ULTRIX

digital

Guide to Writing and Porting VMEbus and TURBOchannel Device Drivers

Part Number: AA–PGL5A–TE

ULTRIX

Guide to Writing and Porting VMEbus and TURBOchannel Device Drivers

Order Number: AA-PGL5A-TE
May 1991

Product Version: ULTRIX Version 4.2
Operating System and Version: ULTRIX Version 4.2 and higher

This guide contains information needed by systems engineers who write and port device drivers for the VMEbus and the TURBOchannel. Systems engineers who write drivers that operate on other buses can find information on driver concepts, interfaces to device driver routines, kernel structures, kernel routines used by device drivers, installation of device drivers, and header files related to device drivers.

digital equipment corporation
maynard, massachusetts

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1991
All rights reserved.

Portions of the information herein are derived from copyrighted material as permitted under license agreements with AT&T and the Regents of the University of California. © AT&T 1979, 1984. All Rights Reserved.

Portions of the information herein are derived from copyrighted material as permitted under a license agreement with Sun MicroSystems, Inc. © Sun MicroSystems, Inc, 1985. All Rights Reserved.

Portions of this document © Massachusetts Institute of Technology, Cambridge, Massachusetts, 1984, 1985, 1986, 1988.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

ALL-IN-1, CDA, DDIF, DDIS, DEC, DECnet, DECstation, DECsystem, DECUS, DECwindows, DTIF, MASSBUS, MicroVAX, Q-bus, ULTRIX, ULTRIX Mail Connection, ULTRIX Worksystem Software, UNIBUS, VAX, VAXstation, VMS, VT, XUI, and the DIGITAL logo.

SunOS, NeWs, and Open Look are registered trademarks of Sun Microsystems. UNIX is a registered trademark of UNIX System Laboratories, Inc.

Contents

About This Manual

1 Introduction to Device Drivers

Part I: OPENbus Hardware and Software Architectures

2 VMEbus Architectures

3 TURBOchannel Architecture

Part II: Structure of a Device Driver

4 Structure of an ULTRIX Device Driver

Part III: Data Structures, Kernel Routines, and Autoconfiguration

5 Data Structures Used by Device Drivers

6 Kernel Routines Used by Device Drivers

7 Device Autoconfiguration

Part IV: Error Handling and Installation

8 Error Handling

9 Installing Device Drivers

Part V: Example Drivers

10 VMEbus Device Driver Examples

11 TURBOchannel Device Driver Examples

Part VI: Porting Issues

12 Porting VMEbus Device Drivers

13 Porting TURBOchannel Device Drivers

Part VII: Appendixes

A Header Files Related to Device Drivers

B Kernel Support Routines

C Summary of Device Driver Routines

Index

Figures

Tables

About This Manual

This manual contains information needed by systems engineers who write and port device drivers for the VMEbus and the TURBOchannel. Systems engineers who write drivers that operate on other buses can find information on driver concepts, interfaces to device driver routines, kernel structures, kernel routines used by device drivers, installation of device drivers, and header files related to device drivers.

Audience

The audience are systems engineers who already know how to write a device driver. Although the manual provides some step-by-step instructions for installing device drivers, it is not a tutorial. This manual is intended for systems engineers who:

- Develop programs in the C language using standard library routines
- Know the Bourne or some other UNIX shell
- Understand basic ULTRIX concepts such as kernel, shell, process, configuration, autoconfiguration, and so forth
- Understand how to use the ULTRIX programming tools, compilers, and debuggers
- Develop programs in an environment involving dynamic memory allocation, linked list data structures, and multitasking
- Understand the hardware device for which the driver is being written
- Understand the basics of RISC hardware architecture including interrupts, Direct Memory Access (DMA) operations, memory mapping, and I/O.

Organization

Chapter 1	Introduction to Device Drivers
	Presents an overview of device drivers
Part One: OPENbus Hardware and Software Architectures	
Chapter 2	VMEbus Architectures
	Presents an overview of the VMEbus hardware and software architectures.
Chapter 3	TURBOchannel Architecture
	Presents an overview of the TURBOchannel software architecture.
Part Two: Sections of a Device Driver	
Chapter 4	Structure of an ULTRIX Device Driver

	Presents descriptions of the sections that make up any device driver.
Part Three: Data Structures, Kernel Routines, and Autoconfiguration	
Chapter 5	Data Structures Used by Device Drivers Describes members of the structures used in input/output (I/O). Only members needed by the device driver writer are described. The chapter also describes VMEbus and TURBOchannel structures.
Chapter 6	Kernel Routines Used by Device Drivers Discusses the kernel routines developed for use with VMEbus and TURBOchannel device drivers. The chapter also discusses newly developed routines that can be used by any device driver.
Chapter 7	Device Autoconfiguration Discusses the sequence of events that occurs during the autoconfiguration of VMEbus and TURBOchannel devices.
Part Four: Error Handling and Installation	
Chapter 8	Error Handling Provides guidelines for handling errors in VMEbus device drivers. In addition, explains an option for testing memory map drivers. Also summarizes when and why you would use the different kernel routines that allow you to write text to an output device.
Chapter 9	Installing Device Drivers Explains how to install VMEbus and TURBOchannel device drivers.
Part Five: Example Drivers	
Chapter 10	VMEbus Device Driver Examples Provides VMEbus device driver examples.
Chapter 11	TURBOchannel Device Driver Examples Provides TURBOchannel device driver examples.
Part Six: Porting Issues	
Chapter 12	Porting VMEbus Device Drivers Describes issues related to porting VMEbus device drivers from another vendor's hardware (for this version, Sun Microsystems) to Digital hardware.
Chapter 13	Porting TURBOchannel Device Drivers Describes issues related to porting Q-bus and UNIBUS device drivers to the TURBOchannel.
Appendix A	Header Files Related to Device Drivers Summarizes the header files used by device drivers.
Appendix B	Kernel Support Routines Presents, in reference page (man) style, descriptions of the kernel

support routines. In addition, describes special files and global variables used by device drivers.

Appendix C	Summary of Device Driver Routines Summarizes the routines for block and character device drivers.

Related Documentation

If this is your first attempt at writing or porting device drivers, you should consult some of the commercial manuals. One such manual is *Writing a UNIX Device Driver*, by Janet I. Egan and Thomas J. Teixeira.

- *Guide to Configuration File Maintenance*

  This guide contains information on how to maintain the system configuration file and how to build a new kernel, either automatically or manually. The configuration file provides you with the ability to configure your system to meet your needs. You should read this manual if you are responsible for maintaining an ULTRIX system. You should also read parts of this manual if you are planning to modify or write device drivers.

- *Guide to the Error Logger*

  This guide contains information about the error logger and how it records and reports errors and other events that occur on your ULTRIX system. The guide gives an overview of the error logger, describes how to control error logger functions, and describes using the Error Report Formatter, `uerf`. You should read this manual if you manage error information on an ULTRIX system.

- *Guide to Languages and Programming*

  This guide describes the compilers and high-level languages that are part of the compiler system. The manual gives an overview of the ULTRIX driver commands and system tools that are provided for the programming environment, and it describes how to program in a POSIX environment. The manual also describes debugging programs and provides security guidelines for programmers. Although this manual discusses implementation details for the supported languages, it does not list the syntax and definition of the elements of each language. You should read this manual if you are a programmer on an ULTRIX system.

- *Kernel Messages Reference Manual*

  This manual describes the messages produced by the files in the ULTRIX kernel. You should refer to this manual if you receive a hardware-detected or software-detected message that is reported through the ULTRIX kernel software.

- *Reference Pages Section 2: System Calls*

  This section contains descriptions of calls such as `open`, `getpagesize`, and `sigvec`. You should refer to these reference pages if you write software that calls the ULTRIX kernel.

- *Reference Pages Section 3: Library Routines*

  The ULTRIX system contains library routines for C and FORTRAN programmers. The library routines are organized into a number of libraries, including libraries for writing international software, standards-conforming software, and math software. The ULTRIX system also contains the Internet network library and various other specialized libraries. You should refer to these reference pages if you write software that calls routines in any of the ULTRIX libraries.

- *Reference Pages Section 4: Special Files*

  These reference pages describe the files related to device driver functions and network support. You should refer to these reference pages if you need information about devices. For example, you might refer to these reference pages if you are a programmer who is writing a device driver or a system administrator who is partitioning a disk.

- *Reference Pages Section 5: File Formats*

  These reference pages describe formats of various files and how the system files are used. The files described include assembler and link editor output, system accounting, and file system formats. Refer to this reference page section if you need information about file formats.

- *Reference Pages Section 8: Maintenance*

  These reference pages describe commands used to create new file systems and to verify the integrity of file systems. Use these reference pages when you perform system administration tasks.

Conventions

The following conventions are used in this manual:

`open`	In text, each mention of a generic device driver routine name is presented in this type.
`xxstrategy`	In text, each mention of an example device driver routine name is presented in this type.
`buf.h`	In text, each mention of a file name, full path name, or relative path name is presented in this type.
`brelse`	In text, each mention of a kernel routine or kernel macro name is presented in this type.
bp	In text and in kernel function definitions, each mention of an argument name is presented in this type.
`bdevsw`	In text, each mention of a structure name or structure member name is presented in this type.
. . .	In syntax descriptions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
[ ]	In syntax descriptions, brackets indicate items that are optional.
. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.

In addition, certain conventions are followed for the kernel routine function definitions presented in Appendix B. These conventions are illustrated in the following example:

int copyin(*user_addr*, *kern_addr*, *nbytes*)
caddr_t *user_addr*;
caddr_t *kern_addr*;
unsigned int *nbytes*;

The kernel function definition gives you this information:

- Return type

  Gives the data type of the return value, if the kernel routine returns data.

- Kernel routine (or macro) name

  Gives the kernel routine (or macro) name, for example, `copyin`. Note that many kernel macro names use uppercase to distinguish them from kernel routines.

- Argument names

  Gives the name of each kernel routine argument name. In the example, the argument names are *user_addr*, *kern_addr*, and *nbytes.*

- Argument types

  Gives the types for each of the arguments. In the example, these types are `caddr_t` and `unsigned int`.

The conventions followed for the driver interface function definitions are similar to those used for the kernel routines in the way argument names and types are represented. However, there are differences in the way return types and names are represented in the driver function definitions. The differences in the conventions are illustrated in the following example:

int *vmeprobe*(*ctrl*, *addr1*, *addr2*)
int *ctrl*;
caddr_t *addr1*;
caddr_t *addr2*;

The driver interface function definition gives you this information:

- Return type

  Gives the data type of the return value, if the driver routine returns data. If the driver routine does not return data, no type appears.

- Driver routine name

  Gives the driver routine name. There are two variations on the name illustrated in the driver function definitions. First, if the driver interface differs according to the bus on which the driver operates, a bus-specific name is used. For example, the interface to a driver's `probe` routine differs according to whether the driver operates on the VMEbus or the TURBOchannel. Therefore, either the name *vmeprobe* or *turboprobe* is used.

  If the driver interface is the same regardless of the bus on which the driver operates, the name *anydrv* followed by the specific interface name is used. For example, the interface to a driver's `open` routine is the same regardless of the bus on which the driver operates. Therefore, the name *anydrvopen* is used.

Note the use of *italics* to indicate that the driver routine name is variable. When you write your driver routines, you should use the naming conventions described in Section 9.1.1.1.

Introduction to Device Drivers 1

This chapter presents an overview of device drivers by discussing:

- The purpose of a device driver
- The types of device drivers
- When a device driver is called
- The place of a device driver in ULTRIX

1.1 The Purpose of a Device Driver

The purpose of a device driver in ULTRIX is to handle requests made by the kernel with regard to a particular type of device. There is a well defined and consistent interface for the kernel to make these requests. By isolating device-specific code in device drivers and by having a consistent interface to the kernel, adding a new device is made easier.

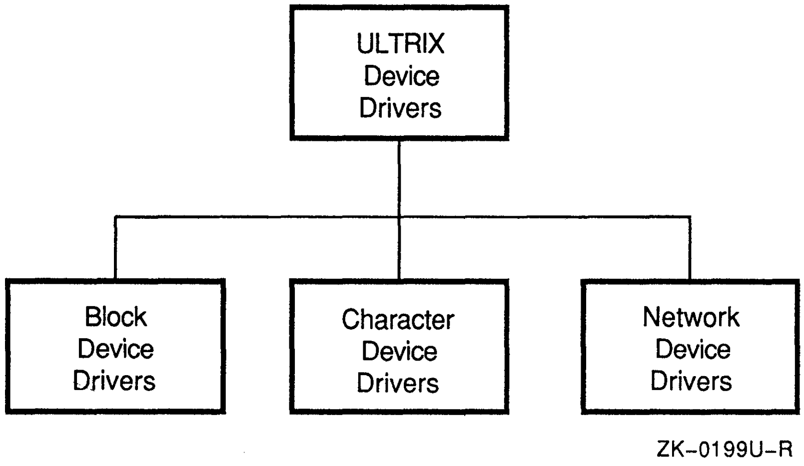
1.2 The Types of Device Drivers

A device driver is a software module that resides within the ULTRIX kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller, tape controller, network controller device, and so forth. In general, there is one device driver for each type of hardware device. Figure 1-1 shows that device drivers can be classified as:

- Block device drivers
- Character device drivers (including terminal drivers)
- Network device drivers

The following sections briefly discuss each type.

Figure 1-1: Types of ULTRIX Device Drivers



1.2.1 Block Device Driver

A block device driver is one that performs I/O using file system block-sized buffers from a buffer cache supplied by the kernel. The kernel also provides support routines for the device driver that copy data between the buffer cache and the address space of a process.

A block device driver is particularly well suited for disk drives, the most common block device. For block devices, all I/O occurs through the buffer cache. During an I/O operation, if the data is not already in the buffer cache the access of the data is not as fast as it could be, because there is an extra move of the data getting to or from the user's process.

1.2.2 Character Device Driver

A character device driver does not handle input and output through the buffer cache. Therefore, these device drivers are not tied to a single approach for handling I/O.

A character device driver can be used for a device such as a line printer that handles one character at a time. However, a character device driver can also be used where it is necessary to copy data directly to or from a user process.

Because of their flexibility, many drivers are character drivers. In addition to line printers, interactive terminals and graphics displays are examples of devices that require character device drivers.

A terminal device driver is actually a character device driver that handles input and output character processing for a variety of terminal devices. Like any character device, a terminal device can accept or supply a stream of data based on a request from a user process. Like any other character device, a terminal device cannot be mounted as a file system and, therefore, does not use data caching.

1.2.3 Network Device Driver

A network device driver attaches a network subsystem to a network interface, prepares the network interface for operation, and governs the transmission and reception of network frames over the network interface. This manual does not discuss network device drivers.

1.3 When a Device Driver Is Called

Figure 1-2 illustrates that the kernel calls a device during:

- Autoconfiguration

  The kernel calls a device driver at autoconfiguration time to determine what devices are available and to initialize them.

- Input/output operations

  The kernel calls a device driver to perform input/output operations on the device. These operations include opening the device to perform reads and writes and closing the device.

- Interrupt handling

  The kernel calls a device driver to handle interrupts generated from devices capable of generating interrupts.

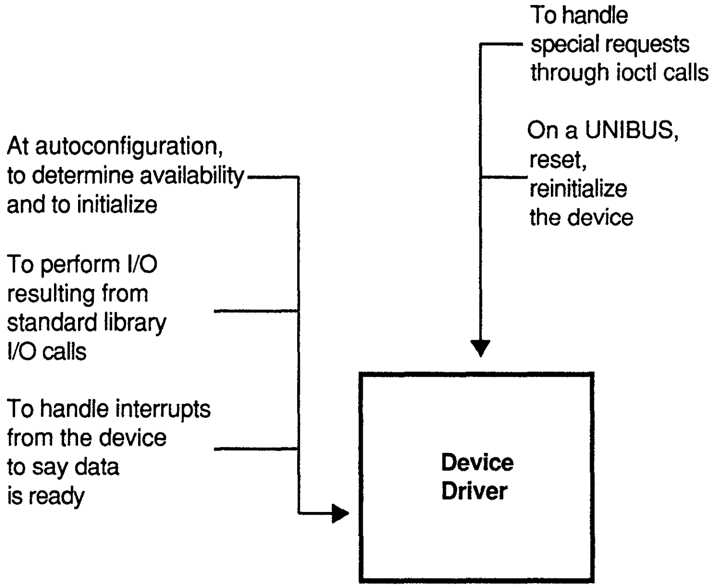
- Special requests

  The kernel calls a device driver to handle such special requests through ioctl calls.

- Reinitialization

  The kernel calls a device driver to reinitialize the driver, the device, or both when the bus (the path from the CPU to the device) is reset.

Figure 1-2: When the Kernel Calls a Device Driver



ZK-0200U-R

Some of these requests, such as input or output, result directly or indirectly from corresponding system calls in a user program. Other requests, such as the calls at autoconfiguration time, do not result from system calls.

1.3.1 The Place of a Device Driver in ULTRIX

Figure 1-3 shows the place of a device driver in ULTRIX relative to some device. Note that the device is in the center and the outer circles represent the distance of the following:

- User program

  A user program makes calls on the kernel but never directly calls a device driver.

- The kernel

  The kernel runs in supervisor mode and does not communicate with a device except through calls to a device driver.

- A device driver

  A device driver communicates with a device by reading and writing to peripheral device registers through a bus.

- Bus

  The bus is the data path between the main processor and the device controller.

- Controller

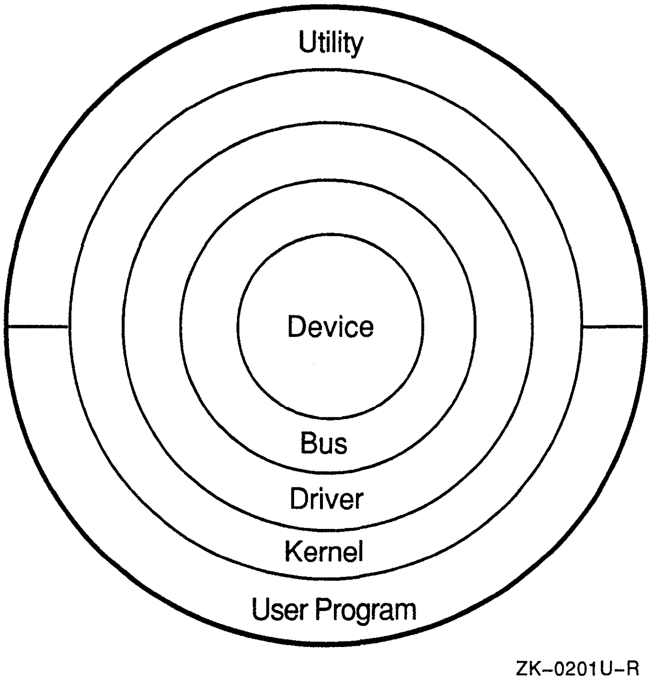
  A controller is a physical interface for controlling one or more devices. Some devices (for example, disk and tape drives) can be connected to the controller.

Other devices (for example, the network) may be integral to the controller.

- Peripheral device

  A peripheral device is a device that can be connected to a controller.

Figure 1-3: The Place of a Device Driver in ULTRIX



The following sections describe these parts with an emphasis on how a device driver relates to them.

1.3.2 User Program

User programs make system calls on the kernel that result in the kernel making requests of a device driver. For example, a user program can make a `read` system call, which calls the driver's `read` routine.

The ULTRIX operating system includes the following:

- The kernel
- The shell
- Utilities that execute ULTRIX commands
- Interpreters, compilers, debuggers, and so forth
- Spooling systems
- Other programs considered for various reasons to be part of the system

From the point of view of writing device drivers, the parts of the operating system other than the kernel are basically like user programs.

1.3.3 The Kernel

The kernel makes requests to a device driver to perform operations regarding a particular device. Some of these requests result directly from requests from a user program. For example:

- Block I/O (open, strategy, close)
- Character I/O (open, write, close)

Autoconfiguration requests, such as `probe` and `attach`, do not result directly from a user program, but as the result of activities performed by the kernel. At boot time, for example, the kernel calls the driver's `probe` routine.

A device driver may call on kernel support routines to support such tasks as:

- Sleeping and waking (process rescheduling)
- Scheduling events
- Managing the buffer cache
- Moving or initializing data

See Appendix B for descriptions of the kernel support routines.

