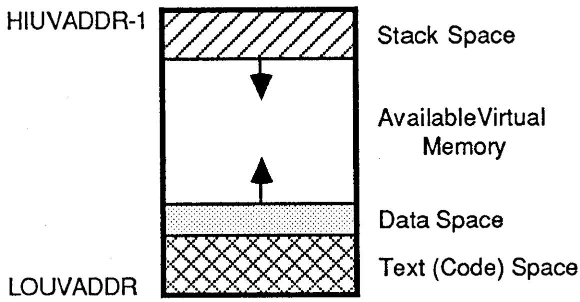
4 CTIX KERNEL TUTORIAL

This chapter contains tutorial information about the CTIX kernel. It is not meant to be an exhaustive treatment of the operating system architecture. Rather, it presents only the material you must have in order to understand what happens when a user process makes an I/O request. In particular, this chapter does not discuss process creation and deletion, scheduling, memory management, interprocess communication, or the CTIX file system. If you do not understand these topics, you should take a class in CTIX (or UNIX System V) internals.

THE USER PROCESS

The user process is the fundamental unit of work in the operating system. It represents the unique execution of a program. If two users are running the same program (or if one user is running the same program twice), CTIX creates and manages two separate processes.

The following diagram illustrates the memory map of a process from the user's perspective.



User Space

The text (code) space starts at user virtual address **LOUVADDR**, which is defined in the master file. The process data space resides in memory immediately above the text space and grows upward; the stack space starts at the top of user virtual memory (**HIUVADDR-1**) and grows downward. Each process has its own, unique data and stack spaces, but the text space is almost always shared among all processes running the same program. Thus, only one copy of the code resides in physical memory; it is mapped into **LOUVADDR** of each associated user process.

The CTIX operating system uses a number of internal queues in order to manage the processes. The most important of these queues are

- The run queue, which is the list of all processes that are ready to execute.
- The sleep queue, which is the list of all processes that are waiting for some event to occur. The **sleep(2K)** function puts a process on the sleep queue; the **wakeup(2K)** function takes it off the sleep queue and puts it back on the run

queue.

CTIX software also keeps data about each process in various tables. The most important of these tables are described in the subsections that follow.

THE PROCESS TABLE

The Process Table contains much of the information that CTIX needs to manage the scheduling of the CPU. Each process has a single entry that is created when the process is created (with the **fork(2)** system call). This entry is used continuously whether the process is running or not, whether it is swapped in or out, until the process ceases to exist. The various queues are implemented as singly linked lists using a field in the process table entry. Since there is only one link field, each process can be on only one queue at a time.

The Process Table entry is defined in the header file <**sys/proc.h**>, portions of which are included here. The vertical dots indicate that lines from the header file have been omitted here. The following code fragment may differ from the include files on your system. In all cases, the files in the latest CTIX operating system release supercede this document.

<sys/proc.h>

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

/*
 * There is one process structure allocated per process. It contains
 * all of the information about the process that could be needed by
 * CTIX while the process is swapped out.
 *
 * Other per process data (user.h) is swapped out with the process.
 */

struct    proc {
    struct    proc *p_link;   /* Linked list of process's queue */
    int       p_flag;         /* Process state */
    char      p_stat;         /* Current state of the process */
    char      p_pri;          /* Priority, negative is high */
    char      p_time;         /* Resident time for scheduling */
    char      p_cpu;          /* CPU usage for scheduling */
    char      p_nice;         /* Nice for CPU usage */
    char      p_szup;         /* Nbr of pages in user page (u.) */
    ushort    p_uid;          /* Real user id */
    ushort    p_suid;         /* Save (effective) user id */
    ushort    p_sgid;         /* Save (effective) group id */
    short     p_pgrp;         /* Process ID of proc grp leader */
    short     p_pid;          /* Unique process ID */
    short     p_ppid;         /* Process ID of parent */
    ushort    p_dieevkey;     /* For notify on process death */
    struct user *p_upage;     /* Ptr to user page (u.) */
    struct pregion *p_region; /* Pointer to process regions */
    short     p_size;         /* Process size in pages */
    ushort    p_mpgneed;      /* Number of memory pages needed */
    long      p_sig;          /* Signals pending to this process */
    union {
        caddr_t  p_cad;       /* Event process is awaiting */
        int  p_int;
    } p_unw;
#define  p_wchan  p_unw.p_cad
#define  p_arg    p_unw.p_int
    struct text *p_textp;     /* Pointer to text structure */
    int       p_clktim;       /* Time to alarm clock signal */
    char      p_frate;        /* Page fault rate */
    char      p_pad[1];       /* Align to a 4 byte boundary */
    char      p_pad2[6];      /* Align to a 64 byte boundary */
};
```

```
        .
        .
        .

/* p_stat codes */
#define  SSLEEP  1          /* Awaiting an event */
#define  SWAIT   2          /* (No longer used) */
#define  SRUN    3          /* Running */
#define  SIDL    4          /* Intermediate state - proc creation */
#define  SZOMB   5          /* Intermediate state - proc termination */
#define  SSTOP   6          /* Process being traced */
#define  SXBRK   7          /* Process being xswapped */

/* p_flag codes */
#define  SLOAD   0x000001 /* In core */
#define  SSYS    0x000002 /* Swapper or pager process */
#define  SLOCK   0x000004 /* Process being swapped out */
#define  SSWAP   0x000008 /* Save area flag */
#define  STRC    0x000010 /* Process is being traced */
#define  SWTED   0x000020 /* Another tracing flag */

        .
        .
        .
```

THE USER AREA

The user area is a single page (4K bytes) of memory containing information about a process that CTIX needs while the process is swapped in. The user area is also called the user page or u-page. It contains both the **user** structure and the supervisor stack. The address of the **user** structure is equal to the base address of the u-page; the supervisor stack starts at the highest address in the page and grows downward. The supervisor stack is referred to as the system call stack or system stack, because CTIX uses it while processing system calls.

> **CAUTION**
>
> There is nothing to prevent the supervisor stack from overrunning the **user** structure. If this happens, CTIX will die in strange and unpredictable ways. **Be very judicious in your use of supervisor stack space**: in particular, note that all of the automatic variables in your device driver consume space on this stack.

When a process is created, CTIX allocates one extra page for the user area. (Currently, only a single page is needed; this could change in the future.) At every context switch, CTIX software writes the page frame number of the process's u-page into the page table entry that describes kernel address 0x7E000 (the current address on the MightyFrame). Thus, the base address of the **user** structure for the current process is always 0x7E000 (on the MightyFrame), and the beginning of the supervisor stack is always 0x7FFFC. This allows CTIX to access the most frequently needed information at the same address no matter which process is running. This location is named **u**; it is declared (in <**sys/user.h**>) as follows:

extern struct user u;

Its address is set in the **ifile** for the operating system. (See the link editor documentation in AT&T's *UNIX System V Support Tools Guide* for more information about the **ifile**.)

The u-page always contains the information for the currently executing process. Be especially mindful of this fact when you design your driver's interrupt handler. Any runnable process could be active when an interrupt occurs: the odds are overwhelming that the u-page visible to the interrupt handler does not describe the process for which the current I/O operation is being carried out. Therefore, interrupt handlers **must not** reference or change the information in the u-page. (Interrupts are processed on the interrupt stack, not the supervisor stack, so there is no conflict there.) Your driver must keep all of the information it needs to service the interrupt in the buffer header data structure associated with the I/O in progress. If there is no associated buffer header structure, the driver must keep the information it needs in its own data space.

The **user** structure is defined in the header file <**sys/user.h**>, portions of which are included here. The vertical dots indicate that lines from the header file have been omitted here. The following code fragment may differ from the include files on your system. In all cases, the files in the latest CTIX release supercede this document.

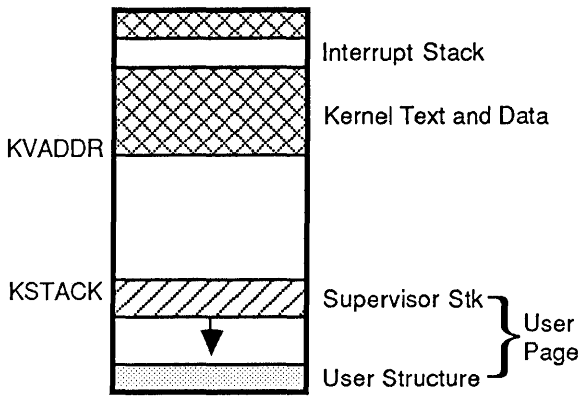
<sys/user.h>

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/inode.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/dir.h>

/*
 * The user structure.
 *
 * There is one allocated per process and it is swapped out
 * with the process. It contains all per-process data that
 * isn't referenced while the process is swapped. It holds
 * the per-user system stack, used during system calls. It
 * is cross-referenced with the proc structure for the same
 * process.
 */

struct user
{
     .
     .
     .
     struct proc *u_procp;      /* Pointer to proc structure */
```

```
    •
    •
    •
    char u_error;               /* Syscall error code */
    •
    •
    •
    union {                     /* Syscall return values */
         struct {
              int  r_val1;
              int  r_val2;
         } r_reg;
         •
         •
         •
    } u_r;
    caddr_t  u_base;            /* Base address for I/O */
    unsigned u_count;           /* Bytes remaining for I/O */
    union {
         off_t ow_offset;       /* Offset in file for I/O */
         •
         •
         •
    } u_ow;
    •
    •
    •
    short     u_fmode;          /* File mode for I/O */
    ushort    u_pbsize;         /* Bytes in block for I/O */
    ushort    u_pboff;          /* Offset in block for I/O */
    dev_t     u_pbdev;          /* Real device for I/O */
    •
    •
    •
    int   u_arg[10];            /* System call arguments */
    •
    •
    •
};
    •
    •
    •
```

KERNEL MEMORY MAP

The following diagram illustrates the kernel virtual address space.



Kernel Space

As documented in Chapter 2, *Architectural Information,* the kernel's virtual memory lies between 0x7F800000 and 0x7FFFFFFF. In a running system, however, the kernel can read and write every virtual memory page: only the kernel's text and data regions lie in the stated address range. The u-page for the current process is always mapped into the page beginning at **u**, as explained above.

SYSTEM CALL PROCESSING

You can think of an operating system as a collection of subroutines that provide various services to user programs. According to this view, a system call is nothing more than a transfer of control to a routine that is memory-resident and available to any process. In a single-tasking operating system such as MS-DOS, this is a fairly accurate picture. In a multitasking system like

CTIX, however, this is very much an oversimplification.

When an operating system supports multiprogramming, it must somehow maintain the illusion that each process has exclusive use of the entire machine. Moreover, when two or more unrelated processes compete for the same resource (such as memory space, or a peripheral device), the system must grant each process some access to the resource, while ensuring that no process monopolizes it to the exclusion of the others. Further, a complex system like CTIX occasionally competes directly with the user processes for resources.

Some protection mechanism is needed to ensure that user processes do not interfere with each other or with the operating system. In the MightyFrame, this separation is provided by memory management hardware that can distinguish between valid and invalid memory references. Each process has associated with it a group of page table entries that describe its access permission for every page of virtual memory. Each page of memory can be marked Read/Write to the kernel only, Read/Write to the kernel and Read-only to the user, or Read/Write to both the kernel and the user. The distinction between kernel and user accesses corresponds to whether the MC68020 is executing in supervisor or user mode. (See the *MC68020 32-Bit Microprocessor User's Manual* for more information about the CPU operating modes.)

CTIX software sets up the page table entries as follows: the kernel can read and write any page of memory in the system; the user can read his text (code) space, but he cannot write it (except in unusual cases, for example, when running under a debugger that allows patching). Finally, the user can read and write the data and stack spaces associated with his process. Given this arrangement, the user cannot issue a direct call to a subroutine in the kernel, since he does not have read (and therefore, execute) access permission on kernel memory. The MC68020 **TRAP** instruction provides controlled access to the operating system. **TRAP** changes the execution level from user mode to supervisor mode and performs a subroutine call to a fixed address in the kernel.

When you compile a program that makes any of the system calls documented in Section 2 of the *CTIX Operating System Manual*, the linker loads a small library routine of the form:

```
syscall:
        movw    &type,%d0
        trap    &0
        rts
```

Here **syscall** is the name of the system call, such as **setuid(2)** or **read(2)**, and **type** is a number uniquely associated with **syscall**. When your program actually issues the call, CTIX uses this number as an index into the system entry point table. This table contains the expected number of arguments and the address of the kernel's handler function for every possible system call. The following example illustrates the system call mechanism.

When you compile a program that makes a call to **setuid(2)**, the linker includes the following code in your program's text space:

```
setuid:
        mov.w   &17,%d0   ;The system call type
        trap    &0        ;Go to kernel in supervisor mode
        bcc     noerror   ;Carry Clear means no error
        jmp     __cerror  ;Couldn't set the uid

noerror:
        clr.l   %d0       ;No error - return 0 to user
        rts               ;Return from setuid() call

__cerror:
        mov.l   %d0,errno ;errno = rtn value from kernel
        moveq   &-1,%d0   ;Return -1 to user
        rts               ;Return from setuid() call
```

When you finally execute your program, your call to **setuid(2)** transfers control to the small library function above. The **TRAP &0** instruction puts the CPU into supervisor mode and transfers control to an assembly language routine in the kernel named **intsys()**. This routine saves the CPU registers and the user stack pointer on the supervisor stack and then calls **systrap()**. Using the &17 (from your program's saved **D0** register) as an index into the system entry point table, **systrap()** determines the number of parameters and the transfer address for the **setuid(2)** function.

Next, **systrap()** copies the parameter (in this case, the new user ID) from the program's stack into the user area and then calls the handler for the **setuid(2)** request. The handler is a kernel routine also named **setuid()**. (The kernel's **setuid()** function is not the same as the one loaded with your program. The kernel's function actually processes the system call: the function in your program is a small assembly language routine that issues a **TRAP &0** instruction and then sets **errno** if an error occurred.)

The kernel's **setuid()** function attempts to set the user ID and then returns to **systrap()**. **Systrap()** cleans up after the request and returns to **intsys()**, which executes an **RTE** (Return from Exception) instruction. This puts the CPU back into user mode and returns to the instruction after the original **TRAP &0** in your program.

Upon return from CTIX, the processor carry bit indicates the success or failure of the system call. If the carry bit is clear, the request was honored without error: registers **D0** and **D1** contain the return values from CTIX. If the carry bit is set, the contents of register **D0** are placed in **errno**, and then **D0** is set to -1. Note that, as documented in the Introduction to Section 2 of the *CTIX Operating System Manual*, **errno** is **not** cleared when a system call succeeds.

SYSTEM CALL EXAMPLES

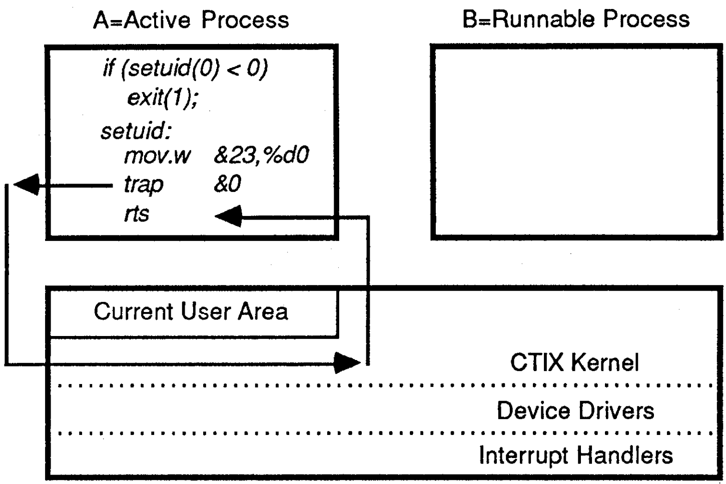
Broadly speaking, system calls can be divided into two overlapping groups: those that can be serviced by CTIX without delay, and those that cannot be serviced until some event occurs. Very often, this event is in the form of an interrupt generated by the completion of an I/O operation. For the purpose of illustration in this document, system calls that can be serviced immediately are referred to as synchronous; those that await the occurrence of some event are called asynchronous. This grouping is tutorial in nature: CTIX makes no such distinction when processing system calls.

This section contains a detailed analysis of two system calls, **setuid(2)**, which is a synchronous request, and **read(2)**, which usually is an asynchronous request. Occasionally, however, the data that the reading program requests is already in a kernel buffer. In this case, the **read(2)** system call can be processed synchronously. This section describes asynchronous **read(2)** requests only.

Synchronous System Call Processing - setuid(2)

A synchronous system call is a request for service that CTIX software can satisfy without any delay. Functions such as **getuid(2)**, **setuid(2)**, and **time(2)** are examples of synchronous requests. In general, synchronous requests either report or change the state of kernel variables.

The following diagram illustrates the flow of control through the **setuid(2)** system call.



Setuid(2): Trap to Kernel - Process System Call

The diagram shows two user processes, A and B. Process A has issued a **setuid(2)** request. Process B is an unrelated process that is ready to run: its presence serves to demonstrate that Process A does not lose the CPU **as a result of** the system call. (However, under the CTIX operating system, a process can lose the CPU at almost any time, unrelated to its system call activity. Synchronous, in the sense that it is used here, simply means that CTIX **does not need to wait** for an event to occur before it can satisfy the request.)

The flow of control for a synchronous call is simple. Process A calls **setuid(2)**, which is a small assembly language routine loaded from the library. This code places the constant 23 into register **D0** and then issues a **TRAP &0** instruction, which places the CPU into supervisor mode and essentially "calls"

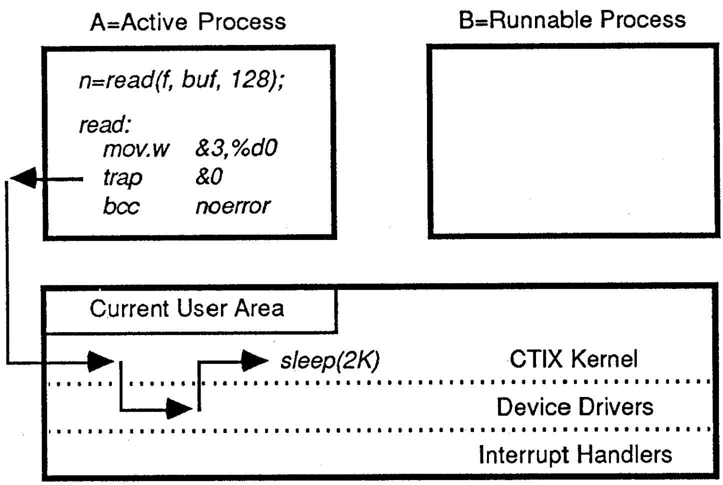
intsys(), the system call trap handler. **Intsys()** saves the user's register set and calls **systrap()**. **Systrap()** gets the parameter (in this case, the new user ID) from the stack and then, using the &17 from the user's **D0** register as an index into the system entry point table, indirectly calls the **setuid()** kernel function to process the request.

When the **setuid()** handler returns, **systrap()** places the return value into the user's **D0** register and returns to **intsys()**. **Intsys()** restores the user's register set and executes an **RTE** instruction, which places the CPU back into user mode and returns to the instruction after the original **TRAP &0**.

Asynchronous System Call Processing - read(2)

An asynchronous system call is a request for service that cannot be satisfied until some event occurs. While the process is waiting, CTIX puts it on the sleep queue and gives the processor to some other process. I/O requests such as **read(2)** and **write(2)** are the most common asynchronous system calls. For these requests, the awaited event is an I/O completion interrupt from the device being accessed.

The following series of diagrams illustrates the flow of control through the **read(2)** system call.



Read(2): Trap to Kernel - Process A Sleeps

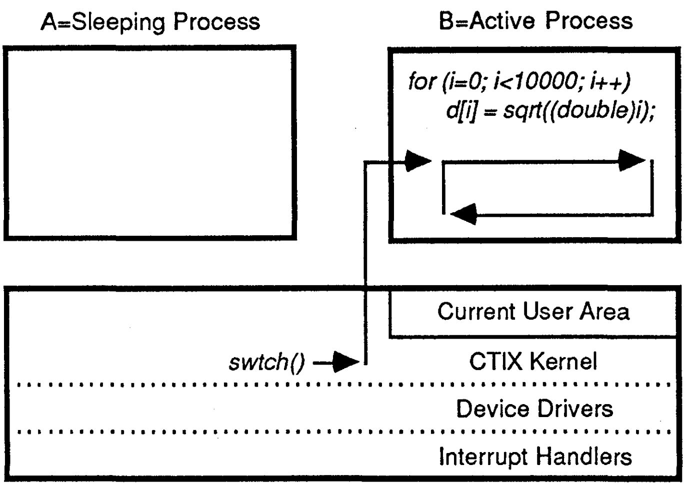
The preceding diagram shows two user processes, A and B. In this case, Process A has issued a **read(2)** request to a file. As before, Process B is an unrelated process that is ready to run. For asynchronous calls, however, Process B actually will run when Process A calls **sleep(2K)** in the device driver.

As you can see from the diagram, the **read(2)** request begins exactly as the **setuid(2)** did: by loading a constant (&3) into register **D0** and then issuing a **TRAP &0** instruction. Again, this places the CPU into supervisor mode and effectively calls **intsys()**, **Intsys()** calls **systrap()**, which gets the **read(2)** parameters (in this case, the file descriptor, the buffer address, and the transfer length) from the stack and then calls the kernel's **read()** handler to process the request.

Through a series of subroutines (which is described in detail in Chapter 7, *Block I/O Tutorial*), CTIX arrives at the **bread()** (block read - pronounced "be read") routine. **Bread()** determines that the requested data block is not in memory and calls the **gdstrategy(2K)** routine to read the block in from the disk. (If the desired block had been in memory, **bread()** would have returned immediately. In this case, the **read(2)** system call would have been processed synchronously.) **Gdstrategy(2K)** sorts the request into the queue of outstanding work and calls the device driver's **devstart(2K)** routine to start I/O on the controller. Finally, **devstart(2K)** returns to **gdstrategy(2K)**, which returns to **bread()**.

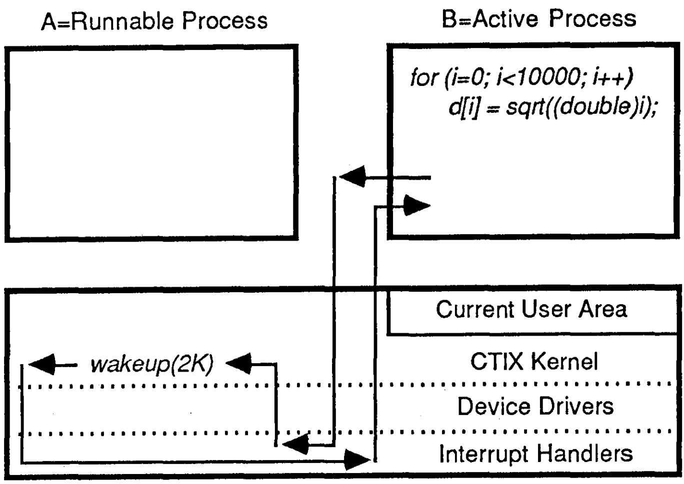
Regardless of whether the controller has other work to perform, the requested disk block will not be available for many microseconds. Since Process A has no further use of the CPU at this time, **bread()** calls **iowait(2K)**, which in turn calls **sleep(2K)**, allowing the process to wait for the I/O to finish. This is the state that is shown in the previous diagram.

The next diagram illustrates the context switch to Process B that occurs as a result of the **sleep(2K)** call in Process A. Process A is moved from the run queue to the sleep queue, and Process B (the highest priority process on the run queue) is placed into execution. The diagram underscores the fact that the kernel is using a new u_page by showing Process B's user area in a new location. Even though the kernel always refers to the **user** structure at virtual address **u**, the data in the structure is associated only with the currently executing process. Each process in the system has a separate page of physical memory dedicated to its u-page: CTIX must remap address **u** every time it performs a context switch.



Read(2): Context Switch - Restart Process B

Process B is CPU-bound; it makes no system requests during its lifetime. As long as CTIX does not intervene, Process B will calculate the square root of the first ten thousand integers and then exit. However, while Process B is running, the disk controller finally accesses the block that Process A requested. The controller issues an interrupt to the CPU to signal the completion of the I/O request. The processing of this interrupt is illustrated in the next diagram.

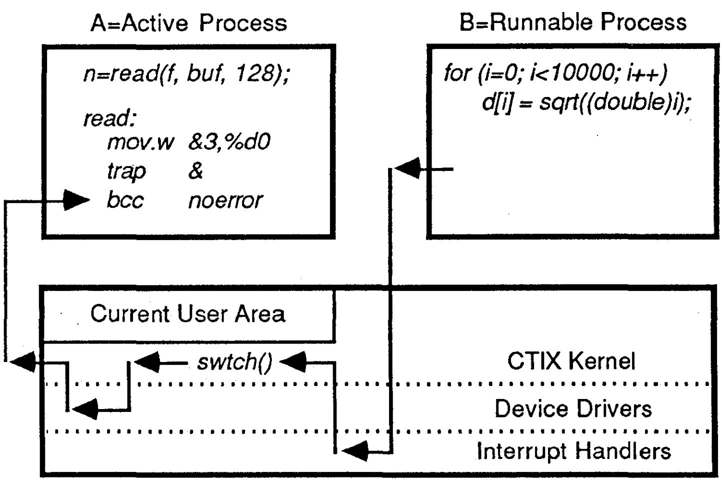


Read(2): I/O Completion Interrupt - Wakeup Process A

The I/O completion interrupt is fielded by CTIX software, and control is passed first to **gdintr(2K)** and then on to the **devintrgd(2K)** routine in the device driver. (For a discussion of low-level interrupt processing, see *Interrupt Processing* in Chapter 2, *Architectural Information.*) **Devintrgd(2K)** determines that the interrupt indicates the successful completion of the outstanding I/O request. It clears the hardware, cleans up after the current request, and returns an I/O done indication to **gdintr(2K)**.

Because **devintrgd(2K)** indicated that the I/O operation was complete, **gdintr(2K)** calls **iodone(2K)** to complete processing on the buffer. **Iodone(2K)** sets the **B_DONE** bit on the buffer header and calls **wakeup(2K)**. **Wakeup(2K)** takes Process A off the sleep queue and places it back on the run queue. This action, by itself, does not cause a context switch. However, if the process being placed on the run queue (in this case, Process A) is still present in memory (it could have been swapped out) and if its priority is higher than the priority of the currently running process, CTIX performs a context switch before returning from the interrupt. In the example, the processes have equal priority, so the kernel gives the CPU back to Process B. The I/O completion interrupt simply caused CTIX to place Process A back on the run queue.

Process B continues to run, calculating square roots, until it finally uses up its allotted portion of CPU time (currently, 16/60ths of a second). The final diagram in this example illustrates the system clock interrupt that causes a context switch back to Process A.



Read(2): System Clock Interrupt - Restart Process A

The clock interrupt handler detects that 16/60ths of a second have passed, and lowers the priority of the current process (Process B). Because the priority has been changed, the clock interrupt handler exits through the scheduler (**swtch()**). **Swtch()** scans the run queue looking for the process with the highest priority. It selects Process A and performs a context switch, mapping in its user page, restoring its context, and giving it the CPU.

Process A simply returns from its **sleep(2K)** call to **iowait(2K)**, as though it had never lost the processor. **Iowait(2K)** returns to **bread()**, which has accomplished its task: the requested block has been read in from the device. **Bread()** returns through a series of subroutines until control arrives back at the **systrap()** routine. **Systrap()** places the return value into the user's **D0** register and returns to **intsys()**. **Intsys()** restores the user's register set and executes an **RTE** instruction, which places the CPU back into user mode and returns to the instruction after the original **TRAP &0**.

The original **read(2K)** request has been serviced. In a way completely transparent to itself, Process A was put to sleep, waited for many microseconds, and then was given the CPU again. The **read(2)** system call was handled asynchronously because the device driver put Process A to sleep, waiting for the I/O completion interrupt to occur.

THE CTIX I/O SYSTEM

The CTIX I/O system presents a consistent, device-independent interface to the programmer. Instead of supporting a unique set of system calls specific to each device, CTIX makes all devices appear as files. To access a device, you must first issue an **open(2)** system call on the file associated with the device. This file, unlike a real data file, is created by the system administrator with the **mknod(1M)** program. These special files usually are located in the **/dev** directory: they have the same access permission bits as regular files.

Once you have opened the special file, you can issue **read(2)** and **write(2)** requests to transfer data between your program and the associated device, exactly as you would to a file. Finally, you must issue a **close(2)** call to inform CTIX that your program no longer requires communication with the device. Unless your program issues a device-specific **ioctl(2)** call, the interaction is exactly the same as though you were accessing a normal data file.

Beneath this file-like layer presented to the programmer, CTIX divides the I/O system into two pieces: the Block I/O system and the Character I/O system. The original designer of UNIX has written that the names should have been "structured I/O" and "unstructured I/O," respectively. (See "UNIX Implementation" by Ken Thompson, in the *Bell System Technical Journal*: July-August, 1978.) There is some basis for the name "block" device, but "character" has nothing whatever to do with the devices included in that class.

The information-node (i-node) for each special file contains a major device number, a minor device number, and a class, indicating whether the associated device should be accessed through the block or the Character I/O system. For each class, CTIX keeps an array of entry points into the device drivers for the members of the class. These arrays are described in detail in the Introduction to Appendix A, *CTIX Interface Manual Pages*. The major device number from the special file is used as an index into the array associated with the device class. The minor device number has no significance to CTIX: it is used to pass device-specific information to the driver.

The following set of steps documents the linkage process between an application program and a character device named **chardev**:

1. The program issues an **open(2)** call on special file **/dev/chardev**.
2. CTIX reads the i-node associated with **/dev/chardev** and verifies that the user has the proper access permission on the file.
3. CTIX determines from the i-node that it is dealing with a character special file and calls the device driver's **devopen(2K)** routine to initialize the device.

4. The program issues a **read(2)** request on the file descriptor returned by the **open(2)** call.
5. CTIX determines that the read is on a character special file and calls **devread(2K)**, the device driver read routine, to perform the data transfer.
6. The program issues a **close(2)** request on the file.
7. CTIX determines that it is dealing with a character special file and calls the device driver's **devclose(2K)** routine to release the device.

THE BLOCK I/O SYSTEM

The Block I/O system is designed to support random access devices that store data in fixed-length "chunks" (1,024 bytes under CTIX on the MightyFrame). A disk drive is the model block device. Tape drives also fit the model, even though they may not be capable of random access. The entire device is treated as an array of uniform 1K blocks, numbered from 0 to N-1 where N is the number of 1K blocks on the device. The driver for a block device receives requests to transfer data to and from these "array elements": its job is to hide the underlying physical structure of the device from the rest of CTIX.

Rather than perform physical reads and writes whenever they are requested, CTIX maintains an in-memory cache of data blocks in least recently used (LRU) order. Each block in the cache contains its own address, that is, its device number and block number. Whenever CTIX receives a request to read a block, the Block I/O system searches the cache first. If the block is found in memory, a copy of it is returned to the user. If the desired block is not found, the oldest block is written out (if it has been modified), and then a request is issued to the appropriate device driver to refill the buffer with the newly requested block. When the device driver completes the read, CTIX returns a copy of the desired data to the user.

When the user issues a **write(2)** request on a block device, the data is not written immediately. Instead, the block is marked modified and placed at the end of the LRU list. The new data is not written out to the device until the buffer is needed to hold a different block. As long as the buffer is still in the cache, any process that needs the data from that block will get it without performing any physical I/O.

The design of the Block I/O system has several effects: first, there is a substantial performance improvement, since many user I/O requests can be satisfied with no access to the device. This is not without cost, however. When the system crashes, a considerable amount of data (every altered, unwritten block in the cache) will be lost. Since some of these blocks contain modified free list and i-node structures, the integrity of the file system itself may be corrupted. Most system administrators run the **update(1M)** program to synchronize the disks periodically, thereby minimizing the effects of a system crash. In the event of a system failure, you can use the **fsck(1M)** program to repair damaged file systems. Nevertheless, any modified user data that is in the cache when the system crashes is lost.

THE CHARACTER I/O SYSTEM

The Character I/O system supports all devices that do not fit within the block system. Typically, this refers to devices such as terminals, printers, and communications lines, which produce or consume nonrepeatable sequences of data. The Character system also supports block devices when they are accessed in an unstructured manner: track-at-a-time reads and writes to disk devices are a good example of this use of Character I/O. From this, you can see that devices do not belong absolutely to the block or Character I/O system: on the contrary, some devices naturally fit into **both** systems.

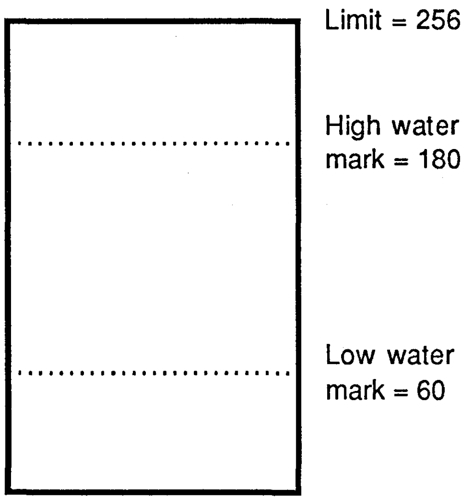
Character Queue Processing

The Character I/O system frequently uses small buffers called character blocks for intermediate storage of incoming and outgoing data. These c-blocks are linked together to form character queues called character lists, or c-lists. There are kernel routines for placing a character on a queue, and for getting a character off a queue. (See **getc(2K), getcb(2K), putc(2K), and putcf(2K)**) in Appendix A, *CTIX Interface Manual Pages*.) The following steps document the flow of control through the character I/O system when accessing an output-only device such as a printer. Refer to the Character Queue Processing diagram for more information.

1. The user process initiates a **write(2)** request to print a line on the device.
2. The driver begins placing characters on the c-list.
3. When the first character is placed on the c-list, the driver sets up the device to begin outputting characters.
4. The driver continues adding characters to the queue.
5. The interrupt handler continues removing characters from the queue and outputting them, sustained by I/O completion interrupts.
6. The driver continues adding characters to the queue until it reaches the high water mark. At this point, the driver issues a **sleep(2K)** request, waiting for the interrupt handler to work down the queue.
7. As the I/O completion interrupts are serviced, the driver's interrupt handler checks the number of characters left in the queue. When the count falls below the low water mark, the interrupt handler issues a **wakeup(2K)** call to restart the upper level of the driver.
8. The driver once again begins adding characters to the queue. As before, when it reaches the high-water mark, the driver sleeps.

9. The entire cycle is repeated until all the characters have been placed onto the queue. At this point, the driver returns, exiting from the kernel and completing the user's **write(2)** request.

10. The interrupt handler continues outputting characters asynchronously until it has emptied the queue.



Character Queue Processing

The processing for input devices is similar, except that the producer and consumer roles are reversed. The device interrupt handler is running asynchronously, placing unasked-for characters on the queue. When the user issues a **read(2)** request, if the queue contains characters, the driver returns them immediately. Otherwise, the driver calls **sleep(2K)**, waiting for the interrupt handler to place an incoming character onto the queue.

Terminal Devices

Terminals are a special class of character device. They have one output queue but two input queues: a raw queue and a canonical queue. The device driver places incoming characters onto the raw queue as it receives them from the serial device. When it receives a new line character, it copies the entire line onto the canonical queue: in the process, the character erase and line kill functions are performed. The reading process can specify either the raw or canonical ("cooked") queue as the source of its input.

Buffered Character I/O

C-list processing is meant for low-speed devices such as terminals and printers. It is not suitable for applications such as communications networks and machine-to-machine links. Drivers for these devices often use the block device buffer management techniques for their intermediate storage requirements. They either "borrow" the needed buffers from the system buffer cache, or they allocate private buffers for their own use.

Physical (Raw) I/O

For most **write(2)** requests, CTIX first copies the data from the user's address space into kernel space. The driver then transfers the data from kernel space out to the device, often using DMA hardware. The reverse happens for **read(2)** requests. This double copy operation is wasteful: it is always more efficient to transfer I/O data directly between the user's memory space and the device. Direct transfers such as this are known as "physical" or "raw" I/O. The device driver for the DR11, which is documented in Chapter 6, *Character Device Example,* uses physical I/O to perform its data transfers.

5 CHARACTER I/O TUTORIAL

This chapter describes the Character I/O system in detail. It also contains two example character device drivers: one example performs character-at-a-time I/O using c-lists; the other performs Physical I/O directly between the user's virtual memory and the device. The examples are written in a C-like pseudocode and include program narratives describing the drivers in detail.

OVERVIEW

The Character I/O system includes all devices that cannot be handled through the Block I/O system: that is, devices that do not support randomly accessible, fixed-length "chunks" of storage. While the Character I/O system can be used to access raw disk or tape drives, it most frequently is used with devices that produce and consume nonrepeatable sequences of characters. Devices such as terminals and printers are handled naturally by the Character I/O system.

The Character I/O system can be subdivided further into low-speed and high-speed devices. Typically, low-speed devices deal with characters one at a time and generate an interrupt for each character in the sequence. High-speed devices usually use DMA hardware to transfer large "chunks" of data in and out: they generate an interrupt at the completion of each DMA operation. Superficially, high-speed devices resemble block devices because they generally deal with data in "chunks," but these "chunks" usually are sequential in nature. On the other hand, it is common to find character drivers for raw disk and tape devices, which are randomly addressable.

The following sections present examples of the two classes of character devices. *Character-at-a-Time I/O* describes a Network Interface board, which is a typical low-speed device. *Physical*

(Raw) I/O describes an Analog-to-Digital board, which is a typical high-speed device. Both sections contain a broad overview of the hypothetical device and the environment in which it is used. Following this introductory material is a tutorial driver for the device.

The example drivers are written in a C-like pseudocode. At times the pseudocode is abstract and general; at other times, it reads almost like an actual C program. These examples are not meant to be exhaustive: in particular, they do not do adequate error detection and recovery. Use these examples as models: simple, straightforward, and to some extent, ideal. Refer to them as you address the problems presented by your device. Your own driver may have more or less functionality than these examples, depending on the complexity of your device and the level of support you decide to provide.

CHARACTER-AT-A-TIME I/O

The principal task of a low-speed character driver is to transmit and/or receive data by performing byte-by-byte transfers, sustained by I/O completion interrupts. To shield the user process from speed variations at the device level, low-speed character drivers store the data temporarily in small buffers within the kernel. Block I/O system buffers are not appropriate for this task, since they form an associative cache, addressed by the block number and device number of the data they contain. It makes no sense to speak of "block number 43" from "terminal number 9," since this character device produces and consumes streams of nonrepeatable data.

The ideal buffer structures for character-at-a-time devices are small queues that allow characters to be added and removed one by one. The CTIX operating system defines structures called character blocks, or c-blocks, that can hold up to 64 characters (the number of characters is implementation-dependent). These c-blocks are linked together into character lists (c-lists), each containing one or more c-blocks. Your driver can add a character to a c-list by calling **putc(2K)** or **sputc(2K)** and can remove a character by calling **getc(2K)**.

> **NOTE**
>
> The kernel functions **getc(2K)** and **putc(2K)** are not the same as the library macros **getc(3S)** and **putc(3S)**. The library macros read characters from and write characters to user I/O streams. The kernel functions place characters on and remove characters from kernel c-lists. Before proceeding, you should read and understand the appropriate manual pages in Section 3 of the *CTIX Operating System Manual* and Appendix A, *CTIX Interface Manual Pages*, in this document.

THE NETWORK INTERFACE DRIVER

The following pages contain pseudocode for a character-at-a-time device driver. The device under consideration is a low-speed, serial Network Interface (**NI**). The device is full duplex: it contains two channels: one dedicated to upstream traffic, and the other dedicated to downstream traffic. The channels are completely independent: the **NI** device can transmit and receive characters simultaneously. Each channel is interrupt-driven, so the driver must acquire two interrupt vectors at initialization time. As long as the driver is open, it must be ready to receive unsolicited input from the network.

The number of c-blocks in the kernel is limited, so it is possible for the driver to run out of queue space. If this happens when outputting a character, the **niwrite()** routine sleeps, waiting for a c-block to become free. This is not possible when the driver receives a character from the network, since an interrupt handler (**niRXintr()**) cannot issue a **sleep(2K)** call. If the driver ever receives characters for which there is no space on the queue, it discards the data and remembers this condition. As soon as a c-block becomes available, the driver enqueues a **CAN** (CANcel) character in place of the lost data. The **CAN** character can represent one or many lost characters. It is the responsibility of the network server to detect and process lost data errors.

Immediately after its invocation, the network server daemon creates a socket for communication and then forks two child processes: a reader and a writer. The writer process loops, issuing **recv(2N)** calls on the socket and writing the resulting messages to the **NI** device. The reader process loops, issuing **read(2)** calls on the **NI** device and sending the resulting messages through the socket.

The narration for the **NI** driver begins on the following page. Throughout the pseudocode, **RX** indicates the Receiver Channel, while **TX** indicates the Transmitter Channel. Also, routines that begin with the characters **hw_** refer to hardware-specific code.

This page intentionally left blank.

niinit()

CTIX software calls **niinit()** to initialize the driver before it allows any process to open the device. If the driver was linked with the kernel, CTIX calls **niinit()** at system initialization time. If the driver is loadable, CTIX calls **niinit()** as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVBIND**. The **lddrv(1M)** program makes this system call.

The driver must make certain that **niinit()** is called only once between calls to **nirelease()**. Testing and setting an initialization flag in **niinit()** and clearing the flag in **nirelease()** is sufficient to accomplish this.

Next, **niinit()** makes certain that the VMEbus Interface board is installed in the MightyFrame, and that the EEPROM on the board has a valid checksum. Finally, the driver searches through the array of device information in the EEPROM, looking for the entry corresponding to the **NI** device.

Then, **niinit()** acquires interrupt vectors for both of its channels. (See *Interrupt Processing,* in Chapter 2, *Architectural Information,* for a discussion of acquiring interrupt vectors. Also see the detailed descriptions of the **get_vec(2K)** and **set_vec(2K)** kernel routines in Appendix A, *CTIX Interface Manual Pages.*)

The **devinit(2K)** routine also must perform the required hardware initialization, which is unique for each hardware device. Generally, you should clear any interrupts and device status information, write the interrupt vector number into a device register, and make the device ready to perform I/O.

In the example, **niinit()** does not set up the device to perform any transfers until it receives an **open(2)** request from the daemon. If your driver must handle unsolicited input, make certain that you limit the amount of system memory it can consume.

```
#include "nidefs.h"
#include <sys/types.h>
#include <sys/spl.h>
#include <sys/tty.h>
#include <sys/user.h>
#include <sys/errno.h>
/*
 * Initialize the driver and the device - see devinit(2K).
 */
niinit(vecnbr)
int vecnbr;
{
	struct vmeeprom *eeprom, *is_eepromvalid();
	int i;

	if (DriverInitialized) {
		printf("niinit: double initialization");
		u.u_error = EBUSY;
		return;
	}
	DevAddress = 0;		/* Initialize */
	/* Make sure VMEbus is present and EEPROM is valid. */
	if (haveVME && ((eeprom = is_eepromvalid()) != 0)) {
		/* Search the EEPROM for our device */
		for (i=0; i<VME_SLOTS; i++) {
			if (eeprom->slots[i].type == VMET_NI) {
				DevAddress = eeprom->slots[i].address;
				break;
			}
		}
	}
	/* No VMEbus, invalid EEPROM, or no device. */
	if (DevAddress == 0) {
		if (eeprom == 0)
			printf("siinit: invalid VMEbus eeprom");
		u.u_error = ENXIO;
		return;
	}
	if (get_vectors()) < 0) {
		u.u_error = EBUSY;
		return;
	}
	hw_init();	/* Initialize the hardware. */
	DriverInitialized = 1;
}
```

nirelease()

The **devrelease(2K)** routine reverses the actions taken by **devinit(2K)**. CTIX software calls **nirelease()** as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVUNBIND**. The **lddrv(1M)** program makes this system call.

If the device is open, it cannot be released: the driver should print a message, set **u.u_error**, and return.

Next, the release routine should do whatever is necessary to the hardware to ensure that it does not cause any unwanted activity after the driver is unloaded. Specifically, **devrelease(2K)** should abort any outstanding I/O and disable any interrupts that the device has been programmed to generate. If the device initiates any activity after the driver has been unbound, it may cause a system crash.