1.3.4 Device Drivers

A device driver, run as part of the kernel software, manages each of the device controllers on the system. Often, one device driver manages an entire set of identical device interfaces. Because the device driver is part of the kernel, it must be configured with the rest of the kernel software. On ULTRIX, you can configure more device drivers than there are physical devices configured into the hardware system. At boot time, the autoconfiguration procedure can determine which of the physical devices are accessible and functional and can produce a correct run-time configuration for that instance of the running kernel.

As stated previously, the kernel makes requests of a driver by calling the driver's standard entry points (such as `probe`, `attach`, `open`, `read`, `write`, `close`). In the case of I/O requests such as read and write, it is typical that the device causes an interrupt upon completion of each I/O operation. Thus, a write system call from a user program may result in several calls on the interrupt entry point in addition to the original call on the write entry point.

Device drivers, in turn, make calls upon kernel support routines to perform the tasks mentioned earlier.

The structure declaration giving the layout of the control registers for a device are part of the source for a device driver. Device drivers (unlike the rest of the kernel) can access and modify these registers.

1.3.5 Buses

When a device driver reads or writes to the hardware registers of a controller, the data travels across a bus.

A bus is a physical communication path and an access protocol between a processor and its peripherals. A bus standard, with a predefined set of logic signals, timings, and connectors, provides a means by which many types of device interfaces (controllers) can be built and easily combined within a computer system. The term OPENbus refers to those buses whose architectures and interfaces are publicly documented, allowing a vendor to easily plug in hardware and software components. The VMEbus and the TURBOchannel can be classified as having OPENbus architectures.

Device driver writers must understand the bus that the device is connected to. Different buses require different approaches to writing the driver. For example, a VMEbus device driver writer must know how to allocate the VMEbus address space. This manual describes what a driver writer must know to write device drivers that communicate with a peripheral device that uses the VMEbus and the TURBOchannel.

1.3.6 Device Controller

Controllers are the hardware interface between the computer and a peripheral device. Sometimes a controller handles several devices. In other cases, a controller is built into the device.

1.3.7 Peripheral Devices

A peripheral device is a piece of hardware that connects to a computer system. It can be controlled by commands from the computer and can send data to the computer and receive data from it. Examples of peripheral devices include:

- A data acquisition device, like a digitizer
- A line printer

For the most part, the distinction between a device and its controller is not important to the driver writer.

Part I: OPENbus Hardware and Software Architectures

VMEbus Architectures 2

The VMEbus is an industry standard high performance bus that supports 8-, 16-, and 32-bit transfers over a nonmultiplexed 32-bit data bus. In addition, the VMEbus supports 16-, 24-, and 32-bit addressing over a separate 32-bit address bus. This chapter presents an overview of the VMEbus hardware and software architectures. Specifically, the chapter discusses the following:

- Processors used with the VMEbus hardware
- VMEbus hardware architecture
- VMEbus software architecture

For detailed information on VMEbus architectures, see the *IEEE Standard for a Versatile Backplane Bus: VMEbus ANSI/IEEE Std 1014-1987.*

2.1 Processors Used with the VMEbus Hardware

The DECstation 5000 Model 200 supports the VMEbus. The VMEbus attaches to the DECstation 5000 Model 200 through an adapter card on the TURBOchannel.

2.2 VMEbus Hardware Architecture

The VMEbus, like other buses, is a computer architecture that defines a computer data path. Unlike other buses, the VMEbus is microprocessor-independent, is easily upgraded from 16-bit to 32-bit processors, and is suitable for a vendor to build compatible products. The following describes VMEbus hardware architecture topics relevant to the device driver writer:

- Address spaces
- Data size
- Byte ordering
- Interrupt vectors
- Interrupt priorities

2.2.1 VMEbus Address Spaces

The VMEbus hardware makes no distinction between I/O space and memory space. The device driver writer must understand which address space the board uses. The VMEbus hardware architecture includes three address spaces:

- 16-bit (A16)
- 24-bit (A24)

- 32-bit (A32)

These address spaces are overlapping, that is, an address (for example, 0xC0) points to the same location in all three address spaces. VMEbus devices can respond to address requests in any of the address spaces.

2.2.2 Data Size

The VMEbus supports 8-bit(D08), 16-bit(D16), and 32-bit(D32) data sizes. A VMEbus device can operate in more than one data space at one time. For example, a VMEbus device may have D16 control registers and D32 memory.

2.2.3 Byte Ordering

While the VMEbus does not specify any particular byte ordering, most devices use the Motorola model, which is big endian. Because the Digital model is little endian, two mechanisms are provided to accomplish byte swapping:

- VMEbus adapter

  The VMEbus adapter provides hardware byte swapping. Digital's adapters provide hardware assist for all DMA transfers and may provide hardware assist for programmed I/O (PIO) transfers on an adapter-dependent basis.

- Software routines

  Kernel routines and library calls accomplish the byte swapping.

See Chapter 6 and Appendix B for information on these byte-swapping routines: `swap_lw_bytes`, `swap_word_bytes`, and `swap_words`.

2.2.4 VMEbus Interrupt Vectors

VMEbus interrupt vectors range from 0x00 to 0xff inclusive. The vectors from 0x00 - 0x3f inclusive are reserved for use by the ULTRIX operating system. The vectors 0x40 - 0xff inclusive are available for use by VMEbus devices.

2.2.5 VMEbus Interrupt Priorities

The VMEbus provides for seven interrupt priorities. On some host implementations, fewer than seven levels may be provided. On those implementations, the VMEbus priorities are mapped to the available host priority levels.

ULTRIX allows the adapter to handle any or all of the VMEbus interrupt levels. In general, you will want the adapter to handle all seven levels. If, however, there is another processor on the VMEbus that you want to handle VMEbus interrupts, you can selectively enable the interrupts handled by Digital's VMEbus adapter. The mechanism for accomplishing this is through the `intr_mask` member of the `vbadata` structure, which is described in Section 5.2.2.

2.3 VMEbus Software Architecture

Before writing device drivers that operate on the VMEbus, you need to consider the following topics associated with the VMEbus software architecture:

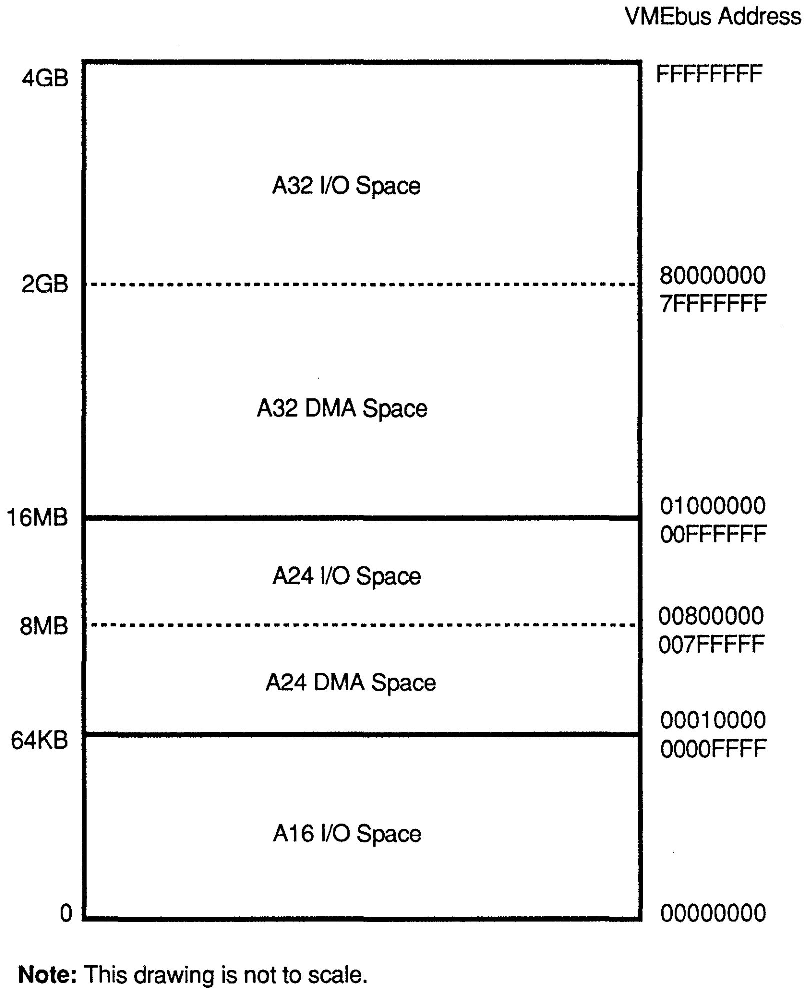
- VMEbus address space

- Direct Memory Access (DMA) support
- Input/Output (I/O) access
- Read-Modify-Write
- Writes to the hardware device register

2.3.1 VMEbus Address Space

The VMEbus supports a 4 gigabytes (GB) address space. ULTRIX divides this address space into overlapping address spaces according to the number of address bits used. A generic layout of the VMEbus address space is illustrated in Figure 2-1. Some adapter configurations, however, modify this generic layout to accommodate their specific mapping requirements.

Figure 2-1: VMEbus Address Space



Note: This drawing is not to scale.

ZK-0197U-R

Note that all device CSRs and onboard memory must be configured in the I/O space of the appropriate VMEbus address space.

By convention, Digital reserves the lower half of the A24 and A32 address spaces for VMEbus-to-system memory DMA transfers. The upper half of the A24 and A32 address spaces and the entire A16 address space are reserved for I/O space (CSRs) and device-to-device DMA transfers.

The figure shows the following overlapping address spaces:

- A 16-bit address space (A16) of 64 kilobytes (KB)

  The entire A16 address space is reserved for I/O space (CSRs) and for device-to-device Direct Memory Access (DMA) transfers.

  Valid VMEbus CSR addresses for the A16 I/O space range from 00000000 to 0000FFFF inclusive.

- A 24-bit address space (A24) of 16 megabytes (MB) - 64 kilobytes (KB)

  The lower half (8 MB - 64 KB) of the A24 address space is reserved for VMEbus-to-system memory DMA transfers. The upper half (8 MB) of the A24 address space is reserved for I/O space (CSRs) and for device-to-device DMA transfers.

  Valid VMEbus CSR addresses for the A24 I/O space are from 00800000 to 00FFFFFF inclusive.

- A 32-bit address space (A32) of 4 GB - 16 MB

  The lower half (2 GB - 16 MB) of the A32 address space is reserved for VMEbus-to-system memory DMA transfers. The upper half (2 GB) of the A32 address space is reserved for I/O space (CSRs) and for device-to-device DMA transfers.

  Valid VMEbus CSR addresses for the A32 I/O space are from 80000000 to FFFFFFFF inclusive.

You allocate the VMEbus address space for DMA by calling `vballoc` or `vbasetup`. These routines return an address from the DMA space (the lower half) that is mapped to the buffer. For the A24 DMA space, the range of valid VMEbus addresses these routines can return is from 00010000 to 007FFFFF inclusive. For the A32 DMA space, the range of valid VMEbus addresses these routines can return is from 01000000 - 7FFFFFFF inclusive. See Chapter 6, Chapter 12, and Appendix B for more information on these routines.

2.3.2 DMA Support

Some VMEbus devices can perform Direct Memory Access (DMA). Using DMA, the host processor informs the device controller about the following:

- The address in VMEbus address space where a data transfer occurs
- The length of the data to be transferred
- When to start the data transfer

The host processor makes no further intervention during the transfer of the data. Upon completion of the data transfer, the device controller interrupts to indicate that transfer has successfully completed.

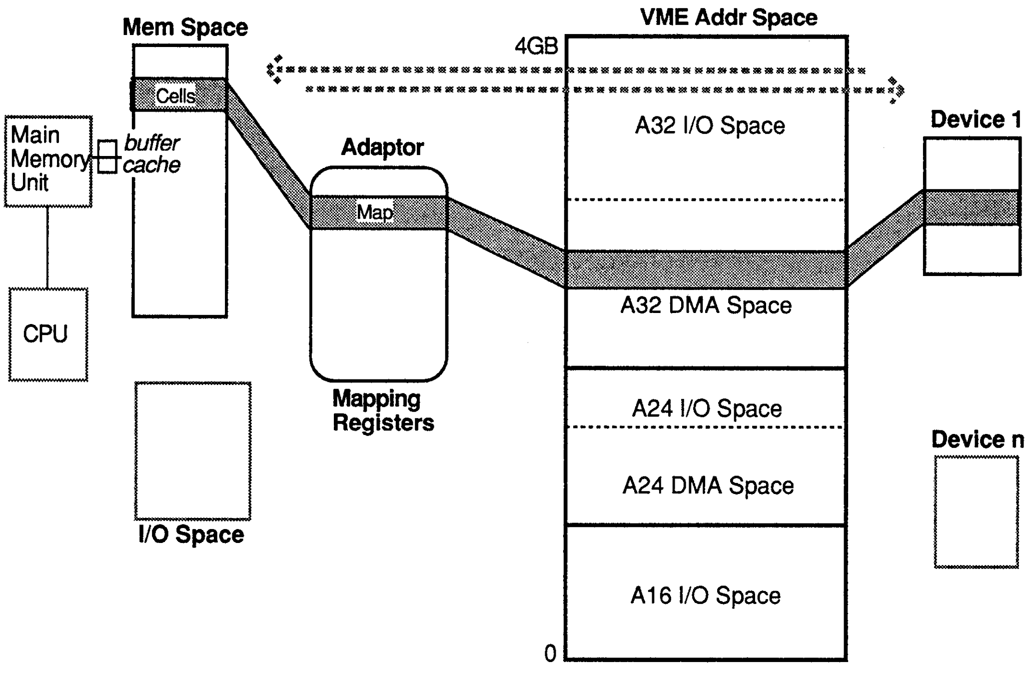
There are these scenarios to consider when dealing with the VMEbus and DMA:

- VMEbus-to and from-host-DMA

- VMEbus device-to-device DMA
- DMA for multiple VMEbus adapters

2.3.2.1 VMEbus-to Host DMA and VMEbus from-Host-DMA – Figure 2-2 illustrates VMEbus-to-host-DMA and VMEbus-from-host-DMA.

Figure 2-2: VMEbus-to and from-Host-DMA



ZK–0255U–R

The figure depicts the following typical VMEbus environment:

- A host CPU and its attendant memory
- The buffer cache
- I/O space
- An adapter that contains the mapping registers
- The VMEbus address space
- One or more devices (represented by Device 1 through Device n)

The figure uses two arrows to indicate that the data transfer can take the following routes:

- The data transfer can originate from the device to the system memory of the host CPU.

  In this route, the host memory is mapped to the A32 DMA space. As stated previously, the lower halves of the A24 and A32 spaces are reserved for VMEbus-to-host-DMA transfers. The transfer continues through the adapter into the host memory.

- The data transfer can originate from the system memory of the host CPU to the device.

  In this route, some memory space in the CPU is mapped to the A32 DMA space. The device reads from the A32 mapped space and the data is fetched by the adapter from mapped host memory.

  If a device performs DMA-to-host memory transfers, the driver must explicitly flush the data cache, because there is no hardware cache coherency mechanism. To flush the data cache, the driver calls the `bufflush` kernel support routine after the DMA completes but before it releases the buffer to the system. See Chapter 6 and Appendix B for descriptions of `bufflush`.

It is important to note that in both routes, the device initiates the data transfer.

Note that not all adapters provide the ability to perform DMA to the entire address range. Table 2-1 lists the maximum size for the A24 and A32 DMA space for the supported adapter. In addition, the table lists the range of addresses that `vballoc` or `vbasetup` can return to the device driver for the supported adapter.

Table 2-1: Maximum Size and Range of Addresses for PMABV-AA Adapter

Address Space	Maximum Size	Range of Addresses
A24	8MB - 64K = 7.936MB	00010000 - 007FFFFF
A32	128MB - 16MB = 112MB	01000000 - 7FFFFFFF

2.3.2.2 VMEbus Device to Device DMA – In addition to VMEbus to and from host DMA, there is VMEbus device-to-device DMA. Digital provides for this type of DMA by designating portions of the VMEbus address spaces as reserved for device-to-device DMA. As stated previously, the upper halves of the A24 and A32 spaces and the entire A16 space are reserved for device-to-device DMA. The VMEbus address space may have holes that are created by device registers and on-board memory. During VMEbus configuration, those areas are removed from the resource allocation map for VMEbus address space and are unavailable for use by any form of DMA.

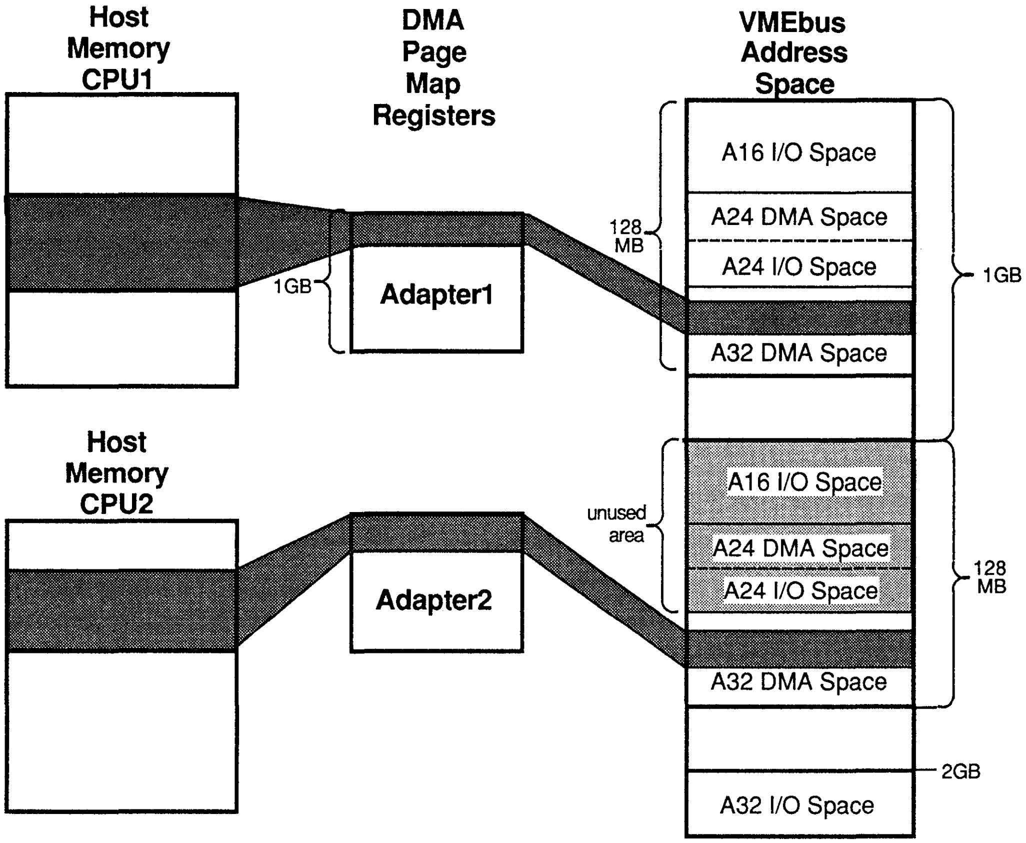
2.3.2.3 DMA for Multiple VMEbus Adapters – The PMABV-AA adapter for the TURBOchannel supports two VMEbus adapters in a single VMEbus backplane. This support exists only if the VMEbus adapters are connected to different host CPUs. To use this feature, the device driver writer must consider:

- The configuration of the DMA Page Map Registers (PMRs)
- The handling of interrupts between the two VMEbus adapters

The following discussion assumes an understanding of the `vbadata` structure and the `vballoc` and `vbasetup` routines. See Section 5.2.2 for descriptions of the members contained in the `vbadata` structure. See Chapter 6 and Appendix B for information on `vballoc` and `vbasetup`.

Figure 2-3 illustrates how the driver writer can configure the DMA PMRs to accommodate the use of two VMEbus adapters.

Figure 2-3: Use of Multiple VMEbus Adapters



ZK–0275U–R

The figure shows a VMEbus environment consisting of:

- Two host CPUs and their attendant memories
- Two adapters: one labeled Adapter1 and the other labeled Adapter2
- The VMEbus address space

Note that this figure shows a modification of the generic VMEbus address space that is illustrated in Figure 2-1. Because the PMABV-AA adapter does not support the entire A32 DMA address space, it makes use of the unused space to provide a mapping area for the second adapter. You can see this arrangement by studying the VMEbus Address Space block in Figure 2-3. The first mapping area resides within the first gigabyte and consists of the A16 I/O space, the A24 DMA space, the A24 I/O space, and the A32 DMA space. The second mapping area resides within the second gigabyte and consists of the same address spaces as the first mapping area except that the address spaces that are shaded cannot be used.

You use the `asc` member of the `vbadata` structure to select either the first gigabyte or the second gigabyte of VMEbus address space for the mapping of the DMA PMRs.

By default, the system sets this member to `VME_MAP_LOW`, which means the `vballoc` or `vbasetup` routine maps the DMA PMRs for the adapter (in this example, Adapter1) to the VMEbus addresses that reside in the range from 0 - 128MB. These addresses reside in the first gigabyte of the VMEbus address space and, specifically, in the A32 DMA address space.

To select the second gigabyte, you set the `asc` member to the constant `VME_MAP_HIGH`. In this case, `vballoc` or `vbasetup` maps the DMA PMRs for the adapter (in this example, Adapter2) to the VMEbus addresses that reside in the range from 40000000 (1GB) - 47FFFFFF (1GB + 128MB). Note that only A32 DMA can be performed if the map registers are mapped to the second GB.

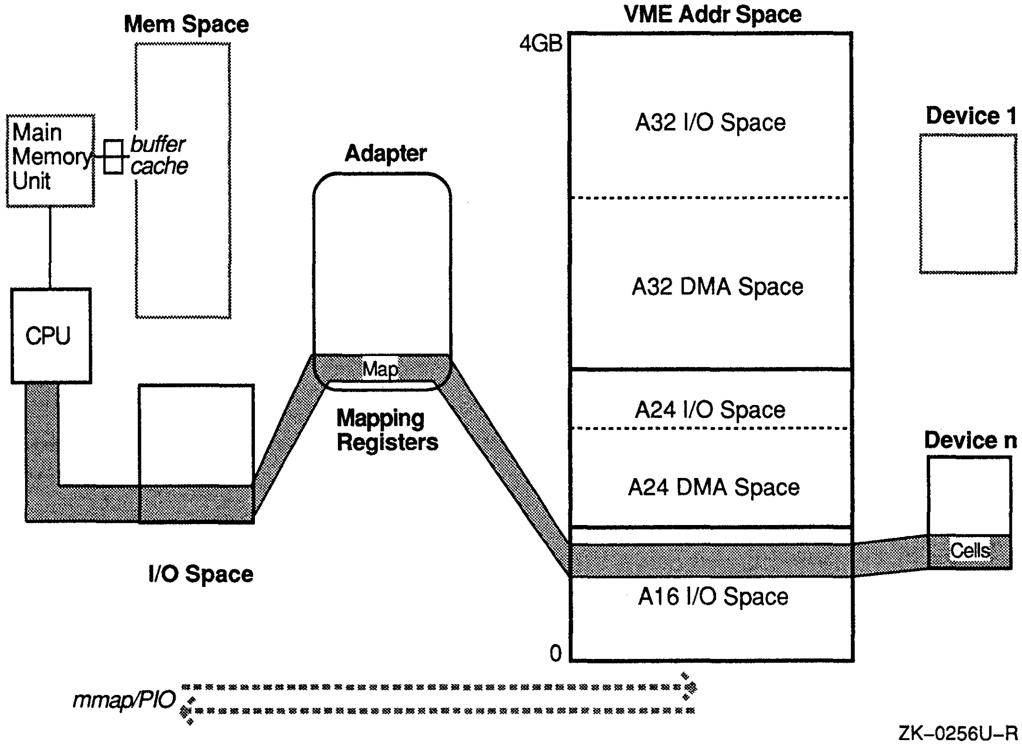
This strategy guarantees that the addresses will not overlap.

The second thing you need to do is to coordinate the handling of the interrupts between the two VMEbus adapters. The `intr_mask` member of the `vbadata` structure must be set so that only one of the VMEbus adapters is handling each interrupt level.

2.3.3 I/O Access

Applications access I/O devices through memory locations in the physical address space of the CPU. Two mechanisms – programmed I/O (PIO) and memory mapping – are provided for transferring data as the result of a data transfer request from an application. These mechanisms are discussed in the following sections. Figure 2-4 illustrates programmed I/O and memory mapping.

Figure 2-4: Programmed I/O



2.3.3.1 Programmed I/O

Programmed I/O – In the PIO mechanism, the device driver performs the data transfer. The device driver has direct access to the CSRs or to device memory. The VMEbus address space for the CSRs or onboard memory is mapped during VMEbus configuration when the device is configured. The sizes and address spaces for the mapped areas are set in these members of the `uba_driver` structure: `ud_addr1_size`, `ud_addr2_size`, `ud_addr1_atype`, and `ud_addr2_atype`. See Chapter 5 for more information on these and other members of the `uba_driver` structure. ULTRIX passes the information contained in these members to the device driver through the `probe` routine. See Section 4.3 for a description of the `probe` routine.

2.3.3.2 Memory Mapping – Many applications make use of memory mapping in which the mapped I/O space (or some portion of it) is mapped into the user address space. This allows applications to access VMEbus devices implicitly, through memory references. For example, an application can map a portion of the VMEbus address space and point to the base of an array at that mapped area. Any memory reference to that array in the application causes the corresponding part of VMEbus address space to be accessed. This is a commonly used technique for logic simulators and array processors.

An application maps VMEbus space with the `mmap` system call. The `mmap` system call invokes a kernel routine that, in turn, calls the device driver's memory mapping routine so that the driver actually performs the mapping. See Section 4.12 for more information on the tasks performed by the `mmap` system call and the memory mapping routine.

2.3.4 Read – Modify – Write

Some applications, mainly those using semaphores, require a way to perform atomic read and write operations. The VMEbus specification provides for these operations through the read-modify-write cycle on the bus. This operation allows an application to read a location, check if the location is available for writing, and to write data back to the location if the location is available. The DECstation 5000 Model 200 cannot support this operation in hardware because the TURBOchannel does not support read-modify-write operations. Because the TURBOchannel does not support the read-modify-write operations, you cannot use the system main memory for read-modify-write transactions.

To support read-modify-write operations, the `vme_rmw` routine is provided. This routine allows read-modify-write operations to VMEbus memory. See Appendix B for a description of `vme_rmw`.

2.3.5 Writes to the Hardware Device Register

Whenever a VMEbus device driver writes to a hardware device register, the write is delayed by the system write buffer used to synchronize the CPU on the TURBOchannel. A subsequent read of that register does not wait for the write to complete. To ensure that a write to I/O space completes, the driver calls the `wbflush` kernel support routine. See Chapter 6 and Appendix B for descriptions of `wbflush`.

TURBOchannel Architecture 3

The TURBOchannel is a synchronous, asymmetrical I/O channel that is supported by the DECstation 5000 Model 200.

The device driver writer is not required to be intimately familiar with the details of the TURBOchannel hardware. Therefore, this chapter discusses the following aspects of the software architecture for a TURBOchannel device driver:

- Structure of a TURBOchannel device driver
- Include files
- Writes to hardware device register
- DMA-to-host memory transfers
- Device interrupt line to the processor

3.1 Structure of a TURBOchannel Device Driver

In general, you structure a TURBOchannel device driver much like a UNIBUS or Q-bus driver. This means you declare and initialize a `uba_driver` structure in the declarations section of the TURBOchannel driver. In addition to the `uba_driver` structure, you also use these other uba structures: `uba_device` and `uba_ctlr`. See Chapter 5 for descriptions of these structures.

Note

> Even though the `uba` data structures are used, TURBOchannel device drivers do not need to use mapping registers, because the TURBOchannel address space is included in the system address space.

3.2 Include Files

TURBOchannel device drivers, in addition to the usual header files required by ULTRIX device drivers, need this header file:

```
"../io/tc/tc.h"
```

See Chapter 4 for information on header files.

3.3 Writes to the Hardware Device Register

Whenever a TURBOchannel device driver writes to a hardware device register, the write is delayed by the system write buffer used to synchronize the CPU on the TURBOchannel. A subsequent read of that register does not wait for the write to complete. To ensure that a write to I/O space completes, the driver calls the `wbflush` kernel support routine. See Chapter 6 and Appendix B for descriptions of `wbflush`.

3.4 Direct Memory Access (DMA)-to-Host Memory Transfers

If a device performs DMA-to-host memory transfers, the driver must explicitly flush the data cache, because there is no hardware cache coherency mechanism. To flush the data cache, the driver calls the `bufflush` kernel support routine after the DMA completes but before it releases the buffer to the system. See Chapter 6 and Appendix B for descriptions of `bufflush`.

3.5 Device Interrupt Line

If a device needs to have its interrupts enabled or disabled during configuration or during operation, a TURBOchannel device driver can call the `tc_enable_option` and `tc_disable_option` routines. See Chapter 6 and Appendix B for descriptions of `tc_enable_option` and `tc_disable_option`.

Part II: Structure of a Device Driver

Structure of an ULTRIX Device Driver 4

This chapter describes the sections that make up an ULTRIX device driver. Figure 4-1 illustrates the sections that a character device driver can contain and the possible sections for a block device driver. Both types of drivers contain an include files section, a declarations section, an autoconfiguration support section, an open and close device section, an ioctl section, and a strategy section (which often is not defined for character devices). Note that the strategy section for the character device driver is for nbufio, and the strategy section of the block device driver is for queuing I/O requests. (The concept of nbufio is not discussed in this manual.)

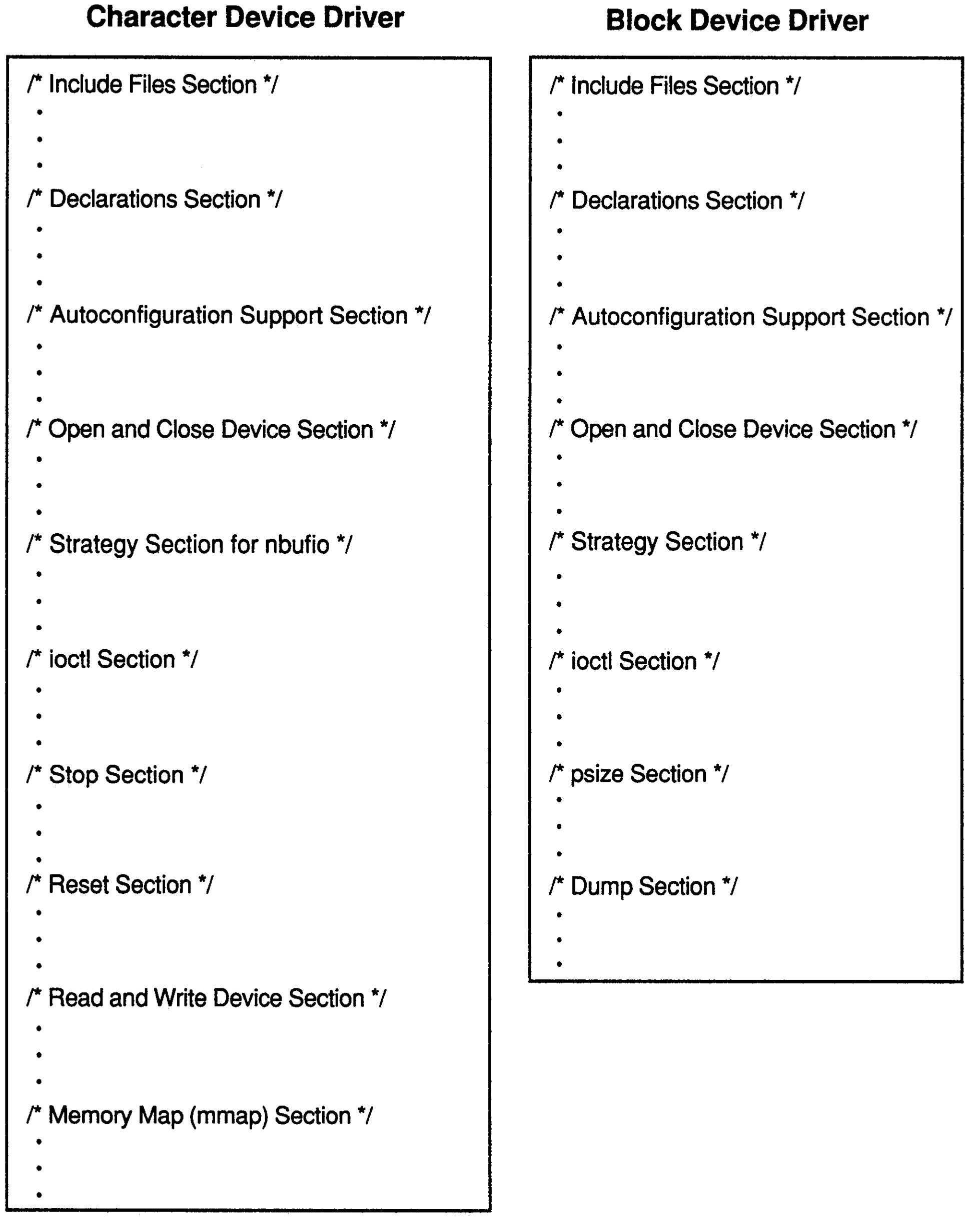
The character device driver contains a read and write device section. The block device driver does not contain either of these sections. Although raw block devices require a read and write device section, their driver entry points are specified through the `cdevsw`, not the `bdevsw`. In other words, the device driver for the raw block device is both a block and a character driver. When accessed as a block device, the system uses the driver's `strategy` routine as the entry point. When accessed as a character device, the driver's `read` and `write` routines are used as the entry points. (See Section 9.1.1 for descriptions of the `cdevsw` and `bdevsw` tables.) The character device driver can contain a reset section, a stop section, and a memory map (mmap) section. The block device driver does not contain any of these sections.

Note

> The `psize` routine is no longer used. It has been superseded by driver ioctl calls that are used to obtain disk geometry information. Previously, the routine determined the location on the disk where ULTRIX should perform a dump.
>
> ULTRIX supports dumping only to disks that it can boot from. In most cases, ULTRIX uses dump routines located in the console subsystem. Because ULTRIX does not support booting from a VMEbus disk, dumping to disk is not used in a VMEbus device.

Each device driver section is described following Figure 4-1.

Figure 4-1: Sections of a Character Device Driver and a Block Device Driver



```
/* Include Files Section */
.
.
.
/* Declarations Section */
.
.
.
/* Autoconfiguration Support Section */
.
.
.
/* Open and Close Device Section */
.
.
.
/* Strategy Section for nbufio */
.
.
.
/* ioctl Section */
.
.
.
/* Stop Section */
.
.
.
/* Reset Section */
.
.
.
/* Read and Write Device Section */
.
.
.
/* Memory Map (mmap) Section */
.
.
.
```

```
/* Include Files Section */
.
.
.
/* Declarations Section */
.
.
.
/* Autoconfiguration Support Section */
.
.
.
/* Open and Close Device Section */
.
.
.
/* Strategy Section */
.
.
.
/* ioctl Section */
.
.
.
/* psize Section */
.
.
.
/* Dump Section */
.
.
.
```

ZK-0204U-R

The conventions followed for the driver interface function definitions are similar to those used for the kernel routines in the way argument names and types are represented. However, there are differences in the way return types and names are represented in the driver function definitions. The differences in the conventions are illustrated in the following example:

int *vmeprobe*(*ctrl*, *addr1*, *addr2*)
int *ctrl*;

caddr_t *addr1*;
caddr_t *addr2*;

The driver interface function definition gives you this information:

- Return type

  Gives the data type of the return value, if the driver routine returns data. If the driver routine does not return data, no type appears.

- Driver routine name

  Gives the driver routine name. There are two variations on the name illustrated in the driver function definitions. First, if the driver interface differs according to the bus on which the driver operates, a bus-specific name is used. For example, the interface to a driver's `probe` routine differs according to whether the driver operates on the VMEbus or the TURBOchannel. Therefore, either the name *vmeprobe* or *turboprobe* is used.

  If the driver interface is the same regardless of the bus on which the driver operates, the name *anydrv* followed by the specific interface name is used. For example, the interface to a driver's `open` routine is the same regardless of the bus on which the driver operates. Therefore, the name *anydrvopen* is used.

Note the use of *italics* to indicate that the driver routine name is variable. When you write your driver routines, you should use the naming conventions described in Section 9.1.1.1.

4.1 Include Files Section

Data structures are defined in header files that the device driver writer includes in the driver source code. The following lists the header files most frequently used by any device driver, including VMEbus and TURBOchannel device drivers:

```
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h"
```

Device drivers should use relative path names, not explicit path names. For summary descriptions of the contents of the header files listed in this and subsequent sections, see Appendix A.

The header file `types.h` defines system data types used to declare members in the data structures referenced by device drivers. To store values in these structure members, the driver writer must declare the variable using the appropriate system data type, or cast the stored value. Table 4-1 lists the system data types most frequently used by device drivers.

Table 4-1: System Data Types Frequently Used by Device Drivers

Data Type	Meaning
`daddr_t`	Block device address
`caddr_t`	Main memory virtual address

Table 4-1: (continued)

Data Type	Meaning
`ino_t`	Inode index
`label_t`	Vector for `setjmp/longjmp`
`dev_t`	Device major and minor numbers
`off_t`	File offset
`paddr_t`	Main memory physical address
`time_t`	System time
`u_short`	unsigned short

4.1.1 Include Files for VMEbus Device Drivers

In order, the minimal header files needed by VMEbus device drivers are:

```
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h"
#include "../io/vme/vbareg.h"
```

Note that `vbareg.h` is used exclusively by VMEbus device drivers.

4.1.2 Include Files for TURBOchannel Device Drivers

In order, the minimal header files needed by TURBOchannel device drivers are:

```
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h"
#include "../io/tc/tc.h"
```

Note that `tc.h` is used exclusively by TURBOchannel device drivers.

4.2 Declarations Section

The declarations section of a block or character device driver contains:

- Variable and structure declarations
- Definitions of symbolic names
- Declarations of the specific driver routines

The following example illustrates the declarations section of a VMEbus device driver:

```
 .
 .
 .
 /* Symbolic definitions */
```

```
#define SKREGSIZE 256 /* First csr area */
#define SKUNIT(dev) (minor(dev)) /* Device minor number */
.
.
.
/* Structure and variable declarations */

struct uba_device *skdinfo[NSK];
.
.
.
/* Driver routines declarations */
int skprobe(), skattach(), skintr(), skmmap();
.
.
.
```

The following variables or data structures should be declared as volatile by VMEbus and TURBOchannel device drivers:

- Any variable or data structure that can be changed by a controller or processor other than the system CPU
- Variables that correspond to hardware device registers
- Any variable or data structure shared with a controller or coprocessor

When declaring a variable or data structure as volatile, use the compiler key word `volatile` in the declaration. For example:

```
volatile int hrdwrereg;
struct register_for_some_device {
           volatile char stub_0;     /* Base address */
           volatile char V;          /* First readable, always V */
           volatile char stub_1;     /* Data is only on every other word */
           volatile char M;          /* Second readable */
};
```

4.3 Autoconfiguration Support Section

The autoconfiguration support section applies to both character and block device drivers. It can contain:

- A `probe` routine
- A `slave` routine
- An `attach` routine

You define the entry point for each of these routines in the `uba_driver` structure. See Section 5.1.4 for a description of this structure.

Each of these routines is discussed in the following sections.

4.3.1 The Probe Routine

A device driver's `probe` routine performs all the tasks necessary to determine if the device exists and is functional on a given system. At boot time, the kernel performs checks to determine if the device is present before calling the `probe` routine. The kernel calls the `probe` routine for each device that was defined in the system configuration file.

The `probe` routine typically checks some device status register to determine whether the physical device is present. To perform this check, the `probe` routine calls the `BADADDR` macro. If the device is not present, the device is not initialized and not available for use. The `probe` routine returns the size of the control/status register address space for the autoconfiguration routines to use.

The interface to the `probe` routine differs according to the bus on which the driver operates. Therefore, the interfaces to the `probe` routine for the VMEbus and the TURBOchannel are discussed separately.

4.3.1.1 Probe Routine Interface for VMEbus Driver

4.3.1.1 Probe Routine Interface for VMEbus Driver – For VMEbus device drivers, the interface to the `probe` routine is expressed in the following function definition:

int *vmeprobe*(*ctrl*, *addr1*, *addr2*)
int *ctrl*;
caddr_t *addr1*;
caddr_t *addr2*;

ctrl	Specifies the controller or device number associated with this device. You specified this number in the system configuration file.
addr1	Specifies the System Virtual Address (SVA) for the device. This SVA corresponds to the first CSR address that you specified for the device in the system configuration file.
addr2	Specifies the System Virtual Address (SVA) for the onboard memory. This SVA corresponds to the second CSR address, if present, that you specified in the system configuration file. If you did not specify a second CSR address, the value of this argument is zero (0).