After the driver has shut down the hardware, it should return any interrupt vectors that it acquired in **devinit(2K)**. Also, if the driver has any outstanding **timeout(2K)** calls, it should call **untimeout(2K)** to clear them.

Before it exits, **devrelease(2K)** should clear the initialization flag, allowing the next call to **devinit(2K)** to succeed.

```
/*
 * Release the driver and the device - see devrelease(2K).
 */
nirelease()
{
	SDEC;

	/* Cannot release device if it is open. */
	if (DevOpen) {
		printf("nirelease: attempt to release open device");
		u.u_error = EBUSY;
		return;
	}
	hw_shutdown();		/* Shut down the hardware */
	SPL_NI;
	/* Reset the acquired interrupt vector(s). */
	reset_interrupt_vectors();
	SPLX;
	/* Re-allow niinit() calls. */
	DriverInitialized = 0;
}
```

niopen()

The Network Interface board is an exclusive use device: only one **open(2)** call at a time should succeed. There must be an intervening **close(2)** call before another **open(2)** call. This is simple to achieve by testing and setting a flag in the **devopen(2K)** routine, and clearing the flag in **devclose(2K)**.

Niopen() resets the hardware to ensure that the driver starts out in a known state. This is an important step in any **devopen(2K)** routine.

Next, **niopen()** enables I/O on the RX channel. This involves little more than enabling the Receiver Interrupt on the RX channel of the device.

Finally, **niopen()** sets the device open flag to ensure exclusive access.

niclose()

Niclose() reverses the actions taken by **niopen()**. First, it stops I/O on the transmitter and receiver channels, and resets the hardware. Then, **niclose()** flushes any leftover characters from both the transmitter and receiver queues. It does this by repeatedly calling **getcb(2K)** to remove a c-block from the queue, and then calling **putcf(2K)** to place the c-block onto the freelist. Finally, **niclose()** clears the exclusive use flag, allowing another **open(2)** call to succeed.

```
/*
 * Open the device - see devopen(2K).
 */
niopen(dev)
dev_t dev;
{
	/* Exclusive use device - only one open(2) at a time. */
	if (DevOpen) {
		u.u_error = EBUSY;
		return;
	}
	hw_reset();		/* Reset the hardware. */
	/* Start I/O on the RX channel. */
	niRXstart();
	/* Lock out niopen() calls until niclose(). */
	DevOpen = 1;
}


/*
 * Close the device - see devclose(2K).
 */
niclose(dev, flag)
dev_t dev;
int flag;
{
	struct cblock *cp;

	/* Stop I/O on the TX and RX channels. */
	niTXstop();
	niRXstop();

	hw_reset();		/* Reset the hardware. */
	/* Flush the TX and RX queues. */
	while ((cp = getcb(&TX_q.clist)) != 0)	/* Remove c-block from Q */
		putcf(cp);		/* Place it on freelist */
	TX_q.flags = 0;
	while ((cp = getcb(&RX_q.clist)) != 0)
		putcf(cp);
	RX_q.flags = 0;

	/* Clear the exclusive use flag. */
	DevOpen = 0;
}
```

niread()

The Network Interface device is interrupt-driven: while **niread()** is taking characters off one end of the c-list, **niRX-intr()** could be adding new data on the other end. Whenever a conflict such as this can occur, the base level portion of the driver (above the interrupt level) must mask off interrupts to prevent the corruption of your driver's data structures. Queues of I/O requests and data are the most common structures at risk.

Niread() fills the user's buffer from the queue of incoming characters. It sleeps whenever there are no characters available. The RX interrupt handler issues a **wakeup(2K)** call whenever it fills the queue above the low-water mark and detects that the reader is sleeping.

Upon entry, **niread()** masks off RX interrupts from the device. It then enters a loop conditioned on **u.u_count** and **u.u_error**. The loop continues until the user's transfer count is exhausted or until an error occurs. If at any time there are no more characters in the queue, **niread()** sets the **SLEEPING** flag and sleeps.

When the reader process goes to sleep in this manner, CTIX gives the CPU to another process (Process X) to run. At some point, CTIX software, or another device driver enables interrupts from the NI device. This allows the receiver interrupt handler to continue placing characters on the RX queue until it surpasses the low-water mark. When this occurs, **niRXintr()** checks the **SLEEPING** flag. If the bit is set, the interrupt handler issues a **wakeup(2K)** call to restart the reader process. For a complete discussion of context switching and interrupt processing, see Chapter 4, *CTIX Kernel Tutorial.*

As long as there are characters present, **niread()** attempts to dequeue them and store them into the user's buffer. When the user's buffer is full (or when an error occurs), **niread()** restores the original IPL and returns to the caller.

```
/*
 * Fill the user's buffer from the queue - see devread(2K).
 */
niread(dev)
dev_t dev;
{
	SDEC;		/* Declaration for SPL */
	int c;

	/* Mask off Network Interface interrupts. */
	SPL_NI;

	/* Fill the user's buffer - stop on error. */
	while ((u.u_count!=0) && (u.u_error==0)) {
		c = getc(&RX_q.clist);
		if (c < 0) {			/* Queue empty*/
			RX_q.flags |= SLEEPING;		/* Wait for data */
			(void)sleep((caddr_t)&RX_q, NI_PRI);
		} else if (subyte(c, u.u_base++)) {	/* Store byte */
			u.u_error = EFAULT;	/* Bad buffer address */
		} else {
			u.u_count--;		/* One less to do */
		}
	}

	/* Restore the original interrupt mask. */
	SPLX;
}
```

niwrite()

The **niwrite()** routine enqueues the contents of the user's buffer onto the transmitter queue, one character at a time. It then enables the transmitter to perform the output.

First, **niwrite()** raises the processor priority level to mask out interrupts from the Network Interface device. The base-level portions of the driver must do this whenever they manipulate data structures that the interrupt handlers also change.

Next, **niwrite()** enters the loop that enqueues the buffer contents. As long as there is data remaining in the buffer and no errors have occurred, the loop continues. **Niwrite()** fetches a byte from the user's buffer and calls **niTXputc()** to enqueue it. This routine sleeps if there is no space available for the character. The loop continues until the user's buffer is exhausted.

With the message safely on the queue, **niwrite()** enables the transmitter and then restores the previous processor priority level before returning to the user.

```
/*
 * Write a message to the network - see devwrite(2K).
 */
niwrite(dev)
dev_t dev;
{
    SDEC;           /* Declaration for SPL */
    int c;

    /* Mask off Network Interface interrupts. */
    SPL_NI;

    /* Copy the user's buffer onto the TX queue, stop on error. */
    while ((u.u_count--!=0) && (u.u_error==0)) {
        /* Get a character from user space */
        c = fubyte(u.u_base++);
        if (c < 0)              /* Access error */
            u.u_error = EFAULT;
        else
            niTXputc(c);        /* Put char on TX queue */
    }

    /* Start the TX */
    niTXstart();

    /* Restore original interrupt mask */
    SPLX;
}
```

niRXstart

These routines start and stop I/O on the TX and RX channels. The code for all four routines is similar. The start routines enable the channel if it is disabled. The stop routines disable the channel if it is enabled.

All four routines disable interrupts upon entry and restore the processor priority level when they exit. This is to prevent contention with the interrupt handlers, which also alter the state of the enabled flags.

```
niRXstart()
{
    SDEC;

    SPL_NI;
    if (!(RX_q.flags & ACTIVE)) {
        hw_RXenable();          /* Enable the RX channel */
        RX_q.flags |= ACTIVE;
    }
    SPLX;
}
niTXstart()
{
    SDEC;

    SPL_NI;
    if (!(TX_q.flags & ACTIVE)) {
        hw_TXenable();          /* Enable the TX channel */
        TX_q.flags |= ACTIVE;
    }
    SPLX;
}

niRXstop()
{
    SDEC;

    SPL_NI;
    if ((RX_q.flags & ACTIVE)) {
        hw_RXdisable();         /* Disable the RX channel */
        RX_q.flags &= ~ACTIVE;
    }
    SPLX;
}
niTXstop()
{
    SDEC;

    SPL_NI;
    if ((TX_q.flags & ACTIVE)) {
        hw_TXdisable();         /* Disable the TX channel */
        TX_q.flags &= ~ACTIVE;
    }
    SPLX;
}
```

niRXintr()

NiRXintr() runs whenever the Network Interface board receives a character from the network. Since this happens asynchronously, the receiver interrupt is left enabled from the time the driver is opened until it is closed. Since the amount of kernel memory reserved for c-blocks is limited, the reader process in the network daemon must constantly read the NI device, or incoming characters may be lost.

Immediately upon entry, **niRXintr()** reads the available character from the device register and attempts to put it on the queue. The interrupt handler does not need to know if there was space for the character: lost data errors are handled by **niRXputc()**.

After attempting to enqueue the data, **niRXintr()** checks to see whether the reader process is asleep, waiting for input. If the queue is above the low-water mark and the reader is sleeping, **niRXintr()** calls **wakeup(2K)**. If the queue is below the low-water mark, there is too little data to awaken the reader.

niTXintr()

NiTXintr() runs whenever the TX channel is ready to transmit another character across the network. When the interrupt occurs, there may or may not be a character to transmit.

First, **niTXintr()** attempts to dequeue a character. If there is no character available, the interrupt handler calls **niTXstop()** to disable the transmitter channel. If there is a character in the queue, **niTXintr()** outputs it to the device.

Next, **niTXintr()** checks to see whether the writer process is asleep, waiting for queue space. If so, and if the queue is now below the low-water mark, **niTXintr()** calls **wakeup(2K)**. If the queue is above the low-water mark, there is too little space to awaken the writer.

```
/*
 * Process an RX interupt - see devintr(2K).
 */
niRXintr()
{
	int c;

	c = hw_input();			/* Get char from device. */

	/* Try to enqueue it. */
	niRXputc(c);

	/* If the Q is above LO_WATER and niread() is sleeping, wake it up */
	if ((RX_q.clist.c_cc>NI_LO_WATER) && (RX_q.flags&SLEEPING)) {
		RX_q.flags &= ~SLEEPING;
		wakeup((caddr_t)&RX_q);
	}
}


/*
 * Process a TX interupt - see devintr(2K).
 */
niTXintr()
{
	int c;

	if ((c = getc(&TX_q.clist)) < 0)	/* Queue empty */
		niTXstop();			/* Disable TX */
	else					/* Got char */
		hw_output(c);			/* Output to device */

	/* If count is below LO_WATER and niwrite() is sleeping, wake it up. */
	if ((TX_q.clist.c_cc<NI_LO_WATER) && (TX_q.flags&SLEEPING)) {
		TX_q.flags &= ~SLEEPING;
		wakeup((caddr_t)&TX_q);
	}
}
```

niRXputc()

The receiver interrupt handler calls **niRXputc()** to place a newly received character on the RX queue. Since queue space is limited, **niRXput()** must handle the possibility that it will not be able to enqueue the data.

Upon entry, **niRXputc()** checks the **QFULL** flag to see if there was room in the queue for the **previous** character. If it is set, **niRXputc()** tries to enqueue a **CAN** character, to inform the reader process that data was lost. If **niRXputc()** does enqueue the **CAN** character, it clears the **QFULL** flag.

After reporting any lost data, **niRXputc()** attempts to enqueue the current character. If the **QFULL** flag is set, there is no room for the data, so **niRXputc()** does not call **putc(2K)**. If the flag is clear, **niRXputc()** attempts to enqueue the character. If **putc(2K)** fails, the queue is full, and **niRXputc()** sets the **QFULL** flag.

It is possible for the **QFULL** flag to be set upon entry, cleared when **putc(2K)** succeeds in placing the **CAN** character on the queue, and then set once more, because **putc(2K)** fails to place the current character on the queue.

niTXputc()

Niwrite() calls **niTXputc()** to place each character on the output queue. Before enqueueing the character, **niTXputc()** checks the high-water mark and, if the queue is too full, starts the transmitter and sleeps. When the TX interrupt handler reduces the queue below the low-water mark, it reawakens **niTXputc()**. Whether or not it slept, **niTXputc()** then calls **sputc(2K)** to enqueue the character. **Sputc(2K)** will sleep again if it cannot get a c-block to expand the queue.

```
/*
 * Try to put a character on the RX queue - called from niRXintr().
 */
niRXputc(c)
int c;
{
	/* If no space for previous character, try to put a CAN */
	if (RX_q.flags & QFULL) {
		/* If room for CAN char, clear the flag */
		if (putc(CAN, &RX_q.clist) == 0)
			RX_q.flags &= ~QFULL;
	}
	/* If there is space now, try to enqueue the current char. */
	if (!(RX_q.flags & QFULL)) {
		/* If no space for current char, set flag for next time */
		if (putc(c, &RX_q.clist) < 0)
			RX_q.flags |= QFULL;
	}
}


/*
 * Try to put a character on the TX queue - called from niwrite().
 */
niTXputc(c)
int c;
{
	/* Sleep while the TX queue is above the high-water mark. */
	while (TX_q.clist.c_cc > NI_HI_WATER) {
		niTXstart();			/* Be sure TX is running */
		TX_q.flags |= SLEEPING;		/* Wait for space on the Q */
		(void)sleep((caddr_t)&TX_q, NI_PRI);
	}

	/* Put the character on the queue - sleep (again) if needed. */
	(void)sputc(c, &TX_q.clist, 1);
}
```

PHYSICAL (RAW) I/O

The principal task of a high-speed character driver is to transmit and/or receive data by performing large, block transfers, sustained by DMA completion interrupts. In order to achieve the highest possible data rates, high-speed character drivers do not buffer the data within the kernel. Instead, these drivers perform DMA directly into or out of buffers located in the memory space of the user process. This form of transfer is known as physical I/O or raw I/O. The CTIX operating system has special facilities to support this high-speed interface.

Physical I/O is performed directly between user memory and the device: there are no associated kernel buffer structures. Since the transfers are performed in "chunks," the buffer header structure from the Block I/O system is useful in describing the DMA operation to the device. These headers are not linked into the associative cache, however, since there are no block numbers associated with their contents. Like their low-speed cousins, high-speed character devices often deal with unstructured data: usually read-once or write-once sequences of characters.

THE SPEECH INTERFACE DRIVER

The following pages contain pseudocode for a DMA-based device driver. The device under consideration is a high-speed, digital-to-analog (D/A) and analog-to-digital (A/D) Speech Interface board (**SI**). The **SI** device is used to digitize and record human speech, and then to reconvert the speech to analog and play it back. It forms the heart of a speech synthesis workstation. The device contains two channels: one (the A/D side) dedicated to digitizing the speech input; the other (the D/A side) dedicated to reconverting the digitized waveform to analog and playing it back.

The controlling process interacts with the user in a manner similar to a tape recorder: it is called the **tape recorder** throughout this section. Symbolic "buttons" (menu selections, perhaps) allow the user to RECORD, PLAY, REVERSE, and FAST FORWARD the "tape," as well as to power off the recorder (exit from the program).

Only one channel of the **SI** device can be active at a time: the device can either record or play, but it cannot do both at the same time. Each channel is DMA-driven, and produces an interrupt at the completion of each DMA operation. Since simultaneous play and record is not supported, the **SI** driver need acquire only one interrupt vector at initialization time.

The driver does not support unsolicited input: instead, the Tape Recorder initiates a RECORD operation with a **read(2)** system call. Recording continues with each **read(2)** until the Tape Recorder issues an **ioctl(2)** call. The **SI** device contains a small, on-board FIFO to provide some buffering capability, but the Tape Recorder must issue its **read(2)** requests quickly enough to ensure that no data is lost.

The Tape Recorder initiates a PLAY operation with a **write(2)** system call. As long as the **SI** device is not in the process of recording, playback is started immediately. If the Tape Recorder issues a **write(2)** request while the driver is recording, the request fails with an **EBUSY** error.

The narration for the **SI** driver begins on the following page. Throughout the pseudocode, **RC** indicates the RECORD (or A/D) channel, while **PC** indicates the PLAY (or D/A) channel. In addition, routines that begin with the characters **hw_** refer to hardware-specific code.

siinit()

CTIX software calls **siinit()** to initialize the driver before it allows any process to open the device. If the driver was linked with the kernel, CTIX calls **siinit()** at system initialization time. If the driver is loadable, CTIX calls **siinit()** as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVBIND**. The **lddrv(1M)** program makes this system call.

The driver must make certain that **siinit()** is called only once between calls to **sirelease()**. Testing and setting an initialization flag in **siinit()** and clearing the flag in **sirelease()** is sufficient to accomplish this.

Next, **siinit()** makes certain that the VMEbus interface board is installed in the MightyFrame, and that the EEPROM on the board has a valid checksum. Finally, the driver searches through the array of device information in the EEPROM, looking for the entry corresponding to the **SI** device.

Next, **siinit()** attempts to allocate system page table entries, so that it can remap the user's buffer into kernel virtual memory before the I/O is started. If **siinit()** cannot acquire the necessary space, **siinit()** prints an error message, sets **u.u_error**, and returns.

Then, **siinit()** acquires an interrupt vector. (See *Interrupt Processing,* in Chapter 2, *Architectural Information,* for a discussion of acquiring interrupt vectors. Also see the detailed descriptions of the **get_vec(2K)** and **set_vec(2K)** kernel routines in Appendix A, *CTIX Interface Manual Pages.*)

The **devinit(2K)** routine also must perform the required hardware initialization, which is unique for each hardware device. Generally, you should clear any interrupts and device status information, write the interrupt vector number into a device register, and make the device ready to perform I/O.

```
#include "sidefs.h"

/* Initialize the driver and the device - see devinit(2K). */
siinit()
{
	struct vmeeprom *eeprom, *is_eepromvalid();
	int i;

	if (DriverInitialized) {
		printf("siinit: double initialization");
		u.u_error = EBUSY;
		return;
	}
	DevAddress = 0;		/* Initialize */
	/* Make sure VMEbus is present and EEPROM is valid. */
	if (haveVME && ((eeprom = is_eepromvalid()) != 0)) {
		/* Search the EEPROM for our device */
		for (i=0; i<VME_SLOTS; i++) {
			if (eeprom->slots[i].type == VMET_SI) {
				DevAddress = eeprom->slots[i].address;
				break;
			}
		}
	}
	/* No VMEbus, invalid EEPROM, or no device. */
	if (DevAddress == 0) {
		if (eeprom == 0)
			printf("siinit: invalid VMEbus eeprom");
		u.u_error = ENXIO;
		return;
	}
	/* Allocate kernel virtual memory space */
	SI_VAddr = (char *)sptalloc(dtop(MAXBLK)+1, (PG_V|PG_KW), -1);
	if (SI_VAddr == 0) {
		printf("siinit: sptalloc() failed");
		u.u_error = ENOMEM;
		return;
	}
	if ((SIvecnbr = get_vec(Drv_id, dr11intr)) < 0) {
		u.u_error = EBUSY;
		return;
	}
	hw_init();	/* Initialize the hardware. */
	DriverInitialized = 1;
}
```

sirelease()

The **devrelease(2K)** routine reverses the actions taken by **devinit(2K)**. CTIX calls **sirelease()** as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVUNBIND**. The **lddrv(1M)** program makes this system call.

If the device is open, it cannot be released: the driver should print a message, set **u.u_error**, and return.

Next, the release routine should do whatever is necessary to the hardware to ensure that it does not cause any unwanted activity after the driver is unloaded. Specifically, **devrelease(2K)** should abort any outstanding I/O and disable any interrupts that the device has been programmed to generate. If the device initiates any activity after the driver has been unbound, it may cause a system crash.

After the driver has shut down the hardware, it should return any interrupt vectors that it acquired in **devinit(2K)**. Also, if the driver has any outstanding **timeout(2K)** calls, it should call **untimeout(2K)** to clear them.

Next, **sirelease()** frees the system page table entries that **siinit()** acquired.

Before it exits, **devrelease(2K)** should clear the initialization flag, allowing the next call to **devinit(2K)** to succeed.

```
/*
 * Release the driver and the device - see devrelease(2K).
 */
sirelease()
{
    SDEC;

    /* Cannot release device if it is open. */
    if (DevOpen) {
        printf("sirelease: attempt to release open device");
        u.u_error = EBUSY;
        return;
    }
    hw_shutdown();          /* Shut down the hardware */
    SPL_SI;
    /* Reset the acquired interrupt vector(s). */
    reset_interrupt_vectors();
    SPLX;
    /* Return kernel virtual memory. */
    sptfree(SI_VAddr, dtop(MAXBLK)+1, 0);
    /* Re-allow siinit() calls. */
    DriverInitialized = 0;
}
```

siopen()

The Speech Interface board is an exclusive use device: only one **open(2)** call at a time should succeed. There must be an intervening **close(2)** call before another **open(2)** call. This is simple to achieve by testing and setting a flag in the **devopen(2K)** routine, and clearing the flag in **devclose(2K)**.

Siopen() resets the hardware to ensure that the driver starts out in a known state. This is an important step in any **devopen(2K)** routine.

Finally, **siopen()** sets the device open flag, to ensure exclusive access.

siclose()

Siclose() reverses the actions taken by **siopen()**. First, it resets the hardware. Then, **siclose()** clears the exclusive use flag, allowing another **open(2)** call to succeed.

```
/*
 * Open the device - see devopen(2K).
 */
siopen(dev)
dev_t dev;
{
	/* Exclusive use device - only one open(2) at a time. */
	if (DevOpen) {
		u.u_error = EBUSY;
		return;
	}
	hw_reset();		/* Reset the hardware. */
	/* Lock out siopen() calls until siclose(). */
	DevOpen = 1;
}


/*
 * Close the device - see devclose(2K).
 */
siclose(dev, flag)
dev_t dev;
int flag;
{
	/* Reset the hardware. */
	hw_reset();

	/* Clear the exclusive use flag. */
	DevOpen = 0;
}
```

siread() - siwrite()

The Speech Interface device performs DMA-driven, physical I/O. All of the work for reads and writes is handled by **physio(2K)** and the **devio(2K)** routine, that is, **siio()**. Both **siread()** and **siwrite()** consist of calls to **physio(2K)** with the address of the **siio()** routine passed as a parameter.

```
/*
 * Fill the user's buffer directly from the device - see devread(2K).
 */
siread(dev)
dev_t dev;
{
	physio(siio, &SI_Buf, dev, B_READ);
}


/*
 * Write the user's buffer directly to the device - see devwrite(2K).
 */
siwrite(dev)
dev_t dev;
{
	physio(siio, &SI_Buf, dev, B_WRITE);
}
```

siio()

Physio(2K) calls **siio()** to set up and start the DMA transfer. First, **siio()** calls **setmap(2K)** to remap the user's buffer into kernel virtual address space. Essentially, this assigns a second set of page table entries (in kernel space) to the user's physical buffer space. **Setmap(2K)** copies the page frame numbers from the user's page table entries into the kernel's page table entries. The kernel's page table entries were allocated in **siinit()**, by a call to **sptalloc(2K)**.

Next, **siio()** sets up the DMA registers to describe the pending transfer, sets the **SI_Active** flag, and starts the DMA transfer.

Siio() then returns to **physio(2K)**, which sleeps if necessary until **siintr()** calls **iodone(2K)** on the buffer header.

```
/*
 * Set up and start the DMA operation - see devio(2K).
 */
VOID
siio(bp)
struct buf *bp;
{
    /* Remap the user's buffer into kernel virtual memory. */
    KernVaddr = setmap(bp, SI_Vaddr, 0, bp->b_bcount);

    /* Setup the DMA transfer. */
    rw = (bp->b_flags & B_READ) ? SI_READ : SI_WRITE);
    hw_setup(KernVaddr, bp->b_bcount, rw);

    /* DMA is active */
    SI_Active = 1;
    /* Enable interrupts and start the DMA operation. */
    hw_go();
}
```

siintr()

CTIX calls **siintr()** whenever it receives an interrupt from the Speech Interface board.

Upon entry, **siintr()** checks to see if DMA is active. If it is, **siintr()** calls **iodone(2K)**, which sets the **B_DONE** bit and wakes up **physio(2K)** and any other process that is sleeping on the address of the buffer header. Then, **siintr()** clears the DMA active flag.

If DMA is not active, **siintr()** prints an error message. Note that the interrupt handler does not set **u.u_error**, since the current u-page probably belongs to a process that is unrelated to the **SI** device.

Finally, **siintr()** returns (exits from the interrupt).

```
/*
 * Process a DMA completion interupt - see devintr(2K).
 */
siintr(vecnbr)
int vecnbr;
{
	/* Check if valid interrupt. */
	if (SI_Active) {
		if (hw_status == OK) {/* No errors on DMA operation */
			SI_Buf.b_resid = 0;
			SI_Buf.b_error = 0;
		} else {		/* Some kind of error */
			SI_Buf.b_flags |= B_ERROR;
			SI_Buf.b_resid = SI_Buf.b_bcount;
			SI_Buf.b_error = EIO;
		}
		iodone(&SI_Buf);
		hw_clrint();		/* Clear the interrupt */
		SI_Active = 0;
	} else {
		printf("siintr: spurious interrupt");
	}
}
```

siioctl()

Siioctl() processes **ioctl(2)** calls from the user. It generally handles user requests that are specific to the device. The routine often is little more than a large **switch** statement, with one **case** for each legal request.

```
/*
 * Process user ioctl(2) call - see devioctl(2K).
 */
siioctl(dev, cmd, addr, flag)
dev_t dev;
int cmd;
caddr_t addr;
int flag;
{
	switch(cmd) {
		break;
	case SISTOP:
		/* Stop recording */
		hw_stop();
		break;
	default:
		u.u_error = EINVAL;
		break;
	}
}
```

6 CHARACTER DEVICE EXAMPLE

This chapter contains the annotated source listing of the device driver for the Ikon 10084 DR11-W Emulator. This is an actual driver that runs under the CTIX operating system on the MightyFrame.

The DR11W is a high-speed, DMA-driven parallel interface that can be used for intermachine linkage between a variety of computer systems. There are DR11-like devices available for the UNIBUS, QBUS, Multibus, VERSAbus, and VMEbus.

The device driver that follows is a simple one. Several features could have been added that would have provided more functionality at the cost of greater complexity. Still, the driver serves as an interesting example because it is interrupt-driven, it performs physical (raw) I/O between the device and the user's memory space, and it detects and handles hung transfers.

See the *Ikon 10084 DR11-W Emulator Hardware Manual* for a complete description of the hardware and its functionality.

Throughout this chapter, source code appears on the right hand page, while the annotations to it are on the left.

DR11 INCLUDE FILES

This page contains the include files and all **extern** declarations. The driver routines also are declared for the purpose of documentation. All of the include files are found in **/usr/include/sys**. Their general contents are as follows:

sys/param.h contains fundamental system constants that change very rarely from machine to machine.

sys/systm.h contains **extern** declarations for the most important system variables, data structures, and functions in the CTIX operating system.

sys/buf.h contains the declaration for the buffer header structure **buf**, and the flag definitions of the form **B_FLAG**.

sys/user.h contains the declaration for the **user** structure. This holds the per-process information not needed by CTIX while the process is swapped out. It also contains the per-process supervisor stack, used during system call processing.

sys/page.h contains fundamental memory management constants and the declaration of the software and hardware page table entry structures, **pte** and **hpte**.

sys/errno.h contains the system error constants as described in the Introduction to Section 2 of the *CTIX Operating System Manual.*

sys/spl.h contains the "set priority level" macros as described in **SPL(2K)** in Appendix A, *CTIX Interface Manual Pages.*

The **DFLT_ID/Drv_id** mechanism is handled completely by the loader. Simply include these two lines in every driver, and the driver ID will be assigned properly, whether it is loadable or is configured in with the kernel.

```
/*******************************************************
 * dr11.c
 *
 * CTIX 5.0 driver for Ikon 10084 VMEbus DR11-W Emulator
 *******************************************************/
#include "sys/param.h"
#include "sys/systm.h"
#include "sys/buf.h"
#include "sys/iobuf.h"
#include "sys/dir.h"
#include "sys/user.h"
#include "sys/page.h"
#include "sys/errno.h"
#include "sys/spl.h"

#define VOID int  /* To document routines returning no value */

VOID    dr11open();    /* Open the device and initialize the hardware */
VOID    dr11close();   /* Close the device, no actions taken w/hardware */
VOID    dr11read();    /* Call physio(2K) to do a read */
VOID    dr11write();   /* Call physio(2K) to do a write */
VOID    dr11io();      /* Setup and start the DMA operation */
VOID    dr11ioctl();   /* Reset the interface to clear hung driver  */
VOID    dr11intr();    /* Service an interrupt */
VOID    dr11status();  /* Print a status message on the console */
VOID    dr11timer();   /* Called periodically to complete timed out I/Os */
VOID    dr11init();         /* Initialize the interface and setup driver */
VOID    dr11release(); /* Disengage the driver, terminate operations */

/* Externs used by the driver */
extern  int       chkbusflt();
extern  VOID      iodone();
extern  VOID      physio();
extern  int       reset_vec();
extern  int       set_vec();
extern  caddr_t   setmap();
extern  caddr_t   sptalloc();
extern  VOID      sptfree();
extern  int       timeout();
extern  VOID      untimeout();

extern  int  haveVME;     /* Non-zero if VMEbus present in system */
extern  int  DFLT_ID;     /* Needed to make the driver loadable */
static int Drv_id = (int) &DFLT_ID;
```

The **ikon** data structure defines the device interface registers. A pointer to this structure (dr_aux->dr_addr) is initialized to the physical address of the hardware (**RA_PHYS**) in **dr11init()**. Thereafter, reads and writes to the members of the structure actually reference the hardware.

Briefly, the fields and their meanings are as follows:

status — is the status register when read and the control register when written. The relevant bit definitions for each case are given below.

data — is the 16-bit read/write data register.

modvec — contains two 8-bit values: the VMEbus address modifier (AM) bits in the upper byte, and the programmable interrupt vector number in the lower byte.

pulse — is a write-only copy of the control bits in the **status** register. Writing a 1 to any of these bits activates this function only, freeing the programmer from carrying around a copy of the status bits. Writing a 0 to any of these bits does nothing.

lowadr — contains the low-order 16 bits of the DMA address (bits 15-00).

range — contains the DMA transfer count in 16-bit words.

highadr — The low-order byte of the register contains the high-order 8 bits of the DMA address (bits 23-16). The high-order byte of the register is ignored.

Consult the *Ikon 10084 Hardware/Software Manual* for more details.

```
/* Ikon 10084 device structure.  It maps onto the hardware registers */
struct ikon {
	ushort	status,		/* Control and status register */
		data,		/* Input/output data register */
		modvec,		/* Addr Modifier/Int vector */
		pulse,		/* Pulse command register */
		word08,		/* Unused */
		word0a,		/* Unused */
		word0c,		/* Unused */
		word0e,		/* Unused */
		word10,		/* Unused */
		lowadr,		/* Low DMA address register */
		range,		/* DMA range counter */
		lcuradr,	/* Lowadr when read */
		word18,		/* Unused */
		highadr,	/* High DMA address register */
		word1c,		/* Unused */
		word1e,		/* Unused */
		hcuradr		/* Highadr when read */
};
/*
 * Control register values.
 */
#define RA_ZERO	0x0000	/* Clear all status bits */
#define RA_RDMA		0x8000	/* Reset DMAF and BERR flags */
#define RA_RATN		0x4000	/* Reset ATTN flag */
#define RA_RPER	0x2000	/* Reset PERR flag */
#define RA_CLEAR	(RA_RDMA|RA_RATN|RA_RPER) /* Clear flags */
#define RA_GO	0x0001	/* GO bit (start DMA transfer) */
#define RA_START	(RA_CLEAR|RA_GO)	/* Clear errs & GO */
#define RA_INIT	0x1000	/* Master clear the board */
/*
 * Status register values.
 */
#define RA_DONE		0x8000	/* DMA done */
#define RA_ERROR	0x4000	/* ATTN from model one (unused)*/
#define RA_PARITY	0x2000	/* Reset parity errors (unused) */
#define RA_READY	0x0080	/* Interface ready */
#define RA_IENB		0x0040	/* Interrupt enable */
#define FCN1		0x0002	/* Function bit 1 */
#define FCN2		0x0004	/* Function bit 2 */
#define FCN3		0x0008	/* Function bit 3 */
#define RA_WRITE	0x0000	/* No function code when writing */
#define RA_READ		FCN2	/* Set FCN2 when reading */
```

The driver control structure **dr_aux** contains general information about the device, any active transfer, and the timeout parameters. The variables have been gathered into a structure to make it easy to extend the driver. If one or more additional DR11's are added to the system, the structure will become an array indexed by the minor device number.

Dr11buf is a buffer header dedicated to this device. **Physio(2K)** uses it to describe the I/O that it sets up. When more DR11's are added to the system, **dr11buf** will become an array.

Dr11vad is a pointer that is set by **dr11init()** to reference a region of kernel virtual memory allocated by **sptalloc(2K)**.

RA_PHYS is the address of the device in I/O space. The DR11 must have an I/O address between 0xC0000000 and 0xC0FFFFFF, since it is an A24 device. Placing it at 0xC0C00000 means that it can be accessed directly by a user process using the VMEbus Protection register. However, since it is a DMA device, the user must not do this. See Chapter 2, *Architectural Information*, for more information.

The defines labelled **Hardware Constants** reflect the settings of hardware straps on the DR11 board itself.

SPLDR11 serves to localize the hardware interrupt level to one place in the driver. If it changes, only this one line must be modified.

```
/*
 * Driver control structure.
 */
struct dr_aux {
     struct ikon *dr_addr;     /* Address of Ikon board */
     struct buf *dr_actf;      /* Currently active buf */
     ushort dr_flags;          /* Open, active, ... */
     int  dr_timing,           /* Countdown till timeout */
          dr_timeout;          /* ID from last timeout(2K) call */
} dr_aux;

/*
 * Values for dr_flags.
 */
#define  DR_OPEN     1     /* The driver is open */

/*
 * Driver data.
 */
struct buf dr11buf;      /* Buffer header for transfers */
char      *dr11vad;      /* Virtual address to map transfers */
int       vecnbr;        /* Interrupt vector number to use */

/*
 * Hardware constants.
 */
#define RA_PHYS ((struct ikon *) 0xc0c00000)  /* VMEbus address */
#define RA_MODBITS 0x3d00                     /* Address modifier bits */
#define RA_MODVEC (RA_MODBITS|vecnbr)  /* AM bits & Int Vector */

#define  SPLDR11SPL2                  /* Our interrupt mask level */
```

DR11OPEN()

Dr11open() sets the **DR_OPEN** flag, resets the hardware, and then returns. **RA_ZERO** disables interrupts. **RA_CLEAR** resets the **DMAF**, **BERR**, **ATTN**, and **PERR** status flags.

In order to implement an exclusive use device such as a line printer, **dr11open()** would test **dr_flags** and, if the driver was open, would set **u.u_error** to **EBUSY** and return.

DR11CLOSE()

Dr11close() simply clears the **DR_OPEN** flag.

The **devclose(2K)** routine is called only when the **last close(2)** is issued on the device. If three processes open a device, CTIX does not call the **devclose(2K)** routine when either of the first two processes closes it.

```
/*
 * dr11open() - open the DR11.
 *
 * Set DR_OPEN and reset the hardware.
 */
VOID
dr11open(dev, flag)
dev_t dev;
int  flag;
{
     dr_aux.dr_flags |= DR_OPEN;
     RA_PHYS->status = RA_ZERO;
     RA_PHYS->pulse = RA_CLEAR;
}
```

```
/*
 * dr11close() - close the device.
 *
 *  Clears the DR_OPEN flag.
 */
VOID
dr11close(dev, flag)
dev_t    dev;
int flag;
{
     dr_aux.dr_flags &= ~DR_OPEN;
}
```

DR11READ() - DR11WRITE()

Both **dr11read()** and **dr11write()** use **physio(2K)** to set up the buffer for the data transfer. Normally, **physio()** is used by block devices to perform raw I/O directly to the user process.

All of the information about the original **read(2)** or **write(2)** system call is contained in the **user** structure of the requesting process. CTIX software sets up two fields in particular: **u.u_base** contains the virtual address of the data buffer in the user's memory, and **u.u_count** contains the transfer length.

Upon entry, **physio(2K)** verifies the count parameter and checks that the user has the required access permission on the buffer. **Physio()** then calls **pglock()**, which faults in all of the buffer pages from the swap device, and locks them into memory. Since the DMA operation will take place into or out of this memory, it must be present physically. After locking the pages, **pglock()** makes a copy of the user's page table entries that point to the buffer.

Next, **physio()** checks the state of the **B_BUSY** flag in the buffer header. If it is set, **physio()** sleeps until the buffer is available (**B_DONE** is set). It then sets up the buffer header to describe the transfer; that is, it sets **b_addr**, and **b_bcount**.

Finally, **physio()** calls the **devio(2K)** routine to perform the I/O. When **dr11io()** returns, **physio()** sleeps, waiting for the driver's interrupt handler to call **iodone(2K)**. This will cause the user's process to be rescheduled.

After the I/O is complete, **physio()** unlocks the user's buffer pages, allowing them to swap again. If the **B_WANTED** bit is set in the buffer header, it issues a **wakeup(2K)** call; then it sets **u.u_count** to **b_resid**, thus returning the number of bytes of data that were not transferred. Finally, it sets **u.u_error** to **EIO** if the **B_ERROR** bit is set in the buffer header.

```
/*
 * dr11read() - read some data using physio(2K).
 *
 * Call physio(2K) with the address of dr11io()
 * as the devstrategy(2K) routine and the B_READ
 * flag.
 */
VOID
dr11read(dev)
{
    physio(dr11io, &dr11buf, dev, B_READ);
}


/*
 * dr11write() - write some data using physio(2K).
 *
 * Call physio(2K) with the address of dr11io()
 * as the devstrategy(2K) routine and the B_WRITE
 * flag.
 */
VOID
dr11write(dev)
{
    physio(dr11io, &dr11buf, dev, B_WRITE);
}
```

DR11IO()

Dr11io() is called by **physio(2K)** as a result of a **read(2)** or **write(2)** system call. Its main purpose is to program the hardware to perform the requested DMA operation.

The call to **setmap(2K)** is important. The original **read(2)** or **write(2)** request referenced a buffer in the user's virtual address space. But the transfer actually takes place when some other process is running, because **physio(2K)** calls **sleep(2K)** and gives up the CPU after its call to **dr11io()**. Since each process has its own set of page table entries, virtual addresses in user space are valid only when that process is running.

The driver cannot program the DMA hardware with the original virtual address of the buffer. In fact, it cannot reference user virtual memory at all, since that changes whenever there is a context switch. The DMA hardware must use kernel virtual memory, which always is valid. This kernel virtual memory is reserved by a call to **sptalloc(2K)** in **dr11init()**. **Sptalloc()** allocates a contiguous region of kernel virtual address space to serve as a "window" on the user's I/O buffer in physical memory. The pointer to this "window" is kept in **dr11vad**.

In the discussion of **dr11read()/dr11write()** above, it was pointed out that the **pglock()** function made a copy of the page table entries pointing to the user's I/O buffer. The address of those saved pte's was stored in **bp->b_pt**. Now, the **setmap(2K)** routine takes the page frame numbers from each of these saved page table entries and writes them into the pte's reserved by **sptalloc(2K)**. In this way, the user's I/O buffer acquires a kernel virtual address, in addition to its user virtual address. This new kernel address is used to program the DMA device on the DR11 board.

Dr11io() also sets up the timeout value in **dr_aux.dr_timing**. This ensures that hung DMA transfers will be aborted, and the requesting process notified of the failure.

```
/*
 * dr11io - setup and start the DMA operation.
 * Remap the user's buffer into kernel virtual memory,
 * program the Ikon's DMA registers, set the timeout,
 * and start the transfer.
 * This routine is called by physio(2K) to start the I/O.
 */
VOID
dr11io(bp)
struct buf *bp;
{
	register unsigned count;
	unsigned int addr;
	register short flag;
	register dev_t dev;
	short cstatus;

	flag = bp->b_flags;
	dr_aux.dr_actf = bp;
	count = ((bp->b_bcount + 1) >> 1) - 1;
	dev = bp->b_dev;
	/*
	 * Remap the user's buffer into kernel virtual memory.
	 */
	addr = (unsigned int) setmap(bp, dr11vad, 0, bp->b_bcount);
	/*
	 * Program the low and high address registers, and count.
	 */
	RA_PHYS->lowadr = (short) (addr >> 1);
	RA_PHYS->highadr = ((short) (addr >> 17) & 0x3f);
	RA_PHYS->range = (short) count;
	/* Clear any lurking ATTN or DMA interrupt flags. */
	RA_PHYS->pulse = RA_CLEAR;
	RA_PHYS->modvec = RA_MODVEC;
	/* Setup the software timeout. */
	dr_aux.dr_timing = 10;

	/* Enable interrupts start the DMA operation. */
	if ((flag & B_READ) == B_READ)
		RA_PHYS->pulse = (RA_READ|RA_START|RA_IENB);
	else
		RA_PHYS->pulse = (RA_WRITE|RA_START|RA_IENB);
}
```

DR11INTR()

The interrupt handler is called from two places for two entirely different events. First, it is called from **perint()** in CTIX whenever an interrupt is received from a device supplying the vector number that was reserved for the DR11 by the **get_vec(2K)** call in **dr11init()**. Also, it is called from **dr11timer()** when that routine detects a hung DMA transfer.

In either case, the result is the same. First, **dr11intr()** checks to see that there is a buffer active. If not, it prints a diagnostic message and returns. Next, it checks to see if the **RA_READY** bit is set in the device status register. Whenever **dr11intr()** is called as a result of a hardware interrupt, this bit will be set. This test is present only to catch the unlikely event that the transfer timed out but then completed normally before **dr11timer()** called **dr11intr()**.

If **dr_aux.dr_timing** is less than zero, it indicates that a timeout has occurred. This causes a diagnostic to be printed on the console and the **B_ERROR** bit ,iw "**B_ERROR**" to be set in the buffer header. Next, **dr11intr()** cancels the timer. (But this is a soft cancel: **dr11timer()** continues to run periodically, because of its call to **timeout(2K)**.)

Finally, and most importantly, **dr11intr()** calls **iodone(2K)** to set the **B_DONE** bit in the buffer header and issue a **wakeup(2K)** call. This restarts the original process in the **physio(2K)** routine. (It also restarts any process that set the **B_WANTED** bit in the buffer header and issued a **sleep(2K)** call.) When it gets rescheduled, **physio()** passes back the status of the transfer in the user area and returns, either to **dr11read()** or **dr11write()**.

Notice that the **devintr(2K)** routine cannot reference the user area. Since it runs asynchronously, the process that issued the original I/O request is no longer active. The current user area belongs to an entirely different process.

If your driver has a timeout feature, you should reinitialize the hardware whenever a timeout occurs.

```
/*
 * dr11intr() - interrupt service routine.
 *
 * Service an interrupt. Note: the interrupt is a
 * one-shot and must be re-enabled for each read or
 * write.
 *
 * This routine also is called from dr11timer()
 * to complete an I/O that has timed out.
 */
VOID
dr11intr()
{
    register struct buf *bp;
    register ushort cstatus;

    cstatus = RA_PHYS->status;
    /* Check if valid interrupt. */
    if ((bp = dr_aux.dr_actf) == (struct buf *)0) {
        dr11status("Spurious dr11 Interrupt", cstatus);
        return;
    }
    /* Check if interface is ready. */
    if (cstatus & RA_READY)
        dr_aux.dr_timing = 0;

    /* Check if software interrupt. */
    if (dr_aux.dr_timing < 0 ) {
        dr11status("timeout", cstatus);
        bp->b_error |= B_ERROR;
    }
    /* Cancel the timeout. */
    dr_aux.dr_timing = 0;
    iodone(bp);
    dr_aux.dr_actf = 0;
}
```

DR11STATUS()

Dr11status() prints uniformly formatted status messages on the system console.

Dr11ioctl() simply resets the hardware, in order to clear a hung condition. This is unusually brief for a **devioctl(2K)** routine, but this, more than any other part of the driver, is yours to use as you see fit.

```
/*
 * dr11status() - print status message.
 *
 * Print a formatted status message on the console.
 */
VOID
dr11status(s, cstat)
char *s;
ushort cstat;
{
	printf("[dr11: %s %x]", s, cstat);
}


/*
 * dr11ioctl() - device-specific I/O control.
 *
 * Reinitialize the Ikon board to recover it from
 * hung conditions.  (This should not happen.)
 */
VOID
dr11ioctl(dev, cmd, addr, flag)
dev_t dev;
int   cmd;
caddr_t addr;
int   flag;
{
	RA_PHYS->status = RA_INIT;
}
```

DR11TIMER()

Dr11timer() runs periodically as a result of a **timeout(2K)** call. The first call is performed in **dr11init()** when the driver is loaded. The sustaining call is done by **dr11timer()** itself. In both cases, the ID returned by **timeout()** is saved so that the timeout can be cancelled when the driver is released.