See Section 9.1.3.3 for information on how to specify a controller's name and logical unit number and the first and second CSR addresses in the system configuration file. See Section 9.1.3.5 for information on how to specify a device's name and logical unit number and the first and second CSR addresses in the system configuration file.

4.3.1.2 Probe Routine Interface for TURBOchannel Driver – For TURBOchannel device drivers, the interface to the `probe` routine is expressed in the following function definition:

turboprobe(*addr*, *ctrl*)
caddr_t *addr*;
struct uba_ctlr * *ctrl*;

addr	Specifies the System Virtual Address (SVA) control/status registers for the device.
ctrl	Specifies a pointer to a `uba_ctlr` or a pointer to a `uba_device` structure. (The function definition shows a pointer to a `uba_ctlr` structure.)

4.3.2 The Slave Routine

A device driver's `slave` routine is called only for controller devices. This routine is called once for each slave attached to the controller. You specify the attachments of these slave devices in the system configuration file. The interface to the `slave` routine differs according to the bus on which the driver operates. Therefore, the interfaces to the `slave` routine for the VMEbus and the TURBOchannel are discussed separately.

4.3.2.1 Slave Routine Interface for VMEbus Driver – For VMEbus device drivers, the interface to the `slave` routine is expressed in the following function definition:

vmeslave(*ui*, *addr1*, *addr2*)
struct uba_device * *ui*;
caddr_t *addr1*;
caddr_t *addr2*;

ui	Specifies a pointer to a `uba_device` structure. This structure contains such information as the logical unit number of the device, whether the device is functional, the bus number the device resides on, the address of the control/status registers, and so forth. See Section 5.1.6 for more information on this structure.
addr1	Specifies the System Virtual Address (SVA) for the device. This SVA corresponds to the first CSR address that you specified for the device in the system configuration file.
addr2	Specifies the System Virtual Address (SVA) for the onboard memory. This SVA corresponds to the second CSR address, if present, that you specified in the system configuration file. If you did not specify a second CSR address, the value of this argument is zero (0).

See Section 9.1.3.3 for information on how to specify the first and second CSR addresses in the system configuration file.

4.3.2.2 Slave Routine Interface for TURBOchannel Driver – For TURBOchannel device drivers, the interface to the `slave` routine is expressed in the following function definition:

turboslave(*ui*, *reg*)
struct uba_device**ui*;
caddr_t *reg*;

ui	Specifies a pointer to a `uba_device` structure. This structure contains such information as the logical unit number of the device, whether the device is functional, the bus number the device resides on, the address of the control/status registers, and so forth. See Section 5.1.6 for more information on this structure.
reg	Specifies the System Virtual Address (SVA) control/status registers for the device.

4.3.3 The Attach Routine

The `attach` routine usually performs the tasks necessary in establishing communication with the actual device. At boot time, this routine is called by the autoconfiguration code under the following conditions:

- If the device is connected to a controller, the `attach` routine is called if the controller's `slave` routine returns a nonzero value, indicating that the device exists.

- If the device is not connected to a controller, the `attach` routine is called if the `probe` routine returns a nonzero value, indicating that the device exists.

The `attach` routine is passed a `uba_device` structure for this device.

The tasks performed by the `attach` routine may include initializing a tape drive, putting a disk drive on line, or some other similar action. In addition, the `attach` routine initializes any global data structures used by the driver. This routine need not return a value. The interface to the `attach` routine is the same regardless of the bus on which the driver operates.

For VMEbus and TURBOchannel device drivers, the interface to the `attach` routine is expressed in the following function definition:

anydrvattach(*ui*)
struct uba_device * *ui*;

ui	Specifies a pointer to a `uba_device` structure. This structure contains such information as the logical unit number of the device, whether the device is functional, the bus number the device resides on, the address of the control/status registers, and so forth. See Section 5.1.6 for more information on this structure.

4.4 Open and Close Device Section

The open and close device section applies to both character and block device drivers. It contains:

- An `open` routine
- A `close` routine

You define the entry point for a driver's `open` and `close` routines in the `cdevsw` table for character devices and the `bdevsw` table for block devices. See Section 9.1.1 for descriptions of the `cdevsw` and `bdevsw` tables.

Each of these routines is discussed in the following sections.

4.4.1 The Open Routine

A device driver's `open` routine is called when a process opens a special device file whose major device number serves as an index into either the `cdevsw` or `bdevsw` table. You specify the entry for the driver's `open` routine in the `cdevsw` for character device drivers and the `bdevsw` for block device drivers.

A block device driver's `open` routine opens a device to prepare it for I/O operations. This routine usually verifies that the device was identified during autoconfiguration. For tape devices, this identification may consist of bringing the device on line and selecting the appropriate density.

A character device driver's `open` routine performs similar tasks to those performed by the block device driver. If the character device provides raw access to a block device, the `open` routine is usually the same. Almost all character device drivers provide an `open` routine; however, some block devices do not require this routine. For terminal devices, the `open` routine may block waiting for the necessary modem signals, for example, carrier detect.

Other tasks performed by the `open` routine for a block or a character device driver are to:

- Determine the logical unit number from the minor device number
- Check that the logical unit number is that of a valid device that is functional.
- Check the state of the device or the *flag* argument if the device is to be an exclusive open, that is, nonblocking open, read-only, or write-only
- Start any device bookkeeping activities, for example, by setting any software flags and state variables

The return status of the `open` routine will eventually be the return status from the `open` system call. The interface to the `open` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `open` routine is expressed in the following function definition:

int *anydrvopen*(*dev*, *flag*)
dev_t *dev*;
int *flag*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device that is to be opened.
flag	Specifies the access mode of the device. The access modes are represented by flag constants defined in `/usr/sys/h/file.h`. The following describes some flag constants that you can pass to this argument:

Value	**Meaning**
`O_RDONLY`	The device is open for reading.
`O_RDWR`	The device is open for reading and writing.
`O_WRONLY`	The device is open for writing.

4.4.2 The Close Routine

A device driver's `close` routine is called when the last file descriptor that is open and associated with this device is closed via the `close` system call. A block device driver's `close` routine closes a device that was previously opened by the `open` routine. This routine is called only after making the final open reference to the device.

A character device driver's `close` routine performs similar tasks to those performed by the block device driver. If the character device provides raw access to a block device, the `close` routine is usually the same. Almost all character device drivers provide a `close` routine; however, some block devices do not require this routine.

Other tasks performed by the `close` routine for a block or a character device driver are to:

- Determine the logical unit number for this device from the minor device number
- Turn off interrupts for the device
- Clean up the software state and flag

The interface to the `close` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `close` routine is expressed in the following function definition:

anydrvclose(*dev*, *flag*)
dev_t *dev*;
int *flag*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device that is to be closed.
flag	Specifies the access mode of the device. The access modes are represented by flag constants defined in `/usr/sys/h/file.h`. Typically, the `close` routine does not use this argument.

4.5 Read and Write Device Section

The read and write device section applies only to character device drivers. This section contains:

- A `read` routine
- A `write` routine

You define the entry point for a character driver's `read` and `write` routines in the `cdevsw` table. See Section 9.1.1 for a description of the `cdevsw` table.

Each of these routines is discussed in the following sections.

4.5.1 The Read Routine

A character device driver's `read` routine is called from the I/O system as the result of a `read` system call. The driver's `read` routine reads data from a device. If there is no data available, the `read` routine puts the calling process to sleep until data is available. If data is available, `read` copies it from the private kernel buffer to the user's process using the `uiomove` kernel routine.

In the case of raw block devices, the `read` routine calls the `physio` kernel routine, passing to it the device-specific parameters. For terminal-oriented devices, the driver passes the read request to the generic terminal interface read routine.

The `read` routine returns an error number to the process's `read` system call if there was a failure. Otherwise, it returns the number of bytes actually read.

The interface to the `read` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `read` routine is expressed in the following function definition:

int *anydrvread*(*dev*, *uio*)
dev_t *dev*;
struct uio * *uio*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device on which the read operation will be performed.

uio	Specifies a pointer to a `uio` structure. This structure contains the information for transferring data to and from the address space of the user's process. You typically pass this structure unchanged to the `uiomove` or `physio` routines. See Section 5.1.3 for information on the `uio` structure.

4.5.2 The Write Routine

A character device driver's `write` routine is called from the I/O system as the result of a `write` system call. A character device driver's `write` routine checks the software state of the device to determine if the device is in a state that permits the write operation. If not, `write` places the device into a writable state and writes data to the device. (Note that read/write permission is checked at the file system level, not in the device driver.)

If necessary, `write` allocates a private kernel buffer. It copies the data of the user process into the private kernel buffer using the `uiomove` kernel routine. It then sets up the software state of the device for the current output transfer and starts the hardware transferring the data. Following this, `write` puts the process to sleep and awakes it after all of the data in the current transfer has been sent to the device.

If the device is a raw block device, the `write` routine calls the `physio` kernel routine to accomplish the write. For terminal-oriented devices, the device driver passes the write request to the generic terminal interface write routine.

The `write` routine returns an error number to the process's `write` system call if there was a failure. Otherwise, it returns the number of bytes actually written.

The interface to the `write` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `write` routine is expressed in the following function definition:

int *anydrvwrite*(*dev*, *uio*)
dev_t *dev*;
struct uio * *uio*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device on which the write operation will be performed.
uio	Specifies a pointer to a `uio` structure. This structure contains the information for transferring data to and from the address space of the user's process. You typically pass this structure unchanged to the `uiomove` or `physio` routines. See Section 5.1.3 for information on the `uio` structure.

4.6 ioctl Section

The ioctl section applies to both character and block device drivers. This section contains an `ioctl` routine, which is a general purpose device control routine. This routine typically performs all device-related operations other than read or write operations. A device driver's `ioctl` routine is called as a result of an `ioctl` system call. Only those ioctl commands that are device-specific or that require action on the part of the device driver result in a call to the driver's `ioctl` routine.

You define the entry point for the driver's `ioctl` routine in the `cdevsw` for character device drivers and the `bdevsw` for block device drivers. See Section 9.1.1

for a description of the `cdevsw` and `bdevsw` tables.

Some of the device-related operations performed by the `ioctl` routine are to:

- Return device attributes and parameters in response to queries by user programs

  In general, all device drivers have an `ioctl` routine that identifies the device type, controller name, and other related parameters. For example, user programs may request information about disks, in which case the `ioctl` routine returns disk geometry information. For user program requests about terminal devices, the `ioctl` routine might return the current values of the terminal line attributes. User program requests about tape drives, on the other hand, can result in the return of such attributes as tape density. For more information, see `devio` in the *Reference Pages Section 4: Special Files.*

- Return the status of a device

  The device status for a tape drive, for example, might consist of the tape mark encountered, end of media encountered, positioning at the bottom of the tape, device is write protected, and so forth.

- Allow for the setting of device-related parameters

  The device settings for a terminal device, for example, may consist of baud rate, parity, and so forth. For disk drives, the partition-related information may be specified by the ioctl interface. For tape drives, the `ioctl` routine performs tape repositioning commands, such as rewinding and forward or backward skipping of tape marks and tape records.

The `ioctl` routine returns an error number if there was a failure; otherwise, it returns zero (0). This is the return value of the process's `ioctl` system call.

The interface to the `ioctl` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `ioctl` routine is expressed in the following function definition:

int *anydrvioctl*(*dev*, *cmd*, *data*, *flag*)
dev_t *dev*;
int *cmd*;
caddr_t *data*;
int *flag*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device on which the ioctl operation will be performed.
cmd	Specifies the ioctl command as specified in `/usr/sys/h/ioctl.h` or in another include file defined by the device driver writer. Many ioctl commands are handled by the I/O system and do not result in a call to the device driver's `ioctl` routine. However, when some commands require a device-specific action, this information is passed to the driver's `ioctl` routine. One of the values you can pass to this argument is `DEVIOCGET`. For information on the `DEVIOCGET` ioctl request, see Appendix B.
data	Specifies a pointer to ioctl command-specific data that is to be passed to the device driver, or filled in by the device driver. The particular ioctl command implicitly determines the action to be taken. The size of this data cannot exceed 128 bytes.

This argument is a kernel address. The `ioctl` system call performs all the necessary copy in and copy out operations by calling the `copyin` and `copyout` kernel routines.

flag Specifies the access mode of the device. The access modes are represented by flag constants defined in `/usr/sys/h/file.h`. The following describes some flag constants that you can pass to this member:

Value	Meaning
`O_RDONLY`	The device is open for reading.
`O_RDWR`	The device is open for reading and writing.
`O_WRONLY`	The device is open for writing.

4.7 Strategy Section

The strategy section applies to both character and block device drivers. This section contains a `strategy` routine, which initiates read and write operations. You define the entry point for a driver's `strategy` routine in the `cdevsw` table for character devices and in the `bdevsw` table for block devices. See Section 9.1.1 for descriptions of the `cdevsw` and `bdevsw` tables.

Typically this routine is not called directly from user-level programs; instead, the routine is called from different routines within the kernel. For the block driver, it is the `strategy` routine that implements the concept of disk partitions. Disk partitions involve subdividing the physical disk into smaller logical disk partitions. Through the use of partition tables that define partition boundaries, the `strategy` routine maps read and write requests to the correct disk offset.

The main user of the block device is the file system. File system reads and writes are usually handled through the kernel routines `bread` and `bwrite`. Through these routines and the routines that they call, the data is read from or written to the data cache. When the data being read is not present in the data cache, the block device `strategy` routine will be called to initiate a data transfer to read in the data from the disk. When a decision is made to flush the written data out of the data cache to the disk media, the block driver `strategy` routine is called to initiate the transfer.

For the character device driver, data transfer operations (reads and writes) are initiated by the driver's `read` and `write` routines. These routines will call the `strategy` routine indirectly to initiate the data transfer operation.

The interface to the `strategy` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `strategy` routine is expressed in the following function definition:

anydrvstrategy(*bp*)
struct buf * *bp*;

bp Specifies a pointer to a `buf` structure. This structure contains information such as binary status flags, the major/minor device numbers, the address of the associated buffer, and so forth. See Section 5.1.1 for more information on the `buf` structure.

4.8 Stop Section

The stop section applies only to character device drivers and it contains a `stop` routine. The `stop` routine is used by terminal device drivers to suspend transmission on a specified line. You define the entry point for a character driver's `stop` routine in the `cdevsw` table. See Section 9.1.1 for a description of the `cdevsw` table.

The `stop` routine is called when the terminal driver has recognized a stop character such as ^S. There are also specific ioctl calls that request output on a terminal line be suspended. These ioctl calls result in the general terminal driver interface calling the associated device driver's `stop` routine.

The interface to the `stop` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `stop` routine is expressed in the following function definition:

anydrvstop(*tp*, *flag*)
struct tty * *tp*;
int *flag*;

tp	Specifies a pointer to a `tty` structure. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth.
flag	Specifies whether the output is to be flushed or suspended. ULTRIX device drivers do not use this argument. However, the pseudo-terminal driver does use this field for its own purposes. The argument is included here for use in your terminal drivers.

4.9 Reset Section

The reset section applies only to character device drivers and it contains a `reset` routine. You define the entry point for a character driver's `reset` routine in the `cdevsw` table. See Section 9.1.1 for a description of the `cdevsw` table.

The `reset` routine is used to force a device reset to place the device in a known state after a bus reset. The bus adapter support routines call the `reset` routine after completion of a bus reset.

For a terminal device driver, the `reset` routine may consist of reenabling interrupts on all open lines and resetting the line parameters for each open line. Following a reset of terminal state and line attributes, transmission may resume on the terminal lines.

The interface to the `reset` routine is the same regardless of the bus on which the driver operates. Note, however, that the reset section would not be used by VMEbus and TURBOchannel device drivers. The interface to the `reset` routine is expressed in the following function definition:

anydrvreset(*busnum*)
int *busnum*;

busnum	Specifies the logical unit number of the bus on which the bus reset occurred.

4.10 Interrupt Section

The interrupt section applies to both character and block device drivers and it contains an `interrupt` routine. You define a driver's `interrupt` routine or routines in the device definitions part of the system configuration file when you define controllers and devices. The following sections describe the specification of `interrupt` routines in the system configuration file:

- Section 9.1.3.3

  Describes the controller specification for controllers associated with the VMEbus

- Section 9.1.3.4

  Describes the controller specification for controllers associated with the TURBOchannel

- Section 9.1.3.5

  Describes the device specification for devices that run on the VMEbus

- Section 9.1.3.6

  Describes the device specification for devices that run on the TURBOchannel

ULTRIX fields interrupts from devices and dispatches the appropriate device driver interrupt routine to service the interrupt. Typically, interrupt service routines handle the transfer of data to and from a device. On output, the `interrupt` routine may be called to notify the completion of a Direct Memory Access (DMA) output request. Similarly on input, the `interrupt` routine is called when there is input data available from the device.

The `interrupt` routine may also be called for device status reporting purposes. These events may be caused by the generation of device-specific errors. For terminal devices, the `interrupt` routine may be called to report transitions of modem signals.

The interface to the `interrupt` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `interrupt` routine is expressed in the following function definition:

anydrvinterrupt(*unit*)
int *unit*;

unit	Specifies the logical unit number of the controller or device that is interrupting. You specified this logical unit number in the system configuration file. This logical unit number is used as an index into the driver's data structures to obtain per-device state and information. See Section 9.1.3.3 for information on how to specify a controller's name and logical unit number and the first and second CSR addresses in the system configuration file. See Section 9.1.3.5 for information on how to specify a device's name and logical unit number and the first and second CSR addresses in the system configuration file.

4.11 Select Section

The select section applies only to character device drivers, and it contains a `select` routine. You define the entry point for a character driver's `select` routine in the `cdevsw` table. See Section 9.1.1 for a description of the `cdevsw` table.

The `select` routine determines whether data is available for reading and whether space is available for writing data. The `select` system call is most frequently associated with terminal devices. The `select` system call is used to determine that there are characters available in the terminal input queue for reading. This system call is also used to indicate that there is available space in the terminal's output queue to accept bytes to be output to the terminal device. For most terminal device drivers, the `select` routine is implemented by the general kernel terminal interface select routine called `ttselect`.

For nonterminal type character devices that do not support nbufio, the `select` routine is implemented by the kernel routine `seltrue`, which returns true for any select request. In this situation, the `select` routine returns true because all transfers are synchronous operations and it should always be possible to read and write to the device.

For nonterminal type character devices that do support nbufio, the `select` routine is implemented by the kernel routine `asyncsel`. This is applicable to disk and tape drivers. The `asyncsel` routine returns a value of 1 to indicate that there is a nonbusy buffer available for reading or writing purposes. If all buffers used for nbufio are presently busy, the `asyncsel` routine returns zero (0) to note this busy status.

The interface to the `select` routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the `select` routine is expressed in the following function definition:

anydrvselect(*dev*, *rwflag*)
dev_t *dev*;
int *rwflag*;

dev	Specifies the major and minor device numbers for this device. The minor device number is used to determine the logical unit number for the device on which the select operation will be performed.
rwflag	Specifies the read/write flag. You can set the *rwflag* argument to one of these constants:

Value	**Meaning**
`FREAD`	Select on input data
`FWRITE`	Select on device being ready to accept more output

4.12 Memory Map Section

The memory map section applies only to character device drivers and it contains an `mmap` routine. You define the entry point for a character driver's `mmap` routine in the `cdevsw` table. See Section 9.1.1 for a description of the `cdevsw` table.

A device driver's memory map routine is invoked by the kernel as the result of an application calling the `mmap` system call. An application calls `mmap` to map a character device's memory into user address space. The user address space is inherited on a fork and is unmapped automatically on a process exit or `exec`. (An application can also explicitly unmap a previously mapped device memory by calling the `munmap` system call. See the *Reference Pages Section 2: System Calls* for descriptions of the `mmap` and `munmap` system calls.)

You need to consider the following when writing a memory map routine for your driver:

- The interface to the memory map routine
- Mapping to nonexistent memory
- Reading from nonexistent memory
- Writing to nonexistent memory

Each of these considerations is discussed in the following sections.

4.12.1 The Memory Map Routine

The interface to the memory map routine is the same regardless of the bus on which the driver operates. For VMEbus and TURBOchannel device drivers, the interface to the memory map routine is expressed in the following function definition:

int *anydrvmmap*(*dev*, *off*, *prot*)
dev_t *dev*;
off_t *off*;
int *prot*;

dev	Specifies the major and minor device number for this device. The minor device number is used to determine the logical unit number for the character device whose memory is to be mapped.
off	Specifies the offset in bytes into the character device's memory. The offset must be a valid offset into device memory.
prot	Specifies the protection flag for the mapping. The protection flag is the bitwise inclusive OR of these valid protection flag bits defined in `/usr/sys/h/mman.h`:

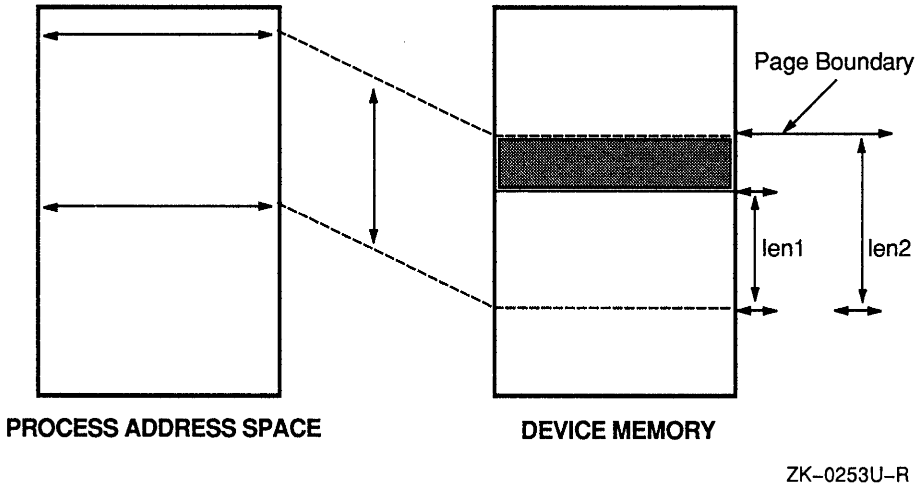
Value	**Meaning**
`PROT_READ`	Pages can be read
`PROT_WRITE`	Pages can be written

The memory map routine, if successful, returns the page frame number corresponding to the page at the byte offset specified by the *off* argument. Otherwise, the memory map routine returns –1.

4.12.2 Mapping to Nonexistent Memory

Using the memory map interface, a user process can map nonexistent device memory into its address space. One way this can occur is when the device memory being mapped does not begin or end on a page boundary, as illustrated in Figure 4-2.

Figure 4-2: Mapping Nonexistent Device Memory



The figure shows the following:

- The address space of the calling process where the device memory is to be mapped.
- The memory for some character device. The len1 symbol represents the number of bytes the calling process wants to map into its address space. However, the number of bytes that is actually mapped is represented by the len2 symbol and includes the shaded area. This shaded area can be nonexistent device memory or it can belong to another device. The reason that len2 bytes get mapped is that the requested length (len1) does not begin and end on a page boundary.

A second way that a user process can map nonexistent device memory into its address space is by making a single call to the `mmap` system call to map both CSRs and device memory. However, if the CSRs and the device memory are not contiguous, nonexistent memory can be mapped.

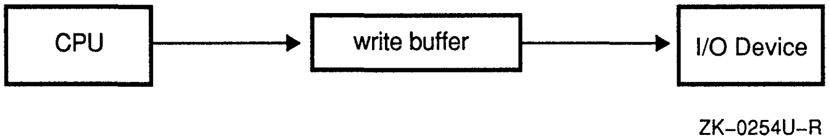
4.12.3 Reading from Nonexistent Memory

When a user process initiates a read from nonexistent device memory, the kernel delivers synchronously to this process a `SIGBUS` (bus error) signal. The default action of the `SIGBUS` signal is to terminate (kill) the process that initiated the read.

4.12.4 Writing to Nonexistent Memory

The way writes to nonexistent memory are dealt with is machine-dependent. On some hardware architectures, including Digital RISC, a write to I/O space is buffered by hardware as illustrated in Figure 4-3.

Figure 4-3.: Writes to I/O Space on Digital RISC Architecture



On such architectures, a write to nonexistent memory has the following characteristics:

- The hardware generates a bus timeout.
- The bus timeout is asynchronous to the user process initiating the write.
- The hardware provides only minimal state information, namely the physical address at which the timeout occurred.
- The hardware does not provide any information on whether the timeout was caused by a kernel or user reference.

An ideal policy for dealing with bus timeouts is the following:

- If a timeout is caused by a user reference, the kernel machine check code locates and kills the process that initiated the write.
- If a timeout is caused by a kernel reference, the kernel machine check code crashes the processor. A kernel access can arise from the device driver, as noted.

This policy cannot be implemented because:

- The hardware provides only the physical address at which the timeout occurred. And, since the physical address can be mapped by more than one process, it is impossible to determine the exact process that caused the timeout.
- A kernel write cannot be distinguished from a write by a user level process.

Because of these restrictions, ULTRIX uses the following policy. First, an attempt is made to kill all the processes that map the physical address, not just the process that caused the timeout. If no such processes are found, the write is assumed to originate from the kernel, and the kernel machine check code crashes the machine.

Part III: Data Structures, Kernel Routines, and Autoconfiguration

Data Structures Used by Device Drivers 5

Data structures are the mechanism used to pass information between the ULTRIX kernel and device driver routines. Because device drivers written for devices connected to the VMEbus or TURBOchannel are structured like UNIBUS or Q-bus drivers, they use some of the same structures. This chapter describes the existing ULTRIX structures pertinent to VMEbus and TURBOchannel device drivers. In addition, the chapter describes newly defined structures used exclusively by VMEbus device drivers.

Specifically, the chapter discusses the following:

- Data structures used by both VMEbus and TURBOchannel device drivers
- Data structures used only by VMEbus device drivers

5.1 Data Structures Used by VMEbus and TURBOchannel Device Drivers

The data structures discussed in this section are used in I/O operations. Any device driver, including VMEbus and TURBOchannel device drivers, can reference these structures. The data structures used in I/O are as follows:

- `buf`
- `file`
- `uio`

The section also discusses the following UNIBUS data structures used by VMEbus and TURBOchannel drivers:

- `uba_driver`
- `uba_ctlr`
- `uba_device`

5.1.1 The buf Structure

The `buf` structures describe arbitrary I/O, but are usually associated with block I/O and `physio`. A systemwide pool of `buf` structures exists for block I/O; however, many device drivers also include locally defined `buf` structures. Table 5-1 lists the members of the `buf` structure that a device driver can reference.

Table 5-1: Members of the buf Structure Used by Device Drivers

Member Name	Data Type	Description
`b_flags`	`long`	Specifies binary status flags.
`b_forw`	`struct buf *`	Specifies a hash chain.
`b_back`	`struct buf *`	Specifies a hash chain.
`av_forw`	`struct buf *`	Specifies the position on the free list if the `b_flags` member is not set to `B_BUSY`.
`av_back`	`struct buf *`	Specifies the position on the free list if the `b_flags` member is not set to `B_BUSY`.
`b_bcount`	`long`	Specifies the size of the requested transfer, in bytes.
`b_error`	`short`	Specifies that an error occurred on this data transfer.
`b_dev`	`dev_t`	Specifies the major/minor device number.
`b_blkno`	`daddr_t`	Specifies the block number on the partition of a disk.
`b_addr`	`caddr_t`	Specifies the address of the associated buffer.
`b_resid`	`long`	Specifies the data (in bytes) not transferred because of some error.
`b_iodone`	`int (*b_iodone) ()`	Specifies the routine called by `iodone`.

The following explains some of these members in more detail.

b_flags

The `b_flags` member contains binary status flags. These flags indicate how a request is to be handled and the current status of the request. The following flags are applicable to device drivers:

Flag	Meaning
`B_READ`	This flag is set if the operation is read and cleared if the operation is write.
`B_DONE`	This flag is cleared when a request is passed to a driver `strategy` routine. The device driver writer must set this flag when the operation has been completed or aborted.
`B_ERROR`	Specifies that an error occurred on this data transfer.
`B_BUSY`	This flag indicates that the buffer is in use.
`B_PHYS`	This flag indicates that the associated data is in user address space.

Flag	Meaning
`B_WANTED`	If this flag is set, it indicates that some process is waiting for this buffer. The device driver should issue a call to the `wakeup` kernel routine when the buffer is freed by the current process, passing the address of the buffer as an argument to it.

av_forw and av_back

The `av_forw` and `av_back` members specify the position on the free list if the `b_flags` member is not set to `B_BUSY`. If `b_flags` is set to `B_BUSY`, a device driver can use the `av_forw` and `av_back` members for other purposes besides queueing.

b_error and b_resid

The `b_error` member specifies that an error occurred on this data transfer. The `b_resid` member specifies the data (in bytes) not transferred because of some error. When a data transfer does not complete, the device driver should do the following:

- Set the error code in `b_error` to one of the values defined in `/usr/sys/h/errno.h`.
- Set the `b_resid` member to the number of bytes that could not be transferred.
- Set the flag `B_ERROR` in the `b_flags` member.

b_dev

The `b_dev` member specifies the major/minor device number. Device drivers often use the minor number to select one unit or drive when several are attached to the identical controller. You can use the `major` and `minor` macros to obtain the major and minor number. See Appendix B for descriptions of these macros.

b_iodone

The `b_iodone` member specifies the routine called by `iodone`. The driver routine calls the `iodone` routine when a data transfer completes. The `iodone` routine then calls the routine pointed to by the `b_iodone` member.

5.1.2 The file Structure

There is one `file` structure for each open file in the system. ULTRIX allocates and initializes this `file` structure when a file is opened. Table 5-2 lists the member of the `file` structure that a device driver can reference.

Table 5-2: Member of the file Structure Used by Device Drivers

Member Name	Data Type	Description
`f_flag`	`int`	Specifies file descriptors associated with the open file. These descriptors are represented by constants defined in `/usr/sys/h/file.h`.

5.1.3 The uio Structure

The `uio` structure describes I/O, either single vector or multiple vectors. Table 5-3 lists the members of the `uio` structure that a device driver can reference. Typically, device drivers do not manipulate the members of this structure. However, they are presented here for the purpose of understanding the `uiomove` kernel routine, which operates on the members of the `uio` structure.

Table 5-3: Members of the uio Structure Used by Device Drivers

Member Name	Data Type	Description
`uio_iov`	`struct iovec *`	Specifies a pointer to the first `iovec` structure. The `iovec` structure has two members: one that specifies the address of the segment and the other that specifies the size of the segment. The system allocates these `iovecs` contiguously.
`uio_iovcnt`	`int`	Specifies the number of `iovec` structures.
`uio_offset`	`int`	Specifies the offset within the file.
`uio_segflg`	`int`	Specifies the value that indicates the segment type. This member can be set to one of these values: `UIO_USERSPACE` (the segment is from the user data space); `UIO_SYSSPACE` (the segment is from the system space); or `UIO_USERISPACE` (the segment is from the user I space).
`uio_resid`	`int`	Specifies the number of bytes that still need to be transferred.
`uio_flag`	`int`	Contains file descriptor flags associated with the file for this I/O operation. This member gets set by `read` and `write` system calls according to the corresponding field in the file descriptor. Possible values are contained in `/usr/sys/h/file.h`.

5.1.4 The uba_driver Structure

The `uba_driver` structure is used by ULTRIX to probe a device and to tie device driver code to ULTRIX code. The device driver writer must correctly initialize the members of this structure in the device driver code. Table 5-4 lists the members of the `uba_driver` structure that a device driver can reference.

Table 5-4: Members of the uba_driver Structure Used by Device Drivers

Member Name	Data Type	Description
`ud_probe`	`int (*ud_probe) ()`	Specifies a pointer to the driver's `probe` routine.
`ud_slave`	`int (*ud_slave) ()`	Specifies a pointer to a `slave` routine located within the device driver.
`ud_attach`	`int (*ud_attach) ()`	Specifies a pointer to an `attach` routine located within the device driver.
`ud_dgo`	`int (*ud_dgo) ()`	Specifies a pointer to a `go` routine located within the device driver. This routine is not used by VMEbus and TURBOchannel device drivers.
`ud_addr`	`u_short *`	Specifies the device's CSR address. This member is not used by VMEbus and TURBOchannel device drivers.
`ud_dname`	`char *`	Specifies the name of the device.
`ud_dinfo`	`struct uba_device **`	Specifies an array of pointers to `uba_device` structures accessed by this device driver. This array is indexed with the unit number, as specified in the `ui_unit` member of the `uba_device` structure.
`ud_mname`	`char *`	Specifies the name of the controller.
`ud_minfo`	`struct uba_ctlr **`	Specifies an array of pointers to `uba_ctlr` structures accessed by this device driver. This array is indexed with the controller number as specified in the `um_ctlr` member of the `uba_ctlr` structure.
`ud_xclu`	`short`	Specifies the driver's need to exclusively use buffer data paths (bdps). This member is not used by VMEbus device drivers.

Table 5-4: (continued)

Member Name	Data Type	Description
`ud_addr1_size`	`int`	Specifies the size in bytes of the first CSR area. This area is usually the control status register of the device.
`ud_addr1_atype`	`int`	Specifies the address space and data size of the first CSR area.
`ud_addr2_size`	`int`	Specifies the size in bytes of the second CSR area. This area is usually the data area and is used with devices that have two separate CSR areas.
`ud_addr2_atype`	`int`	Specifies the address space and data size of the second CSR area.

You can set the `ud_addr1_atype` and `ud_addr2_atype` members to the bitwise inclusive OR of:

- One of the nine address space and data size constants
- One of the four byte swapping constants

These constants appear in this table:

Value	Meaning
`VMEA16D16`	Specifies a request for the 16-bit address space and the 16-bit data size.
`VMEA16D32`	Specifies a request for the 16-bit address space and the 32-bit data size.
`VMEA24D08`	Specifies a request for the 24-bit address space and the 8-bit data size.
`VMEA24D16`	Specifies a request for the 24-bit address space and the 16-bit data size.
`VMEA24D32`	Specifies a request for the 24-bit address space and the 32-bit data size.
`VMEA32D08`	Specifies a request for the 32-bit address space and the 8-bit data size.
`VMEA32D16`	Specifies a request for the 32-bit address space and the 16-bit data size.
`VMEA32D32`	Specifies a request for the 32-bit address space and the 32-bit data size.
`VME_BS_NOSWAP`	Specifies no byte swapping.
`VME_BS_BYTE`	Specifies byte swapping in bytes.
`VME_BS_WORD`	Specifies byte swapping in words.

Value	Meaning
`VME_BS_LWORD`	Specifies byte swapping in long words.

You need to declare and initialize a `uba_driver` structure in your device driver, so you need to be more familiar with this structure than with other structures discussed in this chapter. The `uba_driver` structure declaration is as follows:

```
struct uba_driver {
  int     (*ud_probe)();
  int     (*ud_slave)();
  int     (*ud_attach)();
  int     (*ud_dgo)();
  u_short *ud_addr;
  char    *ud_dname;
  struct  uba_device **ud_dinfo;
  char    *ud_mname;
  struct  uba_ctlr **ud_minfo;
  short   ud_xclu;
  int     ud_addr1_size;
  int     ud_addr1_atype;
  int     ud_addr2_size;
  int     ud_addr2_atype;
};
```

The following example shows the declaration of a `uba_driver` structure for a VMEbus device driver:

```
  .
  .
  .
struct uba_driver xxdriver = {xxprobe, 0, 0, 0, 0, "xx", xxdinfo, "0",
                              NULL, 0, 0x20, VMEA16D16, 0,0};
  .
  .
  .
```

In the example code, the `xxdriver` structure members are initialized as follows:

- The `ud_probe` member is initialized to a probe routine called `xxprobe`.
- The `ud_slave`, `ud_attach`, and `ud_dgo` members are initialized to zero (0), because this driver does not use any of these routines.
- The `ud_addr` member is initialized to zero (0), because this member is not used by VMEbus device drivers.
- The `ud_dname` member is initialized to the name of the device, which is `xx`.
- The `ud_dinfo` member is initialized to the name of the pointer to an array of `uba_device` structures, which is `xxdinfo`.
- The `ud_mname` member is initialized to zero (0) because there is no controller for this device.
- The `ud_minfo` member is initialized to NULL, because this device driver does not reference any information in the `uba_ctlr` structures.

- The `ud_xclu` member is initialized to zero (0), because this member is not used by VMEbus device drivers.
- The `ud_addr1_size` member is initialized to the size of the first CSR area, which is 0x20 bytes.
- The `ud_addr1_atype` member is initialized to the address space and data size of the first CSR area, which is the constant `VMEA16D16`. This constant represents the A16 address space and a 16-bit data size.
- The `ud_addr2_size` is initialized to zero (0), because it is not used by this device driver.
- The `ud_addr2_atype` is initialized to zero (0), because it is not used by this device driver.

This example shows the declaration of a `uba_driver` structure for a TURBOchannel device driver:

```
  .
  .
  .
struct  uba_driver qacdriver =
        { qacprobe, 0, qacattach, 0, qacstd, "qac", qacinfo };
  .
  .
  .
```

In the example code, the `qacdriver` structure members are initialized as follows:

- The `ud_probe` member is initialized to a probe routine called `qacprobe`.
- The `ud_slave` member is initialized to zero (0), because this driver does not use a slave routine.
- The `ud_attach` member is initialized to an attach routine called `qacattach`.
- The `ud_dgo` member is initialized to zero (0), because this driver does not use a go routine.
- The `ud_addr` member is initialized to `qacstd`, which is an array of type `u_short`. The `qacstd` declaration is as follows:

  ```
    .
    .
    .
  u_short qacstd []={0};
    .
    .
    .
  ```

  This declaration indicates that the field must be filled in with the address of an array. The array has just one zero entry to indicate that this member is not used.
- The `ud_dname` member is initialized to the name of the device, which is `qac`.
- The `ud_dinfo` member is initialized to the name of the `uba_device` structure declared in this device driver, which is `qacinfo`.

5.1.5 The uba_ctlr Structure

The `uba_ctlr` structure contains members that store hardware resources information and commands for communication between ULTRIX and the device driver. The following describes characteristics of the `uba_ctlr` structure pertinent to device driver writers:

- Each `uba_ctlr` structure contains a back pointer to a bus header structure. For the VMEbus, the bus header structure is `vba_hd`.
- Each `uba_ctlr` structure contains at least one System Virtual Address (SVA) of the device CSRs in onboard memory.

Table 5-5 lists the members of the `uba_ctlr` structure that a device driver can reference. Note that `config` generates the values for members from `um_driver` to `um_ivnum` from information provided in the system configuration file.

Table 5-5: Members of the uba_ctlr Structure Used by Device Drivers

Member Name	Data Type	Description
`um_driver`	`struct uba_driver *`	Specifies a back pointer to a `uba_driver` structure.
`um_ctlrname`	`char *`	Specifies the name of the controller.
`um_ctlr`	`short`	Specifies the controller index into the device driver, for example, td0.
`um_adpt`	`int`	Specifies the adapter number (consecutive adapter number).
`um_nexus`	`short`	Specifies the nexus on the I/O bus that the controller is on.
`um_rctlr`	`short`	Specifies the remote controller number.
`um_ubanum`	`short`	Specifies the uba number the controller is on.
`um_vbanum`	`short`	Specifies the VMEbus adapter number as specified in the system configuration file. For example, a VMEbus entry would have these specifications: vba0, vba1, vba2, and so forth. (Note that this member stores only the VMEbus adapter number.)
`um_alive`	`short`	Specifies whether the controller exists. The value 1 indicates the controller exists and the value zero (0) indicates the controller does not exist.
`um_intr`	`int (**um_intr) ()`	Specifies an array of interrupt handlers. These interrupt handlers are called when the device generates interrupts.

Table 5-5: (continued)

Member Name	Data Type	Description
`um_addr`	`caddr_t`	Specifies the System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file.
`um_addr2`	`caddr_t`	Specifies the System Virtual Address (SVA) corresponding to the second CSR specified in the system configuration file.
`um_bus_priority`	`int`	Specifies the configured VMEbus priority level of the device.
`um_ivnum`	`int`	Specifies the first configured VMEbus device interrupt vector number for this device.
`um_priority`	`int`	Specifies the main bus request level of the VMEbus device. Device drivers use this member for synchronizing (through the `splx` kernel routine) to the corresponding VMEbus devices and in blocking out interrupts.
`um_physaddr`	`caddr_t`	Specifies the physical address of the device in I/O space. This member corresponds to the member that stores the SVA, `um_addr`.
`um_hd`	`struct uba_hd *`	Specifies a back pointer to a `uba_hd` structure.
`um_vbahd`	`struct vba_hd *`	Specifies a back pointer to a `vba_hd` structure.
`um_ubinfo`	`int`	Saves the UNIBUS or VMEbus mapping register information.
`um_tab`	`struct buf`	Specifies a `buf` structure used as a queue of devices for this controller and a queue for pending transfers.