When there is no I/O outstanding, **dr11timer()** runs every 10 seconds (that is, 10 * **HZ**). When there is I/O active, **dr11rtimer()** runs every second (that is, 1 * **HZ**).

The **SPLDR11/SPLX** macros are used to mask out interrupts from the DR11. This is because the **dr_timing** variable also is manipulated by **dr11intr()**, and disaster would result if it interrupted **dr11timer()**. Also, the **dr11intr()** routine is called from here. If this call were not protected by the **SPLDR11**, **dr11intr()** could interrupt itself. A close look at the code will tell you that it was not designed to support this. See **timeout(2K)** for more information about IPL management in functions that it calls.

Notice the **SDEC;** declaration. This must be included to provide storage for the previous value of the processor status word for **SPLDR11** and **SPLX**. See **SPL(2K)** for a complete discussion of these macros.

```
/*
 * dr11timer() - timeout routine setup by dr11init().
 *
 * As long as the driver is installed, this routine
 * is called periodically. When no DMA is in progress
 * it is called every 10 seconds.  When DMA is in
 * progress it is called every second.
 *
 * If a transfer times out, call dr11intr() to
 * complete the operation.
 */
VOID
dr11timer(arg)
int arg;
{

	int next = 10*HZ;
	SDEC;

	SPLDR11;
	if (dr_aux.dr_timing > 0) {
		next = HZ;
		if(--dr_aux.dr_timing == 0) {
			dr_aux.dr_timing = -1;
			dr11intr();
		}
	}
	dr_aux.dr_timeout = timeout(dr11timer, 0, next);
	SPLX;
}
```

DR11INIT()

The **haveVME** flag is an external that indicates the presence of the VMEbus expansion board. The call to **chkbusflt(2K)** verifies the presence of the DR11 board at the **RA_PHYS** address. (Actually, it only indicates the presence of **something** at that address.)

Dr11init() uses the value of the **dr11vad** pointer to determine if the driver has been bound already.

The call to **sptalloc(2K)** allocates a contiguous region of kernel virtual memory only; no physical memory is allocated. Later, the driver sets these page table entries to point to the physical memory that contains the user's I/O buffer. Note that the size of this region is one more than the number of pages required to hold the largest allowable transfer for block devices (**MAXBLK** 1K blocks). The extra page allows the buffer to start in the middle of a page and still be **MAXBLKs** long. If the user issues a **read(2)** or **write(2)** request for more than the maximum allowable size, **physio(2K)** fails with **EFAULT**.

Since the DR11 board supports software-programmable interrupt vector generation, the driver issues a **get_vec(2K)** call to allow CTIX software to assign an available vector. If the board required strapping the vector number, **dr11init()** would have called **set_vec(2K)** with a constant equal to the strapped vector number.

The **timeout(2K)** call starts the deadman timer running with a 10 second timeout. **Dr11timer()** continues the timer with another **timeout()** call.

Finally, **dr11init()** sets up the VMEbus address modifier bits and Interrupt Vector register and initializes the hardware.

```
/* dr11init() - initialize a loadable driver.
 *
 * This routine is called by drvbind() in response to a
 * syslocal(2) call with a parameter of SYSL_BINDDRV and
 * an argument of DRVBIND.
 *
 * If the driver is loaded already, or any errors are
 * encountered during initialiation, u.u_error is set,
 * terminating the loading of the driver.
 *
 * This routine sets up the virtual address for mapping data
 * for read and writes, and initializes the hardware.
 */
 VOID
dr11init()
{
	if (!haveVME |chkbusflt(RA_PHYS, 0)) {
		printf("dr11: no VME or dr11 board installed");
		u.u_error = ENXIO;
		return;
	}
	if (dr11vad != NULL) {
		printf("dr11: driver already installed");
		u.u_error = EBUSY;
		return;
	}
	if ((dr11vad = (char *) sptalloc(dtop(MAXBLK)+ 1,
			(PG_V|PG_KW), -1)) == NULL) {
		printf("dr11: cannot allocate memory");
		u.u_error = ENOMEM;
		return;
	}
	if ((vecnbr = get_vec(Drv_id, dr11intr)) < 0) {
		printf("dr11: cannot get interrupt vector");
		u.u_error = EBUSY;
		return;
	}
	dr_aux.dr_timeout = timeout(dr11timer, 0, 10*HZ);
	dr_aux.dr_addr = RA_PHYS;

	RA_PHYS->modvec = RA_MODVEC;
	RA_PHYS->status = RA_INIT;
}
```

DR11RELEASE()

If the device is open, it can't be released. If it is not open, **dr11release()** simply deallocates the kernel virtual memory region that **dr11init()** acquired, cancels the outstanding **timeout(2K)** request, and gives back the interrupt vector number.

```
/*
 * dr11release() - release a loadable driver.
 *
 *  This routine deallocates the memory used by the
 *  driver, cancels any outstanding timeout(2K) call,
 *  and gives back the device's interrupt vector.
 */
VOID
dr11release()
{
	SDEC;

	if (dr_aux.dr_flags & DR_OPEN) {
		u.u_error = EBUSY;
		return;
	}
	sptfree(dr11vad, dtop(MAXBLK)+1, 0);
	SPLDR11;
	/*
	 * Cancel the timer.
	 */
	untimeout(dr_aux.dr_timeout);
	/*
	 * Give back the interrupt vector.
	 */
	reset_vec(Drv_id, vecnbr);
	SPLX;
}
```

7 BLOCK I/O TUTORIAL

This chapter describes the Block I/O system in detail, including the system buffer cache, and the general disk driver. The chapter also contains an example device driver for a general disk-type block device. The example is written in a C-like pseudocode and includes a program narrative describing the driver in detail.

OVERVIEW

The Block I/O system supports random access devices that transfer data in fixed length "chunks." It is sometimes called the buffered I/O system, since it makes use of the buffer cache to reduce the amount of physical I/O in the system. Disk drives are the most common block devices. Tape drives also can be supported here, but they are frequently classified as raw character devices. In actual practice, fewer and fewer nondisk devices are handled by the Block I/O system.

The user rarely opens a block device directly: the most common interface to the Block I/O system is through the file system. When the user opens a data file, CTIX software reads the i-node from the disk and determines that it is not a special (device) file. From this point on, the kernel routes all **read(2)** and **write(2)** requests through the file system, rather than directly to a device driver.

When the user requests to read data from a file, CTIX first searches the buffer cache. If the data is present, CTIX returns it immediately, with no disk I/O activity. If the data is not present, CTIX acquires the "oldest" buffer in the cache to hold the new data. If this buffer contains data that has not yet been written to disk, CTIX calls the device driver's **devstrategy(2K)** routine to write the buffer. When the driver completes the write, the system is free to reuse the buffer.

Once it has acquired a free buffer, CTIX software calls the device driver's **devstrategy(2K)** routine to fill it with the data that the user requested. When the read operation is complete, CTIX copies the data from the system buffer to the user's buffer and returns. When it has read in a block from disk, the Block I/O system attempts to keep it in the buffer cache as long as possible.

When the user requests to write data to the file, the operating system first checks to see if the file offset lies exactly on a block boundary and if the transfer length is an even multiple of blocks long. If either of these criteria is not met, CTIX must first preread a block and merge the new data into it. As with a normal read, if the data is present in the cache, CTIX does not need to call the driver to perform physical I/O.

Whether or not CTIX performed a preread, the write request is satisfied asynchronously. The new or modified buffer is inserted into the cache: it is not written to disk immediately. The Block I/O system leaves the unwritten data in the cache as long as possible. CTIX writes the data to disk only when it needs to reuse the buffer for another request. The asynchronous nature of the Block I/O system makes it impossible to report physical write errors to the correct process.

Because data written to block devices is retained in memory as long as possible, CTIX is prone to file system corruption when the system crashes. In order to minimize the effects of crashes, CTIX provides the **sync(2)** request, which writes all of the modified cache blocks to the disk. System administrators usually run the **update(1M)** program to synchronize all disks periodically.

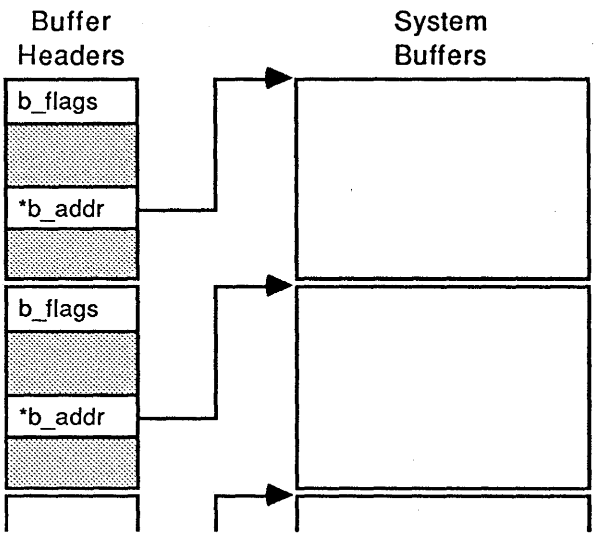
SYSTEM BUFFER CACHE

The system buffer cache is composed of **buf** structures known as buffer headers, and blocks of kernel memory that are used as buffers. Each buffer header describes the contents (including the block and device number) of the associated buffer. The header also contains two pairs of pointers: **b_forw/b_back**, and **av_forw/av_back**. Each of these pointer pairs can be used to place the buffer on a doubly linked list or queue.

BASIC STRUCTURE

CTIX software allocates space for the buffer cache at system initialization time. The cache consumes at least 15 percent of all available memory. If available memory is limited, CTIX ensures that the cache contains at least 16 buffers. The cache is made up of an array of **buf** structures and a separate array of buffers. Each buffer is the length of one file system block, which is 1,024 bytes on the MightyFrame. This may or may not be the same length as one physical sector on a disk drive. The header file **<sys/buf.h>** contains the definition of the **buf** structure.

The following diagram illustrates the basic structure of the system buffer cache. Note that each member of the cache is composed of both a **buf** structure (or buffer header) and a buffer.



System Buffer Cache

NOTE
Many drivers, especially character drivers that perform physical I/O, allocate one or more **buf** structures for their own internal use. These buffer headers are not part of the system buffer cache.

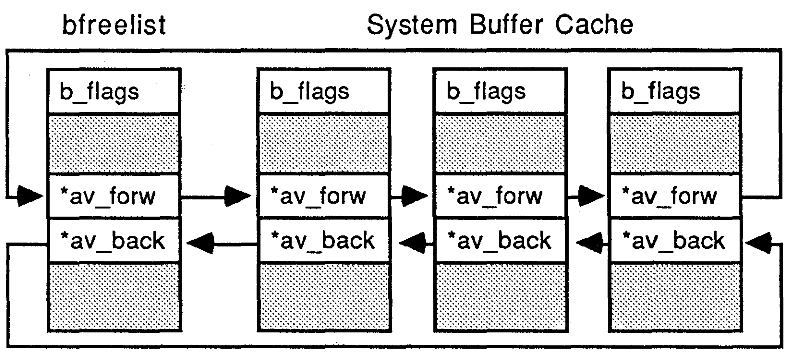
AVAILABLE (FREE) LIST

After allocating memory for the buffer cache and linking each buffer header to a buffer, CTIX places all of the buffers onto the available (or free) list. Each buffer on the available list is available for use by the Block I/O system.

Initially, of course, all of the buffers on the free list are empty: CTIX can use any of them for data. After the system has been running for some time, however, most of the buffers on the free list contain data that various users have read or written. These "full" buffers are also available in least recently used (LRU) order. When the Block I/O system needs a buffer, it selects the oldest one on the available list: the buffer that has been on the list for the longest time. If the buffer contains data that has not been written to disk, CTIX calls the appropriate **devstrategy(2K)** routine to write the contents before reusing the buffer.

The available list is a doubly linked list of buffer headers, with a separate **buf** structure named **bfreelist** serving as the list head. **Bfreelist** does not have an associated buffer, and it is not part of the system buffer cache: it is simply used to point to the head and tail of the free list. The **b_bcount** field of **bfreelist** contains the number of buffers on the list. (In a normal buffer header, this field contains the number of characters in the associated buffer.)

The following diagram illustrates the system available (free) list: **bfreelist** is on the left, the buffer headers are on the right. Only three headers are pictured. Also, to clarify the drawing, the buffers themselves are not shown. Note that each linked list is circular: the forward pointer of the final member in the list points to **bfreelist**, and the backward pointer of **bfreelist** points to the final list member.



System Available (Free) List

The only time a buffer is not on the free list is when a **devstrategy(2K)** routine has placed it on an I/O queue, waiting for a driver to transfer data into or out of it. The I/O queues are documented below.

HASH LISTS

Whenever a user issues a request that results in a read or write to a block device, CTIX software must search the buffer cache to see if the requested block is already in memory. This would be a very slow process if the buffers were simply linked together in a list. In order to reduce the search time, CTIX hashes the device and block number associated with every buffer in the cache. The device and block numbers together serve as the hash key.

The process of hash searching is simple:

1. Apply some function to the key value to transform it into a small, positive integer.

2. Use the integer directly as an index into a table of the objects you wish to search.

If the data base has a large number of key values, however, it may not be possible to find a function that produces a unique index for every key. In this case, the hashed value can be used as an index into a table of pointers. Each of these pointers serves as the head of a linked list of objects with key values that hash to the same index value. The new hash search algorithm is only slightly more complex:

1. Apply some function to the key value to transform it into a small, positive integer.

2. Use the integer directly as an index into a table of linked-list heads.

3. Examine the selected linked-list sequentially, comparing the key value of each member with the desired key.

This is exactly the scheme that the operating system employs to search the buffer cache.

At system initialization time, CTIX allocates space for an array of **hbuf** (hash buffer) structures of the following form:

```
struct hbuf
{
    int  b_flags;
    struct      buf *b_forw;   /* Forward pointer */
    struct      buf *b_back;   /* Backward pointer */
};

struct hbuf hbuf[NHBUFS];
```

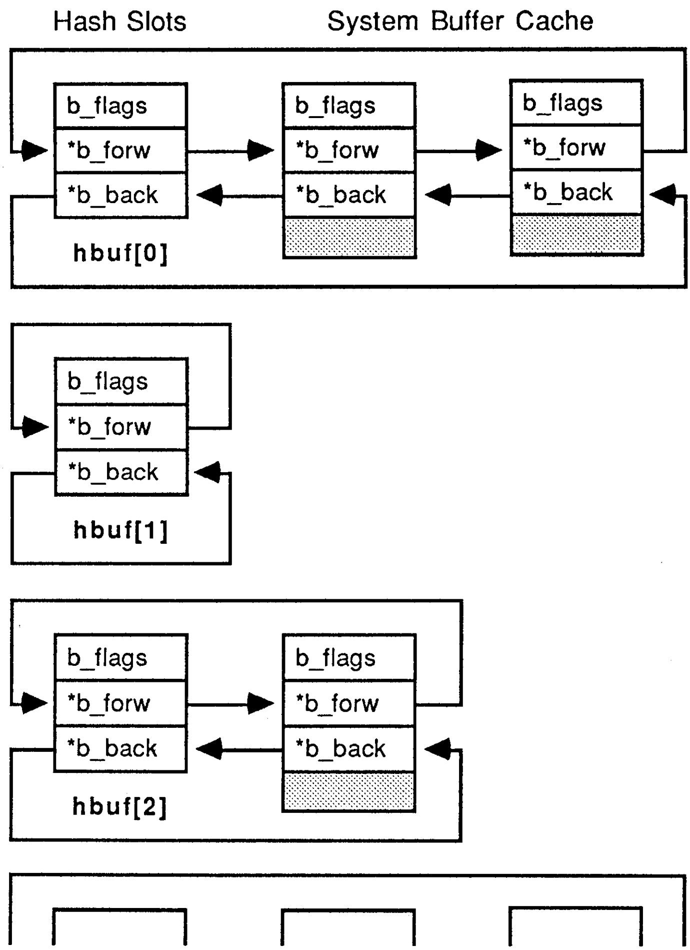
Each **hbuf** array member is called a hash slot in this document. Initially, each hash slot is empty: no buffer headers are linked to it. For each **read(2)** or **write(2)** request to a block device, CTIX software

1. Applies the hash function to the block and device number referenced in the request;
2. Uses the resulting number as an index into the **hbuf** array;
3. Uses the selected hash slot as a list head; then
4. Searches the selected linked list, comparing the block and device numbers in each buffer header with those of the requested block.

If CTIX does not find the requested block in the list, it is not in the buffer cache. In this case

- If the user is reading data, the operating system must allocate an available buffer and call the appropriate **devstrategy(2K)** routine to initiate a physical read operation.
- If the user is writing data, CTIX must allocate an available buffer to hold the new information. The new data will not be written out until this buffer again becomes the oldest on the list and needs to be reused for another request.

The following diagram illustrates the system hash lists.



System Hash Lists

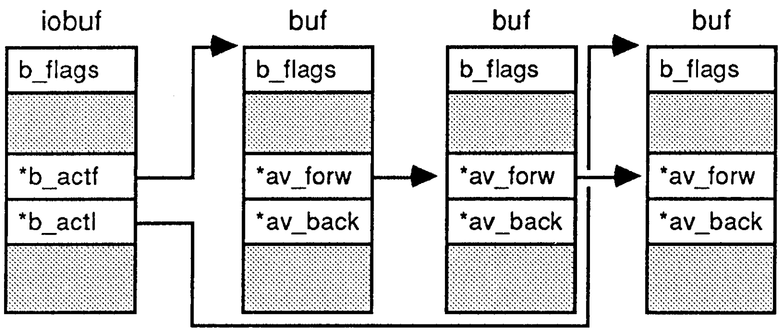
The data structures down the left side of the drawing are **hbuf** structures (hash slots): in their totality, they represent the **hbuf** array. The data structures on the right side of the drawing are **buf** structures: in their totality, they represent the system buffer cache. The buffers themselves are not shown.

Note that each linked list is circular: the forward pointer of the final member in the list points to the appropriate hash slot, and the backward pointer of each **hbuf** member points to the final list member. Note also that the hash list uses the **b_forw/-b_back** pointer pair, not the **av_forw/av_back** pair that the free list uses. A buffer can be (and usually is) on **both** the free list and a hash list at the same time.

I/O QUEUES

There is one I/O queue for each block device or DMA channel in the system. A disk controller that supports two drives concurrently has two associated I/O queues, not one. The I/O queue contains a linked list of all outstanding work for the driver to perform. The **devstrategy(2K)** routine places new work onto this queue, typically sorting it according to some algorithm that ensures the most efficient device access. The driver's **devintr(2K)** routine removes entries from the I/O queue when the requested transfer is complete. For general disk-type devices, **gdstrategy(2K)** and **gdintr(2K)** add and remove entries.

The following diagram illustrates the I/O queue for a block device.



I/O Queue - One per Block Device

The structure on the left of the drawing is an **iobuf**, which is defined in <**sys/iobuf.h**>. This structure is the list head of the I/O queue. The list members are themselves buffer headers, which are part of the buffer cache. The buffers themselves are not shown.

Note that the list is singly linked (the **av_back** pointer is unused) and that it is not circular (the **av_forw** pointer on the last buffer header is **NULL**). The pointers in the I/O queue head point to the first and last entries, unlike other buffer cache list heads.

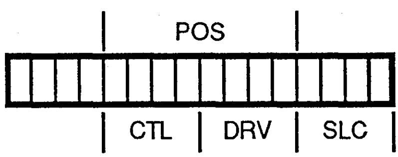
Since **av_forw** is used for the I/O queue, a buffer cannot be on both the available list and an I/O queue at the same time. This is consistent with the information presented previously: a buffer remains on the available list until it is placed on an I/O queue by the **devstrategy(2K)** routine.

While the buffer is on the I/O queue, the **B_BUSY** bit is set in the header, indicating that the buffer is unavailable. When the device driver calls **iodone(2K)** to report that the I/O transfer is complete, CTIX clears **B_BUSY** and places the buffer back on the end of the available list. This is now the "youngest" available buffer in the cache. It must "age" through the entire list before it is chosen for reuse. (This is true in general, but there are exceptions that are beyond the scope of this document.)

GENERAL DISK I/O QUEUE STRUCTURE

General disk-type devices use a slightly more complex structure for their I/O queues. Each disk drive has an I/O queue called a drive queue, with exactly the same structure as any block device I/O queue. There is one I/O queue per physical drive, **not** per slice (partition) within a drive. The heads of all of the drive queues are **iobuf** structures as they are for any other I/O queue: they are contained in an array named **gdutab**, which is declared in <**sys/space.h**>. The name **gdutab** means General Disk Unit Table.

The **gdpos()** macro, which is defined in <**sys/gdisk.h**>, takes a major + minor device number as a parameter and returns an index into **gdutab**. The following diagram illustrates the fields within a major + minor device number for a general disk-type device.



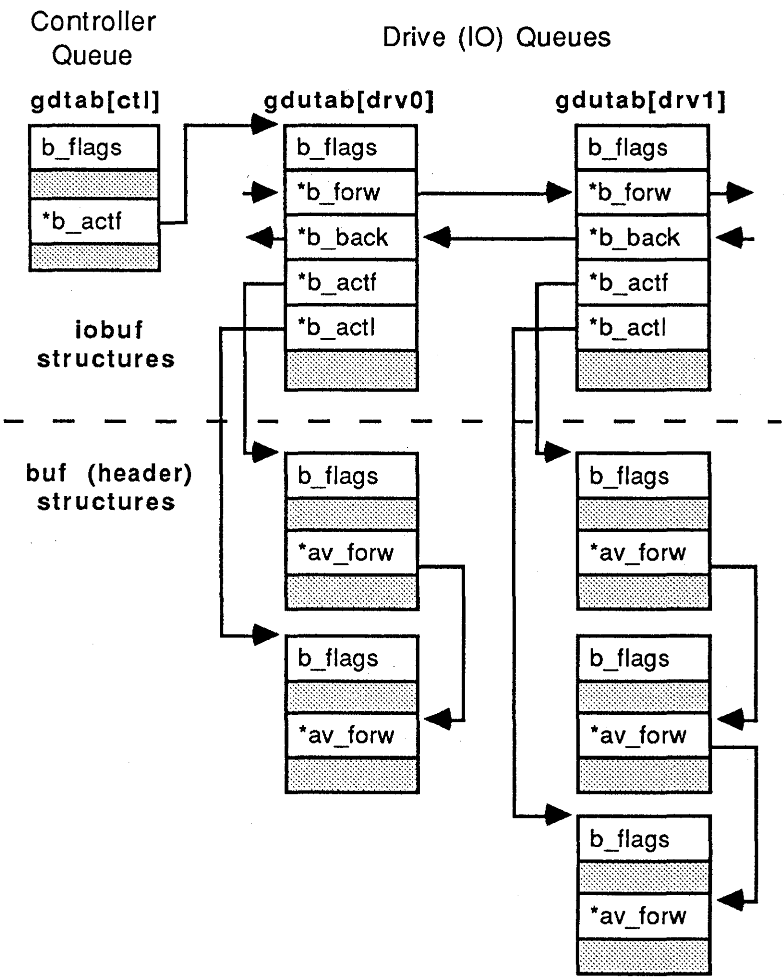
Major + Minor Device Number Fields
General Disk-Type Devices

Many disk controllers support two or more physical drives. These controllers frequently can perform simultaneous I/O operations on their drives. The operating system provides a second, higher level structure above the normal I/O queues to provide support for controllers of this type.

Each disk controller in the system has a controller queue, which is a linked list of all active I/O queues for the drives that are connected to the controller. The head of the controller queue is an **iobuf** structure: these list heads are kept in an array named **gdtab**, which is declared in <**sys/space.h**>. The forward and backward pointers in an **iobuf** structure actually refer to buffer headers: these must be cast to pointers to **iobufs** for the controller queue. The **gdctl()** macro, which is defined in <**sys/gdisk.h**>, takes a major + minor device number as a parameter and returns an index into **gdtab**.

The following diagram illustrates the multiply linked controller and I/O queue structures for one disk controller. The controller queue is represented by the three **iobuf** structures across the top of the figure. Note that the controller queue is circular, but the queue head points only to the first member. To simplify the drawing, the complete paths of the forward and back links between the first and last members are not shown.

The drive queues are drawn vertically: they are headed by **iobuf** structures at the top of the diagram, and contain several buffers in a singly linked list. Only the buffer headers are shown: the actual buffers have been omitted for clarity.



General Disk I/O Queue Structure
One per Disk Controller

SUMMARY

A portion of kernel memory is set aside for block device buffers. Each buffer can hold one file system block, which is not necessarily the same as one physical disk sector. Each buffer has a buffer header associated with it that describes the contents of the buffer and/or the I/O transfer parameters.

There are three queues associated with the system buffer cache:

- The available or free list.
- The hash lists.
- The I/O queues.

Buffers spend most of their time on two out of the three lists:

- Buffers that contain valid data are on both the hash list and the available list.
- Buffers that are waiting for a device driver to perform or complete an I/O transfer are on both the hash list and an I/O queue.

A buffer cannot be on both the available list and an I/O queue at the same time, because each list is implemented with the **av_forw/av_back** pointer pair. (Technically, the I/O queue is singly linked and uses only **av_forw**. In practice, however, various block device drivers "steal" **av_back** for data storage: for example, the general disk driver uses it to hold the track and sector address of the requested block.)

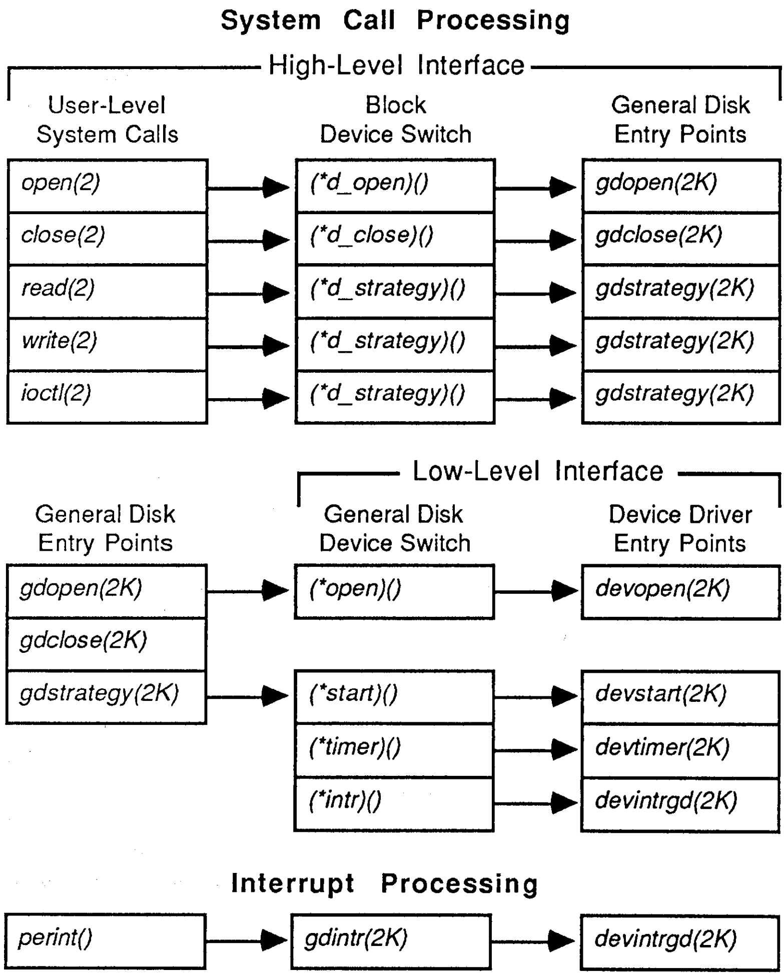
GENERAL DISK DRIVER

The general disk drive provides a device-independent interface to disk-like devices. It is exactly like a normal block device driver in its interface to the CTIX kernel: its entry points are inserted into each **bdevsw** array entry that corresponds to a disk-like device. Using the major device number as an index, CTIX calls the general disk open, close, print, and strategy routines whenever it needs to access a disk-like device.

The general disk driver differs from the normal block device driver in that it is not able to control any specific piece of hardware. Its sole purpose is to perform the device-independent tasks that are common to all disk-like drivers. When the general disk driver has done all of the work that it can, it calls the appropriate low-level (physical) device driver to carry out the actual I/O operation. The general disk driver uses the **gdpos()** macro with the major + minor device number to obtain an index into the **gdsw** array: **gdsw** contains the addresses of the entry points of the low-level device drivers.

Routines that start with the characters **gd** are part of the general disk driver: they are documented in Appendix A, *CTIX Interface Manual Pages.*

The following diagram (reproduced from the Introduction to Appendix A, *CTIX Interface Manual Pages*), illustrates the linkages between a user process and the general disk driver, and between the general disk driver and the low-level drivers for disk-like devices.



General Disk Driver Linkage

AN SMD DEVICE DRIVER

This section contains an example of a device driver for a hypothetical Storage Module Drive (**SMD**) controller: it is a typical general disk-type device. The section begins by describing the device and its environment. The tutorial driver follows this introductory material.

DEVICE ARCHITECTURE

This example driver is meant as a simple, tutorial introduction to the real disk driver presented in Chapter 8, *Block Device Example.* The **SMD** controller in this example supports only one drive. It can perform SEEK, READ, WRITE, and FORMAT operations only. Unlike the V/SMD 3200 Controller described in Chapter 8, *Block Device Example*, **SMD** does not support implied SEEKs: each SEEK command must be issued explicitly.

The READ and WRITE commands transfer one physical sector of information only. The FORMAT command formats the entire device into 1,024 byte physical sectors. Thus, each physical sector is equivalent to one file system block.

The **SMD** controller generates interrupts for only one condition: operation complete. A status register on the controller indicates whether or not the requested operation completed successfully. The controller performs its own timeout function. It generates an operation complete interrupt when it detects a hung condition. The status bits in a controller register indicate an operation timeout error.

THE PSEUDOCODE DRIVER

The example driver is written in C-like pseudocode. At times the pseudocode is abstract and general; at other times, it reads almost like an actual C program. This example is not meant to be exhaustive: in particular, it does not do adequate error detection and recovery. Unlike the example in Chapter 8, this driver does not perform dynamic bad block forwarding. Routines that begin with the characters **hw_** refer to hardware-specific code.

Use this example as a model to understand the V/SMD driver in Chapter 8. Study it before you attempt to read the more complex example. Your own driver almost certainly will have more functionality than this example.

smdopen()

Since general disk-type device drivers are not loadable, the work normally performed by the **devinit(2K)** routine must be performed by **devopen(2K)**. This work should be done only the very first time the device is opened, though, so it is made conditional on the variable **firsttime**. This one-time initialization is common for all VMEbus devices:

1. Check the **haveVME** flag to determine if the VMEbus Interface board is present in the MightyFrame.
2. Check that the EEPROM on the VMEbus Interface board has a valid checksum.
3. Search the information array in the EEPROM for the address of the desired device.
4. Check that the board is present at the VMEbus address by calling **probevme(2K)**. (Note that this really only determines that something is present at the address.)
5. Acquire any required interrupt vectors.
6. Perform any necessary hardware initialization.

```
/*
 * Open and initialize the SMD device - see devopen(2K).
 */
smdopen()
{
	struct vmeeprom *eeprom, *is_eepromvalid();
	static int firsttime = 1;
	int i;

	if (firsttime) {
		SMDAddress = 0;		/* Initialize */
		/* Make sure VMEbus is present and EEPROM is valid. */
		if (haveVME && ((eeprom = is_eepromvalid()) != 0)) {
			/* Search the EEPROM for our device */
			for (i=0; i<VME_SLOTS; i++) {
				if (eeprom->slots[i].type == VMET_SMD) {
					SMDAddress = eeprom->slots[i].address;
					break;
				}
			}
		}
		/* No VMEbus, invalid EEPROM, or no device. */
		if (SMDAddress == 0) {
			if (eeprom == 0)
				gdprint(dev, "Invalid VMEbus eeprom");
			u.u_error = ENXIO;
			return(0);
		}
		/* Check for presence of board. */
		if (probevme(SMDAddress) {
			gdprint(dev, "SMD Controller not present");
			u.u_error = ENXIO;
			return(0);
		}
		if ((vector = get_vec(Drv_id, gdintr)) < 0) {
			u.u_error = EBUSY;
			return(0);
		}
		hw_init(vector);
		firsttime = 0;
	}
	return(1);		/* Success */
}
```

smdstart()

The purpose of **smdstart()** is to start the next I/O request if the controller is not already busy. **Gdstrategy(2K)** calls **smdstart()** whenever it enqueues new work for the low-level driver to perform. **Gdintr(2K)** calls **smdstart()** whenever the **devintrgd(2K)** routine (**smdintr()**) reports that the current I/O request is complete. See the manual page for **devintrgd(2K)** for a discussion of the difference between I/O requests and I/O operations.

Smdstart() first checks for a null I/O queue (**dp**). It returns if there is no I/O queue.

The driver then checks the state of the controller and returns if it is already active servicing another request.

As a precaution, **smdstart()** checks for an empty I/O queue. If the I/O queue is empty, it should not have been enqueued onto **gdtab**. The **gdpanic(2K)** call reports a fatal problem in the high- and low-level disk driver interface.

Next, **smdstart()** sets up the parameters for the current transfer in the **XferInfo** structure. Note that the general disk driver supports only these three commands: **CMD_FORMAT**, **CMD_READ**, and **CMD_WRITE**. In keeping with the CTIX design philosophy, the general disk driver deals with a simple model of disk activity: it is up to the low-level device driver to map the model to the real world.

Finally, **smdstart()** calls **smdxfer()** to perform the I/O.

```
/*
 * Start the next I/O operation - see devstart(2K).
 */
smdstart(dev)
dev_t dev;
{
	struct gdsw *gds = &gdsw[gdpos(dev)];
	struct iobuf *gdt = &gdtab[gdctl(dev)];
	struct iobuf *dp = (struct iobuf *)gdt->b_actf;
	struct buf *bp;

	/* No work queued. */
	if (dp == NULL) {
		blkacty &= ~(1<<gdctl(dev));
		return;
	}
	if (dp->b_flags & DP_ACTIVE)
		return;		/* The controller is already active. */
	if ((bp = dp->b_actf) == NULL)
		gdpanic("smdstart: I/O queue empty");
	/* Setup information for the transfer. */
	dp->b_flags |= DP_ACTIVE;
	XferInfo.rpts = 0;		/* No retries yet */
	XferInfo.xfrcnt = 0;			/* Nothing transferred */
	XferInfo.cyl = bp->cylin;
	XferInfo.dma_addr = bp->b_un.b_addr;
	XferInfo.trk = (ushort)bp->trksec / gds->sectrk;
	XferInfo.sec = (ushort)bp->trksec % gds->sectrk;
	XferInfo.tcnt = (bp->b_bcount + gds->dsk.sectorsz - 1) /
			gds->dsk.sectorsz;
	if (bp->b_flags & B_FORMAT) {
		XferInfo.mode = CMD_FORMAT;
		XferInfo.retries = 1;
	} else if (bp->b_flags & B_READ) {
		XferInfo.mode = CMD_READ;
		XferInfo.retries = GDRETRIES;
	} else {
		XferInfo.mode = CMD_WRITE;
		XferInfo.retries = GDRETRIES;
	}
	/* Setup/Start the transfer (if the controller is free) */
	smdxfer(XferInfo.cyl, XferInfo.trk, XferInfo.sec,
		XferInfo.tcnt, XferInfo.mode, dev);
}
```

smdxfer()

Smdxfer() breaks up the I/O request into I/O operations and attempts to start the next operation. A typical disk request must be carried out in several steps (called I/O operations in this document):

1. A SEEK operation to position the heads over the desired cylinder. This step is not required if the heads are already in place from a previous operation.
2. A READ or WRITE operation to transfer the requested data. This step will be broken up into several operations if the requested data should span a track boundary.

First, **smdxfer()** checks that the drive is still online. If not, it sets the appropriate error indications in the buffer header. It sets the **B_ERROR** bit to indicate a failure, it sets **b_resid** to the number of bytes originally requested minus the number of bytes already transferred, and it sets **b_error** to the generic I/O error code. Finally, **smdxfer()** calls **gdiodone()** to complete the request. Note that this is the only time the low-level disk driver calls **gdiodone()**. In the normal case, **gdintr(2K)** makes the call at the completion of the request.

If the drive is still online, **smdxfer()** checks the transfer against the track limit and adjusts the transfer sector count **tcnt** if necessary. As a precaution, it checks the new transfer count against zero and calls **gdpanic(2K)** in case of problems.

```
/*
 * Perform an I/O request, perhaps involving several I/O operations.
 */
smdxfer(cyl, trk, sec, tcnt, mode, dev);
ushort    cyl;
ushort    trk;
ushort    sec;
uint tcnt;
ushort    mode;
dev_t     dev;
{
     struct gdsw *gds = &gdsw[gdpos(dev)];
     struct iobuf *gdt = &gdtab[gdctl(dev)];
     struct iobuf *dp = &gdutab[gdpos(dev)];
     struct buf *bp = dp->b_actf;
     int ind;
     int ctl = gdctl(dev);

     /* In case the drive went off line */
     if (!(gds->v_flags & GD_OPENED)) {
          dp->b_flags &= ~DP_ACTIVE;   /* release the controller */
          /* Mark the I/O as an error */
          bp->b_flags |= B_ERROR;
          bp->b_resid = bp->b_bcount - XferInfo.xfrcnt;
          bp->b_error = EIO;
          bp->b_flags &= ~B_START;
          /* Mark the I/O as done, take it off the queue */
          gdiodone(bp, dp, gdt);
          return;
     }

     /* Does I/O cross a track boundry? */
     if ((sec+ tcnt) > gds->sectrk)
          tcnt = gds->sectrk - sec;
     if (tcnt == 0)
          gdpanic("smdxfer: zero transfer");
```

smdxfer() continued

Smdxfer() continues setting up for the current I/O operation. After setting up, if the driver detects that the controller is in use, it sets the **DP_WAITING** flag and returns. The low-level interrupt handler will detect the waiting request and issue it.

Next, for READ and WRITE commands, the driver issues a SEEK if required and returns. If no SEEK is required, the driver checks for physical I/O (I/O directly into or out of user space) and calls **setmap(2K)** to remap the user's buffer if required.

Finally, **smdxfer()** issues the READ, WRITE, or FORMAT command and returns.

```
/* smdxfer() continued */

/* Setup the current transfer */
XferInfo.rptcyl = cyl;
XferInfo.rpttrk = trk;
XferInfo.rptsec = sec;
XferInfo.rpttcnt = tcnt;
XferInfo.rptmode = mode;
if (gdt->b_flags & DT_INUSE) {  /* Controller is in use */
     dp->b_flags |= DP_WAITING;
     return;
}
gdt->b_flags |= DT_INUSE;
gdt->b_dev = dev;
gdt->b_actf = (struct buf *)dp;
if (mode != CMD_FORMAT) {
     if (XferInfo.curcyl != cyl) {
          dp->b_flags |= DP_SEEKING;
          hw_seek(cyl, trk);  /* Seek to the desired cyl */
          return;
     }
     if (bp->b_flags & B_PHYS)  /* Called from physio(2K)? */
          XferInfo.dsk_dma_addr = setmap(bp, dp->vaddr,
               (XferInfo.xfrcnt * gds->dsk.sectorsz),
               (tcnt * gds->dsk.sectorsz));
     else
          XferInfo.dsk_dma_addr = XferInfo.dma_addr;
}
dp->b_flags |= DP_ACTIVE;
gdtab[gdctl(dp->b_dev)].b_flags |= DT_DMAON;
switch (mode) {
case CMD_READ:
     hw_read(cyl, trk, sec);
     break;
case CMD_WRITE:
     hw_write(cyl, trk, sec);
     break;
case CMD_FORMAT:
     hw_format(cyl, trk);
     break;
default:
     gdpanic("smdxfer: invalid mode");
}
}
```

smdintr()

CTIX calls the **gdintr(2K)** routine whenever it receives an interrupt from the SMD controller. **Gdintr(2K)** verifies that the interrupt is expected (because the controller is active) and then calls **smdintr()** to handle the interrupt.

It is the responsibility of **smdintr()** to

1. Determine the reason for the interrupt. The SMD controller generates interrupts for two reasons:
   - SEEK, READ, WRITE, or FORMAT operation complete (with or without error).
   - The drive has gone offline.
2. Perform the required action based upon the interrupt reason and drive status. Possible actions are
   - Retry a failed SEEK, READ, or WRITE operation.
   - Issue a READ or WRITE operation after a SEEK completes.
   - Issue another in a series of READ or WRITE operations for an I/O request that crossed a track boundary.
3. Return a request complete or incomplete indication to **gdintr(2K)**.

If **smdintr()** indicates that the current I/O request is complete, **gdintr(2K)** calls **gdiodone()** on the buffer and then calls **smdstart()** to begin processing the next I/O request.

The processing of the current interrupt is governed completely by the **switch** statement: it handles the interrupt reasons described above. In the case of a drive offline interrupt, **smdintr()** calls **binval()** to invalidate all of the system cache blocks related to the current drive. Then it marks the error condition in the current buffer header and returns 0 to **gdintr(2K)**. This tells the general disk driver to call **gdiodone()** on the buffer.

```
/*
 * Process completion interrupts from the SMD controller.
 * Returns:
 *    0 - current operation is complete: call gdiodone();
 *    1 - current operation is continued or retried.
 */
smdintr(bp, dev, vec)
struct buf *bp;
dev_t dev;
int vec;
{
	register struct iobuf *gdt = &gdtab[gdctl(dev)];
	register struct gdsw *gds = &gdsw[gdpos(dev)];
	ushort status;
	int continuef;

	status = hw_status();
	hw_clrint();	/* Clear the interrupt. */

	gdt->b_flags &= ~(DT_INUSE |DT_DMAON); /* The controller is free */

	/* Process completion status. */
	switch (status) {
	case DRVOFFL:		/* The drive went offline */
		gds->v_flags &= ~GD_READY;
		binval(dp->b_dev);	/* Invalidate all cache blocks */

		/* Free the drive */
		dp->b_flags &= ~(DP_SEEKING|DP_ACTIVE);

		/* Mark an error in the buffer */
		bp->b_flags |= B_ERROR;
		bp->b_resid = bp->b_bcount - gdr->xfrcnt;
		bp->b_error = EIO;
		bp->b_flags &= ~B_START;
		return(0);
		/* NOTREACHED */
```

smdintr() continued

The next case handles fatal errors on I/O operations. First, the driver formats an appropriate error message. Then it decrements the retry count. If the count is zero, the I/O request has failed: **smdintr()** sets the error indication in the buffer header and returns 0. This tells **gdintr(2K)** to call **gdiodone()** on the buffer. If the retry count is not 0, **smdintr()** calls **smdxfer()** to repeat the I/O operation.

```
/* smdintr() continued */

case FTLERR:        /* A fatal error occurred */
    fmtberr();      /* Print an error message */
    gdr->retries--;
    gdr->rpts++
    if (gdr->retries == 0) {
        dp->b_flags &= ~DP_ACTIVE;
        /* Mark the error in the bp */
        bp->b_flags |= B_ERROR;
        bp->b_resid = bp->b_bcount - gdr->xfrcnt;
        bp->b_error = EIO;
        bp->b_flags &= ~B_START;
        return(0);      /* I/O is done */
    } else {
        smdxfer(XferInfo.cyl, XferInfo.trk, XferInfo.sec,
            XferInfo.tcnt, XferInfo.mode, dev);
        return(1);      /* Retrying the I/O */
    }
    /* NOTREACHED */
    break;
```

smdintr() continued

These two cases handle successful I/O operations. If there was a recoverable error (an error that the controller itself could rectify), the driver formats an error message, and then falls into the **NOERR** case.

If the last operation was a SEEK, the driver resets its internal state flags, updates the current cylinder information, and breaks from the **switch** statement.