5.1.6 The uba_device Structure

The `uba_device` structure has the following characteristics pertinent to device driver writers:

- There is one `uba_device` structure for each data device. The device can be a slave or a pure device.
- Each `uba_device` structure contains back pointers to `uba_hd`, `uba_ctlr`, `uba_driver`, and `vba_hd` (for VMEbus) structures.
- Each `uba_device` structure contains at least one System Virtual Address (SVA) and physical address of the device CSRs.

Note that `config` generates the values for members from `ui_driver` to `ui_ivnum` from information provided in the system configuration file.

Table 5-6 lists the members of the `uba_device` structure that a device driver can reference.

Table 5-6: Members of the uba_device Structure Used by Device Drivers

Member Name	Data Type	Description
`ui_driver`	`struct uba_driver *`	Specifies a back pointer to a `uba_driver` structure.
`ui_devname`	`char *`	Specifies the name of the device.
`ui_unit`	`short`	Specifies the unit number of the device on the system.
`ui_adpt`	`int`	Specifies the adapter number (consecutive adapter number).
`ui_nexus`	`short`	Specifies the nexus on the I/O bus.
`ui_rctlr`	`short`	Specifies the remote controller number.
`ui_ubanum`	`short`	Specifies the uba number the device is on.
`ui_vbanum`	`short`	Specifies the VMEbus adapter number as specified in the system configuration file. For example, a VMEbus entry would have these specifications: vba0, vba1, vba2, and so forth. (Note that this member stores only the VMEbus adapter number.)
`ui_ctlr`	`short`	Specifies the controller number associated with this device, if it exists. If it does not exist, this member contains the value –1.
`ui_slave`	`short`	Specifies the slave device number on the controller.
`ui_intr`	`int (**ui_intr) ()`	Specifies an array of interrupt handlers. These interrupt handlers are called when the device generates interrupts.
`ui_addr`	`caddr_t`	Specifies the System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file.
`ui_addr2`	`caddr_t`	Specifies the System Virtual Address (SVA) corresponding to the second CSR specified in the system configuration file.

Table 5-6: (continued)

Member Name	Data Type	Description
`ui_dk`	`short`	If this member is greater than or equal to zero (0), then it can be used as an index into the set of dk arrays defined in `/usr/sys/h/dk.h`. These arrays are used to hold performance data displayed by the `iostat` command.
`ui_flags`	`int`	Saves the flags from the system configuration file, if any flags were specified.
`ui_bus_priority`	`int`	Specifies the configured VMEbus priority level of the device.
`ui_ivnum`	`int`	Specifies the first configured VMEbus device interrupt vector number for this device.
`ui_priority`	`int`	Specifies the main bus request level of the device.
`ui_alive`	`short`	Specifies whether the device exists.
`ui_type`	`short`	Specifies driver-specific type information.
`ui_physaddr`	`caddr_t`	Specifies the physical address for standalone (dump) code.
`ui_forw`	`struct uba_device *`	Specifies a list of devices on a controller.
`ui_mi`	`struct uba_ctlr *`	Specifies a back pointer to a `uba_ctlr` structure. If connected to the device, this `uba_ctlr` structure identifies the controller.
`ui_hd`	`struct uba_hd *`	Specifies a back pointer to a `uba_hd` structure.
`ui_vbahd`	`struct vba_hd *`	Specifies a back pointer to a vba_hd structure.

5.2 VMEbus Data Structures

In addition to the structures discussed previously, the VMEbus device driver writer must understand these structures:

- `vba_hd`
- `vbadata`

The members of these structures pertinent to VMEbus device drivers are discussed in the following sections.

5.2.1 The vba_hd Structure

The `vba_hd` structure holds a pointer to the interrupt vector table for the VMEbus adapter and the VMEbus adapter's address in physical and virtual memory. At boot time, ULTRIX determines which devices are attached to the VMEbus adapters and fills in the interrupt vectors associated with each device as specified in the system configuration file. During normal operation, ULTRIX allocates resources and returns them to the `vba_hd` structure. Table 5-7 lists the members of the `vba_hd` structure that a VMEbus device driver can reference.

Table 5-7: Members of the vba_hd Structure Used by Device Drivers

Member Name	Data Type	Description
`next`	`struct vba_hd *`	Specifies a pointer to the next `vba_hd` structure.
`vba_type`	`int`	Specifies the VMEbus adapter type. For the DECsystem 5000 Model 200 processor, this member is set to `VBA_3VIA` (the PMABV-AA adapter supported by the DECsystem 5000 Model 200 processor).
`vbanum`	`int`	Specifies the VMEbus adapter number as provided in the system configuration file for this VMEbus adapter.
`adptnum`	`int`	Specifies the adapter number (consecutive adapter number).
`vbavirt`	`caddr_t`	Specifies the virtual address of the VMEbus adapter.
`vbaphys`	`caddr_t`	Specifies the physical address of the VMEbus adapter.
`pio_base`	`caddr_t`	Specifies the base of PIO mapped space.
`vbadata`	`struct vbadata *`	Specifies a pointer to a `vbadata` structure for this VMEbus adapter.
`intr_vec`	`int (**intr_vec) ()`	Specifies the interrupt vector routines for the DECstation 5000 Model 200 processor.
`vbavec_page`	`int (**vbavec_page) ()`	Specifies the interrupt vector routines for other processors.
`vba_err`	`int (*vba_err) ()`	Specifies a pointer to the error routine for this VMEbus adapter.
`vba_vmewant`	`short`	Specifies that some process is waiting for VMEbus mapping resources.

5.2.2 The vbadata Structure

The `vbadata` structure is used by ULTRIX to customize a variety of VMEbus parameters. Table 5-8 lists the members of the `vbadata` structure.

Table 5-8: Members of the vbadata Structure

Member Name	Data Type	Description
`vme_brl`	`int`	Specifies the VMEbus request level for adapter master cycles.
`arb_to`	`unsigned int`	Specifies the arbitration timeout period.
`arb_type`	`int`	Specifies the arbitration method, for example, round-robin arbitration, single level arbitration, and so forth.
`intr_mask`	`int`	Specifies the interrupt priority levels handled by the adapter. You can set this member to the bitwise inclusive OR of the valid interrupt priority levels to be handled by this adapter. These are defined in `/usr/sys/data/vba_data.c`.
`syscon`	`int`	Specifies if the VMEbus adapter is the VMEbus system controller.
`release`	`int`	Specifies the VMEbus release modes.
`asc`	`int`	Specifies whether the DMA PMRs are mapped to the first or second gigabyte of VMEbus address space.

The members of the `vbadata` structure are initialized to values that should provide proper VMEbus operation for most applications. You should be careful about making any modifications to the initialized values for these members, because not all adapters support all of these values.

Table 5-9 lists the initialized values for the members of the `vbadata` structure. If you need to modify the values for any of these members, see the file `/usr/sys/data/vba_data.c`.

Table 5-9: Initialized Values of the vbadata Structure

Value	Description
`VME_BR_3`	Bus request level for master cycles is level 3.
`VME_ARBTO_64US`	Arbitration time out is 64 microseconds.
`VME_ARB_RR`	Arbitration is round robin.
`VME_ALL_IPL`	All interrupt levels are handled by the adapter.
`VME_SYS_CONTROLLER`	The adapter is a VMEbus controller.
`VME_ROR`	VMEbus release mode is release on request.

Table 5-9: (continued)

Value	Description
`VME_MAP_LOW`	The DMA PMRs for this adapter are mapped to the first gigabyte in the VMEbus address space.

Kernel Routines Used by Device Drivers 6

This chapter describes when and why you would use the kernel routines developed for use with VMEbus and TURBOchannel device drivers. In addition, the chapter discusses when and why you would use certain other kernel routines that can be used by any device driver. The chapter provides brief examples (and references to more complete examples when they appear in other chapters) to illustrate how to use these routines in device drivers. For complete descriptions of the definitions and arguments for these and other kernel routines, see Appendix B.

Specifically, the chapter discusses kernel routines used by:

- VMEbus device drivers
- TURBOchannel device drivers
- Any device driver

6.1 Kernel Routines Used by VMEbus Device Drivers

When writing device drivers for the VMEbus, you need to be familiar with the kernel routines that:

- Allocate VMEbus address space (for DMA)
- Release VMEbus address space (for DMA)
- Obtain the VMEbus address
- Perform byte swapping operations
- Perform read-modify-write operations

The two kernel routines that allow VMEbus drivers to log errors are discussed in Chapter 8.

6.1.1 Allocating VMEbus Address Space

Direct Memory Access (DMA) is a mechanism for allowing a peripheral device to access main memory without the help of the CPU.

In ULTRIX, you can allocate the DMA space and then set up the mapping registers for DMA transfer by calling the `vballoc` or the `vbasetup` routines or both. The primary difference between the two routines is that `vbasetup` takes a pointer to a `buf` structure as an argument, while `vballoc` takes an address and the number of bytes as arguments. You would use `vbasetup` when a `buf` structure is provided to the driver. All file system I/O and most user I/O occur using a `buf` structure. You would use the `vballoc` routine for driver-initiated I/O, for example, device command packets. Each of these routines returns a VMEbus address that is mapped to the buffer. If the requested mapping could not be performed, each of these routines returns a value of zero (0).

The following code fragments illustrate the similarities and differences between the call to the two routines:

```
/*******************************************************************/
/* Code fragment for call to vballoc */
 .
 .
 .
 /* Declarations */
 .
 .
 .
 #define BUFSIZ 512
 .
 .
 .
 unsigned int vmeaddr;
 register struct uba_device *devptr;
 char buffer[BUFSIZ];
 .
 .
 .
 /* Call to vballoc */
 vmeaddr = vballoc (devptr->ui_vbahd, [1]
                    buffer, BUFSIZ, [2]
                    VME_DMA | VMEA32D32 | VME_BS_NOSWAP, [3]
                    0); [4]
 .
 .
 .
/*******************************************************************/
/* Code fragment for call to vbasetup */
 .
 .
 .
 /* Declarations */
 .
 .
 .
 struct buf *bp;
 .
 .
 .
 unsigned int vmeaddr;
 register struct uba_device *devptr;
 /* Call to vbasetup */
 vmeaddr = vbasetup (devptr->ui_vbahd, [1]
                     bp, [2]
                     VME_DMA | VMEA32D32 | VME_BS_NOSWAP, [3]
                     0); [4]
 .
 .
 .
```

[1] The code fragments show that both routines take as the first argument a back pointer to the `vba_hd` structure associated with this device. Note that the back pointer is accessed through the `ui_vbahd` member of `devptr`, which is a pointer to a `uba_device` structure.

[2] The second argument passed to `vballoc` is an argument (*buffer*) that represents the beginning virtual address of the buffer to be mapped. In addition, a third argument (`BUFSIZ`) that specifies the byte count (size) of this buffer is passed.

For `vbasetup`, the second argument is a pointer to a `buf` structure.

3 Both routines pass the bitwise inclusive OR of the valid VMEbus flags bits: `vballoc` passes the bits as the fourth argument and `vbasetup` passes the bits as the third argument.

Some devices may want to perform DMA operations with another VMEbus device. To manage the addresses used for these DMA operations, you can set the *flags* bits argument for the `vbasetup` and `vballoc` routines to `VME_RESERV`. This value reserves space in the VMEbus I/O space (the A16 I/O space). The VMEbus address returned will be used in the VMEbus I/O space for the specified VMEbus address space.

4 Both routines pass a value to indicate some address in the VMEbus address space: `vballoc` passes this value as the fifth argument and `vbasetup` passes the value as the fourth argument. In the code frgaments, the value passed is zero (0), which indicates that these routines use the next available VMEbus address in the A24 or A32 DMA space. It is possible to pass a nonzero value, in which case these routines attempt to map the buffer to the requested VMEbus address.

See Section 10.2.6 for a more detailed example of how to call the `vbasetup` routine in a DMA driver.

6.1.2 Releasing VMEbus Address Space

To release the VMEbus address space allocated in a previous call to `vballoc` or `vbasetup`, use `vbarelse`. This routine releases the resources (map registers) used to map the specified VMEbus address.

The only situation in which you would not release the resources is when the memory needs to be mapped for an extended length of time (for example, common data structures). The following code fragment illustrates a call to `vbarelse` based on the code fragments presented in the previous section for `vballoc` and `vbasetup`:

```
/***********************************************************************/
/* Code fragment for call to vbarelse */
 .
 .
 .
 vbarelse (devptr->ui_vbahd, vmeaddr); 1
 .
 .
 .
```

1 The first argument is the `vba_hd` structure on which the map registers were allocated in a previous call to `vballoc` or `vbasetup`.

The second argument is the VMEbus address that was mapped to the specified buffer. This address was returned in a previous call to `vballoc` or `vbasetup`.

6.1.3 Obtaining the VMEbus Address

There are situations when your device driver may need to know the VMEbus address that corresponds to the System Virtual Address (SVA) that was passed to the driver's `probe` routine. To retrieve this address, you call the `vba_get_vmeaddr` routine. Typically, you call this routine to retrieve the VMEbus address used in device-to-device DMA. The following code fragment illustrates a call to `vba_get_vmeaddr`:

```
/*******************************************************************/
/* Code fragment for call to vba_get_vmeaddr */
 .
 .
 .
 caddr_t vmeaddr;
 u_long addr;
 register struct uba_device *devptr;
 addr = devptr->ui_addr;
 .
 .
 .
 vmeaddr = vba_get_vmeaddr (devptr->ui_vbahd, [1]
                                  addr); [2]
 .
 .
 .
```

[1] The first argument to `vba_get_vmeaddr` is a back pointer to a `vba_hd` structure. The back pointer is accessed through the `ui_vbahd` member of the `uba_device` structure pointed to by `devptr`.

[2] The second argument is the SVA for the device. This argument is set to the value stored in the `ui_addr` member of the `uba_device` structure associated with this device. In addition, the `ui_addr2` member of the `uba_device` structure associated with this device would have been used if the driver wanted the second CSR space.

6.1.4 Performing Byte Swapping Operations

The VMEbus does not specify any particular byte ordering. Because most devices use the big endian model and the Digital model is little endian, the following kernel routines are provided for drivers to perform byte swapping operations:

- `swap_lw_bytes`

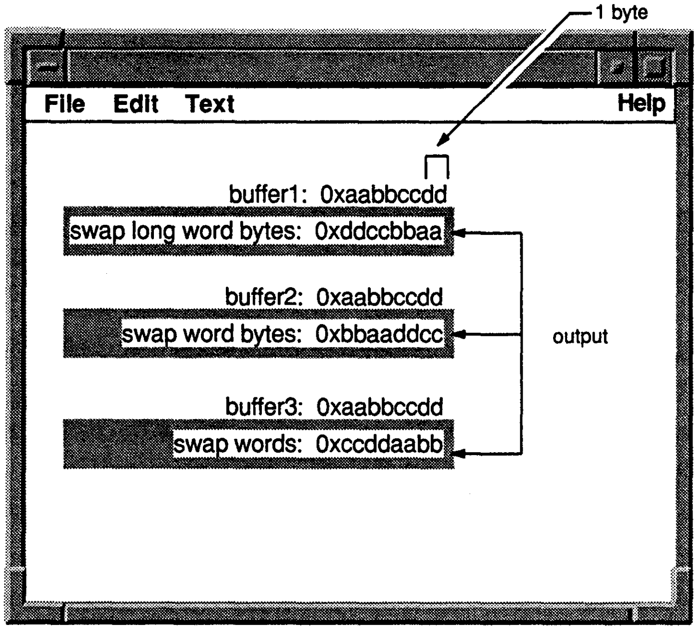
  Performs a long word byte swap
- `swap_word_bytes`

  Performs a short word byte swap
- `swap_words`

  Performs a word byte swap

Figure 6-1 illustrates a 32-bit (4 bytes) quantity that the following code fragments will swap.

Figure 6-1: Results of Byte Swapping Routines



```
             buffer1: 0xaabbccdd
swap long word bytes: 0xddccbbaa

             buffer2: 0xaabbccdd
     swap word bytes: 0xbbaaddcc

             buffer3: 0xaabbccdd
          swap words: 0xccddaabb
```

ZK-0258U-R

The figure also shows what the 32-bit quantity looks like after calling each of the byte swapping routines and after executing the `printf` statements.

```
/***********************************************************************/
/* Code fragment for call to swap_lw_bytes */
 .
 .
 .
 unsigned int buffer; 1
 unsigned int result; 2
 unsigned int *bufpt; 3
 .
 .
 .
 bufpt = &buffer; 4
 *bufpt = 0xaabbccdd; 5
 .
 .
 .
 /* Byte swap using swap_lw_bytes */ 6
 printf("\n              buffer1: 0x%x\n",*bufpt);
 result = swap_lw_bytes (*bufpt);
 printf("swap long word bytes: 0x%x\n\n",result);
 .
 .
 .
 /* Byte swap using swap_word_bytes */ 7
 printf("\n              buffer2: 0x%x\n",*bufpt);
 result = swap_word_bytes (*bufpt);
 printf("     swap word bytes: 0x%x\n\n",result);
 .
 .
 .
```

```
/* Byte swap using swap_words */ [8]
printf("\n             buffer3: 0x%x\n",*bufpt);
result = swap_words (*bufpt);
printf("         swap words: 0x%x\n\n",result);
swap_word_bytes (buffer);
```

[1] This line declares a 32-bit (4 bytes) quantity that will be swapped by the byte swapping routines.

[2] This line declares an argument in which the result of the byte swapping operation will be stored.

[3] This line declares a pointer to a buffer pointer.

[4] This line initializes the buffer pointer to the address of *buffer*.

[5] This line initializes the buffer to the 32-bit quantity (aabbccdd).

[6] The first call to the `printf` kernel routine prints the value pointed to by the *bufpt* argument. This value is the 32-bit quantity. The `swap_lw_bytes` routine performs a long word byte swap and returns the result in the *result* argument. The second call to the `printf` statement prints the result of the byte swap, as illustrated in Figure 6-1. Note that `swap_lw_bytes` swaps all four bytes.

[7] These lines perform the same tasks as those described previously except `swap_word_bytes` performs a short word byte swap, as illustrated in Figure 6-1. The figure shows that `swap_word_bytes` swaps the individual bytes that make up each byte of the 32-bit quantity.

[8] These lines perform the same tasks as those described previously except `swap_words` performs a word byte swap, as illustrated in Figure 6-1. The figure shows that `swap_words` swaps the two words.

6.1.5 Performing Read-Modify-Write Operations

There are situations when your device driver may need to perform a read-modify-write to VME-side memory. The `vme_rmw` routine is an interlock primitive that emulates a hardware read-modify-write cycle. You can use it to lock a portion of memory, read some specified data that resides in that portion of memory, and modify (write) that portion of memory with new data. The following code fragment illustrates a call to `vme_rmw`. The context is a device driver that implements its own locking scheme on an address space:

```
/********************************************************************/
/* Code fragment for call to vme_rmw */
 .
 .
 .
 #define DATA_LOCKED -1
 #define SUCCESS 0
 #define DRIVER_LOCK_MASK 0x00000001
 .
 .
 .
 clear_location (vhp, address_p) [1]
 struct vba_hd *vhp;
 unsigned int *address_p;

 .
 .
 .
```

```
{
int new_data = 0;
int lock_mask = DRIVER_LOCK_MASK;
        /* Perform a read-modify-write */ [2]

        if vme_rmw (vhp, address_p, new_data, lock_mask,) != 0)
               return (DATA_LOCKED);
        else
               return (SUCCESS);
}
 .
 .
 .
```

[1] This routine clears a location and returns zero (0) for success. Note that it takes two arguments: the first a pointer to a `vba_hd` structure and the second a pointer to the data to be cleared.

[2] The code fragment shows that the first argument to `vme_rmw` is a pointer to the `vba_hd` structure associated with this device.

The second argument passed to `vme_rmw` is a pointer to the data to be modified.

The third argument is the new data to be written to this memory location.

The fourth argument is a lock mask that specifies which bits to check to determine if the data is locked.

6.2 Kernel Routines Used by TURBOchannel Device Drivers

When writing device drivers for the TURBOchannel, you need to be familiar with these kernel routines: `tc_enable_option` and `tc_disable_option`.

The `tc_enable_option` routine enables a device's interrupt line to the processor. A device driver uses this routine only if the device must have its interrupts enabled during configuration. The ULTRIX kernel automatically enables the device's interrupts after configuration, depending on what you specified in the `tc_option` data table. See Section 9.3 for instructions on setting the tc option table so that the kernel enables the device's interrupts after configuration.

The `tc_disable_option` routine disables a device's interrupt line to the processor. A device driver uses this routine only if the device must have its interrupts alternately enabled and disabled during configuration or during operation.

The following code fragment illustrates calls to `tc_enable_option` and `tc_disable_option`:

```
 .
 .
 .
/*****************************************************************/
/* Code fragment for calls to tc_enable_option */
/* and tc_disable_option */
 .
 .
 .
extern struct uba_device *cfbinfo[];
int     cfb_curs_vsync = 0;
struct  uba_device *cfbinfo[1];
 .
 .
 .
```

```
          case QIOWLCURSOR:
                  cfb_curs_vsync = 1;
                  *(cfbp->framebuffer + IREQ_OFFSET) = 0;
                  tc_enable_option(cfbinfo[0]); [1]
                  while (cfb_curs_vsync)
                      sleep(&cfb_curs_vsync, TTIPRI); [2]
                  tc_disable_option(cfbinfo[0]); [3]
                  break;
  .
  .
  .
```

[1] This code fragment uses a switch statement whose corresponding case values represent some task performed by this driver. The code fragment picks up with the `QIOWLCURSOR` case value and it illustrates the use of the `tc_enable_option` and `tc_disable_option` routines. The single argument passed to `tc_enable_option` is the pointer to the `uba_device` structure associated with device unit 0. Device unit 0 is the device whose interrupt line to the processor is enabled.

[2] While the `cfb_curs_vsync` value is true, the process sleeps.

[3] The interrupt line to the processor for device unit 0 is disabled.

6.3 Kernel Routines That Can Be Used by Any Device Driver

When writing device drivers for any bus, including VMEbus and TURBOchannel, you need to be familiar with the kernel routines that perform these tasks:

- Flushing the processor data cache
- Ensuring a write to I/O space completes
- Obtaining the page frame number (for memory mapping)

6.3.1 Flushing the Data Cache

The `bufflush` routine flushes the processor data cache. A device driver must explicitly flush the processor data cache if the device performs DMA-to-host-memory. The reason for this is that there is no hardware cache coherency mechanism on some RISC processors. For example, the 5800 systems support hardware cache coherency, while the DECsystem 5400 and DECsystem 5000 Model 200 systems do not.

The following code fragment illustrates a call to `bufflush`:

```
/*****************************************************************/
/* Code fragment for call to bufflush */
 .
 .
 .
 struct buf *bp;
 .
 .
 .
 .
 .
 .
 if (bp->b_flags & B_READ) bufflush(bp); [1]
```

```
.
.
.
```

[1] The argument passed to `bufflush` is the pointer to the `buf` structure. In this fragment, if the result of the bitwise AND operation produces a one (indicating a read operation), `bufflush` is called to flush the data cache.

6.3.2 Ensuring a Write to I/O Space Completes

The `wbflush` routine ensures a write to I/O space has completed. Whenever a device driver writes to I/O space, the write may be intermittently delayed through the imposition of a hardware-dependent system write buffer. Subsequent reads of that location will not wait for a delayed write to complete. Either the original or the new value may be obtained. Subsequent writes of that location may replace the previous value, either in I/O space or in the system write buffer, if its writing had been delayed. In this case, the previous value would never have been actually written to I/O space.

Whether a given write to I/O space is delayed and how long this period is depends upon the existence of a system write buffer, its size, and its content. In general, delayed writes are not a problem. Device drivers need not call `wbflush` except in the following special situations:

- The write causes a state change in the device, and the change is indicated by a subsequent device-induced change in the value of the location being written by the device driver. This situation normally exists only during initialization of certain devices.
- The value being written is permanently consumed by the act of writing it. This situation exists only for certain specific devices, including some terminal devices.

The following code fragment illustrates a call to `wbflush`:

```
/*****************************************************************/
/* Code fragment for call to wbflush */
 .
 .
 .
 .
 .
 .
 .
 .
 .
 if (reg->csr & ERROR)
 {
     return(0);
 }
 reg->csr=0;
 wbflush(); [1]
 .
 .
 .
```

[1] This code fragment shows that if the result of the bitwise AND operation produces a nonzero value (that is, the error bit is set), then the value zero (0) is returned. If the result of the bitwise AND operation is a zero value (that is, the error bit is not set), then the device's control status register is set to zero (0) and the `wbflush` routine is called to ensure that a write to I/O space completes.

Note that `wbflush` takes no arguments.

6.3.3 Obtaining the Page Frame Number

When writing a device driver that provides a memory mapping routine, you need to obtain the page frame number associated with the address of the device. To accomplish this task, you use the `vtokpfnum` routine. This routine obtains the page frame number for the page in the character device's memory that was mapped to the kernel virtual address. The following code fragment illustrates a call to `vtokpfnum`:

```
/*******************************************************************/
/* Code fragment for call to vtokpfnum */
 .
 .
 .
 register struct sk_reg_t *sk_reg;
 .
 .
 .
 u_int kpfnum;
 .
 .
 .
 vtokpfnum (sk_reg+off); 1
 .
 .
 .
```

1 The argument passed to `vtokpfnum` is the kernel virtual address whose page frame number is to be returned. This address is the result of the expression whose operands consist of the pointer to the structure that represents the device's registers and the offset into the device's memory. You pass these arguments to the driver's memory map routine.

Device Autoconfiguration 7

Autoconfiguration is the process by which the ULTRIX operating system determines what hardware devices might be present on the system. This chapter describes autoconfiguration for devices connected to the VMEbus and TURBOchannel. The chapter consists of the following:

- Autoconfiguration overview
- Autoconfiguration for VMEbus devices
- Autoconfiguration for TURBOchannel devices

7.1 Autoconfiguration Overview

ULTRIX supports a variety of hardware devices that must be configured during system startup. It is not possible to configure all of these devices in advance, because on different systems these devices are present in different numbers, at different addresses, and in different combinations. To solve this problem, ULTRIX supports a static configuration procedure and a dynamic configuration procedure. The static procedure defines the set of hardware devices that might be on the system and the dynamic procedure identifies the set of hardware devices that are actually present on the system. This section presents an overview of the dynamic procedure, which is usually referred to as autoconfiguration. For information on the static procedure, see the *Guide to Configuration File Maintenance.*

In general, the autoconfiguration procedure requires that device drivers supply:

- A `probe` routine
- A `slave` routine
- An `attach` routine

The implementation of these (and possibly additional) routines to accomplish the autoconfiguration procedure can differ, depending on the bus for which the device driver is being written. The following sections discuss the specifics of autoconfiguration for devices connected to the VMEbus and the TURBOchannel.

7.2 Autoconfiguration for Devices Connected to the VMEbus

The autoconfiguration procedure for VMEbus devices consists of the following:

- Controller configuration
- Device configuration

7.2.1 Controller Configuration

The controller configuration routine does the following when it is called:

- Calls the adapter code, which maps CSR addresses into VMEbus address space
- Invokes the controller's `probe` routine
- Fills in the configuration database controller entry, if the `probe` routine detects the presence of a controller
- Prints information about the controller to the console and error log file
- Sets the controller alive bit in the `vba_ctlr` structure
- Initializes the interrupt vector table
- Initializes the controller priority field
- Searches the configuration database and for each configured slave device on a controller:
  - Calls the controller `slave` routine
  - Sets the device alive bit in the configuration database, if the `slave` routine detects a device connected to the controller
  - Fills in the configuration database device entry
  - Sets the device as alive in the `uba_device` structure
  - Calls the device driver's `attach` routine

7.2.2 Device Configuration

The device configuration routine does the following when it is called:

- Calls the adapter code, which maps the CSR addresses into VMEbus space
- Determines if the device is present
- Invokes the device driver's `probe` routine
- Fills in the configuration database device entry, if the `probe` routine detects a device
- Prints information about the device to the console and error log file
- Initializes the device priority field
- Sets the device as alive in the `uba_device` structure
- Initializes the interrupt vector table
- Calls the device driver's `attach` routine

7.3 Autoconfiguration for Devices Connected to the TURBOchannel

Each TURBOchannel device (option module) has the following characteristics, which are defined in the `tc_slot` structure:

- The name of the I/O module as it appears in read only memory (ROM) on the device
- The name of the controller or device attached to the TURBOchannel
- The TURBOchannel I/O slot number
- The number of slots occupied by the I/O module
- A pointer to the `interrupt` routine
- The unit number of the device
- The base physical address of the device

The ULTRIX operating system uses the information contained in the `tc_slot` structure to perform the following tasks during autoconfiguration:

- Probe TURBOchannel option slots
- Obtain the I/O module's name
- Map TURBOchannel slot numbers

Following the discussion of these tasks, there is a brief discussion of the `tc_option` table and the system configuration file as it affects TURBOchannel device driver writers. You can find the `tc_option` table in `/usr/sys/data/tc_option_data.c`.

7.4 Probing TURBOchannel Option Slots

During system startup, ULTRIX searches the TURBOchannel address space to determine which slots actually contain an I/O module. Each TURBOchannel I/O slot is at a fixed and known physical address. Thus, ULTRIX can search the TURBOchannel I/O slots by their known physical addresses. If the slot contains an I/O module, the driver's `probe` routine performs device-specific setup and initialization that may include forcing the device to interrupt.

Each I/O module must have a ROM with a known format. ULTRIX reads that ROM to determine the I/O module's width (that is, the number of slots it occupies) and to obtain the I/O module's name.

7.4.1 Obtaining the I/O Module's Name

After probing the TURBOchannel I/O slots, ULTRIX looks up the module name in the `tc_option` data table to obtain the device or controller name as it is specified in the system configuration file. This is an internal table that maps TURBOchannel module names to names as they appear in the system configuration file. This internal table contains a structure entry for each of the TURBOchannel I/O options on the system. The following example illustrates a sample entry in the system configuration file:

```
 device           qac0       at ibus?    vector qacvint
```

The following example illustrates the corresponding entry in the `tc_option` data table:

```
struct tc_option tc_option [] =
{
    /*  module          driver  intr_b4 itr_aft         adpt    */
```

```
    /*  name            name    probe   attach  type    config  */
    /*  ------          ------  ------- ------- ----    ------  */
    {   "PMAG-BA ",     "qac",    0,      0,    'D',    0},     /* QAC */
    /*
     * Do not delete any table entries above this line or your system
     * will not configure properly.
     *
     * Add any new controllers or devices here.
     * Remember, the module name must be blank padded to 8 bytes.
     */
    /*
     * Do not delete this null entry, which terminates the table or your
    * system will not configure properly.
     */
    {   "",             ""      }       /* Null terminator in the table */
};
```

ULTRIX compares the device names found in the I/O slots and the `tc_option` table (optional as well as fixed devices) with the names given in the system configuration file. These device names appear in the `ubminit` table (an array of `uba_ctlr` structures) and the `ubdinit` table (an array of `uba_device` structures). Each entry in the system configuration file specifies the `interrupt` routine name for the device. In the previous example, the interrupt routine is called `qacvinit`.

The name of the `interrupt` routine is placed in the `ubminit` and `ubdinit` tables by the configuration program.

For information on how to make an entry in this file, see Section 9.3.

7.4.2 Mapping TURBOchannel Slot Numbers

If ULTRIX matches a device name in the `tc_option` table with a device name in the system configuration file, ULTRIX puts an entry in the `tc_slot` table.

If ULTRIX finds a module name in a module ROM that is not in the `tc_option` data table, then the system warns that the device is unknown.

If ULTRIX finds a device name that was not in the system configuration file, that device will not be configured. That is, it will not have its `probe` or `attach` routines called, and its interrupt line will be disabled.

For properly configured and recognized controllers and devices, the ULTRIX operating system calls the `probe`, `attach`, and `slave` routines through the ‘‘ibus’’ configuration routines. The ibus configuration routines obtain the names of the `probe`, `attach`, and `slave` routines from the device driver's `uba_driver` structure.

Adapters are handled in a similar way as devices and controllers. Adapters have an adapter line in the system configuration file with no `interrupt` routine name. The ULTRIX operating system configuration code looks up the adapter module name in the `tc_option` data table and obtains the name of the adapter configuration routine to call. One of the arguments passed to the adapter configuration routine is an address where that configuration routine places the address of the interrupt handling routine.

7.4.3 Considerations for TURBOchannel Device Driver Writers

The `tc_option` table and the system configuration file provide a flexible mechanism for adding third-party devices and device drivers. This table allows third-party device driver writers to map additional device names with their associated names in the system configuration file. Third-party or customer device drivers must conform to standard ULTRIX operating system conventions. For instance, drivers must have a `uba_driver` structure with the name of the device `probe` routine, `attach` routine, device name, and so forth. The qac, for example, has a `uba_driver` structure that looks like this:

```
 .
 .
 .
struct  uba_driver qacdriver =
        { qacprobe, 0, qacattach, 0, qacstd, "qac", qacinfo };
 .
 .
 .
```

The corresponding entry in the system configuration file looks like this:

```
 device          qac0       at ibus?    vector qacvint
```

Part IV: Error Handling and Installation

Error Handling 8

The ULTRIX programming environment provides a variety of debugging tools, some of which are listed in Table 8-1.

Table 8-1: ULTRIX Debugging Tools

Tool	Description
`ctrace`	Allows you to watch program flow and to observe changes to variables
`dbx`	Invokes an interactive debugger
`error`	Inserts error messages from a compiler or language processor into a source file at the point of error
`gcore`	Creates a core image file of a running process
`lint`	Checks C source files for waste, errors, and nonportable code
`trace`	Traces the system calls made by a command

See the *Guide to Languages and Programming* for descriptions and examples of each tool.

This chapter discusses error handling and some topics associated with error handling for VMEbus device drivers. Specifically, the chapter discusses the following:

- Logging errors associated with the VMEbus
- Testing memory map drivers
- Writing text to an output device

You accomplish most of these tasks by calling kernel routines. The chapter provides brief examples to illustrate how to use these routines in device drivers. For complete descriptions of the function definitions and argument descriptions for these and other kernel routines, see Appendix B.

8.1 Logging Errors Associated with the VMEbus

Error log events are initiated by hardware errors, informational events, the ULTRIX kernel, or applications. Appropriate information is gathered by the applicable driver, ULTRIX kernel, or application to form an error log event that is temporarily stored in the memory resident error log buffer. The error log daemon, `elcsd`, retrieves those events and transfers them to an error log file for permanent storage.

The data collection routines responsible for collecting pertinent data that is formed into an error log event exist in device drivers, the ULTRIX kernel, or an application. For VMEbus device drivers, two kernel routines are provided that allow you to log controller and device error events into the errorlog file. You would use these routines when you want to record VMEbus-specific error events in the errorlog and later use the `uerf` error report formatter to print these error events. For information on the error logging subsystem and the `uerf` error report formatter, see the *Guide to the Error Logger*. You can also find reference information on this utility in the *Reference Pages Section 8: Maintenance.*

To log controller error events, use the `log_vme_ctlr_error` kernel routine. To log device error events, use the `log_vme_device_error` kernel routine. Both routines allocate a message packet that includes the ASCII text supplied by the driver and the VMEbus adapter registers. The difference between the routines is that `log_vme_ctlr_error` includes controller information in its message packet, while `log_vme_device_error` includes device information in its message packet.

The following lists some of the controller information in the message packet provided by `log_vme_ctlr_error`:

- The controller index into the device driver (stored in the `um_ctlr` member of the `uba_ctlr` structure)
- The System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file (stored in the `um_addr` member of the `uba_ctlr` structure)
- The System Virtual Address (SVA) corresponding to the second CSR specified in the system configuration file (stored in the `um_addr2` member of the `uba_ctlr` structure)
- VMEbus adapter information

The following is the device information provided by `log_vme_device_error` in its message packet:

- The unit number of the device on the system (stored in the `ui_unit` member of the `uba_device` structure)
- The System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file (stored in the `ui_addr` member of the `uba_device` structure)
- The System Virtual Address (SVA) corresponding to the second CSR specified in the system configuration file (stored in the `ui_addr2` member of the `uba_device` structure)
- VMEbus adapter information

The following code fragments illustrate calls to these routines:

```
/*******************************************************************/
/* Code fragment for call to log_vme_ctlr_error */
 .
 .
 .
 char driver_text[] = "xx0: device fatal error";
 register struct vba_hd *vhp;
 register struct uba_ctlr *umptr;
 log_vme_ctlr_error (driver_text, vhp, umptr); [1]
```

```
 .
 .
 .

/*****************************************************************/
/* Code fragment for call to log_vme_device_error */
 .
 .
 .
 char driver_text[] = "xx0: device fatal error";
 register struct vba_hd *vhp;
 register struct uba_ctlr *umptr;
 log_vme_device_error (driver_text, vhp, umptr); [1]
 .
 .
 .
```

[1] The first argument is the ASCII text you want the `log_vme_ctlr_error` or `log_vme_device_error` routine to log. If you do not supply a message, `log_vme_ctlr_error` supplies this message: NO ERROR MESSAGE ENTERED BY DRIVER.

The second argument for both routines is the pointer to the `vba_hd` structure associated with this controller or device. These routines use the pointer to the `vba_hd` structure to determine the VMEbus adapter type and VMEbus adapter number.

The third argument for `log_vme_ctlr_error` is the pointer to the `uba_ctlr` structure associated with this controller, while the third argument for `log_vme_device_error` is the pointer to the `uba_device` structure associated with this device.

To obtain all VMEbus adapter and VMEbus controller and device errors from the error log, type the following:

```
/etc/uerf -A vba
```

To obtain all controller and device errors, including VMEbus controller and device errors, type:

```
/etc/uerf -r 104
```

To obtain all adapter errors, including VMEbus adapter errors, type:

```
/etc/uerf -r 105
```

8.2 Testing Memory Map Drivers

When debugging memory map device drivers, you may need to change the default behavior of the kernel when it responds to a write to nonexistent device memory. (See Section 4.12.2 for a discussion of mapping to nonexistent device memory.) By default, the kernel tries to locate and kill all processes that used the `mmap` system call to map the memory of the device into their address space.

You can change the kernel's default behavior by specifying the `MMAPDRV_DEBUG` option in the options definitions part of the system configuration file. By specifying this option, you ensure that the compiled kernel does not kill any processes, but causes the machine to crash. See Section 9.1.3 for a description of the system configuration file and the parts related to device drivers.

You would use this option when debugging a device driver to verify that the driver is correctly accessing the device. If you do not use this option, the kernel assumes that any write to an invalid address must be generated by a user process. The kernel then searches for any process that has mapped the area of memory where the invalid access occurred. If no user process has mapped that memory, the kernel assumes the request came from the device driver and crashes the system.

8.3 Writing Text to an Output Device

In handling errors, you need to be familiar with the kernel routines that allow you to print data to some output device. This section briefly describes when and why you would do this. See Appendix B for the function definitions and additional descriptions for these routines.

The `cprintf` routine prints only to the console terminal. You generally call this routine to report information when there is a problem with the error logging mechanism or to perform debugging.

The `mprintf` routine logs all text to the kernel error log file. This usually happens during hardware failures that are considered soft and corrected.

The `uprintf` routine prints to the current user's terminal. This routine guarantees not to sleep, thereby allowing it to be called by interrupt routines. It does not perform any space checking, so you do not want to use this routine to print verbose messages. The `uprintf` routine does not log messages to the error logger.

The `printf` routine prints diagnostic information directly on the console terminal, and it writes ASCII text to the error logger. Because `printf` is not interrupt driven, all system activities are suspended when you call it.

Installing Device Drivers 9

This chapter discusses how to install VMEbus and TURBOchannel device drivers. It begins with detailed discussions of the system files that you must modify as part of the driver installation. The chapter includes examples relevant to the VMEbus and the TURBOchannel. Because the steps for installing VMEbus and TURBOchannel drivers vary, the chapter discusses how to install each separately.

Specifically, the chapter contains information on:

- Modifying system files associated with device drivers
- Installing VMEbus device drivers
- Installing TURBOchannel device drivers

9.1 Modifying System Files Associated with Device Drivers

To add a device driver, you need to modify the following files used during the building of an ULTRIX kernel:

- `/usr/sys/machine/common/conf.c`
- `/usr/sys/conf/mips/files.mips`
- `/usr/sys/conf/mips/MACHINE`
- `/usr/sys/data/tc_option_data.c` (for TURBOchannel device drivers only)

The following sections discuss the parts of these files pertinent to device driver writers.