If the current I/O operation was not a SEEK, the driver updates the transfer information to skip past the length of the transaction. If there is any more data to transfer, **smdintr()** calls **smdxfer()** to carry on with the next I/O operation.

```
/* smdintr() continued */

case RCOVBERR:        /* Recoverable error occurred */
     fmtberr();      /* Print an error message */
     /* Fall through */
case NOERR:        /* No error occurred */
     if (dp->b_flags & DP_SEEKING) {
          dp->b_flags &= ~DP_SEEKING;
          dp->b_flags |= DP_WAITING;
          XferInfo.curcyl = XferInfo.rptcyl;
          break;
     }

     /* Setup for continuation of transfer */
     XferInfo.xfrcnt + = XferInfo.rpttcnt;
     XferInfo.sec + = XferInfo.rpttcnt;
     XferInfo.tcnt -= XferInfo.rpttcnt;
     if (XferInfo.sec >= (gds->sectrk) {
          XferInfo.sec -= gds->sectrk;
          XferInfo.trk+ +;
          if (XferInfo.trk >= gds->dsk.heads) {
               XferInfo.trk -= gds->dsk.heads;
               XferInfo.cyl+ +;
          }
     }
     /* Increment by bytes, each block is 1,024 bytes (2**10)
     XferInfo.dma_addr + = (XferInfo.rpttcnt << 10);
     if (!XferInfo.tcnt) { /* This operation is done */
          dp->b_flags &= ~(DP_WAITING|DP_ACTIVE);
          /* Clear error flag */
          bp->b_resid = 0;
          continuef = 0;
     } else {
          /* Continue with the I/O */
          smdxfer(XferInfo.cyl, XferInfo.trk, XferInfo.sec,
               XferInfo.tcnt, XferInfo.mode, dev);
          continuef = 1;
     }
     break;

case default:
     gdpanic("smdintr: unknown status");
     break;
}
```

smdintr() continued

If control comes this far, the interrupt handler attempts to start another I/O operation. If the controller is free, and if the **DP_WAITING** flag is set, **smdintr()** calls **smdxfer()** to perform the next I/O operation.

Finally, the low-level interrupt handler returns the **continuef** flag, indicating whether the current I/O request (not operation) is complete.

```
    /* smdintr() continued */

    gds->v_flags |= GD_READY;

    /* Try to perform another I/O */
    if (!(gdt->b_flags & DT_INUSE)) {
        dp = (struct iobuf *)gdt->b_actf;
        if (dp->b_flags & DP_WAITING) {
            dp->b_flags &= ~DP_WAITING;
            if (XferInfo.curcyl != XferInfo.cyl)
                gdpanic("curcyl!=cyl");
            smdxfer(XferInfo.rptcyl, XferInfo.rpttrk,
                XferInfo.rptsec, XferInfo.rpttcnt,
                XferInfo.rptmode, dp->b_dev);
            break;
        }
    }

    /* Tell gdintr() whether the current transfer is still in progress */
    return(continuef);
}
```

smdtimer()

Smdtimer() reports the controller status to the general disk driver. The **gdtimer(2K)** routine calls **smdtimer()** periodically to determine whether or not the indicated drive is online.

```
/*
 * Report controller status back to gdtimer() - see devtimer(2K).
 * Returns:
 *    0 - the drive is NOT ready.
 *    1 - the drive is ready.
 *   -1 - the controller is busy.
 */
smdtimer(dev)
dev_t dev;
{
	struct gdsw *gds = &gdsw[gdpos(dev)];
	struct iobuf *gdt = &gdtab[gdctl(dev)];
	int status;

	if (!(gds->v_flags & GD_OPENED))
		gdprint(dev, "smdtimer: called on unopened drive");

	if (gdt->b_flags & DT_INUSE)
		return(-1);		/* Controller is busy */
	return((hw_status() == DRVOFFL) ? 0 : 1);
}
```

8 BLOCK DEVICE EXAMPLE

This chapter contains the annotated source listing of the device driver for the Interphase V/SMD 3200 Disk Controller. This is an actual driver that runs under the CTIX operating system on the MightyFrame.

The V/SMD 3200 is a very high-speed, DMA-driven disk controller that can support either one or two disk drives. It has an on-board 68000 that provides an intelligent interface to the device driver. In particular, it supports

- On-board sector caching with dynamic sector allocation and deallocation.
- Zero latency reads and writes.
- Overlapped and implied seeks.
- Multiple, programmable vector numbers to speed interrupt processing.
- Software selectable disk sector sizes.

See the *Interphase V/SMD 3200 User's Guide* for a complete description of the hardware and its functionality.

Throughout this chapter, source code appears on the right hand page, while the annotations to it are on the left.

> **NOTE**
>
> Do not be confused by the fact that the routines in this chapter all begin with the letters **gd**. Throughout this document, the generic names for functions that are part of the general disk driver also begin with **gd**: for example, **gdopen(2K)**, **gdstrategy(2K)**, and so on. These generic functions make up the high-level, device-independent layer of every disk driver. All of the routines in this chapter, however, are part of the low-level portion of a driver for one specific general disk-type device. The prefix **gdvs32** identifies the functions as part of a driver for a general disk-type device named the V/SMD 3200.

This page intentionally left blank.

GDVS32.H

The first group of defines are hardware-dependent parameters for the Interphase controller. **VS32_MAXDMA** supports nine pages times 4,096 bytes per page or 36K bytes maximum transfer. The extra page is to allow the transfer to start in the middle of a page.

The **DP_XXXX** flags indicate the state of each I/O queue. **Gdutab** is an array of **iobufs**, each serving as the head of the queue of active I/O requests for one drive.

Gdvs32uib is the V/SMD 3200's Unit Information block. The controller supports software configurable drive parameters: they are communicated to it through the UIB. Before the controller can process any other command, it must receive an INITIALIZE command for each drive on it, with the UIB as a parameter.

The attribute flags are the values for **gdvs32uib.attrib**.

AT_RSK tells the controller to return to track zero and then reseek the target track before reporting a read or write failure.

AT_STC tells the controller to generate an interrupt whenever the drive status changes: for example, when a SEEK operation is complete.

AT_SSE tells the controller to format each track to contain a spare sector. This way, whenever a bad sector is encountered on a track, there is a good chance that the data can be redirected to the spare sector on the same track, thus incurring no extra seek latency for bad block forwarding.

```
/* gdvs32.h - Hardware definitions for the Interphase V/SMD 3200 */
#define VS32_GAPSZ        16   /* Default size of gap1 and gap2 */
#define VS32_MAXCYL            2048 /* Support a max of 2048 cylinders */
#define VS32_MAXHEAD      16   /* Support a maximum of 16 heads */
#define VS32_MAXTRACK (VS32_MAXCYL*VS32_MAXHEAD)
#define VS32_MAXDMA       (9)  /* Number of pgs for DMA (per drive) */
#define VS32_INTLVL  3    /* SPLDISK is 3 */
/*
 * Values in b_flags in gdutab[]. The high 8 bits are reserved
 * for use by gdisk.h.
 */
#define DP_ACTIVE    0x0001   /* Driver operation in progress */
#define DP_SEEKING   0x0002   /* Seek operation in progress */
#define DP_WAITING   0x0008   /* Waiting for controller */
#define DP_DELAYRD   0x0010   /* Reading a delayed bad block */

struct gdvs32uib {
	unchar	v0sh;		/* Volume zero start head number */
	unchar	v0nh;		/* Volume zero number of heads */
	unchar	v1sh;		/* Volume one start head number */
	unchar	v1nh;		/* Volume one number of heads */
	unchar	psectrk;	/* Number of sectors per track */
	unchar	skew;		/* Spiral skewing factor */
	ushort	sectorsz;	/* Bytes per sector */
	unchar	gap1;		/* Number of words in gap1 */
	unchar	gap2;		/* Number of words in gap2 */
	unchar	interleave;	/* Interleave factor */
	unchar	retry;		/* Number of retries on data error */
	ushort	cyls;		/* Number of cylinders */
	unchar	attrib;		/* Attribute flags */
	unchar	unused;		/* Reserved */
	unchar	scil;		/* Status change interrupt level */
	unchar	sciv;		/* Status change interrupt vector */
};

/* Attributes */
#define	AT_RSK	0x01	/* Enable restore and reseek on error */
#define	AT_MBD	0x02	/* Enable the transfer of possibly bad data */
#define	AT_INH	0x04	/* Increment by Head */
#define	AT_DLP	0x08	/* Dual Port */
#define	AT_STC	0x10	/* Status Change */
#define	AT_CE	0x20	/* Enable Sector Caching */
#define	AT_SSE	0x40	/* Allow a spare sector on each track */
```

The **gdvs32iopb** structure describes the I/O parameter block used by the V/SMD 3200 Controller. Every I/O request processed by the controller uses one or more of these fields. The fields generally are self-explanatory. The following list explains a few of the more complex fields.

bufferp contains the address of the buffer where the transfer will take place.

buf_memtype contains a descriptor indicating whether the controller should perform 8-bit, 16-bit, or 32-bit data transfers.

buf_addmod contains the VMEbus address modifier bits that the controller should assert during its DMA transfers.

iopbp When the **CF_LINK_ENABLE** bit is set in **gdvs32iopb.flags**, the V/SMD 3200 processes multiple, linked I/O requests. In this case, **iopbp** points to the next IOPB in the chain. The last IOPB in the chain must have the **CF_LINK_ENABLE** bit cleared.

iopb_memtype contains a descriptor indicating whether the controller should perform 8-bit, 16-bit, or 32-bit data transfers when reading the linked IOPB.

iopb_addmod contains the VMEbus address modifier bits that the controller should assert when reading the linked IOPB.

See the *Interphase V/SMD 3200 User's Guide* for a complete specification of the IOPB.

/* gdvs32.h continued */

```
struct hilo {
     ushort   hi;
     ushort   lo;
};
typedef struct hilo hilo;

struct gdvs32iopb {
     unchar   command;        /* Drive command code */
     unchar   flags;          /* Command flags */
     unchar   status;         /* Status code */
     unchar   error;          /* Error code */
     ushort   cyl;            /* Cylinder number */
     unchar   head;           /* Head number */
     unchar   sec;            /* Sector number */
     ushort   sectors;        /* Sector count */
     hilo     bufferp;        /* Address of the I/O buffer */
     unchar   buf_memtype;    /* Memory type for the I/O buffer */
     unchar   buf_addmod;     /* Address modifier for the I/O buffer */
     unchar   int_level;      /* Interrupt level */
     unchar   int_complete;   /* Normal completion interrupt vector */
     unchar   dmaburst;            /* DMA burst count */
     unchar   int_error;      /* Error interrupt vector */
     hilo     iopbp;          /* Address of the next IOPB */
     unchar   iopb_memtype;        /* Memory type for the next IOPB */
     unchar   iopb_addmod;    /* Address modifier for the next IOPB */
     unchar   sp_skew;        /* Spiral skew */
     unchar   unused;         /* Reserved */
};
```

The **MEM_XXX** defines are used in the two **gdvs32iopb.xxx_memtype** fields. The **AM_NN_K** defines are used in the two **gdvs32iopb.xxx_addmod** fields.

The Interphase controller contains 12K bytes of on-board RAM for sector buffering. It also has a 512-byte area for communication with the driver. **Gdvs32ctl** defines this communications area. The fields are defined as follows:

unit_status contains the status bits for each drive on the controller. Note that they are stored in reverse order: that is, **unit_status**[**0**] contains the status of Drive 1, and vice-versa. This accounts for the code **unit_status**[**drv^1**].

command holds the controller command and status register. This register contains the GO bit, which is used to start an operation, and the BUSY bit, which indicates the end of an operation.

iopb holds the IOPB as defined by the **gdvs32iopb** data structure. This is the only IOPB required by the controller; the **Xiopbs** are used for the convenience of the driver only.

Xiopb is an array of IOPBs, seven for each drive on the controller. The driver sets up one **DC_FETCH_AND_EXECUTE_IOPB** command in **gdvs32ctl.iopb**, it sets the **CF_LINK_ENABLE** bit in **gdvs32iopb.flags**, and then it builds specific I/O requests for each drive in the **Xiopb** array.

Xuib holds the Unit Information blocks for each drive on the controller.

The **US_XXXX** constants define the bits in **gdvs32ctl.unit_status**. The **CS_XXXX** constants define the bits in **gdvs32ctl.command** when the register is read. The **CMD_XXXX** constants define the bits in **gdvs32ctl.command** when the register is written.

```
                    /* gdvs32.h continued */

/* VME address modifiers and memory types */
#define  MEM_32      3          /* Perform 32-bit DMA operations */
#define  MEM_16      2          /* Perform 16-bit DMA operations */
#define  MEM_INTERNAL 1         /* Data is in internal controller memory */
#define  AM_16_K          0x2D          /* 16 bit address space */
#define  AM_24_K          0x3D          /* 24 bit address space */
#define  AM_32_K          0x0D          /* 32 bit address space */
#define  NXIOPB  7              /* Support 7 Auxiliary IOPBs */
/* The VSMD3200 short address space layout */
struct gdvs32ctl {
     unchar    unit_status[2];      /* Unit 1/0 status register */
     ushort    command;             /* Command/status register */
     struct    gdvs32iopb iopb;     /* IOPB buffer area */
     struct    gdvs32iopb Xiopb[2][NXIOPB]; /* Unit 0/1 aux IOPB areas */
     struct    gdvs32uib  Xuib[2];       /* Unit 0/1 aux UIB areas */
     char      unused[52];
};
/* Unit Status Bits */
#define  US_DREADY 0x01       /* Drive is ready */
#define  US_WRPROT 0x02       /* Drive is write protected */
#define  US_BUSY   0x04       /* Drive is busy */
#define  US_FAULT  0x08       /* Drive fault */
#define  US_ONCYL  0x10       /* Drive is on cylinder */
#define  US_BADSEEK     0x20       /* Seek error */
#define  US_PRESENT 0x40      /* Unit is present */
#define  US_UREADY 0x80       /* Unit is ready */
/* Controller Status Bits (read command register) */
#define  CS_ELC      0x10       /* Error Last Command */
#define  CS_OD       0x40       /* Operation was concluded */
#define  CS_BUSY     0x80       /* Begin Operation */
#define  CS_BERR     0x0100     /* Bus Error Has Occured */
#define  CS_BOK      0x4000     /* Board passed self test */
#define  CS_SLED          0x8000     /* Status LED */
/* Status Change Bits */
#define  CS_SCS      0x0200     /* Status Change Source */
#define  CS_SC       0x0400     /* Status was changed */
/* Command Bits (write command register) */
#define  CMD_GO           0x0080     /* Start Operation */
#define  CMD_BDCLR 0x1000     /* Reset the board */
#define  CMD_SFEN  0x2000     /* Enable the assertion of SYSFAIL */
#define  CMD_SLED  0x8000     /* Set/Clear Status LED */
```

The **DC_XXXX** constants define the controller commands. They are written to the **gdvs32iopb.command** field.

The **CF_XXXX** constants define the flag values that can be written to the **gdvs32iopb.flags** field.

The **SR_XXXX** constants define the bit values that are reported in the **gdvs32iopb.status** field.

Gdvs32driver is a pointer to a region of memory that is allocated in **gdvs32open()**. It contains information describing all in-progress I/O on a drive-by-drive basis.

/* gdvs32.h continued */

```
/* Drive Commands (in the IOPB) */
#define DC_CLEARFAULT                   0x97
#define DC_DIAGNOSTIC                   0x70
#define DC_FORMAT_TRACK                 0x84
#define DC_HANDSHAKE                    0x86
#define DC_INITIALIZE                   0x87
#define DC_MAP_SECTOR                   0x90
#define DC_MAP_TRACK                    0x85
#define DC_READ_HEADER                  0x74
#define DC_READ_LONG                    0x71
#define DC_READ_SECTORS                 0x81
#define DC_RESET                        0x8F
#define DC_RESTORE                      0x89
#define DC_RE_FORMAT                    0x8B
#define DC_SEEK                         0x8A
#define DC_TRACK_ID                     0x9A
#define DC_VERIFY_TRACK                 0x99
#define DC_WRITE_LONG                   0x72
#define DC_READ_NON_CACHED              0x94
#define DC_READ_SECTOR_BUFFER           0x79
#define DC_REPORT_CONFIGURATION         0x77
#define DC_VERIFY_SECTORS               0x83
#define DC_WRITE_SECTORS                0x82
#define DC_WRITE_SECTOR_BUFFER          0x78
#define DC_FETCH_AND_EXECUTE_IOPB       0x9B
#define DC_FORMAT_TRACK_WITH_DATA       0x8C
/* Command Flags (in the IOPB) */
#define CF_ECC_ENABLE                   0x01
#define CF_INT_ENABLE                   0x02
#define CF_ERROR_DETECTION_DISABLE      0x04
#define CF_RESERVE_ENABLE               0x08
#define CF_LOGICAL_TRANSLATION          0x10
#define CF_LINK_ENABLE                  0x20
#define CF_VOLUME_NUMBER                0x40
#define CF_UNIT_NUMBER                  0x80
/* Status field (in the IOPB) */
#define SR_OK        0x80
#define SR_CIP       0x81
#define SR_ERR       0x82
#define SR_EXC       0x83
/* Information about the transfer in progress. */
extern struct gddriver  *gdvs32driver;
```

GDVS32.C - PREAMBLE

sys/sysmacros.h	See the **macros(2K)** in Appendix A, *CTIX Interface Manual Pages.*
sys/page.h	contains fundamental memory management constants.
sys/buf.h	contains the declaration for the buffer header structure **buf**.
sys/elog.h	contains declarations to support error logging.
sys/iobuf.h	contains the declaration of the **iobuf** data structure.
sys/user.h	contains the declaration for the **user** structure. This holds all of the per-process information that is not needed by CTIX while the process is swapped out.
sys/errno.h	contains the system error constants defined in the Introduction to Section 2 of the *CTIX Operating System Manual.*
sys/systm.h	contains **extern** declarations for the most important variables, structures, and functions in the CTIX operating system.
sys/gdisk.h	contains declarations and definitions used throughout the general disk (gd) system.
sys/gdvs32.h	contains hardware and programming definitions for the Interphase V/SMD 3200 Controller.
sys/iohw.h	contains defines for I/O system hardware.
sys/hardware.h	contains defines for various hardware registers.
sys/vme.h	contains VMEbus EEPROM structure declaration.

```
/***********************************************
 * gdvs32.c
 *
 * CTIX 5.0 driver for Interphase V/SMD 3200.
 ***********************************************/
static char gdvs32_c[] = "(#)gdvs32.c 5.33";

#include "sys/param.h"
#include "sys/types.h"
#include "sys/sysmacros.h"
#include "sys/page.h"
#include "sys/buf.h"
#include "sys/elog.h"
#include "sys/iobuf.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/systm.h"
#include "sys/gdisk.h"
#include "sys/gdvs32.h"
#include "sys/iohw.h"
#include "sys/hardware.h"
#include "sys/vme.h"
#include "sys/i8259.h"
#include "sys/spl.h"
#include "sys/debug.h"
```

The **DFLT_ID/Drv_id** mechanism is handled completely by the linker. Simply include these two lines in every driver, and the driver ID will be assigned properly, whether it is loadable or is configured in with the kernel. For more information, see Chapter 9, *Integrating the Driver.*

Vctladdr contains the VMEbus address of the controller.

VSindex takes a minor device number and returns an index into the array of **gddriver** structures reserved by the driver. This array is named **gdvs32driver**. There is one **gddriver** structure for each physical device that this driver controls. Each structure describes the current DMA operation in progress on one device.

All **extern** declarations appear next. The driver routines also are declared for the purpose of documentation.

CTIX software sets the **haveVME** flag to nonzero if the VMEbus expansion board is installed in the system.

```
                        /* gdvs32.c */

extern int DFLT_ID;
static int Drv_id = (int) &DFLT_ID;
struct gdvs32ctl *Vctladdr;
/* The index into the driver table (2 drives per controller) */
#define VSindex(X)      (((gd_config[gdctl(X)].cs)<<1) + gddrive(X))
/* Virtual to compressed virtual addresses for 24 bit VME address space */
#define vtocv24(X)      ((caddr_t)((int)X & 0x007FFFFF))

/* Flags for the IOPB */
#define iopbflags(drv)  (CF_ECC_ENABLE|CF_INT_ENABLE |
                        (drv ? CF_UNIT_NUMBER : 0))
#define iopblink(drv)   (CF_LINK_ENABLE|CF_INT_ENABLE |
                        (drv ? CF_UNIT_NUMBER : 0))

#define VOID int  /* To document functions returning no value */
VOID    gdvs32doxfr();          /* Initiate the transfer */
int     gdvs32errors();         /* Diagnose error conditions */
int     gdvs32intr();           /* Device interrupt handler */
int     gdvs32open();           /* Open the device */
VOID    gdvs32seek();           /* Perform a SEEK or RESTORE */
VOID    gdvs32start();          /* Start I/O on the controller */
VOID    gdvs32statuschange();   /* Process Status Change interrupt */
int     gdvs32timer();          /* Return status to gdtimer() */

extern  VOID    binval();
extern  VOID    delay();
extern  VOID    fmtberr();
extern  int     gdaddbadblk();
extern  int     gdaltblk();
extern  int     gdintr();
extern  VOID    gdiodone();
extern  VOID    gdpanic();
extern  VOID    gdprint();
extern  struct  vmeeprom *is_eepromvalid();
extern  VOID    logstray();
extern  VOID    printf();
extern  int     probevme();
extern  int     set_vec();
extern  caddr_t setmap();
extern  caddr_t sptalloc();
extern  caddr_t sptballoc();
extern  int     strcmp();
extern  int     haveVME;
```

GDVS32OPEN()

Gdvs32open() is called as a result of a **mount(2)** or an **open(2)** system call. In either case, the code to handle the request includes a line of the form:

(*bdevsw[bmajor(dev)].d_open)(minor(dev), flag);

which calls **gdopen(2K)** with a **flag** parameter indicating whether the device is to be mounted or opened with WRITE permission. **Gdopen()** verifies its parameters and then calls **gdvs32open()** with a statement of the form:

```
if ((*gdsw[pos].open)(minor_dev) == 0)
    return; /* open failed - it set u.u_error */
```

Firsttime tells **gdvs32open()** to perform one-time initialization for the V/SMD 3200 Controller.

Dp is initialized to point to the **gdutab** structure for the requested device. **Gdutab** is an **iobuf** structure that contains the head of the list of all active **buf** structures for this device.

Smdp is initialized to the base address of the I/O registers and IOPB of the controller. The structure of this region is defined by **gdvs32ctl**.

See **qprintf(2K)** for a complete description of the debugging macros. The header file **<sys/debug.h>** documents the default uses for the various debugging levels.

Gdvs32open() first verifies that the VMEbus Expansion board is installed. Next, it checks **vp** to make certain that the call to **is_eepromvalid(2K)** did not indicate that the VMEbus EEPROM had an error. Then, it verifies the drive number.

```
/*
 * gdvs32open() - open V/SMD 3200-type device.
 *
 * Returns:
 *     0 - Open failed.
 *     1 - Open succeeded.
 */
 int
gdvs32open(dev)
register dev_t dev;
{
      register short ctl = gdctl(dev);
      register struct gdsw *gds = &gdsw[gdpos(dev)];
      static int firsttime = 1;
      short driveno;
      int vector;
      struct vmeeprom *vp = is_eepromvalid();

      register struct iobuf *dp = &gdutab[gdpos(dev)];
      register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
      register struct gdvs32iopb *iopbp;
      register struct gddriver *gdr;

      qprintf("gdvs32open(dev: 0x%x) dp is 0x%x", dev, dp);
      if (!haveVME) {
            gdprint(dev, "VME interface board not present");
            u.u_error = ENXIO;
            return(0);
      }
      if (!vp) {
            gdprint(dev, "EEPROM is not valid");
            u.u_error = ENXIO;
            return(0);
      }
      if (gddrive(dev) > 1) {
            /* Only drives 0, 1 allowed */
            u.u_error = ENXIO;
            return(0);
      }
```

The code that is conditioned by the **if (firsttime)** test is performed only once. The low-level drivers for general disk-type devices do not have a **devinit(2K)** function, because they cannot be loaded by **lddrv(1M)**. Thus, they must use something like the **firsttime** flag to control one-time initialization.

First, **gdvs32open()** searches through the **gd_config** array for the last occurrence of this controller type. The **cs** field of the last entry contains the number of instances of this controller in the system.

The driver calls **sptballoc(2K)** to allocate enough memory to hold the **gddriver** data structures: one for each drive on every V/SMD 3200 Controller. If it can't get the memory the **gdvs32open()** fails.

If the **sptballoc(2K)** call succeeds, **gdvs32open()** clears the **firsttime** flag and proceeds.

```
                /* gdvs32open() continued */

if (firsttime) {
    short i;

    /* Allocate space for the driver tables - get enough
     * space for all instances of this controller.
     *
     * Search the gd_config structure for the LAST occurrence
     * of this driver type - it's "cs" field will be the
     * number of occurrences of this controller.
     */
    for (i=gd_cnt-1; i>= 0; i--)
        if (strcmp(gd_config[ctl].dev, gd_config[i].dev) == 0)
            break;
    if (i < 0)
        gdpanic("No gd_config entry for gdvs32 controller");

    qprintf("found %d occurrences", gd_config[i].cs+ 1);
    /* Need two driver structures per controller */
    gdvs32driver = (struct gddriver *)sptballoc(
        (int)(sizeof(struct gddriver) * 2 *
        (gd_config[i].cs+ 1)));
    if (gdvs32driver == NULL) {
        gdprint(dev, "Could not get space for gdvs32driver");
        u.u_error = ENXIO;
        return(0);
    }
    firsttime = 0;
}
```

The next section of code is performed only if the driver has not yet determined the address of the controller board (**dp->io_addr**).

The **get_vec(2K)** call allocates and assigns an interrupt vector to this device driver. Note that the address of the **gdintr(2K)** routine is passed as the interrupt handler, not **gdvs32intr()**. **Gdintr()** calls this driver's interrupt handler only after it has validated the interrupt vector number and has checked that there is activity expected on the interrupting device. The call in **gdintr()** is with a statement of the form:

```
if ((*gdsw[pos].intr)(bp, bp->b_dev, vec))
    return;        /* I/O retried or continued */
```

The next lines in **gdvs32open()** search the **gdint** array and insert the new vector number. **Gdintr(2K)** uses this table to verify the interrupts as they come in.

Next, **gdvs32open()** searches through the VMEbus EEPROM for an entry describing this controller. This entry contains the VMEbus address for the controller. If the entry is not found, **gdvs32open()** returns with an error.

After getting the address from the EEPROM, **gdvs32open()** probes the bus to make certain that the controller is present. If the board is in the system, **gdvs32open()** clears the controller memory.

```
                    /* gdvs32open() continued */

if (!dp->io_addr) {
     register short i, ctlr, *wp;

     qprintf("initializing controller %x", ctl);
     /* plug the interrupt vectors */
     vector = get_vec(Drv_id, gdintr);
     for (i=0; gdint[i].vec != 0; i++)
          ;
     gdint[i].vec = vector; gdint[i++].ctl = ctl;

     /* Search the EEprom for the correct occurance of this controller */
     for (ctlr=0, i=0; i<VME_SLOTS; i++) {
          if (vp->slots[i].type == VMET_V3200) {
               if (gd_config[ctl].cs == ctlr) {
                    smdp = (struct gdvs32ctl *)vp->slots[i].address;
                    qprintf("controller is slot %d at address %x", i, smdp);
                    break;
               }
               ctlr++;
          }
     }
     if (!smdp) {
          gdprint(dev,
     "address for VSMD3200 controller not found in EEprom");
          u.u_error = ENXIO;
          return(0);
     }

     /* Check for the presence of the controller board */
     if (probevme(smdp)) {
          gdprint(dev, "VSMD3200 controller board not present");
          u.u_error = ENXIO;
          return(0);
     }

     /* Clear memory in the controller */
     for (wp=(short *)&smdp->iopb; wp<(short *)&smdp->unused[0];
               wp++)
          *wp = 0;
```

The code at the top of the page resets the controller. First it writes the **CMD_BDCLR** command, then delays 117 milliseconds (7/60 seconds), and finally clears the **command** register. Then the driver waits for up to 2 seconds (in 117 ms intervals) for the **BUSY** bit to be cleared on the Controller Status register. After this sequence, if the Board OK status bit (**CS_BOK**) is not set, the controller has failed to initialize correctly. Return an error in this case.

Next, the green LED on the controller is lit. It stays on as long as the Board OK status bit is set.

Finally, default values are written to the I/O parameter block on the controller, and the base IOPB is set up with a **DC_-FETCH_AND_EXECUTE_IOPB** command. Note that the interrupt level is set to **VS32_INTLVL**, which is 3 in CTIX.

```
            /* gdvs32open() continued */

/* reset the board */
smdp->command = 0;
smdp->command = CMD_BDCLR;
delay(7);
smdp->command = 0;
/* Wait up to 2 seconds */
for (i=0; i<20; i++) {
      /* Wait 112 ms */
      delay(7);
      if (!(smdp->command & CS_BUSY))
            break;
}

if (!(smdp->command & CS_BOK)) {
      gdprint(dev, "Controller board not ready");
      u.u_error = EIO;
      return(0);
}

/* Turn on the green light */
smdp->command |= CMD_SLED;
smdp->iopb.iopb_memtype = MEM_INTERNAL;
smdp->iopb.iopb_addmod = AM_16_K;
smdp->iopb.int_level = VS32_INTLVL;
smdp->iopb.int_complete = vector;
smdp->iopb.int_error = vector;
smdp->iopb.command = DC_FETCH_AND_EXECUTE_IOPB;
```

The Unit Information blocks for the two drives on the controller are initialized. All of the Auxiliary I/O Parameter blocks are initialized and chained together. As described above, **gdvs32ctl.iopb** always contains a **DC_FETCH_AND_-EXECUTE_IOPB** command. Its **iopbp** field is initialized to point to the first **Xiopb** structure for the relevant drive whenever a command is issued.

The **gdvs32driver.curcyl** field is initialized to -1; this forces the driver to issue a **DC_RESTORE** command before the first I/O request is carried out.

```
                /* gdvs32open() continued */

    /* For now, only drive 0,1 on the first controller are initialized */
    for (driveno = 0; driveno <= 1; driveno++) {
        int j;

        gdr = &(gdvs32driver[VSindex(gdmkdev(gdctl(dev),
            driveno, 0))]);
        smdp->Xuib[driveno].v0sh = 0;
        smdp->Xuib[driveno].v1sh = 0;
        smdp->Xuib[driveno].v1nh = 0;
        smdp->Xuib[driveno].skew = 0;
        smdp->Xuib[driveno].gap1 = gds->dsk2.gap1;
        smdp->Xuib[driveno].gap2 = gds->dsk2.gap2;
        smdp->Xuib[driveno].interleave = 1;
        smdp->Xuib[driveno].retry = 0;
        smdp->Xuib[driveno].attrib = AT_STC |AT_CE |AT_INH;
        smdp->Xuib[driveno].scil = VS32_INTLVL;
        smdp->Xuib[driveno].sciv = vector;

        /* Chain the iopbs in the short address space */
        for (j=0; j<6; j++) {
            iopbp = &smdp->Xiopb[driveno][j];

            iopbp->iopbp.lo = (ushort)(iopbp+1);
            iopbp->iopbp.hi = 0;
            iopbp->iopb_memtype = MEM_INTERNAL;
            iopbp->iopb_addmod = AM_16_K;

            iopbp->buf_memtype = MEM_32;
            iopbp->buf_addmod = AM_32_K;

            iopbp->int_level = VS32_INTLVL;
            iopbp->int_complete = vector;
            iopbp->int_error = vector;

            iopbp->sp_skew = 0;
            iopbp->dmaburst = 8;
        }
        gdr->curcyl = (ushort)-1;
        gdutab[gdmkpos(ctl,driveno)].vaddr = NULL;
        gdutab[gdmkpos(ctl,driveno)].io_addr = (physadr)smdp;
        gdutab[gdmkpos(ctl,driveno)].io_nreg = 16;
    }
} /* if (!dp->io_addr) */
```

This driver can be called from **physio(2K)** to do raw I/O directly into or out of user memory. In this case, the address of the buffer is a user virtual address, which is only valid when the requesting process is running. Since **physio(2K)** sleeps after calling the device strategy routine, the original user process is never running at the time the DMA transfer occurs.

Clearly, then, the driver cannot program the DMA hardware with the original virtual address of the buffer. In fact, the driver cannot reference user virtual memory at all, since it changes whenever there is a context switch. The DMA hardware must use kernel virtual memory, which always is valid. This kernel virtual memory is reserved here by a call to **sptalloc(2K)**. **Sptalloc()** allocates a contiguous region of kernel virtual address space to serve as a "window" on the user's I/O buffer in physical memory. The pointer to this "window" is kept in **dp->vaddr** (that is, in **gdutab.io_s2**: **vaddr** is a **#define** in the file <**sys/gdisk.h**>).

Next, the driver allocates space for the bad block table and the bad block queue for this drive, if they have not been allocated already.

Finally, the driver sets the **DP_READVHB** flag, which forces **gdvs32doxfr()** to issue a **DC_INITIALIZE** command to the controller before performing any other I/O. This command is used to set the software-programmable parameters on the drive: it must be issued before any READ, WRITE, or FORMAT commands are accepted.

```
                    /* gdvs32open() continued */

    if (dp->vaddr == NULL) {
        /* Allocate the DMA area for the drive */
        dp->vaddr = (int)sptalloc(VS32_MAXDMA, PG_V|PG_KW, -1);
        if (dp->vaddr == NULL) {
            gdprint(dev, "Unable to allocate virtual addresses for raw dma");
            u.u_error = EIO;
            return(0);
        }
    }

    /* Allocate space for the bad block tables */
    if (gds->bb == NULL && gds->szbb != 0)
        if ((gds->bb = (struct bbmcell *)sptballoc(GDMAXBBT))
                == NULL) {
            gdprint(dev, "Unable to allocate space for the bad block table");
            u.u_error = EIO;
            return(0);
        }

    if (gds->bbq == NULL && gds->szbbq != 0)
        if ((gds->bbq = (short *)sptballoc((int)(sizeof(short) *
                VS32_MAXCYL))) == NULL) {
            gdprint(dev,
            "Unable to allocate space for the bad block table index");
            u.u_error = EIO;
            return(0);
        }

    /* Mark the vhb as read - this will force re-initialization of the drive */
    dp->b_flags |= DP_READVHB;

    /* All OK */
    return(1);
}
```

GDVS32START()

Gdvs32start() is called from three places to start I/O on the device: **gdstrategy(2K)** calls it to service **bread()** and **bwrite()** requests from the kernel; it calls itself recursively to start as many of the outstanding requests on the drive I/O queue as possible; **gdvs32doxfr()** calls it (indirectly) recursively to flush the drive queue when the drive has gone offline.

There is one **gdtab** entry for each disk controller in the system. Each entry points to a queue of queues of active I/O requests on that controller; that is, each first-level queue entry points to another queue containing all of the active I/O for one of the drives on the controller. Active in this sense means that the I/O has not yet been completed. Some of the requests may be in process, while others are not. The second-level queue of requests is called the drive I/O queue in this document. See Chapter 7, *Block I/O Tutorial*, for more information.

Dp is initialized to point to the head of the queue of queues. This is the per-drive queue; each member is an **iobuf** structure that serves as the head of the drive I/O queue. If the top-level queue is empty, there is no work on any of the drives. The driver resets the block activity flag for this controller and returns.

Next, **gdvs32start()** scans the (top-level) drive queue, looking for drives that are active but do not have I/O started on them. (**Gdstrategy(2K)** sets **b_active** when it enqueues an I/O request buffer. **Gdstart()** sets **DP_ACTIVE** when it sets up the transfer.) The drive queue actually is a circular linked list: the **do while** loop is finished when the forward pointer on the current member equals the forward pointer on the list head. If there are no drives with outstanding I/O, the driver returns.

Next, the **iobuf** pointer **bp** is set to point to the first member of the drive I/O queue. The **gdpanic(2K)** call protects against an empty I/O queue that has the **b_active** flag set.

```
/*
 * Gdvs32start() - start I/O on a device.
 */
 VOID
gdvs32start(dev)
dev_t dev;
{
	register struct iobuf *gdt = &gdtab[gdctl(dev)];
	register struct iobuf *dp = (struct iobuf *)gdt->b_actf;
	register struct gddriver *gdr;
	register struct gdsw *gds;
	register struct buf *bp;

	aaprintf("gdvs32start(dev: 0x%x)", dev);
	if (dp == NULL) {
		/* No work queued.
		 * Major block numbers for gd controllers are from 0..15
		 * I.e. equal to the controller number.
		 */
		blkacty &= ~(1<<gdctl(dev));
		return;
	}

	do {
		if (dp->b_active && !(dp->b_flags & DP_ACTIVE))
			/* I/O is queued, but not started */
			break;
		dp = (struct iobuf *)dp->b_forw;
	} while (dp != (struct iobuf *)gdt->b_actf);

	if (!dp->b_active |(dp->b_flags & DP_ACTIVE))
		/* Must have gone full circle on the circular list */
		return;

	if ((bp = dp->b_actf) == NULL)
		gdpanic("Null I/O queued");
```

Next the driver saves the parameters of the selected I/O request in the **gddriver** structure reserved for this drive. If the **GD_PHYSADDR** flag is set, the driver uses physical track and sector information. Otherwise, it uses the logical values.

Tcnt contains the total number of sectors involved in the transfer. The byte count is rounded up and then converted to a sector count.

Notice that only three commands are possible from the general disk driver: **CMD_FORMAT**, **CMD_READ**, and **CMD_WRITE**. This illustrates how the general disk driver code simplifies the interface for the rest of the kernel. The dozen or more commands that the Interphase controller accepts are hidden from CTIX software.

Next, **gdvs32start()** calls **gdvs32doxfr()** to begin the transfer (if the controller is not busy servicing the other drive). Note that **gdvs32doxfr()** in turn calls **gdvs32start()** recursively if it detects that the drive has gone offline. This has the effect of flushing the current drive queue, since **gdvs32doxfr()** calls **gdiodone(2K)** on the buffer and clears the **DP_ACTIVE** flag before the recursive call.

Finally, this routine calls itself recursively. Since the **DP_ACTIVE** flag is set in this drive queue, **gdvs32start()** selects the next active drive queue to process on each of the succeeding recursive calls. This has the effect of starting I/O on every drive on this controller, if there is anything queued. The recursion ends when **gdvs32start()** can't start any more I/O.

```
                    /* gdvs32start() continued */

    gdr = &gdvs32driver[VSindex(bp->b_dev)];
    gds = &gdsw[gdpos(bp->b_dev)];

    /* Setup information for the first transfer */
    dp->b_flags |= DP_ACTIVE;
    gdr->rpts = 0;
    gdr->xfrcnt = 0;
    gdr->cyl = bp->cylin;
    gdr->dma_addr = bp->b_un.b_addr;

    if (gds->v_flags & GD_PHYSADDR) {
        gdr->trk = (ushort)bp->trksec / gds->dsk.psectrk;
        gdr->sec = (ushort)bp->trksec % gds->dsk.psectrk;
    } else {
        gdr->trk = (ushort)bp->trksec / gds->sectrk;
        gdr->sec = (ushort)bp->trksec % gds->sectrk;
    }
    gdr->tcnt = (bp->b_bcount+ gds->dsk.sectorsz - 1)/gds->dsk.sectorsz;

    if (bp->b_flags & B_FORMAT)
        gdr->mode = CMD_FORMAT;
    else if (bp->b_flags & B_READ)
        gdr->mode = CMD_READ;
    else
        gdr->mode = CMD_WRITE;

    if (bp->b_flags & B_FORMAT) {
        gdr->retries = 1;
    } else {
        gdr->retries = GDRETRIES;
    }

    /* Setup/Start the transfer (if the controller is free) */
    gdvs32doxfr(gdr->cyl, gdr->trk, gdr->sec, gdr->tcnt, gdr->mode, dev);

    /* See if we can start another one */
    gdvs32start(dev);
}
```

GDVS32DOXFR()

Gdvs32doxfr() is called from various places in the driver to initiate an I/O operation on a drive. The operation can be a new operation on behalf of a user request, a continuation operation made necessary because a requested transfer crossed a track boundary, a retry caused by a failure of another transfer, or a WRITE to an alternate block, made necessary when a transfer failed because of a bad block on the disk.

First, turn off the **DP_DELAYRD** flag. This flag controls the delayed assignment of an alternate block when a bad block is encountered on a READ request. This is discussed in detail below.

If the **GD_OPENED** flag is no longer set, the drive has gone offline: fail the transfer request immediately. **GD_OPENED** is cleared by **gdtimer(2K)** as a result of a failure status returned from **gdvs32timer()**. Note that the driver calls **gdiodone(2K)** directly. Normally, this call is made by **gdintr(2K)** when the completion interrupt is received. In this case, since the driver does not even attempt the I/O, there won't be a completion interrupt.

The **b_resid** field is set to the number of bytes remaining in the original transfer request (**b_count**, which is the original transfer length, minus **xfrcnt**, which is the total number of bytes already transferred). **Gdvs32doxfr()** may have been called to perform continuation I/O on a request that was broken up into pieces. Thus, the residue (bytes remaining to transfer) may not be equal to the original transfer length.

Finally, **gdvs32doxfr()** calls **gdvs32start()** to (attempt to) start I/O on all of the remaining drives. This results in (indirect) recursion, since **gdvs32start()** calls **gdvs32doxfr()** to initiate the transfer.

```
/*
 * Gdvs32doxfr() - initiate transfer on the requested drive.
 */
 VOID
gdvs32doxfr(cyl, trk, sec, tcnt, mode, dev)
register ushort cyl, trk, sec, mode;
register uint tcnt;
register dev_t dev;
{
	register struct gdsw *gds = &gdsw[gdpos(dev)];
	register struct iobuf *dp = &gdutab[gdpos(dev)];
	register struct gddriver *gdr = &gdvs32driver[VSindex(dev)];
	register struct iobuf *gdt = &gdtab[gdctl(dev)];
	register struct buf *bp = dp->b_actf;
	int ind;

	int ctl = gdctl(dev);
	register short drive = gddrive(dev);
	register struct gdvs32ctl *smdp = Vctladdr;
	register struct gdvs32iopb *iopbp = &smdp->Xiopb[drive][0];

	aaprintf("gdvs32doxfr(cyl: 0x%x trk: 0x%x sec: 0x%x ", cyl, trk, sec);

	aaprintf("tcnt: 0x%x mode: 0x%x dev: 0x%x)", tcnt, mode, dev);

	dp->b_flags &= ~DP_DELAYRD;
	/* In case the drive went offline */
	if (!(gds->v_flags & GD_OPENED)) {
		/* Release the controller */
		dp->b_flags &= ~DP_ACTIVE;

		/* Mark the I/O as an error */
		bp->b_flags |= B_ERROR;
		bp->b_resid = bp->b_bcount - gdr->xfrcnt;
		bp->b_error = EIO;
		bp->b_flags &= ~B_START;

		/* Mark the I/O as done, take it off the queue */
		gdiodone(bp, dp, gdt);

		/* Get the next I/O */
		gdvs32start(dev);
		return;
	}
```

If the I/O crosses a track boundary, the driver breaks it into multiple operations, one operation per track. The original READ or WRITE request is not complete until all of these subrequests finish. **Tcnt** is the total sector count for the **current** DMA operation (not the current I/O request, only the piece of it that is being done now.) **Gdvs32intr()** is responsible for continuing I/O on subsequent pieces of the original request if it is broken up in this manner.

The outer **if** statement allows redirected bad block accesses to proceed. The disks are formatted with a spare sector at the end of each track. If a bad sector is encountered on a track, **gdaltblk()** is called to assign the nearest spare sector to be used in its place. This alternate block always has a sector number equal to **sectrk**, making special case code necessary here.

If the **GD_PHYSADDR** bit is set, the physical disk parameters are used; otherwise, the logical parameters are used.

```
                    /* gdvs32doxfr() continued */

/* Does I/O cross a track boundry? */
if (tcnt == 1 && sec == gds->sectrk) {
     /* The I/O is to an alternate block - allow it */
} else {
     if (gds->v_flags & GD_PHYSADDR) {
          if ((sec+ tcnt) > gds->dsk.psectrk)
               tcnt = gds->dsk.psectrk - sec;
     } else {
          if ((sec+ tcnt) > gds->sectrk)
               tcnt = gds->sectrk - sec;
     }
}
```

Here, the driver deals with I/O on cylinders known to contain bad sectors. The **if** statement means: "if the disk is in Convergent Technologies format, and if it contains a known bad block." The bad block queue (**bbq**) is a per-drive array of short integers, indexed by cylinder number. If nonzero, the value is the index in this drive's bad block table of the entry corresponding to the first bad block on this cylinder. If zero, there are no (known) bad blocks on this cylinder.

The **for** loop scans the linked list of bad block table entries for the current cylinder. The first **if** statement is TRUE if the bad block referenced by the current table entry falls within the range of requested sectors. The next **if** statement is TRUE if the bad block from the table is equal to the first requested sector; in this case, the I/O must be redirected. Otherwise, the sector count is reduced to break the I/O into three operations: one for the sectors before the bad block, one for the bad block, and one for any remaining block(s) on the cylinder.

If an alternate block has not yet been assigned (**altblk** is equal to zero), the driver is dealing with a delayed block assignment. When a READ fails because of a bad block, the driver marks it in the bad block queue and returns a hard error to the user. It does not assign an alternate block, since a controller can fail to read a block once, but succeed the next time. As long as the driver does not assign an alternate block, the user can continue trying to read the data. When it is clear that the data is lost, the user can write a filler block into the file at the position of the bad sector. When the driver receives a WRITE request on a bad block that has no alternate block assigned, it calls **gdaltblk()** to allocate a spare sector.

If the request is a READ, the driver sets the **DP_DELAYRD** flag and continues. In this case, the user is attempting to read a block that has had a READ failure already. When **gdvs32intr()** processes the READ completion interrupt, if the READ was **successful** but **DP_DELAYRD** is set, the driver allocates an alternate block and writes the data to the spare sector.

```
                    /* gdvs32doxfr() continued */

/* Does I/O cross a bad block ? */
if ((gds->v_flags & GD_CT_FMT) && (ind=gds->bbq[cyl])) {
    register struct bbmcell *gdb = gds->bb;
    register int ssec = trk * gds->dsk.psectrk + sec;
    register int esec = ssec + tcnt;
    register int bb;

    for (; ind != 0; ind = gdb[ind].nxtind) {
        if (((bb=gdb[ind].badblk) >= ssec) && (bb < esec)) {
            if (bb == ssec) {
                if (gdb[ind].altblk == 0) {
                    /* It was a delayed assignment */
                    if (mode == CMD_READ) {
                        /* If the read succeeds, assign it (later) */
                        dp->b_flags |= DP_DELAYRD;
                        continue;
                    } else {
                        if (gdb[ind].altblk=gdaltblk(cyl,dp->b_dev))
                            gdprint(bp->b_dev,
                            "delayed bad block assignment made");
                        else
                            gdprint(bp->b_dev,
            "unable to get alternate block for delayed assignment");
                        gds->bbchanged++;
                    }
                }
                tcnt = 1;
                /* Sector is the last sector on the track */
                sec = gds->dsk.psectrk-1;
                bb = gdb[ind].altblk;
                trk = bb % gds->dsk.heads;
                cyl = bb / gds->dsk.heads;
                break;
            } else {
                tcnt  = imin((int)(bb-ssec), (int)tcnt);
                break;
            }
        }
    }
}
```

If control gets this far and the sector transfer count (**tcnt**) is zero, there is a bug in the driver logic: the **gdpanic(2K)** call reports this fact.

Through the pointer **gdr**, the **gddriver** structure is updated to describe the current transfer.

If the controller is currently in use, set the **DP_WAITING** flag and return. The interrupt handler will start this I/O when the controller becomes free. If the controller is free, grab it by setting the **DT_INUSE** flag.

If the **DP_READVHB** flag is set, the drive parameters need to be reinitialized: clear the flag, set up the fields in the UIB, set up a **DC_INITIALIZE** command in the current Xiopb, and increment the **iopbp** pointer. This is how the driver chains together I/O requests. The **iopbp** pointer always points to the current auxiliary IOPB (Xiopb). When it comes time to issue the command (set the GO bit), the driver clears the **CF_LINK_ENABLE** bit on the last IOPB. This terminates the linked list of IOPBs.

```
                    /* gdvs32doxfr() continued */

if (tcnt == 0)
    gdpanic("zero transfer");

/* Setup the current transfer */
gdr->rptcyl = cyl;
gdr->rpttrk = trk;
gdr->rptsec = sec;
gdr->rpttcnt = tcnt;
gdr->rptmode = mode;

if (gdt->b_flags & DT_INUSE) {
    /* Controller is in use */
    dp->b_flags |= DP_WAITING;
    return;
}
/* Grab the controller */
gdt->b_flags |= DT_INUSE;
gdt->b_dev = dev;
gdt->b_actf = (struct buf *)dp;

if ((dp->b_flags & DP_READVHB)) {
    dp->b_flags &= ~DP_READVHB;

    /* Initialize the UIB for the drive */
    smdp->Xuib[drive].v0nh = gds->dsk.heads;
    smdp->Xuib[drive].psectrk = gds->dsk.psectrk;
    smdp->Xuib[drive].sectorsz = gds->dsk.sectorsz;
    smdp->Xuib[drive].cyls = gds->dsk.cyls;
    smdp->Xuib[drive].gap1 = gds->dsk2.gap1;
    smdp->Xuib[drive].gap2 = gds->dsk2.gap2;

    /* Link this IOPB onto the chain */
    iopbp->bufferp.lo = (ushort)&smdp->Xuib[drive];
    iopbp->bufferp.hi = 0;
    iopbp->buf_memtype = MEM_INTERNAL;
    iopbp->buf_addmod = AM_16_K;
    iopbp->error = iopbp->status = 0;
    iopbp->flags = iopblink(drive);
    iopbp->status = 0;
    iopbp->command = DC_INITIALIZE;
    iopbp++;
}
```

If there is only one drive active, don't bother to do an explicit SEEK. The controller is able to do an implied SEEK with a READ or WRITE request. Setting **gdr->curcyl** equal to **cyl** (the target cylinder for this request) prevents the following code from issuing a SEEK.

If **gdr->curcyl** has been set to -1 (by **gdvs32open()** or **gdvs32errors()**), do a RESTORE instead of a SEEK operation. In either case, call **gdvs32seek()**, set the **DP_SEEKING** flag, and return. The interrupt handler will continue the I/O when the SEEK completion interrupt is received.

If the command is a FORMAT, the driver sets up the current IOPB with a **DC_INITIALIZE** command.

```
                    /* gdvs32doxfr() continued */

if (gdt->qcnt == dp->qcnt) {
    /*
     * We are the only drive with any work to do;
     * do an implied seek with the read
     */
    gdr->curcyl = cyl;
}
/* Seek? */
if ((mode != CMD_FORMAT) && (gdr->curcyl != cyl)) {
    if (gdr->curcyl == -1)
        /* Restore the drive */
        gdvs32seek(iopbp, cyl, trk, dev, 1);
    else
        /* Seek to the desired cyl */
        gdvs32seek(iopbp, cyl, trk, dev, 0);

    dp->b_flags |= DP_SEEKING;
    return;
}
if (mode == CMD_FORMAT) {
    register ushort *bufr=(ushort *)((int)gdr->dma_addr);

    /* Buffer contains psectrk, gap1, gap2 */
    smdp->Xuib[drive].psectrk = (ushort)*bufr++;
    smdp->Xuib[drive].gap1 = (ushort)*bufr++;
    smdp->Xuib[drive].gap2 = (ushort)*bufr;

    /* Link this IOPB onto the chain */
    iopbp->bufferp.lo = (ushort)&smdp->Xuib[drive];
    iopbp->bufferp.hi = 0;
    iopbp->buf_memtype = MEM_INTERNAL;
    iopbp->buf_addmod = AM_16_K;
    iopbp->error = iopbp->status = 0;
    iopbp->flags = iopblink(drive);
    iopbp->status = 0;
    iopbp->command = DC_INITIALIZE;
    iopbp++;

    iopbp->error = iopbp->status = 0;
    iopbp->cyl = cyl;
    iopbp->head = trk;
    iopbp->bufferp.lo = NULL;
```

If the command was not FORMAT, check to see if the routine was called from **physio(2K)**. If so, call **setmap(2K)** to remap the user's I/O buffer into kernel virtual memory. As discussed under **gdvs32open()**, DMA devices cannot reference user virtual memory addresses, because they change at context switch time. Since **physio()** sleeps after calling **gdstrategy(2K)**, the original user process cannot be running at this time. So, the driver must allocate some page table entries in kernel memory, and then copy the page frame numbers from the user's page table entries that point to the I/O buffer. This virtual address range was allocated in **gdvs32open()** through a call to **sptalloc(2K)**; its address was saved in **dp->vaddr**.

Now, **setmap()** copies the page frame numbers so that the kernel virtual memory points to the physical memory containing the user's I/O buffer. In this way, the buffer acquires a kernel virtual address, **in addition** to its user virtual address. The DMA hardware uses this kernel address to perform the transfer.

Next, the driver sets up the current IOPB to describe the transfer. If **gdr->dma_addr** is not on a longword boundary, the controller must use 16-bit transfers. Otherwise, it can use 32-bit transfers.

Iopbflags() clears the **CF_LINK_ENABLE** bit, thus terminating the linked list of IOPBs.

Gdt_wtime is set to the DMA timeout value. **Gdtimer(2K)** decrements this counter and calls **gdpanic(2K)** if it ever reaches zero.

The **DP_ACTIVE** flag is set to indicate that this drive has I/O activity on it. The **DT_DMAON** flag indicates that a DMA operation is in progress.

```
                /* gdvs32doxfr() continued */

} else {    /* mode != CMD_FORMAT */
    /* If physio, we must re-map the DMA */
    if (bp->b_flags & B_PHYS) {
        mprintf("calling setmap(0x%x, 0x%x, 0x%x, 0x%x)",
            bp, dp->vaddr, gdr->xfrcnt*gds->dsk.sectorsz,
            tcnt*gds->dsk.sectorsz);
        gdr->dsk_dma_addr =
            setmap(bp, dp->vaddr, gdr->xfrcnt*gds->dsk.sectorsz,
                tcnt*gds->dsk.sectorsz);
        mprintf("vaddr: 0x%x", gdr->dsk_dma_addr);
    } else
        gdr->dsk_dma_addr = gdr->dma_addr;

    iopbp->bufferp.lo = (ushort)vtocv(gdr->dsk_dma_addr);
    iopbp->bufferp.hi = (ushort)(vtocv(gdr->dsk_dma_addr) >> 16);
    iopbp->buf_addmod = AM_32_K;
    if ((uint)gdr->dma_addr % 4)
        iopbp->buf_memtype = MEM_16;
    else
        iopbp->buf_memtype = MEM_32;

    iopbp->cyl = cyl;
    iopbp->sectors = tcnt;
    iopbp->sec = sec;
    iopbp->head = trk;
}

iopbp->flags = iopbflags(drive);
iopbp->status = 0;

gdt->wtime = gds->DMAto;
dp->b_flags |= DP_ACTIVE;
gdtab[gdctl(dp->b_dev)].b_flags |= DT_DMAON;
```

The driver sets the appropriate command in the **command** field of the current IOPB. Finally, it sets up the linked IOPB information in the base IOPB and sets the GO bit, initiating the transfer.

```
                    /* gdvs32doxfr() continued */

    switch (mode) {
    case CMD_READ:
        iopbp->command = DC_READ_SECTORS;
        break;
    case CMD_WRITE:
        iopbp->command = DC_WRITE_SECTORS;
        break;
    case CMD_FORMAT:
        /* Initialize with new values */
        iopbp->command = DC_FORMAT_TRACK;
        break;
    default:
        gdpanic("Gdvs32doxfr() - invalid mode");
    }

    smdp->iopb.iopbp.lo = (ushort)&smdp->Xiopb[drive][0];
    smdp->iopb.iopbp.hi = 0;
    smdp->iopb.flags = iopblink(drive);
    smdp->iopb.status = 0;
    oprintf(">%d", drive);

    /* Start the command */
    smdp->command |= CMD_GO;
}
```

GDVS32SEEK()

Gdvs32seek() performs SEEK and RESTORE operations, according to the sense of the **restoref** parameter. **Iopbp** is a pointer to the current IOPB, which is always one of the auxiliary IOPBs in **gdvs32ctl**. The routine sets up the current IOPB, sets the desired command in the **command** field, sets the link bit and IOPB pointer address in the base IOPB, and then sets the GO bit, initiating the transfer.

```
/*
 * Gdvs32seek() - perform a SEEK operation.
 */
 VOID
gdvs32seek(iopbp, cyl, trk, dev, restoref)
register ushort cyl, trk;
register dev_t dev;
short restoref;
register struct gdvs32iopb *iopbp;
{
	struct iobuf *dp = &gdutab[gdpos(dev)];
	register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
	register short drive = gddrive(dev);

	aaprintf(
	"gdvs32seek(iopbp: 0x%x cyl: 0x%x trk: 0x%x dev: 0x%x restoref: 0x%x)",
		iopbp, cyl, trk, dev, restoref);
	iopbp->cyl = cyl;
	iopbp->head = trk;
	iopbp->flags = iopbflags(drive);
	iopbp->status = 0;
	if (restoref)
		iopbp->command = DC_RESTORE;
	else
		iopbp->command = DC_SEEK;

	smdp->iopb.iopbp.lo = (ushort)&smdp->Xiopb[drive][0];
	smdp->iopb.iopbp.hi = 0;
	smdp->iopb.flags = iopblink(drive);
	smdp->iopb.status = 0;
	oprintf("#%d", drive);

	/* Start the command */
	smdp->command |= CMD_GO;
}
```

GDVS32INTR()

Gdvs32intr() is called from **gdintr(2K)** whenever an interrupt is received from the V/SMD 3200. Its function is to process I/O completions, I/O continuations (when the original request was broken up into several I/O operations), retries, and status change interrupts (primarily SEEK completions).

Theoretically, it is impossible for both the Status Change (**CS_SC**) bit and the Operation Done (**CS_OD**) bit to be set, since the controller freezes the status register and generates an interrupt request whenever it sets either one of them. The **gdpanic(2K)** call reports that the impossible has occurred.

If it is a Status Change interrupt, the driver sets **iodonef** to indicate that I/O is being continued, and goes to the **cont_io** label. Status Change interrupts are received for the following conditions:

- Drive Ready/Not Ready.
- Drive Fault.
- On Cylinder (SEEK complete with no error).
- SEEK Error.

The **iodonef** flag could have been named **iocontinued**, since it is TRUE when the current I/O request is not complete, and FALSE when it is complete. **Gdvs32intr()** returns this flag to **gdintr()** to indicate whether or not the active I/O is complete. Gdintr() calls **iodone(2K)** on the active buffer whenever the device driver's interrupt handler returns 0.

```
/*
 * Gdvs32intr() - process V/SMD 3200 interrupts.
 *
 * Returns:
 *    0 - Operation complete.
 *    1 - Operation continued or retried.
 */
 int
gdvs32intr(bp, dev, vec)
register struct buf *bp;
register dev_t dev;
register int vec;
{
     register struct iobuf *dp = &gdutab[gdpos(dev)];
     register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
     register struct iobuf *dp = &gdutab[gdpos(dev)];
     register struct iobuf *gdt = &gdtab[gdctl(dev)];
     register struct gddriver *gdr = &gdvs32driver[VSindex(dev)];
     register struct gdsw *gds = &gdsw[gdpos(dev)];
     register short drive = gddrive(dev);
     register struct gdvs32iopb *iopbp = &smdp->Xiopb[drive][0];
     ushort stat_reg, unit_reg, err_reg, sec_reg, ctl_reg;
     int iodonef;

     if ((smdp->status & CS_SC) && (smdp->command & CS_OD))
          gdpanic("double interrupt!");

     aaprintf("gdvs32intr(bp: 0x%x dev: 0x%x vec: 0x%x)", bp, dev, vec);
     if (smdp->status & CS_SC) {
          /* Must have been a status change */
          gdvs32statuschange(dev);

          iodonef = 1;
          goto cont_io;
     }
```

If the current buffer is waiting for the controller (**DP_WAITING** is set) or, if it is not even active (**DP_ACTIVE** is clear), this also indicates a stray interrupt. Clear the bits in the command register and continue the I/O.

If the driver gets past the stray interrupt tests, then the interrupt indicates the completion of some operation. Clear the controller in use and DMA active flags, and save the current contents of the controller command and status registers. The **unit_status** array is arranged such that element 0 contains the status bits for drive 1, and vice versa. The construct [**drive ^ 1**] handles this turnabout.

Finally, the interrupt handler clears the Done and Error bits in the Controller Command register.

```
                /* gdvs32intr() continued */

if ((dp->b_flags & DP_WAITING) |!(dp->b_flags & DP_ACTIVE)) {
     ushort command = smdp->command;
     ushort status = smdp->status;

     /* Clear the interrupt */
     if (command & (CS_OD |CS_ELC)
          smdp->command &= ~(CS_OD|CS_ELC);
     else
     if (status & (CS_SC |CS_SCS)
          smdp->status &= ~(CS_SC|CS_SCS);

     gdprint(dev, "stray logged");
     logstray((physadr)(vec));
     iodonef = 1;
     goto cont_io;
}

/* Operation concluded interrupt.
 * Controller is no longer in use.
 */
gdt->b_flags &= ~(DT_INUSE |DT_DMAON);
oprintf("<%d", drive);

ctl_reg = smdp->command;
unit_reg = smdp->unit_status[drive ^ 1];
err_reg = iopbp->error;
stat_reg = smdp->iopb.status;
sec_reg = smdp->iopb.sec;

/*
 * Clear the interrupt. There could be a status-change interrupt
 * buried behind this one. If so, it will be handled later.
 */
smdp->command &= ~(CS_OD|CS_ELC);
```

If there were any error bits set in the controller registers, the interrupt handler calls **gdvs32errors()** to process them. This routine returns a flag indicating whether the error was fatal, causing the I/O to be completed with error, or whether it can be retried, causing the I/O to be continued.

If there were no errors, set the **GD_READY** flag for the drive, since it just completed an I/O request.

If the **DP_SEEKING** flag is set, the last command was a SEEK. Set **iodonef** to indicate that the current request is not complete.

Gdvs32doxfr() will have set the **DP_DELAYRD** flag if it detected that the user was attempting to READ a block that could not be read one or more times previously. Here, **gdvs32intr()** detects that the flag is set. Since it "knows" at this point that the active I/O request completed successfully, the interrupt handler concludes that the driver has just read a block that it failed to read before. Now that the data from the bad block is known, the driver (finally) can allocate an alternate block and write the good data to it. In this way, the driver removes the marginal block from the disk, but recovers the data it contained.

```
                    /* gdvs32intr() continued */

/* Process errors */
if (  (ctl_reg & CS_ELC) |
      (ctl_reg & CS_BERR) |
      (unit_reg & (US_FAULT|US_BADSEEK)) |
      !(unit_reg & US_DREADY) |(stat_reg == SR_ERR)) {
      iodonef = gdvs32errors(stat_reg, err_reg, unit_reg, ctl_reg,
            sec_reg, dp);
      goto cont_io;
} else
      gds->v_flags |= GD_READY;
switch (stat_reg) {
case SR_OK:
      /* Normal completion */
      break;
case SR_EXC:
      /* Completion with exception - format a recovered error rcd */
      fmtberr(dp, (int)gds->partab[gdslice(bp->b_dev)].strk,
      (int)gds->dsk.heads, (int)gds->dsk.psectrk, (int)gds->ctlr);
      break;
case SR_CIP:
      gdprint(dev, "command in progress");
      break;
default:
      printf("status %x", stat_reg);
      gdpanic("illegal status returned from disk controller");
}
if (dp->b_flags & DP_SEEKING) {
      /* The controller must have issued a SEEK */
      iodonef = 1;
      goto cont_io;
}
if (dp->b_flags & DP_DELAYRD) {
      /*
       * We have successfully read a delayed bad block:
       * reassign and rewrite the block. Redo the I/O
       * (as a WRITE); gdvs32doxfr() re-assigns the bad block.
       */
      gdvs32doxfr(gdr->cyl, gdr->trk, gdr->sec, gdr->tcnt,
            CMD_WRITE, dev);
      iodonef = 1;
      goto cont_io;
}
```

This code alters all of the I/O pointers and counters to skip over the data that was just transferred. Thus, the data structures are set to describe the next part of the transfer request.

If there is no more data to transfer (**tcnt** is equal to 0), the I/O request is complete; clear the waiting and active flags and set **iodonef** to 0, which will cause **gdintr(2K)** to call **iodone(2K)** on the buffer header. This will reawaken the user's process and allow his original **read(2)** or **write(2)** system call to complete.

Otherwise, there is more data to transfer, so the driver calls **gdvs32doxfr()** to initiate the I/O, and sets **iodonef** to 1, indicating to **gdintr()** that the I/O is being continued.

```
                    /* gdvs32intr() continued */

/* Setup for continuation of transfer if necessary */
gdr->xfrcnt += gdr->rpttcnt;
gdr->sec += gdr->rpttcnt;
gdr->tcnt -= gdr->rpttcnt;
if (gdr->sec >= ((gds->v_flags&GD_PHYSADDR) ?
        gds->dsk.psectrk : gds->sectrk)) {
    gdr->sec -= ((gds->v_flags&GD_PHYSADDR) ?
        gds->dsk.psectrk : gds->sectrk);
    gdr->trk++;
    if (gdr->trk >= gds->dsk.heads) {
        gdr->trk -= gds->dsk.heads;
        gdr->cyl++;
    }
}

/* Increment by bytes, each block is 512 bytes (2**9) */
gdr->dma_addr += (gdr->rpttcnt << 9);
if (!gdr->tcnt) {
    dp->b_flags &= ~(DP_WAITING|DP_ACTIVE);
    iodonef = 0;
    /* Clear error flag */
    bp->b_resid = 0;
} else {
    /* Continue with the I/O */
    gdvs32doxfr(gdr->cyl, gdr->trk, gdr->sec, gdr->tcnt,
        gdr->mode, dev);
    iodonef = 1;
}
```

First, **gdvs32intr()** calls **gdvs32statuschange()** to update **curcyl**.

Next, the interrupt handler attempts to start a SEEK operation on the controller. If the controller is not in use (**DT_INUSE** is not set), **gdvs32intr()** chains down the I/O queue looking for a request that is waiting for the controller (**DP_WAITING** is set) and that requires disk head movement (**curcyl** is not equal to **cyl**). If **gdvs32intr()** finds a waiting request, it starts the I/O by calling **gdvs32doxfr()**.

Then, if the controller still is not in use, the interrupt handler tries to start the first I/O it can find. (The fact that the controller still is not busy indicates that **gdvs32intr()** didn't start a SEEK operation in the previous **while** loop.)

Finally, the interrupt handler returns the **iodonef** flag, indicating whether or not **gdintr(2K)** should call **iodone(2K)** on the buffer.

```
                    /* gdvs32intr() continued */

cont_io:
    /* Update curcyl on all drives */
    gdvs32statuschange(dev);
    /* Find a drive to seek on */
    if (!(gdt->b_flags & DT_INUSE)) {
        dp=(struct iobuf *)gdt->b_actf;
        do {
            gdr = &gdvs32driver[VSindex(dp->b_dev)];
            if ((dp->b_flags&DP_WAITING)&&(gdr->curcyl!=gdr->cyl)) {
                dp->b_flags &= ~DP_WAITING;
                gdvs32doxfr(gdr->rptcyl, gdr->rpttrk, gdr->rptsec,
                    gdr->rpttcnt, gdr->rptmode, dp->b_dev);
                break;
            }
            dp = (struct iobuf *)dp->b_forw;
        } while (dp != (struct iobuf *)gdt->b_actf);
    } else
        oprintf("$");

    /* Find a drive to perform I/O on */
    if (!(gdt->b_flags & DT_INUSE)) {
        dp=(struct iobuf *)gdt->b_actf;
        do {
            if (dp->b_flags & DP_WAITING) {
                dp->b_flags &= ~DP_WAITING;
                gdr = &gdvs32driver[VSindex(dp->b_dev)];
                if (gdr->curcyl != gdr->cyl)
                    gdpanic("curcyl!=cyl");
                gdvs32doxfr(gdr->rptcyl, gdr->rpttrk, gdr->rptsec,
                    gdr->rpttcnt, gdr->rptmode, dp->b_dev);
                break;
            }
            dp = (struct iobuf *)dp->b_forw;
        } while (dp != (struct iobuf *)gdt->b_actf);
    } else
        oprintf("$");
    /* Tell gdintr() whether current xfer still in progress. */
    return(iodonef);
}
```

GDVS32ERRORS()

Gdvs32errors() is called by **gdvs32intr()** to process the following error conditions on the drive:

- Bus error.
- Drive fault.
- SEEK error.
- Drive not ready.
- General error on last command status.

If the **GD_QUIET** flag is not set, call **fmtberr()** to build an **eblock** data structure describing the error. This structure is defined in **<sys/erec.h>**.

If the Write Protect bit is turned on in the Controller Unit Status register, the driver prints a message to that effect.

Process the Drive Fault and Drive Not Ready errors here. If the **GD_QUIET** flag is not set, the driver prints the appropriate error message, and then clears the **GD_READY** flag. When **gdintr(2K)** detects that the flag is not set, it closes the drive, effectively performing a dismount on it. This causes all future I/O requests to the drive to fail in **gdstrategy(2K)**.

The driver calls **binval()** to invalidate (set the **B_STALE** and **B_AGE** bits on) all system buffers belonging to this drive.

It clears the **DP_SEEKING** and **DP_ACTIVE** flags on the current drive.

The driver returns an I/O error indication in the buffer header.

It returns 0 to indicate that the I/O operation is complete. This ultimately informs **gdintr()** to call **iodone(2K)**, waking up the original calling process.

```
/*
 * Gdvs32errors() - process errors.
 * Returns:
 *    0 - Operation complete (possibly with error).
 *    1 - Operation continued or retried.
 */
 int
gdvs32errors(stat_reg, err_reg, unit_reg, ctl_reg, sec_reg, dp)
ushort stat_reg, err_reg, unit_reg, ctl_reg, sec_reg;
register struct iobuf *dp;
{
	register struct iobuf *gdt = &gdtab[gdctl(dp->b_dev)];
	register struct gddriver *gdr = &gdvs32driver[VSindex(dp->b_dev)];
	register struct gdsw *gds = &gdsw[gdpos(dp->b_dev)];
	register struct buf *bp = dp->b_actf;
	register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
	register struct gdvs32iopb *iopbp=
		&smdp->Xiopb[gddrive(dp->b_dev)][0];

	oprintf("gdvs32errors(stat_reg: 0x%x err_reg: 0x%x unit_reg: 0x%x",
		stat_reg, err_reg, unit_reg);
	oprintf("       ctl_reg: 0x%x sec_reg: 0x%x dp: 0x%x)",
		ctl_reg, sec_reg, dp);
	if (!(gds->v_flags & GD_QUIET))
		fmtberr(dp, (int)gds->partab[gdslice(bp->b_dev)].strk,
			(int)gds->dsk.heads, (int)gds->dsk.psectrk, (int)gds->ctlr);
	if (unit_reg & US_WRPROT)
		gdprint(bp->b_dev, "Drive is write protected");
	if ((unit_reg & US_FAULT) |!(unit_reg & US_DREADY)) {
		if (!(gds->v_flags & GD_QUIET)) {
			if (!(unit_reg & US_DREADY))
				gdprint(bp->b_dev, "Drive went off line");
			if (unit_reg & US_FAULT)
				gdprint(bp->b_dev, "Drive Faulted, taken off line");
		}
		/* Mark the drive as not ready - gdintr() will close it */
		gds->v_flags &= ~GD_READY;
		binval(dp->b_dev);        /* Invalidate all blocks */
		dp->b_flags &= ~(DP_SEEKING|DP_ACTIVE); /* Free the drive */
		bp->b_flags |= B_ERROR;
		bp->b_resid = bp->b_bcount - gdr->xfrcnt;
		bp->b_error = EIO;
		bp->b_flags &= ~B_START;
		return(0);
	}
```

The driver decrements the remaining retry count and increments the repeat count.

The **switch** statement differentiates among three values for remaining retries:

- None left, in which case the I/O has failed and must be terminated with error.
- One-half the number of retries left (the maximum retry count, **GDRETRIES**, currently is set to ten, **GDRETREST** currently is five), in which case the driver issues a RESTORE command (to recalibrate the drive and clear any fault conditions).
- Any other number of retries, in which case the driver simply reissues the original command.

When the remaining retry count goes to zero, the I/O request has failed. If the disk is in Convergent Technologies format, the driver initiates bad block processing on the offending sector. If the failed transfer was not a READ request, it calls **gdaddbadblk()** to add an entry to the bad block table and allocate a spare sector to be used as an alternate.

Then, the driver reissues the WRITE command to the alternate sector and returns. On the other hand, if the failed request was a READ, the driver adds a bad block entry to the table but **does not** allocate an alternate block. This allows the user to continue trying to read the bad sector as long as necessary. Eventually, the READ may complete without error, since bad sectors often are marginal, failing some times and succeeding others.

```
                    /* gdvs32errors() continued */

gdr->retries--;
gdr->rpts++;

/* Retry count exceeded? */
switch (gdr->retries) {
case 0:
    if (gds->v_flags & GD_CT_FMT) {
        if (gdr->mode != CMD_READ) {
            /* Mark the block as bad & re-assign */
            if (!gdaddbadblk(gdr->rptcyl, gdr->rpttrk, sec_reg,
                    1, dp->b_dev))
                gdprint(bp->b_dev, "Unable to reassign bad block");
            /* Re-do the I/O */
            gdvs32doxfr(gdr->rptcyl, gdr->rpttrk, gdr->rptsec,
                gdr->rpttcnt, gdr->rptmode, dp->b_dev);
            return(1);      /* Retrying the current I/O */
        } else {
            /* Mark the block as bad, but delay re-assignment */
            if (!gdaddbadblk(gdr->rptcyl, gdr->rpttrk, sec_reg,
                    0, dp->b_dev))
                gdprint(bp->b_dev, "Unable to reassign bad block");
        }
    }
    /* Free the drive */
    dp->b_flags &= ~DP_ACTIVE;

    /* Mark the error in the bp */
    bp->b_flags |= B_ERROR;
    bp->b_resid = bp->b_bcount - gdr->xfrcnt;
    bp->b_error = EIO;
    bp->b_flags &= ~B_START;

    if (!(gds->v_flags & GD_QUIET)) {
        printf("Logical Block %d (Cyl %d, Head %d, Sector %d)",
            FsPTOL(bp->b_dev, (bp->b_blkno+gdr->rptsec-sec_reg)),
            gdr->rptcyl, gdr->rpttrk, sec_reg);
        gdprint(bp->b_dev, ":Transfer Failed.");
    }
    return(0);      /* The current I/O is finished */
```

If one-half (**GDRETREST**) of the retry count has been exhausted, issue a RESTORE command to the controller: calling **gdvs32seek()** with a final parameter of 1 performs a RESTORE operation. Turn on the **DP_SEEKING** flag because the RESTORE actually causes a seek to cylinder zero, along with a drive recalibration and a clearing of any outstanding FAULT condition on the drive.

The **default** case handles all of the remaining retries. The driver calls **gdvs32doxfr()** to reissue the original I/O request.

```
                    /* gdvs32errors() continued */

    case GDRETREST:
        /* Do a restore */
        dp->b_flags |= DP_SEEKING;
        gdr->curcyl = (ushort)-1;
        gdt->b_flags |= DT_INUSE;
        gdt->b_dev = dp->b_dev;
        gdt->b_actf = (struct buf *)dp;
        gdvs32seek(iopbp, gdr->rptcyl, gdr->rpttrk, dp->b_dev, 1);
        return(1);      /* Retrying the current I/O */
    default:
        /* Retry the operation */
        gdvs32doxfr(gdr->rptcyl, gdr->rpttrk, gdr->rptsec, gdr->rpttcnt,
            gdr->rptmode, dp->b_dev);
        return(1);      /* Retrying the current I/O */
    }
    /*NOTREACHED*/
}
```

GDVS32STATUSCHANGE()

Gdvs32statuschange() is called from **gdvs32intr()** to process Status Change interrupts from the controller. The main **for** loop processes both drives on the controller. First, the routine reads the controller status byte for the current drive. (The fragment **unit_status[drive ^ 1]** selects byte **1** for drive number 0, and byte 0 for drive number 1, corresponding to the physical layout of the register.)

Next, the routine tests for FAULT conditions. Code will be added here in the future to further process fault conditions.

Next, **gdvs32statuschange()** checks for seek complete interrupts. If the current drive did not have an outstanding SEEK request, the rest of the loop is skipped. If the controller has not set the **US_ONCYL** (on cylinder) bit in the status register, the seek is not complete. Finally, if the drive being processed is not the active drive in **gdtab**, skip the rest of the loop.

If the driver passes all of the tests, it processes the SEEK complete interrupt. In any case, it then clears the interrupt and returns.

```
/*
 * Gdvs32statuschange() - process Status Change interrupt.
 */
 VOID
gdvs32statuschange(dev)
register dev_t dev;
{
	register struct iobuf *dp = &gdutab[gdpos(dev)];
	register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
	register struct gddriver *gdr;
	short ctl = gdctl(dev);
	ushort drive, unit_reg;

	/* Status change interrupt. Update status for both drives */
	for (drive=0; drive<=1; drive++) {
		/* Get the status for the drive */
		unit_reg = smdp->unit_status[drive ^ 1];

		dp = &gdutab[gdmkpos(ctl, drive)];
		gdr = &gdvs32driver[VSindex(dp->b_dev)];

		if (unit_reg & US_FAULT)
			gdprint(dp->b_dev, "drive faulted");
		if (!(dp->b_flags & DP_SEEKING))
			continue; /* Drive not seeking */
		if (!(unit_reg & ON_CYL))
			continue; /* Not ready yet */
		if ((gdtab[ctl].b_flags & DT_INUSE) &&
			(gddrive(gdtab[ctl].b_dev) == gddrive(dp->b_dev)))
			/* No operation complete interrupt yet */
			continue;
		oprintf(" %d-%x", drive, unit_reg);
		/* Drive is now on cylinder */
		dp->b_flags &= ~DP_SEEKING;
		dp->b_flags |= DP_WAITING;
		gdr->curcyl = gdr->rptcyl;
	}
	/* Clear the interrupt */
	if (smdp->status & CS_SC)
		smdp->status &= ~(CS_SC|CS_SCS);
}
```

GDVS32TIMER()

Gdvs32timer() is called periodically by **gdtimer(2K)** as a result of a **timeout(2K)** call. This call **does not** mean that a timeout has occurred: it simply means that time is passing. It gives the driver the opportunity to report the drive status back to the general disk driver. There are three recognized return codes from the **devtimer(2K)** routine: 0 indicates that the drive in question is NOT ready; 1 means that the drive is ready; and -1 means that the controller is busy.

You should be aware that there is a deadman timer available for low-level drivers in the general disk system. When your driver sets the **gdtab.wtime** field for your device to some nonzero value, **gdtimer()** decrements it every time it is entered. When it counts down to zero, **gdtimer()** calls **gdpanic(2K)** to report a DMA operation timeout. In this driver, **gdvs32doxfr()** sets **wtime** to the **DMAto** value from the **gdsw** entry for the controller.

```
/*
 * Gdvs32timer() - return drive status to gdtimer().
 *
 * Returns:
 *    0 - Drive not ready.
 *    1 - Drive ready.
 */
int
gdvs32timer(dev)
register dev_t dev;
{
	register struct gdsw *gds = &gdsw[gdpos(dev)];
	register struct iobuf *gdt = &gdtab[gdctl(dev)];
	register struct iobuf *dp = &gdutab[gdpos(dev)];
	register struct gdvs32ctl *smdp = (struct gdvs32ctl *)dp->io_addr;
	register ushort unit_reg;
	register short drive = gddrive(dev);

	if (!(gds->v_flags & GD_OPENED))
		gdprint(dev, "gdvs32timer called on unopened drive");

	if (!(gdt->b_flags & DT_INUSE)) {
		unit_reg = smdp->unit_status[drive ^ 1];
		if (!(unit_reg & US_PRESENT)) {
			/* Take drive off line */
			return(0);
		}
		if (!(unit_reg & US_DREADY)) {
			/* Take drive off line */
			return(0);
		}

		/* drive is ok */
		return(1);
	}

	/* Controller is busy, we don't know how the drive is */
	return(-1);
}
```

9 INTEGRATING THE DRIVER

This chapter describes the steps you must follow to develop your driver and integrate it into the CTIX operating system. The material is organized into two sections: one for developers who have purchased a CTIX source code license, and another for those who have only a binary license. Each section is complete in itself: you only need to read the section that applies to you.

The chapter also contains information that is useful whether you have a source or a binary license. One section explains how to create the required special files; another contains example **master(4)** file entries.

IF YOU HAVE A SOURCE CODE LICENSE

Use this section if you have purchased a CTIX source code license. It contains all of the information you must have to build and integrate your driver in the CTIX source release environment.

GETTING STARTED

The source files for the CTIX operating system are located in **/usr/src/uts/common** and the subdirectories below it. All references of the form <**directory/filename**> in this section imply that the full pathname is

/usr/src/uts/common/directory/filename.

As you build and integrate your driver, you will be concerned with the following subdirectories of **/usr/src/uts/common**:

cf The configuration files directory. This directory holds the files that customize the generic CTIX operating system for each particular hardware environment. In particular, this directory holds one or more **dfiles**, each of which describes one particular machine configuration.

io The device driver source directory. The source files for all CTIX device drivers are located here. You can develop your driver anywhere, but, when it is time to compile it and link it with the kernel, you must copy the source files into this directory.

sys The kernel header files directory. All of the files referenced by lines of the form **#include <sys/headerfile.h>** are located in this directory. If you have created one or more include files to support your driver, place them in this directory.

INTEGRATING THE DRIVER

In order to get your device driver running, you must compile it, link it with the kernel (unless it is loadable), and create one or more special files to provide access to the device. This subsection describes each of these steps.

Compiling the Driver

Follow these instructions to compile your driver:

1. Develop (or install) the source code for your device driver in **/usr/src/uts/common/io**.
2. Change the definitions of **SRC** and **OBJ** in **<io/Makefile>** to include your driver. For example, if your driver is in **<io/xyz.c>**, you must insert **xyz.c** and **xyz.o** into the definitions for **SRC** and **OBJ** respectively. To be safe, make a copy of the original Makefile before you alter it. Then edit the file, insert the name of your driver, and rewrite the changes.
3. Recreate the Makefile dependency tree by typing **make includes**. This reconstructs **<io/Makeincludes>** to include the dependency tree for your driver.

4. Compile your driver by typing **make**. This will leave the object file for it in the <**io**> directory (assuming there were no compile-time errors).

5. Rebuild the library (**lib2**) so that it includes your driver by typing **make arc**.

Linking the Driver

If your driver is loadable, you don't need to link your driver with the CTIX kernel. Skip this subsection and read the section entitled *Making the Special File(s).*

If your driver is not loadable, you must link it with the rest of the modules in the CTIX kernel. Follow these instructions to link your driver with the kernel:

1. Add a definition line to <**cf/master**> for your device. See **master(4)** for a description of the **master** file. Also see *Some Example Master(4) File Entries,* later in this chapter for specific examples.

2. Add a definition line to <**cf/dfile**> for your device. See **config(1M)** for a complete description of the **dfile** and its contents.

   By convention, **dfile** describes the default "vanilla-flavored" CTIX system: you should not change its contents. Instead, create your own **dfile** with a unique name. To continue with the example above, create <**cf/dfile.xyz**>. The name of the file doesn't matter, but the name of the device does. If you call your new **dfile.xyz** entry **xyz**, then **config(1M)** assumes that the driver entry points are named **xyzopen()**, **xyzclose()**, and so on. You must be consistent with your naming, or you will get "undefined symbol" errors when you attempt to relink the kernel.

3. Change directory to **/usr/sys/cf**.

4. Type **config dfile.xyz**. This runs the configuration program, which creates the files <**cf/conf.c**> and

f3<cf/low.s>.

5. Type **make VER=xyz**. You can set **VER** to anything you like. If you don't specify any value for **VER** (that is, if you just type **make**), it defaults to the value defined in the **cf/Makeflags** file. The resulting object file from the above make command is named **CTIXxyz**.

6. When the make completes (assuming there were no errors), copy the resulting file into the root directory.

7. Bring the system to single-user mode.

8. Remove the existing **/unix** file (if it is a link to some other file. If it is not a link, rename the **/unix** to **/unix.old** or some other nonconflicting name).

9. Link **/unix** to **CTIXxyz** (whatever kernel file resulted from the "make" step above). The new **/unix** is a bootable kernel.

10. Modify the **/etc/system** file to include a line specifying the slot number(s) of your new device, the board type, starting address, and address length, as documented in the *MightyFrame Administrator's Reference Manual.* Make certain that the board type does not conflict with any of the other types defined in <**sys/vme.h**>. It is best (although not required) to place the new board type definition in the header file to help avoid conflicts when other devices are added later.

11. Run **ldeeprom(1M)** to update the VMEbus EEPROM so that your driver can determine its VMEbus address.

IF YOU HAVE A BINARY LICENSE

Use this section if you have not purchased a CTIX source code license. It contains all of the information you must have to build and integrate your driver in the CTIX binary release environment.

GETTING STARTED

The CTIX binary-only release contains two directories of source files that you must have in order to compile and integrate your device driver. These directories are:

/usr/sys/cf The configuration directory. This directory holds the files that customize the generic CTIX operating system for each particular hardware environment. In particular, this directory holds one or more **dfiles**, each of which describes one particular machine configuration.

/usr/include/sys The kernel header files directory. All of the files referenced by lines of the form **#include <sys/headerfile.h>** are located in this directory. If you have created one or more include files to support your driver, place them in this directory.

In addition, you should create the directory **/usr/sys/io** to hold the source code for your driver.

INTEGRATING THE DRIVER

In order to get your device driver running, you must compile it, link it with the kernel (unless it is loadable), and create one or more special files to provide access to the device. This subsection describes each of these steps.

Compiling the Driver

Follow these instructions to compile your driver:

1. Develop (or install) the source code for your device driver in **/usr/sys/io**.
2. Create **/usr/sys/io/Makefile** with the following contents:

```
include ../cf/Makeflags

LIBNAME = ../liblocal

SRC     = xyz.c

OBJ     = xyz.o

all:    $(LIBNAME)

$(LIBNAME): $(OBJ)
        rm -f $(LIBNAME)
        ar qc $(LIBNAME) $(OBJ)
        -chmod 664 $(LIBNAME)
```

3. Type **make** to compile your driver and place it in the library named **/usr/sys/liblocal**.

Linking the Driver

If your driver is loadable, you don't need to link it with the CTIX kernel. Skip this subsection and read *Making the Special File(s)*.

If your driver is not loadable, you must link it with the rest of the modules in the CTIX kernel. Follow these instructions to link your driver with the kernel:

1. Add a definition line to **<cf/master>** for your device. See **master(4)** for a description of the **master** file. Also see *Some Example Master(4) File Entries*, later in this chapter for specific examples.

2. Add a definition line to <**cf/dfile**> for your device. See **config(1M)** for a complete description of the **dfile** and its contents.

   By convention, **dfile** describes the default "vanilla-flavored" CTIX system: you should not change its contents. Instead, create your own **dfile** with a unique name. To continue with the example above, create <**cf/dfile.xyz**>. The name of the file doesn't matter, but the name of the device does. If you call your new **dfile.xyz** entry **xyz**, then **config(1M)** assumes that the driver entry points are named **xyzopen()**, **xyzclose()**, and so on. You must be consistent with your naming, or you will get "undefined symbol" errors when you attempt to relink the kernel.

3. Change directory to **/usr/sys/cf**.

4. Type **config dfile.xyz**. This runs the configuration program, which creates the file <**cf/conf.c**>.

5. Type **make VER=xyz**. You can set **VER** to anything you like. If you don't specify any value for **VER** (that is, if you just type **make**), it defaults to the value defined in the <**cf/Makeflags**> file. The resulting object file from the above make command is named **CTIXxyz**.

6. When the make completes (assuming there were no errors), copy the resulting file into the root directory.

7. Bring the system to single-user mode.

8. Remove the existing **/unix** file, if it is a link to some other file. If it is not a link, rename the **/unix** to **/unix.old** or some other nonconflicting name.

9. Link **/unix** to **CTIXxyz** (whatever kernel file resulted from the "make" step above). The new **/unix** is a bootable kernel.

10. Modify the **/etc/system** file to include a line specifying the slot number(s) of your new device, the board type, starting address, and address length, as documented in the

MightyFrame Administrator's Reference Manual. Make certain that the board type does not conflict with any of the other types defined in <**sys/vme.h**>. It is best (although not required) to place the new board type definition in the header file to help avoid conflicts when other devices are added later.

11. Run **ldeeprom(1M)** to update the VMEbus EEPROM so that your driver can determine its VMEbus address.

MAKING THE SPECIAL FILE(S)

Whether your driver is configured with the kernel or is loadable, you must create one or more special files to provide access to the device. It makes no difference what you call the file, where you locate it, or what access permissions you give it. All that CTIX needs are the major and minor device numbers from the special file's inode. By convention, though, the file is located in **/dev** and is named **xyz**, to match your driver.

To create a special file, follow these instructions:

1. Type **config -t dfile.xyz** to determine the major device number assigned to your device.
2. Use the major device number obtained in the previous step as the parameter to **mknod(1M)** when you create the special files. Assign minor numbers according to the scheme your driver expects.
3. Set the ownership and access permissions on the new special file to provide the appropriate accessibility to your device.

After you run **mknod(1M)**, you are ready to test your driver. Either reboot the system or bind your driver by running **lddrv(1M)**. See Chapter 10, *Debugging the CTIX Kernel,* for more information on how to proceed from this point.

SOME EXAMPLE MASTER(4) FILE ENTRIES

This section contains the **master(4)** file entry for each of the five example drivers in the manual. Each entry is explained in detail.

V/SMD 3200 SMD CONTROLLER

The master(4) file entry for the V/SMD device from Chapter 8, *Block Device Example,* is as follows:

name:	Vsmd3200
mask:	0077
type:	1016
prefix:	gd
block:	0
char:	32
mult:	1
asize:	2
vtype:	2
level:	3

The bits in the **mask** field have the following meanings:

0040 The driver has a power-failure handler.

0020 The driver has a **devopen(2K)** routine (**gdopen(2K)**).

0010 The driver has a **devclose(2K)** routine (**gdclose(2K)**).

0004 The driver has a **devread(2K)** routine (**gdread(2K)**).

0002 The driver has a **devwrite(2K)** routine (**gdwrite(2K)**).

0001 The driver has a **devioctl(2K)** routine (**gdioctl(2K)**).

The bits in the **type** field have the following meanings:

1000 The V/SMD 3200 is a cluster device.

0010 The V/SMD 3200 is a block device.

0004 The V/SMD 3200 is a character device.

0002 The V/SMD 3200 uses a floating interrupt vector.

DR11 PARALLEL INTERFACE

The **master(4)** file entry for the DR11 Parallel Interface is as follows:

name:	dr11
mask:	1136
type:	0406
prefix:	dr11
block:	0
char:	32
mult:	1

The bits in the **mask** field have the following meanings:

1000 The driver is loadable and has a **devrelease(2K)** routine (**dr11release()**).

0100 The driver is loadable and has a **devinit(2K)** routine (**dr11init()**).

0020 The driver has a **devopen(2K)** routine (**dr11open()**).

0010 The driver has a **devclose(2K)** routine (**dr11close()**).

0004 The driver has a **devread(2K)** routine (**dr11read()**).

0002 The driver has a **devwrite(2K)** routine (**dr11write()**).

The bits in the **type** field have the following meanings:

0400 The DR11 is a VMEbus device.

0004 The DR11 is a character device.

0002 The DR11 uses a floating interrupt vector.

SMD - STORAGE MODULE DRIVE DEVICE

The **master(4)** file entry for the SMD device from Chapter 7, *Block I/O Tutorial,* is as follows:

name:	smd
mask:	0037
type:	1416
prefix:	gd
block:	0
char:	32
mult:	1
asize:	2
vtype:	2
level:	3

The bits in the **mask** field have the following meanings:

0020 The driver has a **devopen(2K)** routine (**gdopen(2K)**).

0010 The driver has a **devclose(2K)** routine (**gdclose(2K)**).

0004 The driver has a **devread(2K)** routine (**gdread(2K)**).

0002 The driver has a **devwrite(2K)** routine (**gdwrite(2K)**).

0001 The driver has a **devioctl(2K)** routine (**gdioctl(2K)**).

The bits in the **type** field have the following meanings:

1000 The SMD is a cluster device.

0400 The SMD is a VMEbus device.

0010 The SMD is a block device.

0004 The SMD is a character device.

0002 The SMD uses a floating interrupt vector.

NI - NETWORK INTERFACE DEVICE

The **master(4)** file entry for the NI Parallel Interface is as follows:

name:	netwrk
mask:	1136
type:	0406
prefix:	ni
block:	0
char:	32
mult:	1

The bits in the **mask** field have the following meanings:

1000 The driver is loadable and has a **devrelease(2K)** routine (**nirelease()**).

0100 The driver is loadable and has a **devinit(2K)** routine (**niinit()**).

0020 The driver has a **devopen(2K)** routine (**niopen()**).

0010 The driver has a **devclose(2K)** routine (**niclose()**).

0004 The driver has a **devread(2K)** routine (**niread()**).

0002 The driver has a **devwrite(2K)** routine (**niwrite()**).

The bits in the **type** field have the following meanings:

0400 The NI is a VMEbus device.

0004 The NI is a character device.

0002 The NI uses a floating interrupt vector.

SI - SPEECH INTERFACE DEVICE

The **master(4)** file entry for the Speech Interface Device is as follows:

name:	speech
mask:	1136
type:	0406
prefix:	si
block:	0
char:	32
mult:	1

The bits in the **mask** field have the following meanings:

1000 The driver is loadable and has a **devrelease(2K)** routine (**sirelease()**).

0100 The driver is loadable and has a **devinit(2K)** routine (**siinit()**).

0020 The driver has a **devopen(2K)** routine (**siopen()**).

0010 The driver has a **devclose(2K)** routine (**siclose()**).

0004 The driver has a **devread(2K)** routine (**siread()**).

0002 The driver has a **devwrite(2K)** routine (**siwrite()**).

The bits in the **type** field have the following meanings:

0400 The SI is a VMEbus device.

0004 The SI is a character device.

0002 The SI uses a floating interrupt vector.

10 DEBUGGING THE CTIX KERNEL

This chapter describes the various facilities available for debugging the kernel. The kernel debugger is documented in detail. The **qprintf(2K)** macros are described in general; they are described more fully in Appendix A, *CTIX Interface Manual Pages.* The interactive boot loader is fully documented. Finally, the CTIX debuggers **adb(1)** and **sdb(1)**, and the **crash(1M)** utility are also described briefly. They are documented in the *CTIX Operating System Manual, Volume 1.*

THE KERNEL DEBUGGER

The UNIX kernel has been enhanced by the addition of a full breakpoint and trace debugger under CTIX. This utility provides the capability to

- Single-step the kernel,
- Set and clear breakpoints,
- Examine and modify memory locations,
- Examine and modify registers, and
- Control debugging message output.

You can configure the debugger as part of the kernel load file (**/unix**), or you can use the **lddrv(1M)** utility to load it while the system is running. If the debugger is part of the load file, CTIX transfers control to it before executing the system initialization code. After displaying its banner, the debugger waits about 2 seconds for input from Channel 0. If you type a character in that interval, you will remain in the debugger. If you do not type anything within 2 seconds, the debugger exits, and CTIX continues with its normal initialization sequence.

> **NOTE**
>
> When you want to run with debugging enabled
>
> 1. Reboot the system.
> 2. Hold down any key until the debugger prints its banner.
> 3. Enter the **kd** command to enable the **^B** trap to the debugger.
> 4. Enter the **go** command to continue with the initialization sequence.

If you issue the **sh** command before exiting the debugger at this time, CTIX runs a single-user shell **instead** of running **init(1M)**, ignoring the default run level specified in **/etc/inittab**. See the *CTIX Operating System Manual, Volume 1,* for a complete description of **init(1M)**. Also see Appendix B of the *Mightyframe Administrator's Reference Manual.*

Whether the debugger is configured as part of the load file or is loaded by **lddrv(1M)**, you can transfer control to it at any time by typing **^B** (on a terminal connected to Channel 0 only). This trap is active whether or not any program is reading the keyboard at the time. You can disable the **^B** trap with the debugger's **kd** command. By default, the **^B** trap is disabled if the debugger was linked with the kernel, and enabled if the debugger was loaded with **lddrv(1M)**.

The debugger provides a great deal of control over debugging message output. The following list summarizes the output options.

- You can use the debugger **kp** command to control where the output from kernel **printf(2K)** calls appears. You can route the output to
  - The screen,
  - The printer,
  - The console buffer,
  - The error log file, or
  - Various combinations of these options.
- You can use the debugger **kq** command to control the displaying of selective levels of kernel **printf(2K)** output. To take advantage of this feature, you must use the **qprintf(2K)** macros in your driver to differentiate among various types of debug output.
- You can specify that **printf(2K)** output be paginated, as though it were first piped through the **more(1)** command. This pagination remains in effect for **printf(2K)** output even when you are not in the debugger, so voluminous debugging output won't scroll off the screen.

When output is stopped in page mode (indicated by the ellipsis "..." in the output stream), you can type one of several characters to restart it. The following list describes the options.

- If you press RETURN only, output continues with the next page.
- If you type minus (-) followed by RETURN, output continues with the **previous** page.
- If you type **G** or **g** followed by RETURN (for **GO non-stop**), Page Mode is disabled and output is continuous thereafter. In this case, voluminous debugging output does scroll off the screen.
- If you type **S**, **s**, or **$** followed by RETURN, output to the screen is toggled; that is, it is turned OFF if it was ON, and ON if it was OFF.

- If you type **L** or **l** followed by RETURN, output to the line printer is toggled; again, it is turned OFF if it was ON, and ON if it was OFF.

The debugger makes a distinction between "regular," "temporary," and "automatic" breakpoints. The following list defines the differences:

- You place a **regular** breakpoint when you use the **br** command.
- You place a **temporary** breakpoint when you use the **bx** command.
- The debugger places a **temporary automatic** breakpoint when you use the **to** command.

The following table describes all of the debugger commands and their parameters. Numeric parameters are always ASCII hexadecimal values. The bracket characters [ ] indicate optional parameters. An ellipsis (...) indicates that the parameter can be repeated one or more times.

??	Display the Help menu.
bc [**address**]	Clear the breakpoint at **address**. If **address** is omitted, use the current PC.
bf [**address**]	Set breakpoint at function entry point. If **address** is omitted, use the current PC. The **bf** command adds 4 bytes to the address in order to skip past the LINK instruction located at the entry point of every C-generated function.
bp	Display all current breakpoints in the following format:

ADDRESS	INST	TYPE
AAAAAAAA	IIII	TTTT

AAAAAAAA is the hex address of the breakpoint, **IIII** is the instruction object code at **AAAAAAAA** in hex, and **TTTT** is either **REGULAR**, or **TEMP**. In addition, the keyword **AUTO** is displayed after **TEMP** when the debugger has placed an automatic breakpoint as a result of the **to** command.

br [address] Set a regular breakpoint at **address**. If **address** is omitted, use the current PC.

Every time the breakpoint is encountered, execution is interrupted, and control is returned to the debugger. When the breakpoint is taken and the debugger entered, the instruction at the breakpoint address has not been executed. Regular breakpoints must be cleared explicitly using the **bc** command.

bt [mxframes] Display a stack backtrace consisting of **mxframes** frames. If **mxframes** is not specified, display 16 stack frames. The debugger displays the word **MORE** if there are more (undisplayed) stack frames.

bx [address] Set a temporary breakpoint at **address**. If **address** is not specified, use the current PC contents.

The first time the breakpoint is encountered, execution is interrupted, and control is returned to the debugger. Before giving control to the user, the debugger clears the temporary breakpoint. When the breakpoint is taken and the debugger entered, the instruction at the breakpoint address has not been executed. A temporary breakpoint is taken only once: it is cleared automatically by the debugger.

db address
: Display memory bytes (8 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays 16 bytes in hexadecimal with their ASCII equivalents and waits for input from the keyboard. Enter a RETURN to display the next 16 bytes. Enter a minus sign (-) to display the previous 16 bytes. If you enter anything else, the debugger terminates the **db** command and prompts you to enter the next command.

df
: Display the full register set in hexadecimal.

di [address]
: Display disassembled instructions at **address**. If **address** is omitted, use the current PC contents.

The debugger displays one disassembled instruction and waits for input from the keyboard. Enter a RETURN to display the next instruction. Enter a minus sign (-) to display the previous instruction. If you enter anything else, the debugger terminates the **di** command and prompts you to enter the next command.

dm address
: Display memory longwords (32 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays one longword in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next longword. Enter a minus sign (-) to display the previous longword. If you enter anything else, the debugger terminates the **dm** command and prompts you to enter the next command.

dr

Display the registers one at a time in the following order: D0-D7, A0-A7, Status Register, Program Counter, Interrupt Stack Pointer, Master Stack Pointer, Cache Control Register, Cache Address Register, Vector Base Register, Source Function Code, and Destination Function Code.

The debugger displays one register in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next register. If you enter anything else, the debugger terminates the **dr** command and prompts you to enter the next command.

dw address

Display memory words (16 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays one word in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next word. Enter a minus sign (-) to display the previous word. If you enter anything else, the debugger terminates the **dw** command and prompts you to enter the next command.

go [address]

Resume execution. If **address** is specified, place its value into the program counter before resuming execution. This command takes you out of the debugger.

he Display the Help menu.

kc Enable/disable the 68020 cache. This command toggles the current state of the microprocessor's instruction cache.

kd Enable/disable kernel **^B** entry to the debugger. This command toggles the state of the **^B** debugger trap. When the trap is enabled, typing **^B** on the terminal connected to Channel 0 transfers execution to the debugger. When the trap is disabled, **^B** is ignored.

kl [l] List the kernel trace buffer. If **l** (lowercase L) is specified, list the contents of the kernel buffer on the printer. Otherwise, list the buffer contents on the screen in page mode.

kp Enable/disable kernel **qprintf(2K)** calls. This command advances the state of the **kpflg** variable in the kernel. The states are

Disabled.

Enabled - route data to screen.

Enabled - route data to printer.

Enabled - route data to screen and printer.

Enabled - route data to memory log.

Enabled - route data to logfile.

Enabled - route data to both logfile and screen.

The **kp** command wraps around to the Disabled state after the last Enabled state. See **qprintf(2K)** for a discussion of the usage of the kernel **kpflg**.

kq [**lvl ...**] Enable/disable selective kernel debug levels.

The **kq** command toggles the bits in the **kqflg** variable, thus enabling or disabling one or more selective debug levels in the kernel. Acceptable values for **lvl** are lowercase **a** though **z** and the characters {, |, }, and ~. These values toggle the state of the bits examined by **aprintf()** through **eeprintf()**. By specifying multiple parameters on the command line, you can enable and disable multiple levels at once.

See **qprintf(2K)** for a complete description of the selective debug facility, including the correspondence between **kq** command parameters and debug levels.

mb address Modify memory bytes (8-bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays 1 byte in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next byte. Enter a minus sign (-) to display the previous byte. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **mb** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the low-order 8 bits of the value to **address** and then begins again from the top, redisplaying the new value and waiting for input.

mm address Modify memory longwords (32 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays one longword in hexadecimal and then waits for input from the keyboard. Enter a RETURN only to display the next longword. Enter a minus sign (-) to display the previous longword. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **mm** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the full 32-bit value to **address** and then begins again from the top, redisplaying the new value and waiting for input.

mr Modify the registers one at a time in the following order: D0-D7, A0-A7, Status Register, Program Counter, Interrupt Stack Pointer, Master Stack Pointer, Cache Control Register, Cache Address Register, Vector Base Register, Source Function Code, and Destination Function Code.

The debugger displays one register in hexadecimal and then waits for input from the keyboard. Enter a RETURN only to display the next register. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **mr** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the full 32-bit value to the current register and then begins again from the top, redisplaying the new value of the current register and waiting for input.

mw address Modify memory words (16 bits) starting at **address**. You must specify **address**; there is no default. If **address** is odd, it is rounded down to the nearest 16-bit boundary.

The debugger displays one word in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next word. Enter a minus sign (-) to display the previous word. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **mw** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the low-order 16 bits of the value to **address** and then begins again from the top, redisplaying the new value and waiting for input.

pm Enable/disable page mode output. This command toggles the current state of the page mode output flag. If the flag is enabled, the kernel lists one screenful of information, displays the ellipsis characters (...), and then halts, waiting for input from the keyboard. This is true for any kernel output, whether or not the debugger is currently active. When page mode is disabled, the kernel lists continuous data. Page mode is similar to piping debugging output through the **more(1)** command.

re [-]
Reboot CTIX. This command is equivalent to pressing the RESET button. The disks are not **sync**ed, processes are not halted, and the normal shutdown process is bypassed completely. If the hyphen is present, CTIX will not perform a dump before shutdown. Otherwise, a normal system dump is taken.

sh
Invoke single user shell. If the debugger is configured into the kernel (not loaded as a result of executing **lddrv(1M)**), it runs before CTIX initialization is performed. If you issue a **sh** command before you exit the debugger the first time, CTIX brings up a single user shell **instead** of running **init(1M)**. At any time other than immediately after reboot, the **sh** command does nothing.

to
Trace over a JSR instruction. If the program counter points to a JSR instruction, the debugger places a temporary automatic breakpoint at the instruction after it and resumes execution. If the current instruction is not a JSR, the debugger enters normal trace mode (as though you had entered a **tr** command).

tr
Trace instruction execution; that is, single-step the CPU.

The debugger accomplishes this by setting the **T1** bit in the Program Status register. See the *MC68020 32-bit Microprocessor User's Manual* for more information.

tt Trace change of instruction flow; that is, allow execution until a BRA, JSR, etc., instruction is executed.

The debugger accomplishes this by setting the **T0** bit in the Program Status register. See the *MC68020 32-bit Microprocessor User's Manual* for more information.

wb address Write (without prereading) memory bytes (8 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays the address in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next address. Enter a minus sign (-) to display the previous address. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **wb** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the low-order 8 bits of the value to **address** and then begins again from the top, redisplaying the address and waiting for input.

wm address Write (without prereading) memory longwords (32 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays the address in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next address. Enter a minus sign (-) to display the previous address. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **wm** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the full 32-bit value to **address** and then begins again from the top, redisplaying the address and waiting for input.

ww address Write (without prereading) memory words (16 bits) starting at **address**. You must specify **address**; there is no default.

The debugger displays the address in hexadecimal and then waits for input from the keyboard. Enter a RETURN to display the next address. Enter a minus sign (-) to display the previous address. Otherwise, if you enter anything other than a valid hexadecimal number, the debugger terminates the **ww** command and prompts you to enter the next command.

If you enter a hex number, the debugger writes the low-order 16 bits of the value to **address** and then begins again from the top, redisplaying the address and waiting for input.

QPRINTF(2K) MACROS

The header file <**sys/kprintf.h**> contains a number of macro definitions that are useful in debugging a device driver. Each of these macros is of the form:

#define Qprintf (kpflg&&kqflg&(N<<0))&&printf

where **Q** is one or two letters between **a** and **ff**, and N is a number between 0 and 30.

The macro definition reads like this: "If **kpflg** (the kernel print flag) is nonzero, and if the Nth bit is set in **kqflg**, then call the **printf(2K)** function with the arguments specified with the macro."

The **qprintf(2K)** macros allow both gross and fine control of debugging output. You can disable output altogether by clearing **kpflg**, or you can selectively enable and disable output by setting **kpflg** and one or more of the bits in **kqflg**. The kernel debugger commands **kp** and **kq** are provided to manage these variables.

These macros are documented under **qprintf(2K)** in Appendix A, *CTIX Interface Manual Pages.*

INTERACTIVE BOOT LOADER

The boot loader is the program that is written into the loader area of the disk by **iv(1M)**. It is not the **ld(1)** program, which links object modules together into a runnable process. The boot loader has the responsibility of loading the CTIX operating system at bootstrap time.

CTIX software provides the capability to substitute an interactive boot loader in place of the loader normally supplied with the operating system. This interactive loader allows you to boot from an alternate load file, instead of **/unix**, which is the normal default. This capability is very useful when you are debugging a new device driver.

To install the interactive loader, you must alter the description file and run the **iv(1M)** program. The pathname of the interactive loader is (currently) **/usr/lib/iv/loader11cust**. You must substitute this pathname on the **loader** line of the description file. After you alter the description file, run the **iv(1M)** program to write the interactive loader onto the loader area of the boot disk. Until you change the description file and run **iv(1M)** again, the interactive loader will always run instead of the noninteractive loader.

When you reboot the the system, the interactive loader carries on the following dialog.

- The loader displays its banner line **Mightyframe Loader Version 11**.
- The loader then prompts **Do you to boot anything other than the default?**
- If you respond by typing **n**, the loader searches for and boots from **/unix**.
- If you respond by typing **y**, the loader displays **Select device to load from (0-2=Onboard Disks, T=Tape, 4-7=VME disks)**.
- You must select one of the displayed load devices. After you have entered a valid choice, the loader prompts **Enter filename from which to load.**
- If you enter a directory name, the loader lists the files within the named directory and then starts over, from its banner line.
- If you enter the name of a nonexistent file, the loader displays **Error: no such file, try again**, and then starts over from its banner line.
- If you enter the name of a file within slice one of the named load device, the loader boots that file.

OTHER KERNEL DEBUGGING TOOLS

You can use the **adb(1)** and **sdb(1)** debuggers on the CTIX kernel, but they are much less useful than the kernel debugger. You cannot set breakpoints or trace instruction execution with these programs, since they execute as user processes. They do work well for looking quickly at the state of the kernel or your driver when you do not wish to load the kernel debugger. They also are useful when the system has crashed, and you need to examine a dump.

The **crash(1M)** utility is an invaluable tool for examining a CTIX system image. You can use **crash(1M)** to display every major kernel table, including the linked lists of buffer headers. Like the debuggers, **crash(1M)** works either on a running system, or on a core dump.

See the *CTIX Operating System Manual, Volume 1,* for complete documentation of these utilities.

APPENDIX A: CTIX INTERFACE MANUAL PAGES

INTRODUCTION

This appendix describes all of the kernel calls available to support a device driver. The functions are suffixed with the characters **2K** to indicate that they are system calls of a sort but that they are callable only from within the kernel. There is no direct, user-level access to any of the routines documented in this appendix.

There are three types of routines documented in this appendix:

- Routines that you write as part of your device driver: they begin with the letters **dev**.
- Kernel routines that form part of the general disk driver: they begin with the letters **gd**.
- Other kernel routines that perform specific functions for your device driver. This category includes all of the routines that do not begin with **dev** or **gd**.

Routines that begin with the letters **dev** such as **devread(2K)** and **devclose(2K)** are part of the device driver. When you write your driver, you must substitute the name of your device for the **dev** prefix. For example, the **devread()** routine will be called **dr11read()** if your device is a DR11 parallel interface board: the **devclose()** routine will be **gdvs32close()** if your device is an Interphase V/SMD 3200 Disk Controller, and so on.

KERNEL INTERFACE TO DEVICE DRIVERS

To simplify the task of supporting new types of hardware, the designers of UNIX eliminated all kernel calls directly to the device drivers. In place of direct calls, UNIX and CTIX provide three arrays that describe the device driver entry points. These arrays are

cdevsw The character device switch, which contains the addresses of the entry points for character devices.

bdevsw The block device switch, which contains the addresses of the entry points for block devices.

gdsw The general disk device switch, which contains the addresses of the entry points for disk-like devices.

The declarations for these data structures, which are contained in the header files <**sys/conf.h**> and <**sys/gdisk.h**>, are included below. The vertical dots indicate that lines from the header file have been omitted here. The following code fragment may differ from the include files on your system. In all cases, the files in the latest CTIX release supercede this document.

```
struct cdevsw {
    int  (*d_open)();   /* devopen(2K) routine */
    int  (*d_close)();  /* devclose(2K) routine */
    int  (*d_read)();   /* devread(2K) routine */
    int  (*d_write)();  /* devwrite(2K) routine */
    int  (*d_ioctl)();  /* devioctl(2K) routine */
    struct tty *d_ttys;
};
```

Character Device Switch

```
struct bdevsw {
     int  (*d_open)();     /* devopen(2K)/gdopen(2K) routine */
     int  (*d_close)();    /* devclose(2K)/gdclose(2K) routine */
     int  (*d_strategy)();/* devstrategy(2K)/gdstrategy(2K) routine */
     int  (*d_print)();    /* devprint(2K)/gdprint(2K) routine */
};
```

Block Device Switch

```
struct gdsw {
     /* The fields through "dsk2" are initialized in gdtab.h */
     int  (*intr)();                /* devintr(2K) routine */
     int  (*start)();               /* devstart(2K) routine */
     int  (*open)();                /* devopen(2K) routine */
     int  (*timer)();               /* devtimer(2K) routine */
     short      *bbq;               /* bad block cylinder index */
     struct     bbmcell *bb;        /* bad block table */
     ushort     szbbq;              /* size of bad block cylinder index */
     ushort     szbb;               /* size of bad block table */
     ushort     DMAto;              /* max duration of a disk op */
     short      ctlr;               /* Controller type (see gdioctl.h) */
     struct     gdswprt dsk;        /* disk specific information */
     struct     gdswprt2 dsk2;/* More disk specific info */

     /* The following fields are NOT initialized in gdtab.h */
     .
     .
     .
};
```

General Disk Switch

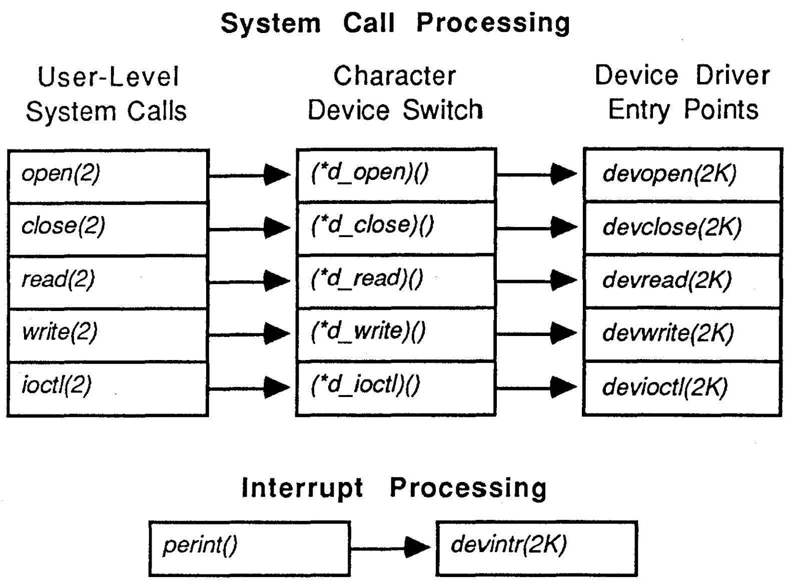
The **cdevsw** and **bdevsw** structures are defined and initialized in the file **<cf/conf.h>**. The **gdsw** structure is itself a member of the **gddefault** structure, which is defined in **<sys/gdisk.h>**. This structure (including **gdsw**) is defined and initialized in **<sys/gdtab.h>**.

In order to add a new character device driver to the kernel, you must

- Insert the addresses of the driver's **devopen(2K)**, **devclose(2K)**, **devread(2K)**, **devwrite(2K)**, and **devioctl(2K)** functions into the **cdevsw** array.
- Run the **mknod(1)** program to create a character special file with the correct major and minor device number.

For character devices, the major device number serves as the index into the **cdevsw** array; the minor device number is used by the driver for whatever it needs. Frequently, the minor device number serves to differentiate among various common devices, but it can also indicate such things as tape density, rewind/no rewind, disk partition (slice), and so on.

The following diagram illustrates the linkage mechanism between the kernel and the character device drivers.



Kernel/Device Driver Linkage
Character Devices

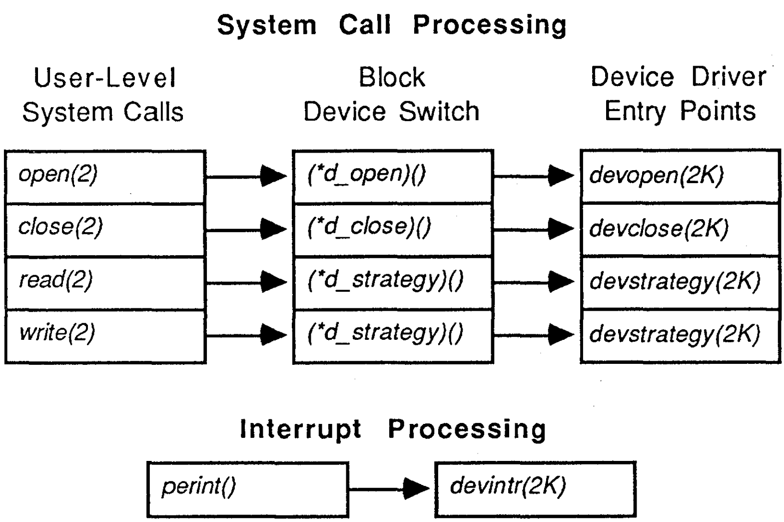
In order to add a new block device driver to the kernel, you must follow almost exactly the same steps outlined above, that is,

- Insert the addresses of the driver's **devopen(2K)**, **devclose(2K)**, **devstrategy(2K)**, **and devprint(2K)** functions into the **bdevsw** array.

- Run the **mknod(1)** program to create a block special file with the correct major and minor device number.

For block devices, the major device number serves as the index into the **bdevsw** array; the minor device number is used by the driver for whatever it needs. The minor device number can be used to differentiate among such things as the channels on a controller, the tape density, whether or not a tape should be rewound, and the disk slice number.

The following diagram illustrates the linkage mechanism between the kernel and the block device drivers.



Kernel/Device Driver Linkage
Block Devices

If your device is (or acts like) a disk drive, it should be treated as part of the general disk driver. (See the next section for details.)

GENERAL DISK-TYPE DEVICES

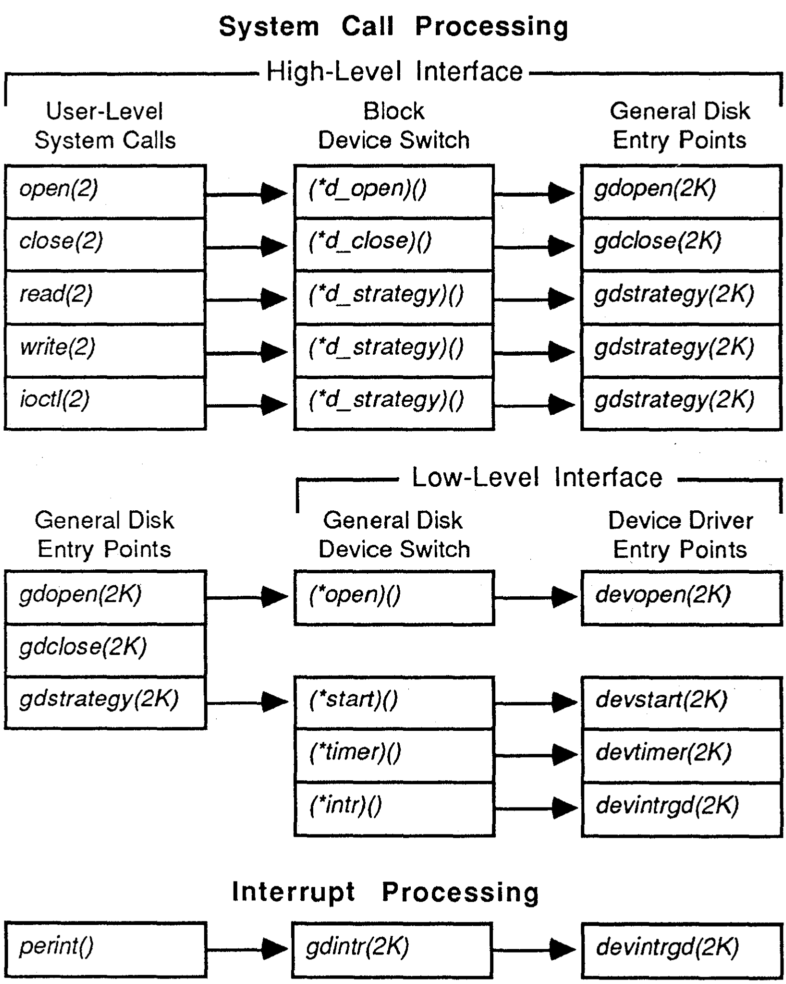
Drivers for disk-like devices are divided into two separate sections within the CTIX kernel: a high-level, device-independent portion, and a low-level, device-specific portion.

The high-level interface exists because much of the code needed to support disk-like devices can be shared among their drivers. This device-independent portion of the driver is called the general disk driver: it includes all of the kernel routines with names that begin with the letters **gd**. For example, **gdstrategy(2K)** and **gdclose(2K)** are part of the general disk driver.

The low-level interface is the actual device driver. It is responsible for issuing the I/O commands to the device and for determining the resulting status. This is the portion of the device driver that you must write yourself.

The linkage mechanism between the kernel and the general disk driver is similar to, but more complex than, the interface for block devices. For general disk-type devices, the **bdevsw** table does not contain the address of the low-level driver's **devstrategy(2K)** routine. Instead, **bdevsw** is set up to point to the addresses of the routines in the general disk driver. After **gdstrategy(2K)** performs all of the device independent work, it calls your device driver's **devio(2K)** routine to perform the actual transfer. The general disk driver uses the **gdsw** array to make the linkage with your driver's entry points.

The following diagram illustrates the linkage mechanism between the kernel, the general disk driver, and the low-level disk drivers.



Kernel/Device Driver Linkage
General Disk-Type Devices

The diagram shows both the high- and low-level interfaces for general disk-type devices. The general disk entry points are shown in two places for continuity. When a user makes a request for service from a general disk-type device, the kernel uses the **bdevsw** as usual to get to the device-independent general disk driver.

The high-level general disk driver performs as much of the work as possible and then calls the low-level disk driver through the **gdsw** table. The device-independent code uses the **gdpos()** macro (defined in <**sys/gdisk.h**>) with the major + minor device number to generate an index into the table. The indirect call results in a transfer to the low-level driver's **dev** routines. You must write these device-dependent routines to perform the physical transfers to and from your device.

Along with certain other driver parameters, the **gdsw** table contains entries for the addresses of your driver's **devintrgd(2K)**, **devstart(2K)**, **devopen(2K)**, and **devtimer(2K)** routines. These are the system interface points for disk-like device drivers. **Gdsw** is contained within another structure named **gddefault**, which is declared in <**sys/gdisk.h**>. **Gddefault** is defined and initialized in <**sys/gdtab.h**>.

NOTE

At the present time, **lddrv(1)** does not support drivers for disk-like devices. You must modify the **gdsw** structure yourself and link your disk driver directly with the kernel. (**Gdsw** is contained within the **gddefault** data structure, which is declared in <**sys/gdisk.h**> and defined in <**sys/gdtab.h**>.)

BUFFER HEADER STRUCTURE

A portion of the header file <**sys/buf.h**> is included below. The most important fields are documented. The vertical dots indicate that lines from the header file have been omitted here. The following code fragment may differ from the include files on your system. In all cases, the files in the latest CTIX release supercede this document.

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

/*
 * The buffer header structure.
 *
 * Each buffer in the pool is usually doubly linked into 2 lists:
 *   - the device it is currently associated with (always)
 *   - the list of blocks available for allocation (usually)
 *
 * A buffer is on the available list and is liable to be reassigned
 * to another disk block if and only if the B_BUSY flag is not set.
 * When a buffer is busy, the available-list pointers can be used for
 * other purposes.
 *
 * Most drivers use the forward ptr as a link in their I/O queue.
 *
 * A buffer header contains all the information needed to perform I/O.
 */

struct buf {
    int       b_flags;         /* see defines below */
    struct    buf *b_forw;     /* position on drive queue */
    struct    buf *b_back;     /* position on drive queue */
    struct    buf *av_forw;    /* position on free list, */
    struct    buf *av_back;    /*  if not B_BUSY */
    dev_t     b_dev;           /* major+minor device name */
    •
    •
    •
    unsigned b_bcount;         /* transfer count */
    union {
        caddr_t b_addr;        /* buffer address */
        •
        •
        •
```

```
    } b_un;

#define   paddr(X) (paddr_t)(X->b_un.b_addr)

    daddr_t  b_blkno;        /* block # on device */
    char     b_error;        /* returned after I/O */
    unsigned int b_resid;    /* bytes not transferred after error */
    •
    •
    •
#define   b_errcnt b_resid   /* while i/o in progress: # retries */
};

/*
 * These flags are kept in b_flags.
 */
#define   B_WRITE    0x0000    /* non-read pseudo-flag */
#define   B_READ     0x0001    /* read when I/O occurs */
#define   B_DONE     0x0002    /* transaction finished */
#define   B_ERROR    0x0004    /* transaction aborted */
#define   B_BUSY     0x0008    /* not on av_forw/back list */
    •
    •
    •
#define   B_WANTED   0x0040    /* issue wakeup when B_BUSY goes off */
    •
    •
    •
#define   B_FORMAT   0x800000 /* perform a format operation */
    •
    •
    •
```

The following is a brief discussion of the meaning and usage of the most important fields in the structure.

b_flags indicates the state of the buffer. One or more of the following bits can be set.

B_WRITE is not really a flag at all. It indicates the absence of the **B_READ** flag. You should test for a WRITE request by saying **if (!bp->b_flags & B_READ)**.

B_READ indicates that the buffer header describes a READ request.

B_DONE indicates that the I/O request specified by the buffer header is finished. There may or may not have been an error in the transfer. You should call **iodone(2K)** to set this flag and wake up the process(es) sleeping on the buffer.

B_ERROR indicates that an error occurred on the transfer. The **b_error** field contains more information when this flag is set.

B_BUSY indicates that the buffer is not on the queue of available buffers; that is, the buffer is in use, describing an I/O request.

B_WANTED indicates that some other process wants to use the buffer when its current I/O request is complete. **Iodone()** sets the **B_DONE** bit and then, if **B_WANTED** is set, it calls **wakeup(2K)** to awaken the process(es) waiting for the buffer.

B_FORMAT indicates that the buffer describes a FORMAT command to a disk-like device.

b_forw Most of the time, each buffer header is on two separate queues: the queue of all buffers available for (re-)use, and the queue of all buffers containing data associated with the same device (the drive queue). The **b_forw/b_back** pair contains the pointers used to maintain the doubly linked list of buffers associated with the same device. When CTIX receives a READ request, it hashes the block and device numbers and uses the resulting value as an index into the system hash list. The selected hash slot points to a (possibly empty) linked list of buffer headers whose block and device numbers hashed to the same value.

CTIX searches this list to see if the requested block is present in memory already. If so, there is no need to access the device.

av_forw The **av_forw/av_back** pair contains the pointers used to maintain the doubly linked list of available buffers. The only time a buffer is **not** on the available list is when the **B_BUSY** bit is set, that is, between the time that CTIX sets up the buffer to describe an I/O request and the time that **devintr()** (or **gdintr()**) calls **iodone()**. This means that buffers are available even when they contain valid data. CTIX maintains the available list in LRU order so that the valid data will be available as long as possible before the buffer is reused. In fact, for most writes, the data is not actually written to the device until the buffer is reused for some other (unrelated) I/O.

b_dev Contains the major+minor device number of the device containing the data to be read or written. The major number is used as an index into the **bdevsw** or **cdevsw** tables. The minor number is used by the driver for its own purposes. Typically, it contains the unit number, which may indicate the controller or the slice (partition) that is being referenced.

b_bcount Contains the number of bytes in the buffer, or the transfer length in bytes.

b_un Is a union describing the pointer to the data area of the buffer. Most commonly, **b_addr** contains the virtual address where the data resides (or will reside).

b_blkno Contains the block number of the data that the buffer contains (or will contain when the I/O request is done).

b_error Contains the error number to be placed into **u.u_error** if the **B_ERROR** bit is set in **b_flags**. The macro **geterror()** in **<sys/buf.h>** sets

u.u_error for you. **Iowait(2K)** calls this macro after I/O is complete. If your driver sets the **B_ERROR** bit but does not set the **b_error** field, **geterror()** sets **u.u_error** to **EIO**.

b_resid Contains the transfer residue after an error occurs, that is, the number of bytes from the original I/O request that were **not** transferred. Normally, it is zero, indicating that no errors occurred. However, you should not use it to determine whether the I/O failed.

USER STRUCTURE

A portion of the header file <**sys/user.h**> is included below. The most important fields are documented. The vertical dots indicate that lines from the header file have been omitted here. The following code fragment may differ from the include files on your system. In all cases, the files in the latest CTIX release supercede this document.

```
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/inode.h>
#include <sys/file.h>
#include <sys/signal.h>
#include <sys/dir.h>

/*
 * The user structure.
 *
 * There is one user structure allocated per process. It is
 * swapped out with the process. It contains all per process
 * data that isn't referenced while the process is swapped. It
 * contains the per-user system stack, used during system
 * calls. It is cross referenced with the proc structure for
 * the same process.
 */

struct    user
{
     •
     •
     •
     struct proc *u_procp;     /* pointer to proc structure */
     •
     •
     •
     char u_error;             /* syscall error code */
     •
     •
     •
     union {                   /* syscall return values */
          struct {
               int  r_val1;
               int  r_val2;
          } r_reg;
          •
          •
          •
     } u_r;
     caddr_t   u_base;         /* base address for I/O */
     unsigned u_count;         /* bytes remaining for I/O */
     union {
          off_t ow_offset;          /* offset in file for I/O */
          •
```

```
        •
        •
    } u_ow;
    •
    •
    •
    short     u_fmode;     /* file mode for I/O */
    ushort    u_pbsize;    /* bytes in block for I/O */
    ushort    u_pboff;     /* offset in block for I/O */
    dev_t     u_pbdev;     /* real device for I/O */
    •
    •
    •
    int  u_arg[10];        /* syscall arguments */
    •
    •
    •
};
    •
    •
    •
```

NAME

bcopy – copy data as efficiently as possible

SYNOPSIS

bcopy(from, to, nbytes)
char *from, *to;
unsigned int nbytes;

DESCRIPTION

Bcopy() copies **nbytes** of data from the **from** address to the **to** address. The routine is optimized for the particular CPU to do its work as efficiently as possible.

Either or both of the source or destination buffers can be in user space; however, **bcopy()** does not verify their accessibility before attempting the transfer. For instance, **copyin(2K)** and **copyout(2K)** call **bcopy()** to perform their data transfers, after they have called **useracc**(2K) to check the accessibility of the destination buffer.

RETURN VALUE

Bcopy() does not return a value.

SEE ALSO

copyin(2K), copyout(2K), useracc(2K).

NAME

ccopyin – copy data from user space to VMEbus EEPROM

SYNOPSIS

ccopyin(from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Ccopyin() copies **nbytes** of data from the **from** address (which may be in user space) to the **to** address in the VMEbus EEPROM, and waits for the EEPROM to accept the data.

It first calls **useracc(2K)** to verify that the user has read permission on the data. Then it calls **probevme(2K)** to verify that the VMEbus address is valid. Next, **ccopyin()** performs the physical copy, sleeping for at least 16 milliseconds between each byte. The EEPROM requires at least 10 milliseconds after each write to capture the data. Finally, **ccopyin()** verifies that the data was accepted by the EEPROM by attempting to read it back. If it was not captured, **ccopyin()** attempts the write once more. If the data still has not been captured, an error is returned.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and **u.u_error** is set as follows:

[EFAULT]	Either the user does not have read permission on the entire buffer, or a read access at the VMEbus address causes a bus fault.
[EIO]	The data was not captured by the EEPROM.

SEE ALSO

copyin(2K), copyout(2K), scopyin(2K), scopyout(2K). Chapter 2, *Architectural Information.*

NOTE

Since **ccopyin()** calls **sleep(2K)**, it should not be called from the interrupt level.

NAME

chkbusflt – check validity of address

SYNOPSIS

int chkbusflt(address, flag)
int *address;
int flag;

DESCRIPTION

Chkbusflt() checks to see whether a read or write access to **address** causes a bus fault. If the value of **flag** is zero, **chkbusflt()** attempts to read a byte of data at **address**. Otherwise, **chkbusflt()** attempts to read and then rewrite a byte of data at **address**.

If the access causes a bus fault, it is caught by this routine, and the normal bus fault handler is not invoked.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of 1 is returned.

SEE ALSO

probevme(2K).

NAME

copyin – copy data from user space to kernel space

SYNOPSIS

copyin(from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Copyin() copies data from user space to kernel space. It first calls **useracc(2K)** to verify that the user has read permission at the **from** address for **nbytes**. Then it calls **bcopy(2K)** to perform the physical copy.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

SEE ALSO

bcopy(2K), copyout(2K), useracc(2K).

NAME

copyout – copy data from kernel space to user space

SYNOPSIS

copyout(from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION

Copyout() copies data from kernel space to user space. It first calls **useracc(2K)** to verify that the user has write permission at the **to** address for **nbytes**. Then it calls **bcopy(2K)** to perform the physical copy.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

SEE ALSO

bcopy(2K), copyin(2K), useracc(2K).

NAME

delay – give up the processor for a time

SYNOPSIS

delay(count)
int count;

ldelay(count)
int count;

pdelay(count)
int count;

DESCRIPTION

Delay() gives up the CPU for a minimum of **count** ticks of the system clock. The frequency of the system clock can be determined from the **HZ** constant, defined in <**sys/param.h**>. A value of 60 for **HZ** indicates a system clock frequency of 60 ticks per second: this yields a tick duration of approximately 16.67 ms.

Delay() first calls **timeout(2K)** with the **count** parameter, and then calls **sleep(2K)** to relinquish the CPU.

If **count** is less than or equal to zero, **delay()** returns immediately.

Ldelay() delays for approximately **count** milliseconds before returning. It is implemented in assembly language as a series of calls to **pdelay()**. It does not call either **timeout()** or **sleep()**

Pdelay() delays for a minimum of 2 microseconds (with a **count** of zero) before returning. Thereafter, each count adds about 250 nanoseconds. Thus, a count of 4 delays for about 3 microseconds; a count of 8 delays about 4 microseconds, and so on. **Pdelay()** is implemented in assembly language as a do-nothing loop. It does not call either **timeout()** or **sleep()**

RETURN VALUE

None of **delay()**, **ldelay()**, or **pdelay()** returns a value.

SEE ALSO

sleep(2K), timeout(2K).

NOTE

Since **delay()** calls **sleep()**, it should not be called from the interrupt level.

Since **delay()** sleeps at a priority of **PZERO** - 1, the process cannot be interrupted by a signal.

Ldelay(), **pdelay()** and **sdelay()** stop all other processor activity while they run. Therefore, you should use them with care.

NAME

devclose – **character and block** device close routine

SYNOPSIS

#include <sys/types.h>

devclose(dev, flag)
dev_t dev;
int flag;

DESCRIPTION

Devclose() is the generic name for a character or block device driver's close routine. If the device is an XYZ, for instance, the actual name would be **xyzclose()**.

Devclose() is called by CTIX when the last process that had the device open issues a **close(2)**.

Dev is the minor device number of the device.

Flag is the value of the **f_flag** field in the file table structure (see **<sys/file.h>** for a complete definition of its contents).

It is the responsibility of the **devclose()** routine to clean up after the device, perhaps disabling interrupt(s), and cancelling any outstanding **timeout(2K)** calls.

RETURN VALUE

Devclose() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

gdclose(2K).

NOTE

CTIX calls **devclose()** only when the device is closed for the **last** time. This is in contrast to **devopen()**, which is called for **every** open on the device.

Gdclose(2K) does not call **devclose()** for block devices that are part of the general disk driver.

NAME

devinit – device driver initialization routine

SYNOPSIS

devinit()

DESCRIPTION

Devinit() is the generic name of the initialization routine for both loadable and configured-in device drivers. If the device is an XYZ, for instance, the actual name would be **xyzinit()**.

For configured-in drivers, **devinit()** is called through the **dev_init** table in **<sys/conf.h>**, if the device was described in the **master** file. For loadable drivers, it is called as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVBIND**. This **syslocal(2)** call is made by the **lddrv(1M)** program.

For VMEbus devices, it is the responsibility of the **devinit()** routine to verify the existence of both the VMEbus interface board and the device that the driver controls. The CTIX kernel sets the external integer variable **haveVME** to nonzero to indicate that the VMEbus interface card is present. Then you can call **probevme(2K)** with the controller address to determine whether your device (actually, **any** responding device) is present.

A more robust test is to call **is_eepromvalid(2K)**, which verifies that the interface board is present and that the checksum in its EEPROM is valid. Normally, the EEPROM will contain information about the device, including the VMEbus address of the controller. You should then call **probevme(2K)** with the controller address to verify that the device is present. The device driver in Chapter 8, *Block Device Example*, contains code that performs these tests.

Devinit() also must make certain that the driver has not been initialized previously. Finally, it should allocate any kernel virtual address space required by the driver through a call to **sptalloc(2K)**, set up the interrupt handler address through a call to **get_vec(2K)** or **set_vec(2K)**, and initialize the hardware.

RETURN VALUE

Devinit() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

lddrv(1M), get_vec(2K), is_eepromvalid(2K), probevme(2K), set_vec(2K), syslocal(2), drivers(7).

NAME

devintr – character and block device interrupt handler

SYNOPSIS

devintr(vecnbr)
short vecnbr;

DESCRIPTION

Devintr() is the generic name for a character or block device driver's interrupt handler. If the device is an XYZ, for instance, the actual name would be **xyzintr()**. If the device is part of the general disk system, the interrupt handler is described under **devintrgd(2K)**.

CTIX calls **Devintr()** when it receives an interrupt with a vector number associated with this device.

Vecnbr is the interrupt vector number supplied by the device when its interrupt was acknowledged.

Typically, **devintr()** removes the source of the interrupt, checking for any error conditions on the device.

For normal character and block devices using buffered I/O, if the original transfer request is complete, the interrupt handler calls **iodone(2K)** to complete the buffer and wake up any process sleeping on it.

If there is more I/O to perform on the current device, **devintr()** starts the next transfer.

RETURN VALUE

Devintr() does not return a value directly. Rather (for buffered I/O), all information about the status of the I/O is returned in the buffer header. If there was no error on the transfer, it sets **bp->b_bcount**, **bp->b_resid**, and **bp->b_error** to zero. If there was an error, it sets the **B_ERROR** bit in **bp->b_flags**, and **bp->b_error** to indicate the cause of the error. Normally, this should be set to **EIO**, since CTIX doesn't (yet) provide for more specific I/O errors. It also sets **bp->b_resid** (buffer residue) to the number of bytes of the original request that were **not** transferred, due to the error.

Whether or not there was an error, it decrements **bp->b_bcount** and increments **bp->b_un.b_addr** by the number of bytes actually transferred.

SEE ALSO

devintrgd(2K), gdintr(2K), iodone(2K), disk(7).

NOTE

Devintrgd() is the interrupt handler for general disk-type devices.

NAME

devintrgd – general disk-type device interrupt handler

SYNOPSIS

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

devintrgd(bp, minor_dev, vecnbr)
struct buf *bp;
dev_t minor_dev;
int vecnbr;

DESCRIPTION

Devintrgd() is the generic name for a general disk-type device's low-level interrupt handler. If the device is an XYZ, for instance, the actual name would be **xyzintr()**.

When an interrupt is received from a general disk device, CTIX first calls **gdintr(2K)** to perform device independent processing. **Gdintr(2K)** then calls **devintrgd()** to perform device dependent processing.

Bp is a pointer to the buffer header structure associated with the IO in progress on the device. The **gdintr(2K)** routine calculates the buffer pointer in the following manner: first, using the vector number, it determines the controller number of the interrupt device. Next, it uses the controller number to determine the head of the associated drive queue. Finally, it takes the buffer pointer from the head of the drive queue.

Minor_dev is the minor device number of the interrupting device.

Vecnbr is the interrupt vector number supplied by the device when its interrupt was acknowledged.

General disk-type device interrupt handlers must differentiate between **I/O requests** (which usually occur as the result of a user program request to read or write data) and **I/O operations** (which are low-level commands issued directly to the controller). It is common for disk I/O requests to require several operations: for instance, a SEEK command to position the read/write head over the correct cylinder, followed by a READ/WRITE command (or several commands, if the requested transfer crosses a track or cylinder boundary). Also, if any given operation fails, a robust driver will retry the operation a number of times before declaring a hard failure on the device.

Interrupts are received at the completion of each I/O operation. Typically, the **devintrgd()** routine removes the source of the interrupt (handling any error conditions on the controller), and then starts the next I/O operation if it is a continuation of the current request. **Devintrgd()** must determine whether the completion of the current operation also marks the completion of the current request: it reports this distinction back to **gdintr(2K)**.

Unlike interrupt handlers for regular character and block devices, **devintrgd()** never calls **iodone(2K)** to complete the buffer and wake up the original requesting process. For general disk-type devices, this call is made by the **gdintr(2K)** routine. Clearly, though, if the current interrupt is only the end of an I/O operation and not the end of an I/O request, **gdintr(2K)** must not make the call. So **gdintr(2K)** uses the return value from **devintrgd()** to determine whether or not to call **iodone(2K)** on the buffer.

RETURN VALUE

Devintrgd() returns 0 when the current I/O request is complete, and nonzero when it is not complete (that is, when there are more I/O operations to perform). When **devintrgd()** returns 0, **gdintr()** calls **iodone()** (indirectly, through **gdiodone()**) to complete the buffer.

All information about the status of the IO is returned in the buffer header. If there was no error on the transfer, **devintrgd()** sets **bp->b_bcount**, **bp->b_resid**, and **bp->b_error** to zero. If there was an error, it sets the **B_ERROR** bit in **bp->b_flags**, and **bp->b_error** to indicate the cause of the error. Normally, this should be set to **EIO**, since CTIX doesn't (yet) provide for more specific I/O errors. It also sets **bp->b_resid** (buffer residue) to the number of bytes of the original request that were **not** transferred, due to the error.

Whether or not there was an error, it decrements **bp->b_bcount** and increments **bp->b_un.b_addr** by the number of bytes actually transferred.

SEE ALSO

devintr(2K), gdintr(2K), iodone(2K), disk(7).

NOTE

Devintrgd() never starts the next I/O request, only the next I/O operation. When the completion of the current operation also completes the current request, (that is, when **devintrgd()** returns a nonzero value) **gdintr(2K)** calls **devstart(2K)** to initiate processing on the next request.

NAME

devio – character device I/O routine (for *physio(2K)*)

SYNOPSIS

#include <sys/buf.h>

devio(bp)
struct buf *bp;

DESCRIPTION

Devio() is the generic name for a character device driver's I/O routine. If the device is an XYZ, for instance, the actual name would be **xyzio()**.

Devio() is called by **physio(2K)** to initiate I/O on a character device. Generally, **physio(2K)** is called either by **devread(2K)** or **devwrite(2K)** to perform physical (raw) I/O.

Bp is a pointer to the buffer structure that describes the I/O to be done. (See **<sys/buf.h>** for a complete description.) The buffer either belongs to the device driver itself, or is a member of the pool of buffers reserved by CTIX for physical I/O. When **devio()** is called, the fields have been set up by **physio(2K)** as follows:

b_un.b_addr is the source or destination buffer address.

b_flags contains flags describing the transfer. In particular, **B_BUSY** is always set, since the buffer is not on the available queue; also, **B_READ** is set if the transfer is a read. Otherwise, **B_READ** is not set (there isn't a real **B_WRITE** flag). Finally, **B_PHYS** is set to indicate that a physical (raw) transfer is in progress.

b_bcount is set to the transfer length in bytes.

b_dev is set to the minor device number of the device on which the transfer is to take place.

b_blkno is set to the block number on the device to transfer. On character devices, this number usually is meaningless.

Typically, **devio()** initiates I/O on the device and returns. **Physio(2K)** then sleeps, waiting for the **B_DONE** bit to be set in **bp->b_flags**.

When the completion interrupt is received from the device, **devintr(2K)** calls **iodone(2K)**, which sets the **B_DONE** bit and issues a **wakeup(2K)**, restarting the requesting process in **physio(2K)**.

RETURN VALUE

Devio() does not return a value directly. Rather, all information about the status of the I/O is returned in the buffer header. If there was no error on the transfer, it sets **bp->b_bcount**, **bp->b_resid**, and **bp->b_error** to zero. If there was an error, it sets the **B_ERROR** bit in **bp->b_flags**, and **bp->b_error** to indicate the cause of the error. Normally, this should be set to **EIO**, since CTIX doesn't (yet) provide for more specific I/O errors. It also sets **bp->b_resid** (buffer residue) to the number of bytes of the original request that were **not** transferred, due to the error.

Whether or not there was an error, it decrements **bp->b_bcount** and increments **bp->b_un.b_addr** by the number of bytes actually transferred.

SEE ALSO

devintr(2K), iodone(2K), wakeup(2K).

NAME

devioctl – character device *ioctl(2)* processor

SYNOPSIS

#include <sys/types.h>
#include <sys/XYZioctl.h>

```
devioctl(dev, cmd, addr, flag)
dev_t dev;
int  cmd;
caddr_t addr;
int  flag;
```

DESCRIPTION

Devioctl() is the generic name for a character device driver's I/O control routine. If the device is an XYZ, for instance, the actual name would be **xyzioctl()**. In the list of header files, the **XYZ** characters in **<sys/XYZioctl.h>** should be replaced by the name of the device. For example, **<sys/gdioctl.h>** contains the I/O control definitions for the general disk driver.

Devioctl() is called by CTIX in response to an **ioctl(2)** call on the device. **Dev** is the minor device number. **Cmd** is the command as defined by the driver itself. **Addr** is the address of a parameter block, and **flag** is a driver-defined value.

Devioctl() is the place to put support for device dependent features. This is the area of the CTIX I/O system that allows you the most flexibility. See Section 7 of the *CTIX Operating System Manual* for examples of **ioctl(2)** calls that various devices support.

Generally, the **devioctl()** routine is little more than a switch statement of the form:

```
switch(cmd) {
case XYZIOCTYPE:
	u.u_rval1 = XYZIOC;
	break;
case XYZGETA:
	/* Return device information to user buffer */
	if (copyout((caddr_t)&devinfo, addr, sizeof devinfo))
		u.u_error = EFAULT;
	break
case XYZSETA:
	/* Set device information from user buffer */
	if (copyin(addr, (caddr_t)&devinfo, sizeof devinfo))
		u.u_error = EFAULT;
	break;
	•
	•
	•
default:
	u.u_error = EINVAL;
	break;
}
```

RETURN VALUE

Devioctl() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

ioctl(2), disk(7), termio(7).

NAME

devioctl – character device *ioctl(2)* processor

SYNOPSIS

#include <sys/types.h>
#include <sys/XYZioctl.h>

```
devioctl(dev, cmd, addr, flag)
dev_t dev;
int  cmd;
caddr_t addr;
int  flag;
```

DESCRIPTION

Devioctl() is the generic name for a character device driver's I/O control routine. If the device is an XYZ, for instance, the actual name would be **xyzioctl()**. In the list of header files, the **XYZ** characters in **<sys/XYZioctl.h>** should be replaced by the name of the device. For example, **<sys/gdioctl.h>** contains the I/O control definitions for the general disk driver.

Devioctl() is called by CTIX in response to an **ioctl(2)** call on the device. **Dev** is the minor device number. **Cmd** is the command as defined by the driver itself. **Addr** is the address of a parameter block, and **flag** is a driver-defined value.

Devioctl() is the place to put support for device dependent features. This is the area of the CTIX I/O system that allows you the most flexibility. See Section 7 of the *CTIX Operating System Manual* for examples of **ioctl(2)** calls that various devices support.

Generally, the **devioctl()** routine is little more than a switch statement of the form:

```
switch(cmd) {
case XYZIOCTYPE:
     u.u_rval1 = XYZIOC;
     break;
case XYZGETA:
     /* Return device information to user buffer */
     if (copyout((caddr_t)&devinfo, addr, sizeof devinfo))
          u.u_error = EFAULT;
     break
case XYZSETA:
     /* Set device information from user buffer */
     if (copyin(addr, (caddr_t)&devinfo, sizeof devinfo))
          u.u_error = EFAULT;
     break;
     •
     •
     •
default:
     u.u_error = EINVAL;
     break;
}
```

RETURN VALUE

Devioctl() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

ioctl(2), disk(7), termio(7).

NAME

devopen – character and block device open routine

SYNOPSIS

#include <sys/types.h>
#include <sys/file.h>

devopen(dev, flag)
dev_t dev;
int flag;

devopen(dev)
dev_t dev;

DESCRIPTION

Devopen() is the generic name for a character or block device driver's open routine. If the device is an XYZ, for instance, the actual name would be **xyzopen()**.

Devopen() is called by CTIX whenever an **open(2)** is issued on the device. In the case of block devices that are part of the general disk system, **devopen()** is called by **gdopen(2K)** as a result of either a **mount(2)** or an **open(2)** system call on the device. In this case, the second form of the call is used.

Dev is the minor device number of the device.

Flag defines whether the device is to be opened with write permission. It contains the bits **F_READ** or **F_WRITE**, as defined in <**sys/file.h**>. This field is present only for devices that are not part of the general disk system.

It is the responsibility of the **devopen()** routine to initialize the device.

RETURN VALUE

For normal character and block devices, **devopen()** does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

For block devices that are part of the general disk system, **devopen()** returns 0 if it fails for any reason. In this case, **u.u_error** contains information about the failure. Otherwise, **devopen()** returns nonzero upon success.

SEE ALSO

gdopen(2K), mount(2), open(2), disk(7).

NOTE

Devopen() is called whenever the device is opened. This is different from **devclose()**, which is called only when the **last close(2)** is issued on the device.

NAME

devprint – block device message print routine

SYNOPSIS

#include <sys/types>

devprint(str, dev)
char *str;
dev_t dev;

DESCRIPTION

Devprint() is the generic name for a block device driver's message print routine. If the device is an XYZ, for instance, the actual name, would be **xyzprint()**.

Devprint() is called by CTIX to format and print a warning message concerning activity on a block device.

Str is a pointer to the message text.

Dev is the minor device number of the device in question.

RETURN VALUE

Devprint() does not return a value.

SEE ALSO

gdprint(2K).

NAME

devread – character device read routine

SYNOPSIS

#include <sys/types.h>

devread(dev)
dev_t dev;

DESCRIPTION

Devread() is the generic name for a character device driver's read routine. If the device is an XYZ, for instance, the actual name would be **xyzread()**.

Devread() is called by CTIX as a result of a **read(2)** system call.

Dev is the minor device number of the device being read.

The I/O request to be processed is described fully in the user area. (See **<sys/user.h>** for a complete description.) The fields will have been set up by CTIX as follows:

u.u_base	equals the destination buffer address.
u.u_count	equals the number of bytes to read.
u.u_segflg	0 indicates that the destination buffer is in kernel space; 1 means that it is in user space.

At the conclusion of the transfer, the **u.u_base** parameter must have been incremented by the number of bytes actually transferred, and **u.u_count** must have been decremented by the same amount.

If the I/O transfer was successful, then **u.u_count** must equal zero. If the transfer was unsuccessful, then **u.u_count** must be greater than zero. If the transfer used an I/O buffer, then **u.u_count** must be equal to

b_resid, the total number of bytes remaining to be transferred when the error occurred.

In either case, **u.u_base** must equal its original value plus the number of bytes transferred.

If you call **iomove(2K)** to transfer to data back to the user's buffer, it will update the user area for you. If you do not call **iomove()**, you must update these fields yourself. In this case you should call **copyout(2K)**, **subyte(2K)**, or **suword(2K)** to write the data into user space.

RETURN VALUE

Devread() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

read(2), copyin(2K), fubyte(2K), fuword(2K), iomove(2K), gdread(2K).

NAME

devrelease – release routine for loadable device drivers

SYNOPSIS

devrelease()

DESCRIPTION

Devrelease() is the generic name for the release routine for a loadable character or block device driver. If the device is an XYZ, for instance, the actual name would be **xyzrelease()**.

Devrelease() is called by CTIX as a result of a call to **syslocal(2)** with a function code of **SYSL_BINDDRV** and an option code of **DRVUNBIND**. The **lddrv(1)** program makes this system call.

If the device is busy (that is, open), **devrelease()** must return a failure indication of **EBUSY**.

If the device is not busy, it is the responsibility of the **devrelease()** routine to deallocate any memory that the driver acquired, cancel any outstanding **timeout(2K)** requests, and clear any pending interrupts from the device. Finally, **devrelease()** should give back the driver's interrupt vector by calling **reset_vec(2K)**.

RETURN VALUE

Devrelease() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

lddrv(1), reset_vec(2K), syslocal(2), untimeout(2K), drivers(7).

NOTE

Without a **devrelease()** routine, a device driver cannot be removed (unloaded) from the system. In this case, the **syslocal(2)** call fails with EBUSY.

NAME

devstart – block device start routine

SYNOPSIS

#include <sys/types.h>

devstart(minor_dev)
dev_t minor_dev;

DESCRIPTION

Devstart() is the generic name for a block device driver's start routine. If the device is an XYZ, for instance, the actual name would be **xyzstart()**.

For normal block devices, **devstart()** is called by **devstrategy(2K)** to start the I/O transfer. For block devices that are part of the general disk system, it is called by **gdstrategy(2K)**.

It is also possible for the **devstart()** routine to be called from the device interrupt handler, in order to carry out the next I/O on the queue. If your driver does this, remember that the **devintr()** routine (and anything it calls) cannot touch the user area, since it belongs to a process other than the one for which the I/O is taking place.

Minor_dev is the minor device number of the device containing I/O requests to be started.

Devstart() scans the I/O queue associated with this device, looking for work to do. If it determines that I/O already is in progress on the device, it returns without doing anything. Otherwise, it sets up the controller to perform the I/O described by the first inactive buffer on the queue and then starts the physical transfer.

Some controllers are able to perform multiple operations in parallel. For instance, some disk controllers can seek on two or more drives simultaneously. If this is true for your device, you should write the **devstart()** routine to call itself recursively. Each successive call finds one non-busy drive with work enqueued, and starts the next operation on that drive. The recursion ends when **devstart()** cannot start any more I/O.

The I/O request to be processed is described fully in the buffer header on the drive queue (see <**sys/buf.h**> for a complete description). The fields have been set up by CTIX as follows:

b_un.b_addr is the source or destination buffer address.

b_flags contains flags describing the transfer. In particular, **B_BUSY** is always set, since the buffer is not on the available queue; also, **B_READ** is set if the transfer is a read. Otherwise, **B_READ** is not set. (There is no real **B_WRITE** flag.)

b_bcount is set to the transfer length in bytes. Usually, this is the block size of the device, but it need not be so.

b_dev is set to the major + minor device number of the device on which the transfer is to take place.

b_blkno is set to the requested block number. On disk drives, blocks are numbered from the start of the partition.

RETURN VALUE

Devstart() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

devstrategy(2K), gdstrategy(2K).

NAME

devstrategy – block device strategy routine

SYNOPSIS

#include <sys/buf.h>

devstrategy(bp)
struct buf *bp;

DESCRIPTION

Devstrategy() is the generic name for a block device driver's strategy routine. If the device is an XYZ, for instance, the actual name would be **xyzstrategy()**.

CTIX calls **devstrategy()** whenever it needs to perform I/O on a block device. In particular, **bread()** (block read) and **bwrite()** (block write) are the most frequent callers. For direct I/O on files, **devstrategy()** is called by **physio(2K)**.

Block devices that are part of the general disk system use **gdstrategy(2K)**; they do not have a separate **devstrategy()** routine.

Bp is a pointer to a buffer structure that describes the I/O to be done. (See **<sys/buf.h>** for a complete description.) When **devstrategy()** is called, the fields have been set up by CTIX as follows:

b_un.b_addr is the source or destination buffer address.

b_flags contains flags describing the transfer. In particular, **B_BUSY** is always set, since the buffer is not on the available queue; also, **B_READ** is set if the transfer is a read. Otherwise, **B_READ** is not set. (There is no real **B_WRITE** flag.)

b_bcount is set to the transfer length in bytes. Usually, this is the block size of the device, but it need not be so.

b_dev is set to the major + minor device number of the device on which the transfer is to take place.

b_blkno is set to the block number on the device to transfer. On disk drives, blocks are numbered from the start of the partition.

There are no **devread(2K)** or **devwrite(2K)** routines for block devices. Instead, the CTIX kernel performs the same function using **devstrategy()**. The underlying assumption is that block devices are able to optimize accesses according to some algorithm other than just first come, first served. For instance, for the general disk driver, **gdstrategy()** implements a modified elevator algorithm to minimize head motion on the drives.

Thus, **devstrategy()** merges new reads and writes into the queue of pending requests and then calls **devstart(2K)** to initiate I/O on the device. When **devstart()** returns, **devstrategy()** also returns. It does not wait for the I/O completion itself; rather, **bread()** and **bwrite()** issue an **iowait(2K)** call, and **physio(2K)** issues a **sleep(2K)** call.

When the completion interrupt is received, **devintr(2K)** calls **iodone(2K)** to complete the buffer and then immediately starts the next I/O from the pending queue. **Iodone(2K)** sets the **B_DONE** bit in **bp->b_flags** and issues a call to **wakeup(2K)**, which restarts the requesting process in **bread()**, **bwrite()**, or **physio(2K)**.

In the case of general disk-type devices, the interrupt is fielded by **gdintr(2K)**, which calls the device driver's **devintrgd()** routine to process the interrupt. This routine returns a flag to **gdintr()** indicating whether or not the I/O is complete. If it is complete, **gdintr()** calls **gdiodone()**, which wakes up the original process as above.

RETURN VALUE

Devstrategy() does not return a value. All information about the status of the I/O is returned to **bread()**, **bwrite()**, or **physio(2K)** in the buffer header. In particular, the **B_ERROR** bit is set in **bp->b_flags**, and **bp->b_error** is set (usually to **EIO**) if an error occurred on the transfer. Also, **bp->b_resid** is set to the number of bytes **not** transferred as a result of the error.

In any case, **bp->b_bcount** is decremented, and **bp->b_un.b_addr** is incremented by the number of bytes actually transferred.

SEE ALSO

devintr(2K), devintrgd(2K), devio(2K), devstart(2K), gdintr(2K), gdstrategy(2K), iodone(2K), iowait(2K), sleep(2K), wakeup(2K).

NAME

devtimer – general disk-type device timer routine

SYNOPSIS

#include <sys/types.h>

devtimer(minor_dev)
dev_t minor_dev;

DESCRIPTION

Devtimer() is the generic name for a disk-type device driver's timer routine. If the device is an XYZ, for instance, the actual name would be **xyztimer()**.

Devtimer() is called periodically by **gdtimer(2K)** to report the status of the drive. It **should not** call **timeout(2K)** itself; rather, it simply reports the drive status when polled by **gdtimer()**.

Devtimer() does not handle DMA timeouts; these too are processed by **gdtimer()**.

RETURN VALUE

Devtimer() returns one of three values as follows:

0 The drive is not ready. This causes **gdtimer()** to remove the device. If the user was not in the process of dismounting the device, the warning message "Disk removed: May be inconsistent" is printed on the console.

1 The drive is ready.

-1 The controller was busy.

SEE ALSO

gdtimer(2K), timeout(2K), disk(7).

NAME

devwrite – character device write routine

SYNOPSIS

#include <sys/types.h>

devwrite(dev)
dev_t dev;

DESCRIPTION

Devwrite() is the generic name for a character device driver's write routine. If the device is an XYZ, for instance, the actual name would be **xyzwrite()**.

Devwrite() is called by CTIX as a result of a **write(2)** system call.

Dev is the minor device number of the device being written.

Devwrite() gets the transfer information from the user area. (See **<sys/user.h>** for a complete description.) CTIX does all of the setup necessary based upon the parameters supplied in the original **write(2)** request. In particular, the following fields are set:

u.u_base	is the source buffer address in user space.
u.u_count	is the number of bytes to write.
u.u_segflg	0 indicates that the source buffer is in kernel space; 1 means that it is in user space.

At the conclusion of the transfer, the **u.u_base** parameter must have been incremented by the number of bytes actually transferred, and **u.u_count** must have been decremented by the same amount.

If the I/O transfer was successful, then **u.u_count** must equal zero. If the transfer was unsuccessful, then

u.u_count must be greater than zero. If the transfer used an I/O buffer, then **u.u_count** must be equal to **b_resid**, the total number of bytes remaining to be transferred when the error occurred.

In either case, **u.u_base** must equal its original value plus the number of bytes transferred.

If you call **iomove(2K)** to transfer to data out of the user's buffer, it will update the user area for you. If you do not call **iomove()**, you must update these fields yourself. In this case you should call **copyin(2K)**, **fubyte(2K)**, or **fuword(2K)** to read the data from user space.

RETURN VALUE

Devwrite() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

write(2), copyout(2K), iomove(2K), gdread(2K), iomove(2K), gdwrite(2K), subyte(2K), suword(2K).

NAME

ftcancel – cancel request for fast (100 microsecond) timer

SYNOPSIS

#include <sys/types.h>

int ftcancel(function, arg)
int (*function)();
caddr_t arg;

DESCRIPTION

Ftcancel() cancels a previous **ftimeout(2K)** request.

Function() and **arg** are the parameters to the original **ftimeout(2K)** call.

RETURN VALUE

Ftcancel() returns the number of 100 microsecond ticks left before **function()** would have been called.

SEE ALSO

ftimeout(2K), timeout(2K), untimeout(2K).

NAME

ftimeout – arrange to call function later (based on fast timer)

SYNOPSIS

#include <sys/types.h>

int ftimeout(function, arg, nticks)
int (*function)();
caddr_t arg;
int nticks;

DESCRIPTION

Ftimeout() arranges for CTIX to call **function** with argument **arg** in **nticks** of the 100 microsecond clock (the fast timer). **Function** is called once, asynchronously, from the fast timer interrupt handler. The **ftimeout()** call itself returns immediately.

Ftimeout() simply validates the request and inserts it in order into the kernel **fcallout** table according to **nticks**. In other words, all of the entries before this one have less time to wait, and all of the entries after it have more time. **Ftimeout()** also adjusts the wait time of this request such that the sum of the wait times of all requests in the table up to and including this one is equal to **nticks**.

The fast timer is programmed to issue an interrupt when the first entry in the table needs to be processed. When the clock "goes off," the interrupt handler removes the first entry from the **fcallout** table and calls its **function()** parameter with argument **arg** at SPL5. The function is free to raise the IPL, but **it must not lower it below IPL5**.

When the function returns, the fast timer interrupt handler repeats this process for each table entry with a wait time of zero. Since it must process these entries in sequence, some of them will wait longer than others, perhaps considerably longer. Thus, **nticks** is the minimum wait time before **function()** is called.

After processing all of the requests that have timed out, the interrupt handler reprograms the clock to interrupt in the number of ticks specified by the first of the remaining entries.

RETURN VALUE

Ftimeout() does not return a value.

SEE ALSO

ftcancel(2K), timeout(2K), untimeout(2K).

NOTE

Ftimeout() calls **panic(2K)** if there is no space left in the **fcallout** table for the new request.

NAME

fubyte – read (fetch) byte from user space

SYNOPSIS

int fubyte(address)
char *address;

DESCRIPTION

Fubyte() reads one byte of data at **address** (which should be in user space).

RETURN VALUE

Upon successful completion, the value of the 8 bits at **address** is returned. Note that the byte value is returned as an integer without sign extension; that is, the return value is guaranteed to lie within the range of 0 to 255. If the user does not have READ access permission at **address**, a value of **-1** is returned.

NAME

fuword – read (fetch) longword from user space

SYNOPSIS

int fuword(address)
int *address;

DESCRIPTION

Fuword() reads one longword of data at **address** (which should be in user space).

RETURN VALUE

Upon successful completion, the value of the 32 bits at **address** is returned. If the user does not have read permission at **address**, a value of -1 is returned.

NAME

gdclose – general disk driver close routine

SYNOPSIS

#include <sys/types.h>

gdclose(dev, flag)
dev_t dev;
int flag;

DESCRIPTION

Gdclose() is part of the general disk driver (see **disk(7)**). It is called as a result of a **umount(2)** system call on a block device with a mounted file system. It also is called as the result of a **close(2)** system call on a block device that is part of the general disk system.

Dev is the minor device number of the device. **Flag** is 0.

The **gdclose()** routine currently does nothing. This means that the **devclose(2K)** routine for block devices that are part of the general disk system is never called.

RETURN VALUE

Gdclose() does not return a value.

SEE ALSO

devclose(2K), close(2), umount(2), disk(7).

NOTE

In the case of block devices that have been the target of **open(2)** system calls, **gdclose()** is called only when the device is closed for the **last** time. This is in contrast to **gdopen()**, which is called for **every** open on the device.

NAME

gdintr – general disk driver interrupt handler

SYNOPSIS

gdintr(vecnbr)
int vecnbr;

DESCRIPTION

Gdintr() is called whenever an interrupt is received from a block device that is a part of the general disk system (see **disk(7)**).

Vecnbr is the vector number supplied by the interrupting device.

Gdintr() checks that the device in question is active, and then calls the **devintrgd(2K)** routine in the device driver to process the interrupt. It then calls **gdiodone()** to complete the buffer, and finally calls **devstart(2K)** to start the next I/O operation on the device.

RETURN VALUE

Gdintr() does not return a value directly. Rather, it sets the **B_ERROR** flag in the **b_flags** and **b_error** fields of the buffer header to indicate a failure. In this case, it also sets **b_resid** (buffer residue) to the number of bytes from the original request that were not transferred.

SEE ALSO

devintr(2K), devstart(2K), get_vec(2K), set_vec(2K), disk(7).

NAME

gdopen – general disk driver open routine

SYNOPSIS

#include <sys/types.h>

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

DESCRIPTION

Gdopen() is called as a result of a **mount(2)** system call on a block device. It also is called as a result of an **open(2)** system call on a block device that is part of the general disk system (see **disk(7)**).

Dev is the minor device number of the device. **Flag** has the low-order bit set to 0 if the special file is to be opened (mounted) read only. If the low-order bit is set to 1, the file may be written. If **gdopen()** was called as a result of a **mount()** system call, the value **FMOUNT** is also present in the **flag** field.

The **gdopen()** routine validates its parameters and then calls **devopen(2K)** to initialize the device itself. **Gdopen()** then attempts to read in the Volume Home Block (VHB) for the file system.

RETURN VALUE

Gdopen() does not return a value directly. Rather, it sets **u.u_error** to indicate a failure.

SEE ALSO

devopen(2K), mount(2), open(2), disk(7).

NAME

gdpanic – report unrecoverable error and reboot

SYNOPSIS

gdpanic(message)
char *message;

DESCRIPTION

Gdpanic() is a part of the general disk driver (see **disk(7)**). It sets the **nosync** kernel flag to suppress a call to **update()** to sync the disks, and then calls **panic(2K)**.

Whenever an unrecoverable error is detected in disk code, there is no way of telling what state the disk queues are in. In this case, it is dangerous to call **update()**, so all drivers for general disk-type devices should call **gdpanic()** instead of **panic(2K)**.

RETURN VALUE

Gdpanic() never returns to the caller. It always causes a system crash by calling **panic(2K)**.

SEE ALSO

panic(2K), disk(7).

NAME

gdprint – general disk driver print routine

SYNOPSIS

#include <sys/types.h>

gdprint(dev, message)
dev_t dev;
char *message;

DESCRIPTION

Gdprint() is a part of the general disk driver (see **disk(7)**). It prints disk-related messages on the console in the following format:

'message' on N controller C drive D, slice S

where **message** is the message text passed to **gdprint()**, **N** is the first six characters of the device name (such as **Vsmd3200**), **C** is the controller number from the major+minor device number, **D** is the drive number, and S is the slice (partition) number.

Dev is the minor device number of the device.

Message is the message text to be printed.

RETURN VALUE

Gdprint() does not return a value.

SEE ALSO

devprint(2K).

NAME

gdread – general disk driver read routine

SYNOPSIS

gdread(dev)
dev_t dev;

DESCRIPTION

Gdread() is called as a result of a **read(2)** system call on the raw partition associated with a block device that is a part of the general disk system (see **disk(7)**.)

Dev is the minor device number of the device.

Gdread() validates its parameters and then calls **physio(2K)** with the **B_READ** flag and the address of the **gdstrategy(2K)** routine.

RETURN VALUE

Gdread() does not return a value directly. Rather, it sets **u.u_error** (indirectly) as follows:

[EFAULT] Either the user does not have read permission on the entire buffer, or the requested transfer length is zero or greater than **MAXBLK**.

SEE ALSO

devstrategy(2K), gdstrategy(2K), physio(2K), disk(7).

NAME

gdstrategy – general disk driver strategy routine

SYNOPSIS

#include <sys/buf.h>

gdstrategy(bp)
struct buf *bp;

DESCRIPTION

Gdstrategy() is called any time CTIX needs to read or write a block of data to or from a block device that is part of the general disk driver (see **disk(7)**). It also is called by **gdread(2K)** and **gdwrite(2K)** as a result of a **read(2)** or **write(2)** system call on a raw device associated with a general disk-type device.

Bp is a pointer to a buffer structure that describes the device and the block of data to read or write. The field **bp->b_blkno** contains the block number relative to the start of the slice (partition).

Gdstrategy() validates its parameters, inserts the new buffer onto the queue of previous requests, and then calls the driver's **devstart(2K)** routine with the minor device number.

RETURN VALUE

Gdstrategy() does not return a value directly. Rather, it sets **u.u_error** as follows:

[ENXIO]	Either the file system was not mounted, the device is not open, or the requested block number was invalid.

SEE ALSO

devstart(2K), disk(7).

NOTE

It is the responsibility of the **gdstrategy()** routine to optimize disk accesses. Therefore, it sorts the new request into the drive queue according its target cylinder number, with the intent of reducing head motion on the target drive.

NAME

gdtimer – general disk driver timer routine

SYNOPSIS

#include <sys/buf.h>

gdtimer(controller)
short controller;

DESCRIPTION

Gdtimer() is part of the general disk driver (see **disk(7)**). It is called once every **GDTIMEOUT** ticks of the system clock, for every disk controller in the system. Currently, **GDTIMEOUT** is set to 2 * HZ, or every two seconds.

Gdopen(2K) makes the initial call to **timeout**(2K) when the first **mount(2)** or **open(2)** system call is issued for a partition on a drive located on **controller**. Thereafter, **gdtimer()** calls **timeout()** itself.

Gdtimer() executes the following code for each drive on the relevant controller. If no partition on the drive is open, it is skipped. If the value of **gdutab.wtime** is zero, nothing is done with it. If it is 1, a DMA transfer on that drive has timed out; call **gdpanic(2K)**. Otherwise, decrement the timer and proceed. **Gdutab.wtime** normally is set by the **devstart(2K)** routine when it starts an operation on a drive.

Next, **gdtimer()** calls the device timer routine to check the drive status, using a statement of the form:

(*gds->timer)(minor_dev);

The recognized return values from the device's timer routine are

0 The drive is not ready. If the **GD_MAYREMOVE** flag is set in **gdsw.v_flags**, invalidate all blocks associated with the device and remove it from the system.

1 The drive is ready. Set the **GD_READY** flag.

-1 The controller was busy; do nothing.

RETURN VALUE

Gdtimer() does not return a value.

SEE ALSO

devtimer(2K), gdopen(2K), gdpanic(2K), timeout(2K), disk(7).

NOTE

Gdtimer() calls **gdpanic(2K)** to report timed out disk transfer operations.

NAME

gdwrite – general disk driver write routine

SYNOPSIS

#include <sys/types.h>

gdwrite(dev)
dev_t dev;

DESCRIPTION

Gdwrite() is a part of the general disk driver (see **disk(7)**). It is called as a result of a **write(2)** system call on a raw device that is associated with a block special file.

Dev is the minor device number of the device to be written.

Gdwrite() validates its parameters and then calls **physio(2K)** with the address of the **gdstrategy**(2K) routine.

RETURN VALUE

Gdwrite() does not return a value directly. Rather, it sets **u.u_error** (indirectly) as follows:

[EFAULT] Either the user does not have write permission on the entire buffer, or the requested transfer length is zero or greater than **MAXBLK**.

SEE ALSO

devstrategy(2K), gdstrategy(2K), physio(2K), disk(7).

NAME

get_vec – acquire an interrupt vector

SYNOPSIS

#include <sys/types.h>

```
int get_vec(drvid, ihandler)
ushort drvid;
int (*ihandler)();
```

DESCRIPTION

Get_vec() acquires a free interrupt vector number and arranges for CTIX to call the device driver interrupt handler **ihandler()** whenever an interrupt is received at that vector.

Drvid is assigned as a result of a **syslocal(2)** call with the parameter **SYSL_ALLOCDRV**. This call typically is made by the **lddrv(1M)** program.

Whether they are to be loaded with **lddrv(1M)** or linked into the kernel, all device drivers under CTIX must have a driver ID assigned. To accomplish this, include the following lines of code in your driver:

```
extern int DFLT_ID;
static int Drv_id = (int)&DFLT_ID;
```

The loader assigns a driver ID of 0 for all device drivers that are linked with the kernel. If you use **lddrv(1M)** to load your driver, **syslocal(2)** assigns a unique driver ID when it performs the BIND operation.

You should use the **get_vec()** call if your device has software programmable interrupt vector generation. If your device supports only hardware strappable interrupt vector generation, you must use **set_vec(2K)**.

RETURN VALUE

Upon successful completion, the interrupt vector number is returned. Otherwise, a value of -1 is returned.

SEE ALSO

set_vec(2K), reset_vec(2K).

NAME

getc – remove character from c-list

SYNOPSIS

#include <sys/tty.h>

int getc(p)
struct clist *p;

DESCRIPTION

Getc() removes and returns the first available character from the c-list addressed by the pointer **p**.

If there are no characters on the c-list, **getc()** fails.

When **getc()** removes the last available character from a c-block, it returns the c-block to the freelist. If **getc()** determines that one or more processes are asleep, waiting for a c-block, it calls **wakeup(2K)** on the address of the freelist.

RETURN VALUE

Getc() returns the first available character if the c-list was not empty. Otherwise, **getc()** returns -1.

SEE ALSO

putc(2K), sputc(2K).

NOTE

Getc() runs at **IPL6**.

NAME

getcb – remove c-block on the freelist

SYNOPSIS

#include <tty.h>

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

DESCRIPTION

Getcb() removes the first c-block linked onto the c-list pointed to by **p**, decrementing **p->c_cc** by the number of characters contained in the c-block.

RETURN VALUE

Getcb() returns a pointer to the c-block that it removed, or **NULL** if the c-list was empty.

SEE ALSO

putcf(2K).

NOTE

Getcb() runs at **IPL6**.

NAME

iodone – complete I/O on buffer

SYNOPSIS

#include <sys/buf.h>

iodone(bp)
struct buf *bp;

DESCRIPTION

Iodone() sets the **B_DONE** flag on **bp**, indicating that I/O is complete; then it calls **wakeup(2K)** to restart any process(es) that are sleeping on the buffer.

Typically, **iodone()** is called by the device driver's interrupt handler, **devintr(2K)**. In the case of block devices that are part of the general disk system, **gdintr(2K)** fields the interrupt and then calls the **devintr()** routine to process it. If the device driver returns a zero value indicating that I/O is complete, **gdintr()** calls **gdiodone()**, which calls **iodone()** to complete the buffer.

RETURN VALUE

Iodone() does not return a value.

SEE ALSO

devintr(2K), gdintr(2K), iowait(2K), wakeup(2K).

NAME

iomove – move I/O-related data and update pointers

SYNOPSIS

#include <sys/types.h>

```
iomove(bufaddr, nbytes, flag)
caddr_t bufaddr;
unsigned int nbytes;
int flag;
```

DESCRIPTION

Iomove() moves I/O-related data between the address specified in **u.u_base** and the **bufaddr** parameter. The **flag** parameter is either **B_READ** to copy data from **bufaddr** to **u.u_base**, or **B_WRITE** to copy data from **u.u_base** to **bufaddr**. After the copy, **iomove()** adds **nbytes** to **u.u_base** and **u.u_offset**, and subtracts **nbytes** from **u.u_count**. Note that **iomove()** does not use either **u.u_offset** or **u.u_count**; it simply changes them as documented above.

If you transfer I/O-related data by some means other than calling **iomove()**, you must update the fields in the user area according to the same formula.

RETURN VALUE

Iomove() does not return a value directly. Rather, it sets **u.u_error** as follows:

[EFAULT] The current user does not have the appropriate access permission at **bufaddr** for **nbytes**.

SEE ALSO

copyin(2K), copyout(2K).

NOTE

If **u.u_segflg** is set to 1, **iomove()** assumes that both source and destination buffers are located in kernel memory. In this case it calls **bcopy(2K)** to perform the move without checking access permissions. If **u.u_segflg** is not 1, **iomove()** assumes that the source/destination buffer is in user memory. In this case, it calls either **copyin(2K)** or **copyout(2K)** to check access permissions and perform the move.

NAME

iowait – wait for I/O completion on a buffer

SYNOPSIS

#include <sys/buf.h>

iowait(bp)
struct buf *bp;

DESCRIPTION

Iowait() sleeps, waiting for the completion of I/O on a buffer. Usually, the completion is signalled by **iodone(2K)**, which is called either by the device driver's interrupt handler, **devintr(2K)**, or by the general disk driver's interrupt handler, **gdintr(2K)**.

RETURN VALUE

Iowait() does not return a value. However, it sets **u.u_error** as follows:

[EIO] The **B_ERROR** flag was set in the buffer header, and **b_error** was not set to indicate a specific error condition.

SEE ALSO

devintr(2K), gdintr(2K), iodone(2K).

NOTE

Iowait() calls **sleep(2K)**, so it should not be called from the interrupt level.

NAME

is_eepromvalid – verify presence of VMEbus interface and checksum of EEPROM

SYNOPSIS

#include <sys/vme.h>

struct vmeeeprom *is_eepromvalid()

DESCRIPTION

Is_eepromvalid() checks to see whether the VMEbus interface board is present in the system and, if it is, it recomputes the checksum of the EEPROM on the board to check its validity.

RETURN VALUE

Upon successful completion, the address of the VMEbus EEPROM is returned. Otherwise, a value of 0 is returned.

SEE ALSO

probevme(2K).

NAME

macros – various useful system macros

SYNOPSIS

#include <sys/sysmacros.h>

KIMAX(val1, val2)
KIMIN(val1, val2)

btoc(nbytes)
btop(nbytes)
btotp(nbytes)

ctob(nclicks)
ptob(npages)

btodb(nbytes)
dbtob(nblocks)
dtop(nblocks)
ptod(npages)

poff(vaddr)

major(dev)
makedev(major, minor)
minor(dev)

#include <sys/page.h>

cvtov(cvaddr)
hclr(hpte)
hispgv(hpte)
hsetpg(hpte, pf, mode)
hsetpte(hpte, pte)
ispgv(hpte)
setpgprot(pte, prot)
setpgv(pte)
vtocv(vaddr)

vtohpte(vaddr)
vtopfn(vaddr)

DESCRIPTION

KIMAX(val1, val2) returns the maximum of two integers.

KIMIN(val1, val2) returns the minimum of two integers.

Btoc(nbytes) converts **nbytes** to clicks (4K bytes), rounding up to the nearest click.

Btodb(nbytes) converts **nbytes** to disk blocks (1K bytes), rounding up to the nearest block.

Btop(nbytes) converts **nbytes** to pages (4K bytes), rounding up to the nearest page.

Btotp(nbytes) converts **nbytes** to pages (4K bytes), truncating down to the nearest page.

Ctob(nclicks) converts **nclicks** to bytes.

Dbtob(nblocks) converts **nblocks** (1K blocks) to bytes.

Dtop(nblocks) converts **nblocks** (1K blocks) to pages (4K bytes).

Major(dev) returns the major device number.

Makedev(major, minor) constructs a device number from its **major** and **minor** parts.

Minor(dev) returns the minor device number.

Poff(vaddr) returns the byte offset within the page containing the virtual address **vaddr**.

Ptob(npages) converts **npages** to bytes.

Ptod(npages) converts **npages** to disk blocks.

Cvtov(cvaddr) converts the compressed virtual address **cvaddr**, which is in the range 0x00000000 to

0x01FFFFFF, to a virtual address, which is in the range 0x00000000 to 0x017FFFFF and 0x7F800000 to 0x7FFFFFFF. Compressed virtual addresses are used by A32 VMEbus DMA devices to access MightyFrame user and kernel address spaces.

Hclr(hpte) clears the hardware page table entry pointed to by **hpte**.

Hispgv(hpte) returns 1 if the page pointed to by the hardware page table entry **hpte** has the valid bit set.

Hsetpg(hpte, pf, mode) sets the hardware page table entry pointed to by **hpte** to reference the page frame numbered **pf**, with access mode **mode**.

Hsetpte(hpte, pte) sets the page table entry pointed to by **pte** to correspond to the hardware page table entry pointed to by **hpte**.

Ispgv(hpte) returns 1 if the page pointed to by the hardware page table entry **hpte** has the valid bit set.

Setpgprot(pte, prot) sets the page protection bits on the page table entry pointed to by **pte** to **prot**.

Setpgv(pte) sets the page valid bit on the page table entry pointed to by **pte**.

Vtocv(vaddr) converts the virtual address **vaddr**, which is in the range 0x00000000 to 0x017FFFFF and 0x7F800000 to 0x7FFFFFFF, to a compressed virtual address, which is in the range 0x00000000 to 0x01FFFFFF. Compressed virtual addresses are used by A32 VMEbus DMA devices to access MightyFrame user and kernel address spaces.

Vtohpte(vaddr) returns a pointer to the hardware page table entry, which references the page containing the virtual address **vaddr**.

Vtopfn(vaddr) returns the page frame number associated with the virtual address **vaddr**.

SEE ALSO

spl(2K).

NAME

panic – report unrecoverable error, sync disks, and reboot

SYNOPSIS

panic(message)
char *message;

DESCRIPTION

Panic() prints a message of the form:

panic: 'message'

on the console log file and then reboots the system. It is used to report unrecoverable errors to the system administrator.

RETURN VALUE

Panic() never returns to the caller: it always exits, either to the debugger if it is enabled, or through **reboot()**, which is documented in **syslocal(2)**. The **reboot()** routine calls **update()** if the **nosync** kernel flag is zero.

SEE ALSO

gdpanic(2K), syslocal(2).

NAME

physio – manage DMA transfers between user space and a device

SYNOPSIS

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

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

DESCRIPTION

Physio() manages DMA transfers directly between a device and user virtual memory.

Strat() is the address of the device driver's strategy routine. This can be **devstrategy(2K)**, **gdstrategy(2K)**, or **devio(2K)**.

Bp is either a pointer to a buffer structure reserved for this device or **NULL**. If **NULL**, **physio()** uses a buffer from a pool reserved for its use.

Dev is the major and minor device number.

Rw is the read/write flag, indicating the transfer direction. **Rw** must be either **B_READ**, which indicates a transfer into user memory, or **B_WRITE**, which indicates a transfer out of user memory. These manifest constants are defined in **<sys/buf.h>**.

Upon entry, **physio()** validates the transfer count, buffer address, and user access permissions. It sets **u.u_error** and returns if it detects any errors (which are defined below). Next it accesses all of the pages that are pointed to by the buffer address (generating page faults as needed to bring them in from swap space) and locks them into memory to prevent them from being swapped out again. Then it allocates a contiguous set of page table entries to reference them.

Physio() then gets a buffer from the **pfreelist** if **bp** is **NULL**. Next it sets up the buffer header fields and calls the device strategy routine, **devstrategy(2K)** for block devices, or **devio(2K)** for character devices, to perform the physical I/O. **Physio()** then sleeps until the interrupt handler (either **devintr(2K)** or **gdintr(2K)**) calls **iodone(2K)** on the buffer.

Physio() wakes up the scheduler if the **runin** kernel flag has been set, unlocks the buffer pages, setting the modified bit on each page if the I/O was a READ, and places the buffer back on the **pfreelist** if it came from there.

Finally, **physio()** sets the user area to return any error indication to the calling process.

RETURN VALUE

Physio() does not return a value directly. Rather, it sets **u.u_error** as follows:

[EFAULT] An invalid transfer count was requested (either 0 bytes or more than **MAXBLK** 1K blocks), the transfer address was not word aligned, or the user does not have read/write access permission to the memory. **MAXBLK** is defined in **<sys/page.h>**. Currently, it is 128.

[EIO] An I/O error occurred on the transfer.

SEE ALSO

devio(2K), devstrategy(2K), gdstrategy(2K).

NOTE

physio() calls **sleep(2K)**, so it should not be called from the interrupt level.

NAME

plug_svec – plug in serial device interrupt vectors

SYNOPSIS

#include <sys/types.h>

plug_svec(drvid, dev, rx, tx, sr, ex)
ushort drvid;
dev_t dev;
int (*rx)();
int (*tx)();
int (*sr)();
int (*ex)();

DESCRIPTION

Plug_svec() arranges for CTIX to call the serial device driver interrupt handlers **rx()**, **tx()**, **sr()**, and **ex()**.

Drvid is assigned as a result of a **syslocal(2)** call with the parameter **SYSL_ALLOCDRV**. This call is made by the **lddrv(1)** program.

Dev is the minor device number of the device.

Rx() is the address of the receiver interrupt handler.

Tx() is the address of the transmitter interrupt handler.

Sr() is the address of the special condition receive interrupt handler.

Ex() is the address of the external status change interrupt handler.

RETURN VALUE

Plug_svec() returns one of three values as follows:

-1 Failed.

0 Succeeded.

1 The caller was the owner. The vectors are not altered.

SEE ALSO

unplug_svec(2K).

PRINTF(2K)

NAME

printf – kernel formatted print routine

SYNOPSIS

printf(format [, arg] ...)
char *format;

DESCRIPTION

Printf() is a scaled down version of the library **printf(3S)** function that prints messages directly on the system console log file. Only the "%s", "%u", "%d", "%o", and "%x" conversion specifications from the library routine are recognized. In addition, "%nP" prints **n** bytes of the contents of memory addressed by pointer **P**. The bytes are printed using the "%x" conversion specification.

RETURN VALUE

Printf() does not return a value.

SEE ALSO

printf(3S).

NAME

probevme – check accessibility of VMEbus address

SYNOPSIS

#include <sys/types.h>

int probevme(address)
caddr_t *address;

DESCRIPTION

Probevme() checks to see whether a read access at **address** causes a bus fault. It is assumed that **address** refers to a VMEbus address. Actually, **probevme()** is simply a call to **chkbusflt(2K)** with a **flag** parameter of 0.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of 1 is returned.

SEE ALSO

chkbusflt(2K), is_eepromvalid(2K).

NAME

psignal – post signal to user process

SYNOPSIS

#include <sys/proc.h>
#include <sys/signal.h>

psignal(procptr, sig)
struct proc *procptr;
int sig;

DESCRIPTION

Psignal() posts the signal **sig** to the process indicated by the process table entry pointed to by **procptr**.

If the process is sleeping at a priority greater than **PZERO**, it is removed from the sleep queue and placed on the run queue.

RETURN VALUE

Psignal() does not return a value.

SEE ALSO

signal(2), sleep(2K).

NAME

putc – add character to c-list

SYNOPSIS

#include <sys/tty.h>

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

DESCRIPTION

Putc() adds the character **c** to the c-list addressed by the pointer **p**.

If the c-list has no c-blocks associated with it, or, if all of the associated c-blocks are full, **putc()** attempts to allocate a new c-block from the freelist. If there are no free c-blocks, **putc()** fails.

RETURN VALUE

Putc() returns 0 if it was successful in adding the character to the c-list. If it could not add the character, **putc()** returns -1.

SEE ALSO

getc(2K), sputc(2K).

NOTE

Putc() runs at **IPL6**.

NAME

putcf – put c-block on the free list

SYNOPSIS

#include <tty.h>

putcf(cp)
struct cblock *cp;

DESCRIPTION

Putcf() adds the c-block pointed to by **cp** to the free list **cfreelist**.

Putcf() calls **wakeup(2K)** on the address of the free list if **cfreelist.c_flag** is nonzero.

RETURN VALUE

Putcf() does not return a value.

SEE ALSO

getcb(2K).

NOTE

Putcf() runs at **IPL6**.

NAME

qprintf – various kernel debugging print macros

SYNOPSIS

#include <sys/kprintf.h>

aprintf(format [, **arg ...**])

•

•

•

jprintf(format [, **arg ...**])
lprintf(format [, **arg ...**])

•

•

•

rprintf(format [, **arg ...**])
tprintf(format [, **arg ...**])

•

•

•

ffprintf(format [, **arg ...**])

DESCRIPTION

Each of these macros is of the form:

```
#define Qprintf (kpflg&&kqflg&(1<<N))&&printf
```

where **Q** is one or two letters between **a** and **ff**, and N is a number between 0 and 30. Essentially, the macro definition says "If **kpflg** (the kernel print flag) is non-zero, and if the Nth bit is set in **kqflg**, then call the **printf(2K)** function with the arguments specified with the macro." This allows both gross and fine control of debugging output. That is, output may be disabled altogether by clearing **kpflg**, or output may be enabled and disabled selectively by setting **kpflag** and one or more of the bits in **kqflg**.

The kernel debugger has commands for manipulating the **kpflg** and **kqflg** variables. The **kp** command sets the state of the kernel print flag (**kpflg**). It controls printing, allowing you to route debugging output to the screen, the printer, the console buffer, the error log file, or various combinations of these options. The **kq** command toggles the bits in the **kqflg** variable. See Chapter 10, *Debugging the CTIX Kernel*, for a complete description of the kernel debugger and its command language.

The following list associates the prefix letter with its corresponding bit number in **kqflg**. It also gives the kernel debugger **kq** command parameter and the default use of each debug level, if one exists.

```
aprintf 0        /* 'a' - Regions */
bprintf 1        /* 'b' - Sptalloc(), etc */
cprintf 2        /* 'c' - Syscall trace */
dprintf 3        /* 'd' - Context swtch */
eprintf 4        /* 'e' - Pte/fault */
fprintf 5        /* 'f' - Trap info */
gprintf 6        /* 'g' - Swap */
hprintf 7        /* 'h' - File system (Direct I/O) */
iprintf 8        /* 'i' - Page hash */
jprintf 9        /* 'j' - Page out */
/* Skip kprintf 10 */
lprintf 11       /* 'l' - Initialization */
mprintf 12       /* 'm' - gdonbd */
nprintf 13       /* 'n' - gd & gdvhb */
oprintf 14       /* 'o' - gd VME */
pprintf 15       /* 'p' - Other VME */
qprintf 16       /* 'q' - VME disk devices */
rprintf 17       /* 'r' - Ram disk/prrfix */
/* Skip sprintf 18 */
tprintf 19       /* 't' - Qici */
uprintf 20       /* 'u' - 232/IOP */
vprintf 21       /* 'v' - 232/IOP */
wprintf 22       /* 'w' - 422 */
```

```
xprintf 23        /* 'x' - Network */
yprintf 24        /* 'y' - 422 */
/* zprintf never prints */
aaprintf 25       /* 'z' - Unused */
bbprintf 26       /* '{' - Unused */
ccprintf 27       /* '|' - Unused */
ddprintf 28       /* '}' - Unused */
eeprintf 29       /* '~' - Unused */
ffprintf 30       /* 'NONE' - Unused */
```

SEE ALSO

printf(2K).

NAME

reset_vec – relinquish an interrupt vector

SYNOPSIS

#include <sys/types.h>

reset_vec(drvid, vecnbr)
ushort drvid;
ushort vecnbr;

DESCRIPTION

Reset_vec() relinquishes the interrupt vector number, **vecnbr**, that was acquired by a previous call to **get_vec(2K)** or **set_vec(2K)**.

Drvid is assigned as a result of a **syslocal(2)** call with the parameter **SYSL_ALLOCDRV**. This call is made by the **lddrv(1M)** program.

RETURN VALUE

Upon successful completion, a value of 0 is returned. A value of -1 is returned if the vector number is invalid (that is, > 255) or if **drvid** does not match the ID of the device driver that owns the interrupt vector.

SEE ALSO

get_vec(2K), set_vec(2K).

NAME

scopyin – copy data from user to VMEbus space

SYNOPSIS

scopyin(from, to, nbytes)
short *from, *to;
int nbytes;

DESCRIPTION

Scopyin() copies data from user space to VMEbus space 16 bits at a time. It first calls **useracc(2K)** to verify that the user has read permission at the **from** address for **nbytes**. Then it calls **probevme(2K)** to verify the VMEbus address. Finally, **scopyin()** performs the physical copy using 16-bit reads and writes.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned. In addition, **scopyin()** sets **u.u_error** as follows:

[EFAULT] Either the user does not have read permission on the entire buffer, or a read access at the VMEbus address causes a bus fault.

SEE ALSO

copyin(2K), ccopyin(2K), copyout(2K), scopyout(2K)

NOTE

From and **to** are pointers to shorts and **nbytes** is a byte count. **Scopyin()** does perform odd-byte copies, so **nbytes** can be odd.

NAME

scopyout – copy data from VMEbus space to user space

SYNOPSIS

scopyout(from, to, nbytes)
short *from, *to;
int nbytes;

DESCRIPTION

Scopyout() copies data from VMEbus space to user space 16 bits at a time. It first calls **useracc(2K)** to verify that the user has write permission at the **from** address for **nbytes**. Then it calls **probevme(2K)** to verify the VMEbus address. Finally, **scopyout()** performs the physical copy using 16-bit reads and writes.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned. In addition, **scopyout()** sets **u.u_error** as follows:

[EFAULT] Either the user does not have write permission on the entire buffer, or a read access at the VMEbus address causes a bus fault.

SEE ALSO

ccopyin(2K), copyout(2K), scopyin(2K)

NOTE

From and **to** are pointers to shorts and **nbytes** is a byte count. **Scopyout()** does perform odd-byte copies, so **nbytes** can be odd.

NAME

set_vec – set interrupt vector number and handler address

SYNOPSIS

#include <sys/types>

set_vec(drvid, vecnbr, ihandler)
ushort drvid;
dev_t dev;
int (*ihandler)();

DESCRIPTION

Set_vec() arranges for CTIX to call the device driver interrupt handler **ihandler()** whenever an interrupt is received with vector number **vecnbr**.

Drvid is the device driver ID.

Whether they are to be loaded with **lddrv(1M)** or linked into the kernel, all device drivers under CTIX must have a driver ID assigned. To accomplish this, include the following lines of code in your driver:

```
extern int DFLT_ID;
static int Drv_id = (int)&DFLT_ID;
```

The linker assigns a driver ID of 0 for all device drivers that are linked with the kernel. If you use **lddrv(1M)** to load your driver, **syslocal(2)** assigns a unique driver ID when it performs the BIND operation.

You must use the **set_vec()** call if your device supports only hardware-strappable interrupt vector generation. If your device has software-programmable interrupt vector generation, use **get_vec(2K)**.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

SEE ALSO

get_vec(2K), reset_vec(2K).

NOTE

For general disk-type devices, the interrupt handler is **gdintr(2K)**, **not** your driver's interrupt handler. You must insert the address of your **devintr(2K)** routine in the **gddefault** data structure located in <**sys/gdtab.h**>. **Gdintr()** calls your interrupt handler when it receives an interrupt from your device.

The following matrix indicates the availability of the interrupt vectors. Those that are in use are marked with a '1': unused vectors are marked with a '0'.

1111 1111 1111 1111	00 - 0F	000 - 015
1111 1111 1111 1111	10 - 1F	016 - 031
1111 1111 1111 1111	20 - 2F	032 - 047
1111 1111 1111 1111	30 - 3F	048 - 063
1111 1111 0000 0000	40 - 4F	064 - 079
1010 1010 1010 1010	50 - 5F	080 - 095
1111 1111 1111 1111	60 - 6F	096 - 111
1111 1111 1111 1111	70 - 7F	112 - 127
1111 1111 1111 1111	88 - 8F	128 - 143
1111 1111 1111 1111	90 - 9F	144 - 159
1111 1111 1111 1111	A0 - AF	160 - 175
1111 1111 1111 1111	B0 - BF	176 - 191
1111 1111 1111 1111	C0 - CF	192 - 207
1111 1111 1111 1111	D0 - DF	208 - 223
1111 1111 1111 1111	E0 - EF	224 - 239
1111 1111 1111 1111	F0 - FF	240 - 255

NAME

setmap – map an I/O transfer onto kernel virtual memory

SYNOPSIS

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

caddr_t setmap(bp, vaddr, offset, count)
struct buf *bp;
caddr_t vaddr;
int offset;
int count;
```

DESCRIPTION

Setmap() sets up a portion of the kernel virtual address space to point to an I/O buffer in physical memory to be used for a raw I/O operation. Before the call to **setmap()**, the buffer has only a user virtual address, which is unusable for raw I/O. After the call to **setmap()**, the buffer has both a user virtual address and a kernel virtual address. The kernel virtual address is passed to the DMA device, which performs the raw I/O operation.

Bp is a pointer to a buffer structure that has had its **b_pt** field set by a call to **physio(2K)**. **Vaddr** is the kernel virtual address on which to map the transfer.

Setmap() does not allocate kernel virtual memory. You should call **sptalloc(2K)** to do this before calling **physio(2K)**.

Setmap() uses the **offset** and **count** parameters to support multiple partial mappings, for instance, when the total amount of data to transfer is larger than the DMA device can process in a single operation. **Setmap()** starts the mapping at **bp->b_un.b_addr+ offset** and maps in **count** bytes. It is your responsibility to ensure that the virtual space pointed to by **vaddr** is large enough to contain **count** bytes. In order to have **setmap()** map the entire transfer in a single call, use the following call:

setmap(bp, vaddr, 0, bp->b_count);

RETURN VALUE

Setmap() returns the virtual address of the start of the buffer.

SEE ALSO

physio(2K), sptalloc(2K).

NOTE

You must not change **bp->b_addr** from within your device driver.

Setmap() does not alter any of the fields in the buffer header, or the data in the buffer itself.

NAME

sleep – give up the processor until an event occurs

SYNOPSIS

#include <sys/types.h>

sleep(channel, pri)
caddr_t channel;
int pri;

DESCRIPTION

Sleep gives up the CPU until a **wakeup(2K)** occurs on **channel**. By convention, **channel** is the address of a data structure (such as a buffer header or a process table entry) associated with an event that the sleeper is awaiting.

When the event occurs, the sleeping process is placed on the run queue at priority **pri**. If **pri** is less than **PZERO**, the sleep will not be interrupted by any signal. On the other hand, if **pri** is greater than or equal to **PZERO**, the sleeping process will be awakened and the signal delivered. In this case, one of two actions will be taken. If the **PCATCH** flag was ORed into the **pri** value, the **sleep()** call will return with a value of 1. If **PCATCH** was not specified, the **sleep()** call never returns. Instead, a nonlocal GOTO is executed (by way of a kernel **longjmp()** call), and control resumes in the system call handler. In this case, if **u.u_error** has not been set, it is set to **EINTR**, to indicate that the system call was interrupted by a signal. Finally, control is returned to the user as though the original system call had completed with error. It is the user's responsibility to check **u.u_error** and reissue the call if it was interrupted by the receipt of a signal.

System priorities and the **PCATCH** flag are defined in **<sys/param.h>**.

Calls to **wakeup(2K)** cause all processes sleeping on **channel** to be rescheduled when, in fact, only one of them may be ready to run. For this reason, you must check that the expected event has occurred before continuing.

For example, consider the case when two device drivers need a buffer structure to perform physical I/O. If there are no buffers available, both processes will set the **B_WANTED** bit in the buffer header and sleep on the address of **pfreelist**.

At some later time, when an I/O completion interrupt occurs on an unrelated transaction, the device driver's interrupt handler will issue a call to **iodone(2K)** on its buffer. **Iodone(2K)** detects that the buffer is wanted and calls **wakeup(2K)** with **channel** equal to the address of **pfreelist**.

Both processes that were waiting for the buffer are then placed back on the run queue, but their order on the queue is indeterminate. The first process to get the CPU, finding the buffer free, sets the **B_BUSY** bit in the buffer header and proceeds with its I/O. When the second process finally runs, the buffer probably will not be free. In this case, it must issue another **sleep()** call and wait again for a free buffer.

This sequence of events describes what happens inside the routine **physio(2K)**.

RETURN VALUE

Sleep() returns a value of 0 if the process actually slept. If the value of **pri** was greater than **PZERO**, the **PCATCH** bit was set, and a signal was delivered (thus interrupting the sleep), it returns a value of 1.

SEE ALSO

signal(2), physio(2K), wakeup(2K).

NOTE

You must not call **sleep()** from the interrupt level. Also, device drivers should **never** sleep at a priority greater than **PZERO**.

NAME

spl – set processor priority level

SYNOPSIS

#include <sys/spl.h>

SDEC;

SPL0, ... SPL7
SPL422
SPLBLK
SPLDSK
SPLSERIAL
SPLTAPE
SPLX

VSPL0, ... VSPL7
VSPL422
VSPLBLK
VSPLDSK
VSPLSERIAL
VSPLTAPE

short spl0(), ... spl7()
short spl422()
short spldsk()
short splhi()
short splserial()
short spltape()

splx(s);
short s;

DESCRIPTION

The **SPL()** calls are used to set the interrupt priority mask in the processor status word. All of the UPPER CASE calls are macros that generate in-line assembly language. They are preferred (for performance reasons) over the traditional lowercase calls that generate (slower) subroutine calls to assembly language routines. In addition, the lowercase calls return the previous contents of the status word.

SPL0, ... SPL7 set the interrupt mask explicitly. They also save the previous contents in a local variable that you must declare using the **SDEC** macro. **SPLX** places the contents of the local variable declared by **SDEC** back into the status word, thereby restoring the previous interrupt level.

VSPL0, ... VSPL7 also set the interrupt level explicitly, but they do not save the old contents of the status word. Consequently, you need not use **SDEC** to declare a local variable, and you cannot call **SPLX**.

Spl0(), ... spl7() are function calls that set the interrupt mask explicitly. They return the previous value of the status word. You must save this value with an assignment statement of the form:

s = spl4();

You can restore the original value by calling **splx(s)**.

SPLDSK, SPL422, SPLTAPE, and **SPLSERIAL** set the appropriate priority level for the device they reference. You may restore the original priority by calling **SPLX**.

SPLBLK sets the priority level at or above the level of every block device in the system.

VSPLDSK, VSPL422, VSPLTAPE, and **VSPLSERIAL** also set the appropriate level as above, but they do not save the previous contents of the status word.

spldsk(), spl422(), spltape(), and **splserial()** are function calls that set the appropriate level as above.

Splblk() is a function call equivalent to **SPLBLK**.

RETURN VALUE

Macros (UPPERCASE) do not return a value. **Spl0()**, **... spl7()** return the previous contents of the processor status word.

NAME

sptalloc – allocate system page table space

SYNOPSIS

#include <sys/types.h>

caddr_t sptalloc(size, mode, base)
int size;
int mode;
int base;

DESCRIPTION

Sptalloc() allocates kernel virtual memory and, depending on the value of **base**, physical memory as well.

Size is the length of the desired memory segment in pages.

Mode is the access mode bits to be written into the page table entry. For example, **PG_V** sets the page valid bit and **PG_UW** provides the user with read/write access permission. In the current implementation, the **mode** parameter is ignored: **sptalloc()** always sets the access bits to **PTE_KW**, to provide kernel read/write access.

Base determines which type of memory allocation to perform. If **base** is less than 0, **sptalloc()** allocates virtual memory only. If **base** is equal to 0, **sptalloc()** allocates both virtual and physical memory and sets up the page table entries to point to the newly allocated memory. If **base** is greater than 0, **sptalloc()** allocates virtual memory and then, using **base** as a beginning page frame number, sets up the page table entries to point to the specified memory.

RETURN VALUE

Sptalloc() returns the virtual address of the allocated page table entries.

SEE ALSO

sptballoc(2K).

NOTE

Sptalloc() calls **panic(2K)** if it cannot allocate virtual memory.

It calls **sleep(2K)** (indirectly) when physical memory is unavailable. Thus, you must not call it from the interrupt level.

NAME

sptballoc – allocate system page table entries in small blocks

SYNOPSIS

#include <sys/types.h>

caddr_t sptballoc(size)
int size;

DESCRIPTION

Sptballoc() allocates virtual and physical memory in increments of 64 bytes.

Size is the length of the desired memory segment in bytes.

RETURN VALUE

If **size** is less than or equal to zero, **sptballoc()** returns **NULL**. Otherwise, it returns the virtual address of the newly allocated memory.

SEE ALSO

sptalloc(2K).

NOTE

Sptballoc() calls **panic(2K)** (indirectly) if it cannot allocate virtual memory.

It calls **sleep(2K)** (indirectly) when physical memory is unavailable. Thus, you must not call it from the interrupt level.

NAME

sptbfree – free system page table entries and memory

SYNOPSIS

#include <sys/types.h>

sptbfree(vaddr, size)
caddr_t vaddr;
int size;

DESCRIPTION

Sptbfree() frees kernel virtual and physical memory that was allocated previously by **sptballoc(2K)**.

Vaddr is the virtual address of the memory segment to free.

Size is the length of the segment in bytes.

RETURN VALUE

Sptbfree() does not return a value.

SEE ALSO

sptalloc(2K), sptballoc(2K), sptfree(2K).

NOTE

Sptbfree() calls **panic(2K)** (indirectly) when either the **vaddr** or **size** parameter points to memory that was not allocated through **sptballoc(2K)**.

NAME

sptfree – free system page table entry

SYNOPSIS

#include <sys/types.h>

sptfree(vaddr, size, flag)
caddr_t vaddr;
int size;
int flag;

DESCRIPTION

Sptfree() frees kernel virtual and physical memory that was allocated previously by **sptalloc(2K)**.

Vaddr is the virtual address of the memory segment to free.

Size is the length of the segment in pages.

A zero value for **flag** indicates that there is no physical memory associated with the virtual segment. A nonzero value indicates that physical memory also must be freed.

RETURN VALUE

Sptfree() does not return a value.

SEE ALSO

sptalloc(2K), sptballoc(2K).

NOTE

Sptfree() calls **panic(2K)** (indirectly) when either the **vaddr** or **size** parameter points to memory that was not allocated through **sptalloc(2K)**.

NAME

sputc – add character to c-list, sleep if necessary

SYNOPSIS

#include <sys/tty.h>

```
sputc(c, p, cansleep)
int c;
struct clist *p;
int cansleep;
```

DESCRIPTION

Sputc() adds the character **c** to the c-list addressed by the pointer **p**.

If the c-list has no c-blocks associated with it, or, if all of the associated c-blocks are full, **sputc()** attempts to allocate a new c-block from the free list. If there are no free c-blocks, **sputc()** checks the value of the **cansleep** flag. If it is zero, **sputc()** fails. Otherwise, **sputc()** sleeps on the address of the free list, waiting for a c-block to become available.

RETURN VALUE

Sputc() returns 0 if it was successful in adding the character to the c-list. If it could not add the character (because **cansleep** was zero), **sputc()** returns -1.

SEE ALSO

getc(2K), putc(2K).

NOTE

Sputc() calls **sleep(2K)**, so you should not call it from the interrupt level. It sleeps at priority TTOPRI (defined in <**sys/param.h**>), so it is interruptible by signals.

Sputc() runs at **IPL6**.

NAME

subyte – write (set) byte in user space

SYNOPSIS

int subyte(address, value)
char *address;
char value;

DESCRIPTION

Subyte() writes the byte **value** at **address** (which should be in user space).

RETURN VALUE

Upon successful completion, value 0 is returned. If the user does not have write permission at **address**, a value of -1 is returned.

NAME

suser – determine if current user is the super user

SYNOPSIS

suser()

DESCRIPTION

Suser() tests to see whether the current user is the super user.

RETURN VALUE

Suser() returns a value of 1 if the current user is the super user. It returns 0 otherwise. In addition, it sets **u.u_error** as follows:

[EPERM] The current user is not the super user.

NAME

suword – store longword to user space

SYNOPSIS

int suword(address, value)
int *address;
int value;

DESCRIPTION

Suword() writes the longword **value** at **address** (which should be in user space).

RETURN VALUE

Upon successful completion, value **0** is returned. If the user does not have write permission at **address**, a value of -1 is returned.

NAME

timeout – arrange to call function later

SYNOPSIS

#include <sys/types.h>

int timeout(function, arg, time)
int (*function)();
caddr_t arg;
int time;

DESCRIPTION

Timeout() arranges for CTIX to call **function** with argument **arg** in **time** divided by **HZ** seconds. **HZ** is defined in <**sys/param.h**>.

Function is called once, asynchronously, from the clock interrupt handler. The **timeout()** call itself returns immediately.

Timeout() simply validates the request and inserts it in order into the kernel **callout** table according to **time**. In other words, all of the entries before this one have less time to wait, and all of the entries after it have more time. **Timeout()** also adjusts the wait time of this request such that the sum of the wait times of all requests in the table up to and including this one is equal to **time**.

When the system clock interrupt handler runs, if it determines that no other interrupts are currently active (that is, that the previous IPL of the processor was 0), it decrements the **time** parameter on the first entry in the **callout** table. If the wait time in the request is now less than or equal to zero, CTIX calls **function()** with argument **arg** at **IPL1**. (Since the interrupt handler made certain that no other interrupts were active, it can manipulate the IPL with no problem. If any other interrupts were active when the clock interrupt occurred, no

callout table processing would have been done.) The function is free to raise the IPL, but it must not lower it below IPL1.

When the function returns, the clock interrupt handler repeats this process for each table entry with a wait time of zero. Since it must process these entries in sequence, some of them will wait longer than others, perhaps considerably longer. Thus, **time** is the minimum wait time before **function()** is called.

After processing all of the requests that have timed out, the clock interrupt handler removes all of the just-processed entries from the **callout** table.

RETURN VALUE

Timeout() returns a unique, 32-bit identifier that can be used as the parameter with a call to **untimeout(2K)**.

SEE ALSO

ftcancel(2K), ftimeout(2K), panic(2K), untimeout(2K).

NOTE

Timeout() calls **panic(2K)** if there is no space left in the **callout** table for the new request.

NAME

unplug_svec – relinquish serial device interrupt vectors

SYNOPSIS

#include <sys/types.h>

unplug_svec(drvid, dev)
ushort drvid;
dev_t dev;

DESCRIPTION

Unplug_svec() relinquishes the interrupt vectors that were acquired by a call to **plug_svec()**.

Drvid is assigned as a result of a **syslocal(2)** call with the parameter **SYSL_ALLOCDRV**. This call is made by the **lddrv(1M)** program.

Dev is the minor device number of the device.

RETURN VALUE

Unplug_svec() does not return a value.

SEE ALSO

plug_svec(2K).

NAME

untimeout – cancel previous *timeout(2K)* call

SYNOPSIS

int untimeout(id)
int id;

DESCRIPTION

Untimeout cancels a previous call to **timeout(2K)** .

Id is the 32-bit identifier returned by the previous **timeout(2K)** call.

RETURN VALUE

Untimeout() does not return a value.

SEE ALSO

devrelease(2K), ftcancel(2K), ftimeout(2K), timeout(2K).

NOTE

It is not an error to call **untimeout()** on a request that already has been performed: in this case, the call does nothing.

NAME

useracc – verify user access permission to memory region

SYNOPSIS

#include <sys/types.h>

useracc(base, count, rw)
caddr_t base;
int count;
int rw;

DESCRIPTION

Useracc() verifies that the current user has read or write access to a memory region.

Base() is the address of the memory region.

Count is the length of the region in bytes.

Rw is the read/write flag that indicates the transfer direction. **Rw** must be either **B_READ**, which indicates a transfer into user memory, or **B_WRITE**, which indicates a transfer out of user memory. These constants are defined in <**sys/buf.h**>.

RETURN VALUE

Useracc() returns a value of 0 if the user does not have the required access permission. Otherwise, it returns a value of 1.

NAME

wakeup - reactivate all processes sleeping on *channel*

SYNOPSIS

#include <sys/types.h>

wakeup(channel)
caddr_t channel;

DESCRIPTION

Wakeup() causes all processes that have issued a **sleep(2K)** call on **channel** to be placed on the run queue at the priority specified in their original **sleep(2K)** call.

By convention, **channel** is the address of a data structure (such as a buffer header or a process table entry) associated with an event that the sleeper is awaiting.

Wakeup(2K) causes all processes sleeping on **channel** to be rescheduled when, in fact, only one of them may be ready to run. For this reason, the caller must check that the expected event has occurred before continuing.

For example, consider the case when two device drivers need a buffer structure to perform physical I/O. If there are no buffers available, both processes will set the **B_WANTED** bit in the buffer header and sleep on the address of **pfreelist**.

At some later time, when an I/O completion interrupt occurs on an unrelated transaction, the device driver's interrupt handler will issue a call to **iodone(2K)** on its buffer. **Iodone(2K)** detects that the buffer is wanted and calls **wakeup(2K)** with **channel** equal to the address of **pfreelist**.

Both processes that were waiting for the buffer are then placed back on the run queue, but their order on the queue is indeterminate. The first process to get the

MPU, finding the buffer free, sets the **B_BUSY** bit in the buffer header and proceeds with its I/O. When the second process finally runs, the buffer probably will not be free. In this case, it must issue another **sleep()** call and wait again for a free buffer.

This sequence of events describes what happens inside the routine **physio(2K)**.

RETURN VALUE

Wakeup() does not return a value.

SEE ALSO

panic(2K), sleep(2K).

NOTE

Wakeup() calls **panic(2K)** if it detects a process on the sleep queue in a state other than **SSLEEP** or SSTOP.

GLOSSARY.

Associative Cache. A short-term storage location whose cells are accessed not by their storage address, but by the data they contain.

Block I/O System. The portion of the CTIX I/O system that interfaces with devices that contain a fixed number of randomly addressable "chunks" of data. Disk and tape drives are the most common block devices.

Buffer Header. A **buf** structure, defined in <**sys/buf.h**>. It points to a buffer, either within the system buffer cache or elsewhere. Its fields describe the contents of the buffer or the I/O operation that must be performed on the buffer.

Buffered I/O System. Another name for the Block I/O system.

C-block. A **cblock** structure, defined in <**sys/tty.h**>. C-blocks provide short-term storage for low speed character devices. The structure contains a small queue of characters (currently 64 on the Mightyframe), and pointers to the addresses within the queue where a character can be added or removed.

C-list. A **clist** structure, defined in <**sys/tty.h**>. C-lists link c-blocks together into larger queues of characters. The structure contains a count of characters held in the queue and pointers to the first and last c-blocks in the queue.

Canonical Queue. A c-list associated with terminal input. It contains all of the input characters after "erase" and "kill" processing has been performed. The process that is reading the terminal input can select whether it wants to receive input from the raw queue or the canonical queue.

Character Block. See C-block.

Character I/O System. The portion of the CTIX I/O system that interfaces with devices that do not fit within the Block I/O system. Frequently, character devices process data in asynchronous, nonrepeatable sequences. Devices such as terminals and printers are typical character devices.

Character List. See C-list.

Controller Queue. A linked list of drive queues, one member for each active drive associated with a particular disk controller. A drive is considered active when it has one or more I/O requests to service. The head of the controller queue is an **iobuf** structure, defined in <**sys/iobuf.h**>.

Critical Region. A section of code that must run without being interrupted. (Technically, on a processor that supports multiple priority levels, a critical region must not be pre-empted by an interrupt at or above a given level.)

Drive Queue. A linked list of **buf** structures, which are defined in <**sys/buf.h**>. A drive queue contains a list member for each I/O request outstanding on the drive. Another name for an I/O queue. A drive queue is headed by an **iobuf** structure, which is defined in <**sys/iobuf.h**>.

EEPROM. Electrically eraseable programmable read-only memory. A ROM device that can be erased and rewritten a limited number of times. Normally used to store information that changes very infrequently, and yet must be changed quickly and easily.

Hash Slot. An entry in the **hbuf** array. Each slot serves as the head of a linked list of buffer headers containing data for a block number and device that hashed to the same value.

High-Water Mark. An arbitrary limit within a c-list. When a process that is adding characters to a c-list causes the list to surpass the high-water mark, it is put to sleep until the driver works down the list past

I-Node. An information (or index) node within the file system. Each i-node describes one file or device (special file) within the system. The in-memory **inode** structure is defined in <**sys/inode**>. Most of the information in the in-memory copy is read in from the disk copy of the i-node.

I/O Operation. One part of an I/O request. Typically, a disk driver must break apart a request into several smaller actions, for instance, a SEEK command, followed by a READ command. Each of these simpler actions is an I/O operation.

I/O Queue. A linked list of all I/O requests associated with one device. The I/O queue is headed by an **iobuf** structure, which is defined in <**sys/iobuf.h**>. The queue members are buffer headers that describe I/O requests to be performed.

IPL. Interrupt Priority Level. In the MC68020 CPU, the IPL is determined by the state of three input lines, which encode the priority of a device requesting service. An Interrupt Priority Level of 0 indicates that no device is requesting service.

Least Recently Used. An algorithm that sorts resources into order depending upon their usage. Whenever a resource is used, it is moved to the end of the list. Thus, the resources near the head of the list are the "oldest," while resources near the end of the list are the "youngest." When a new resource is requested, the "oldest" resource in the list is chosen to satisfy the request.

Low-Water Mark. An arbitrary limit within a c-list. When a driver that is removing characters from a c-list causes the list to fall below the low-water mark, it wakes up any processes that caused the queue to surpass the high-water mark.

LRU. See least recently used.

Major Device Number. A small, positive integer, kept in the Special File for each device. CTIX uses the major device number as an index into either the **cdevsw** or **bdevsw** arrays. These arrays contain the entry point addresses for every device driver in the system.

Minor Device Number. A small, positive integer, kept in the Special File for each device. CTIX passes the minor device number to the device driver as a parameter. Typically, the driver uses the minor device number to differentiate among various devices on one controller, various partitions on a disk, recording density on a tape, and so on.

NMI. Non-Maskable Interrupt. An interrupt that cannot be ignored under software control. Typically, the non-maskable interrupt is used to report impending power failures to the CPU.

Physical I/O. An I/O transfer directly between a DMA-driven device and the user's buffer. No intervening kernel buffering is used. Also called direct I/O or raw I/O.

Raw I/O. Another name for physical I/O.

Raw Queue. A c-list associated with terminal input. It contains all of the input characters exactly as they were typed. The process that is reading the terminal input can select whether it wants to receive input from the raw queue or the Canonical Queue.

Run Queue. The queue of all processes that are ready to run and waiting for the CPU. The list is headed by **runq**, a pointer to a **proc** structure, which is defined in **<sys/proc.h>**. See sleep queue.

Sleep Queue. The queue of all processes that are sleeping, waiting for the occurrence of some asynchronous event. The list is hashed: the heads of the hash chains are kept in **sqhash**, which is an array of pointers to **proc** structures, defined in **<sys/proc.h>**. See Run Queue.

Special Files. I-nodes that are used to provide file-like access to hardware devices. A special file contains two pieces of information that CTIX uses to make the linkage to the appropriate device driver:

1. A bit indicating whether the device is part of the Character I/O system or the Block I/O system.

2. A small, positive integer called the Major Device Number.

CTIX obtains a third piece of information, called the Minor Device Number, from the i-node and passes it to the driver as a parameter.

System Buffer Cache. An associative cache, kept in LRU order, that contains recently accessed blocks from the devices in the Block I/O system. Each cache block is addressed by its device and block numbers.

System Call Stack. The stack used to process system calls within the kernel. It corresponds to the supervisor stack. It is located at the highest address within the user page and grows downward.

System Stack. See System Call Stack.

U-Page. See User Page.

User Page. An area of memory, unique to each process, that contains all of the data about the process that CTIX needs when the process is swapped in. The user page contains both the system call stack and the **user** structure. The base address of the **user** structure is equal to the base address of the user page. The system call stack is located at the highest address within the user page and grows downward.

INDEX

USER'S COMMENT SHEET

Writing MightyFrame Device Drivers, First Edition
09-00619-01   DAC-120

We welcome your comments and suggestions. They help us improve our manuals. Please give specific page and paragraph references whenever possible.

Does this manual provide the information you need? Is it at the right level? What other types of manuals are needed?

Is this manual written clearly? What is unclear?

Is the format of this manual convenient in arrangement, in size?

Is this manual accurate? What is inaccurate?

Name ______________________________ Date ______________

Title ______________________________ Phone ______________

Company Name/Department ______________________________

Address ______________________________

City ______________ State ________ Zip Code __________

Thank you. All comments become the property of Convergent Technologies, Inc.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT #1309 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

Convergent Technologies
Attn: Technical Publications
2700 North First Street
PO Box 6685
San Jose, CA 95159-6685

Fold Here

Convergent Technologies®
Printed in USA