9.1.1 The conf.c File

The `conf.c` file contains two device switch tables called `cdevsw` and `bdevsw`. The device switch tables have the following characteristics:

- They are arrays of structures containing device driver entry points. These entry points are actually the addresses of the specific routines within the drivers.
- They may contain stubs for device driver entry points for devices that do not exist on a specific machine.
- They contain major device numbers that the kernel uses as indexes into this array of structures.

9.1.1.1 The cdevsw Table – The `cdevsw` table contains device driver entry points for each character mode device supported by the system. In addition, the table can contain stubs for device driver entry points for character mode devices that do not exist or for entry points not used by a device driver.

The following example shows the `cdevsw` structure defined in `/usr/sys/h/conf.h`:

```
struct cdevsw
{
  int      (*d_open)();
  int      (*d_close)();
  int      (*d_read)();
  int      (*d_write)();
  int      (*d_ioctl)();
  int      (*d_stop)();
  int      (*d_reset)();
  struct tty *d_ttys;
  int      (*d_select)();
  int      (*d_mmap)();
  int      (*d_strat)();
  int      (d_affinity);
};
```

The `d_open`, `d_close`, `d_read`, `d_write`, `d_ioctl`, and `d_select` members point to device driver routines. For example, a call to the driver from the kernel `read` system call on a device calls the driver routine pointed to by `d_read` in the appropriate `cdevsw` entry.

The `d_stop` member points to a routine used by communication devices.

The `d_reset` member points to a routine that is used to reset the bus.

The `d_ttys` member is used by communication devices.

The `d_mmap` member points to a routine used to perform memory mapping.

The `d_strat` member points to a strategy routine used for nbufio.

The `d_affinity` member specifies whether the CPU runs the driver as a Symmetric Multi-Processing (SMP) driver. The value zero (0) indicates that the CPU runs this driver as a non-SMP driver. The system treats a nonzero value as a mask of which CPUs can run the SMP driver. For example:

d_affinity Value	Valid CPUs
1	only CPU 0
5	only CPU 0 and CPU 2
0x11	only CPU 0 and CPU 4
.	
.	
.	
0xffffffff	CPUs 0 – 31

The following example illustrates a sample `cdevsw` switch table. Note that major device numbers 25-29 are marked reserved to local sites for character mode devices:

```
struct cdevsw   cdevsw[] =
{
.
.
.
{lpopen,        lpclose,        nodev,          lpwrite,         /*15*/
nodev,          nodev,          lpreset,        0,
```

```
 seltrue,        nodev,           0,        0},
 .
 .
 .
 /* 25-29 reserved to local sites */
 {gpibopen,       gpibclose,       gpibread,        gpibwrite,      /*25*/
 gpibioctl,      nulldev,         nodev,           0,
 seltrue,        nodev,           0,       0},
 .
 .
 .
{propen,         nulldev,        nulldev,         nulldev, /*75*/
prioctl,         nulldev,        nulldev,         0,
nodev,           nodev,          0,      0},
 /* TURBOchannel driver entry */
 {qacopen,       qacclose,        qacread,         qacwrite, /*76*/
 qacioctl,       qacstop,         nulldev,         0,
 asyncsel,       nodev,           nodev,           0},
 .
 .
 .
 /* VMEbus driver entry */
 {skopen,        skclose,         nodev,          nodev, /*77*/
 nodev,        nodev,          nulldev,        0,
 asyncsel,       skmmap,           nulldev,     0},
};
```

The example shows that major device number 76 is a TURBOchannel driver with the following entries:

- An open routine called `qacopen` and a close routine called `qacclose`.
- A read routine called `qacread` and a write routine called `qacwrite`.
- An ioctl routine called `qacioctl`.
- A stop routine called `qacstop`.
- A nulldev entry, which represents the `nulldev` routine. The `nulldev` routine returns zero (0). You should specify `nulldev` when it is appropriate for the routine to be called, but the driver has no functionality for this device. In this example, the reset routine has no functionality for the qac device; therefore, the `nulldev` entry is specified.
- The value zero (0) to indicate that the qac device does not support the `ttys` entry.
- A select routine called `asyncsel`. This driver routine is implemented by the kernel for nonterminal type character devices that support nbufio.
- A nodev entry, which represents the `nodev` routine. The `nodev` routine returns an `ENODEV` (error, no such device). You should specify `nodev` when it is not appropriate to call that routine for a particular driver. In this example, it is not appropriate to call a memory mapping routine for a qac device; therefore, the `nodev` entry is specified.
- A `nodev` entry to indicate that it is not appropriate to call a strategy routine.
- The value zero (0) to indicate that the kernel treats this as a non-SMP driver.

The example also illustrates the naming conventions used for device driver routines:

- A prefix that represents the name of the driver. For example, `qac` represents the name of some device.

- The name of the routine, for example, `read`, `write`, and so forth.

Note that each routine entry in the example corresponds to an appropriate member of the `cdevsw` structure. For example, `qacopen` corresponds to the `d_open` member.

9.1.1.2 The bdevsw Table

The bdevsw Table – The `bdevsw` table contains device driver entry points for each block mode device supported by the system. In addition, the table can contain stubs for device driver entry points for block mode devices that do not exist or for entry points not used by a device driver.

The following example shows the `bdevsw` structure defined in `/usr/sys/h/conf.h`:

```
struct bdevsw
{
  int      (*d_open)();
  int      (*d_close)();
  int      (*d_strategy)();
  int      (*d_dump)();
  int      (*d_psize)();
  int      (*d_flags)();
  int      (*d_ioctl)();
  int      (d_affinity);
};
```

The `d_open`, `d_close`, `d_strategy`, `d_dump`, `d_psize`, and `d_ioctl` members point to device driver routines. For example, a call to the driver from the kernel `open` system call on a device calls the driver routine pointed to by `d_open` in the appropriate `bdevsw` entry.

The `d_flags` member points to a value that describes the type of device driver. For tape drivers, this value is `B_TAPE`, which gets set in the `b_flags` member of the `buf` structure. For all other drivers, this member is set to 0.

The `d_affinity` member specifies whether the CPU runs the driver as a Symmetric Multi-Processing (SMP) driver. The value zero (0) indicates that the CPU runs this driver as a non-SMP driver. The system treats a nonzero value as a mask of which CPUs can run the SMP driver. For example:

```
d_affinity Value            Valid CPUs

1                           only CPU 0
5                           only CPU 0 and CPU 2
0x11                        only CPU 0 and CPU 4
 .
 .
 .
0xffffffff                  CPUs 0 - 31
```

The following example illustrates a sample `bdevsw` switch table:

```
struct bdevsw   bdevsw[] =
{
.
.
.
{ rlopen,       nodev,          rlstrategy,     rldump,         /*14*/
rlsize,       0,       rlioctl,    0 },
.
.
.
```

```
/* TURBOchannel driver entry */
{rzopen,        nulldev,        rzstrategy,        rzdump, /*21*/
rzsize,        0,          rzioctl,        0},
.
.
.
/* VMEbus driver entry */
.
.
.
{xxopen,        xxclose,        xxstrategy,        nodev, /*22*/
nodev,        0,          nulldev,        0},
};
```

The example shows that major device number 22 defines the following entries for a VMEbus driver:

- An open routine called `xxopen`.
- A close routine called `xxclose`.
- A strategy routine called `xxstrategy`.
- A nodev entry, which represents the `nodev` routine. The `nodev` routine returns an `ENODEV` (error, no such device). You should specify `nodev` when it is not appropriate to call that routine for a particular driver.
- A second nodev entry.
- A flags entry that is set to zero (0).
- A nulldev entry, which represents the `nulldev` routine. The `nulldev` routine returns zero (0). You should specify `nulldev` when it is appropriate for the routine to be called, but the driver has no functionality for this device.
- The value zero (0) to indicate that the kernel treats this as a non-SMP driver.

9.1.2 The files.mips File

The `files.mips` file contains lines that indicate:

- When the driver is to be loaded in the kernel
- Driver source code location
- Whether the device driver sources are supplied

The following example illustrates a sample `files.mips` file:

```
 .
 .
 .
 machine/mips/autoconf.c         standard device-driver Binary
 machine/common/conf.c           standard
 machine/mips/cons_sw.c          standard Binary
 machine/mips/kn01.c             optional cpu DS3100 Binary
  /* TURBOchannel driver source entry */
 io/tc/qac.c  optional qac device-driver Binary
 .
 .
 .
  /* VMEbus driver source entry */
io/vme/xcm.c    optional    xcm    device-driver    Notbinary
 .
 .
 .
```

The file in the example contains:

- The location of the source code for the device driver. For example, the source code for the qac, a TURBOchannel driver, is located in `io/tc/qac.c`.
- The key word `standard` or the key word `optional`. The `standard` key word indicates that this software module will be included in every kernel. The key word `optional` indicates that this software module will be included in those kernels whose system configuration files have the key string that follows the key word `optional`. For example, the module `io/tc/qac.c` will be included in those kernels whose system configuration files have the key string `qac`.
- The key word `device-driver`, which indicates to the makefile that builds the kernel what C compiler flags to use when compiling the device driver. This key word is mandatory for all device driver entries.
- The key word `Binary` or the key word `Notbinary`. The `Binary` key word causes symbolic links to be made in the `/usr/sys/conf/mips/MACHINE` directory to existing object modules. That is, `ln -s ../mips/BINARY.mips/`*filename* commands are added to the makefile. Device drivers supplied by Digital will use the key word `Binary`, which means that no driver sources are supplied.

  The `Notbinary` key word causes the `config` program to include `cc` as inline commands to be added to the makefile. Device drivers written by third party vendors can use either key word, depending on whether they want to supply the driver sources. This may be particularly applicable to VMEbus and TURBOchannel drivers. Note that the VMEbus entry in the example specifies `Notbinary`, which means that the driver sources will be used to generate the object file.

9.1.3 The MACHINE File

The `MACHINE` file (referred to as the system configuration file) identifies all of the device driver source code that needs to be compiled into the kernel, as well as some system parameters that influence how the kernel operates. The system configuration file has these parts:

- Global definitions
- Options definitions
- Makeoptions definitions
- System image definitions
- Device definitions
- Pseudodevice definitions

This section discusses the options definitions and device definitions parts of the system configuration file as they apply to device drivers written for the VMEbus and the TURBOchannel. Therefore, it supplements the information contained in the *Guide to Configuration File Maintenance*, which discusses each of the listed parts in detail.

The options definitions part of the system configuration file contains values that specify optional code to be compiled into the system. However, you can remove any of the options if they do not pertain to your site or if your system is short on physical memory space.

The syntax for the options definitions is:

`options` *optionlist*

The following option is useful for debugging new device drivers:

```
options MMAPDRV_DEBUG
```

This option allows you to change the default behavior of the kernel when it responds to a write to nonexistent memory. By default, the kernel tries to locate and kill all user processes that used the `mmap` system call to map the failing address into their address space. If the kernel does not find any such processes, it causes the machine to crash.

By specifying this option, you ensure that the compiled kernel does not kill any processes, but only causes the machine to crash. This behavior is desirable when debugging device drivers, especially drivers that can generate writes to nonexistent memory. See Section 4.12.2 for more information on mapping to nonexistent memory.

The device definitions part of the system configuration file contains descriptions of each current or planned device on the system. That is, these definitions describe such things as adapter, controller, device, disk, and tape mnemonics and logical unit numbers. You need to add these definitions for devices that were not on the system at installation time.

Because the syntax for the definitions varies according to whether the device runs on the VMEbus or the TURBOchannel, the discussion of the syntax is divided into the following sections, each separated into a section on VMEbus and a section on TURBOchannel:

- Adapter specification
- Controller specification
- Device specification
- Disk specification
- Tape specification

9.1.3.1 **Adapter Specification for VMEbus –** The following is the syntax for specifying the adapter that connects to the VMEbus:

`adapter vba`*`n`*` at nexus?`

`adapter`	Specifies the key word that precedes a system bus mnemonic and its associated unit number. An adapter identifies a physical connection to a bus. In this case, the bus is the VMEbus.
`vba`	Specifies the mnemonic for the VMEbus adapter.
n	Specifies the unit number of the adapter.
`nexus?`	Specifies the key word that identifies the nexus. A nexus is the hardware through which each physical connection to the system bus is connected. The question mark allows the system to pick the appropriate nexus.

This example shows an adapter entry for the VMEbus:

```
adapter vba0 at nexus?
```

9.1.3.2 Adapter Specification for TURBOchannel – The following is the syntax for specifying the adapter that connects to the TURBOchannel:

`adapter ibus`*n* `at nexus?`

`adapter`	Specifies the key word that precedes a system bus mnemonic and its associated unit number. An adapter identifies a physical connection to a bus. In this case, the system bus is the TURBOchannel.
`ibus`	Specifies the mnemonic for the TURBOchannel adapter.
n	Specifies the unit number of the adapter.
`nexus?`	Specifies the key word that identifies the nexus. A nexus is the hardware through which each physical connection to the system bus is connected. The question mark allows the system to pick the appropriate nexus.

This example shows an adapter entry for the TURBOchannel. Each TURBOchannel slot is configured as an IBUS:

```
# ibus entries for DECstation 5000 Model 200
# IO option slots
adapter        ibus0   at nexus?
adapter        ibus1   at nexus?
adapter        ibus2   at nexus?
adapter        ibus3   at nexus?
adapter        ibus4   at nexus?
adapter        ibus5   at nexus?
adapter        ibus6   at nexus?
adapter        ibus7   at nexus?
```

9.1.3.3 Controller Specification for VMEbus – The following is the syntax for specifying a controller definition associated with the VMEbus. (Note that you should specify the controller entry on one line in the system configuration file.)

```
controller dev at condev csr addr [ csr2 addr2 ] [ flags flg_val ]
priority prilevel vector vec... vec#
```

`controller`	Specifies the key word that precedes a controller mnemonic and its associated logical unit number. A controller identifies either a physical or a logical connection with one or more slaves attached to it.
dev	Specifies the controller's name and logical unit number. You specify the controller name with a character mnemonic.
`at`	Specifies the key word that appears after the `controller` key word and its associated mnemonic and logical unit number.
condev	Specifies the name and logical unit number of the adapter to which the controller is connected.
`csr`	Specifies the key word that precedes a control status register value for some device.

addr	Specifies the address of the control status register for the device. The address needed here must be in the I/O space of the VMEbus address space. See Section 2.3.1 for a discussion of the VMEbus address space.
`csr2`	Specifies the key word that precedes a second control status register value. Many VMEbus devices support direct access to both device registers and to onboard memory. It is likely that the locations for the device registers and to onboard memory will be in different VMEbus address spaces. To accommodate this, a `csr2` key word has been added.
addr2	Specifies the address of the second control status register area or onboard memory for the device. The address needed here must be in the I/O space of the VMEbus address space. See Section 2.3.1 for a discussion of the VMEbus address space.
`flags`	Specifies the key word that precedes some value that directs the system to perform some request.
flg_val	Specifies the value for the flag. Possible values are decimal numbers and hexadecimal numbers. The format of the hexadecimal number is 0x*nn*, where *nn* is a hexadecimal number consisting of digits from 0 to 9 inclusive and of the letters a to f inclusive.
`priority`	Specifies the key word that precedes a VMEbus priority level.
prilevel	Specifies the VMEbus priority level. Valid VMEbus priority levels range from 1 to 7 inclusive.
`vector`	Specifies the key word that precedes the name or names of the interrupt handlers for a device.
vec...	Specifies the name or names of the interrupt handlers for a device.
vec#	Specifies the interrupt vector number. Vector numbers can range from 0x00 to 0xFF inclusive. Interrupt vector numbers 0x00 to 0x3F inclusive are reserved for Digital. If a device has more than one interrupt handler, the system assigns each with the next sequential vector number that follows the number you specify here. For example, if you have two interrupt handlers and specify 0x40 as the interrupt vector number, the system assigns the following:

Interrupt Vector Number	Interrupt Handler
0x40	xxintr1
0x41	xxintr2

This example builds on the adapter example by showing you the adapter entry followed by a controller entry for a device connected to the VMEbus:

```
adapter vba0 at nexus?
controller td0 at vba0 csr 0x8020 priority 1 vector tdintr 0x45
```

9.1.3.4 Controller Specification for TURBOchannel – The following is the syntax for specifying a controller definition associated with the TURBOchannel:

`controller` *dev* `at` *condev* `vector` *vec...*

`controller`	Specifies the key word that precedes a controller mnemonic and its associated logical unit number. A controller identifies either a physical or a logical connection with one or more slaves attached to it.
dev	Specifies the controller's name and logical unit number. You specify the controller name with a character mnemonic.
`at`	Specifies the key word that appears after the `controller` key word and its associated mnemonic and logical unit number.
condev	Specifies the name and logical unit number of the adapter to which the controller is connected.
`vector`	Specifies the key word that precedes the name or names of the interrupt handlers for a device.
vec...	Specifies the name or names of the interrupt handlers for a device.

This example builds on the adapter example by showing you the adapter entries followed by some controller entries for a device connected to the TURBOchannel:

```
# ibus entries for DECstation 5000 Model 200
# IO option slots
adapter         ibus0    at nexus?
adapter         ibus1    at nexus?
adapter         ibus2    at nexus?
adapter         ibus3    at nexus?
adapter         ibus4    at nexus?
adapter         ibus5    at nexus?
adapter         ibus6    at nexus?
adapter         ibus7    at nexus?
controller      asc0     at ibus?        vector ascintr
controller      asc1     at ibus?        vector ascintr
controller      asc2     at ibus?        vector ascintr
controller      asc3     at ibus?        vector ascintr
```

9.1.3.5 Device Specification for VMEbus – The following is the syntax for specifying a device that runs on the VMEbus. (You should specify the device entry on one line in the system configuration file.)

`device` *dev* `at` *condev* `csr` *addr* `[ csr2` *addr2* `] [ flags` *flg_val* `]`
`priority` *prilevel* `vector` *vec... vec#*

`device`	Specifies the key word that precedes a device name and its associated logical unit number.
dev	Specifies the device's name and logical unit number. You specify the device name as a character mnemonic.
`at`	Specifies the key word that appears after the `device` key word and its associated mnemonic and logical unit number.
condev	Specifies the name and logical unit number of the adapter or controller to which the device is attached. You specify the adapter or controller name as a character mnemonic. For the VMEbus, the adapter mnemonic is `vba`.

`csr`	Specifies the key word that precedes a control status register value for some device.
addr	Specifies the address of the control status register for the device. The address needed here must be in the I/O space of the VMEbus address space. See Section 2.3.1 for a discussion of the VMEbus address space.
`csr2`	Specifies the key word that precedes a second control status register value. Many VMEbus devices support direct access to both device registers and to onboard memory. It is likely that the locations for the device registers and to onboard memory will be in different VMEbus address spaces. To accommodate this, a `csr2` key word has been added.
addr2	Specifies the address of the second control status register area or onboard memory for the device. The address needed here must be in the I/O space of the VMEbus address space. See Section 2.3.1 for a discussion of the VMEbus address space.
`flags`	Specifies the key word that precedes some value that directs the system to perform some request.
flg_val	Specifies the value for the flag. Possible values are decimal numbers and hexadecimal numbers. The format of the hexadecimal number is 0x*nn*, where *nn* is a hexadecimal number consisting of digits from 0 to 9 inclusive and of the letters a to f inclusive.
`priority`	Specifies the key word that precedes a VMEbus priority level.
prilevel	Specifies the VMEbus priority level. Valid VMEbus priority levels range from 1 to 7 inclusive.
`vector`	Specifies the key word that precedes the name or names of the interrupt handlers for a device.
vec...	Specifies the name or names of the interrupt handlers for a device.
vec#	Specifies the interrupt vector number. Vector numbers can range from 0x00 to 0xFF inclusive. Interrupt vector numbers 0x00 to 0x3F inclusive are reserved for Digital. If a device has more than one interrupt handler, the system assigns each with the next sequential vector number that follows the number you specify here. For example, if you have two interrupt handlers and specify 0x40 as the interrupt vector number, the system assigns the following:

`Interrupt Vector Number`	`Interrupt Handler`
`0x40`	`xxintr1`
`0x41`	`xxintr2`

This example builds on the adapter example by showing you the adapter entry followed by a device entry for a device connected to the VMEbus:

```
adapter vba0 at nexus?
device xcm0 at vba0 csr 0xa000 priority 3 vector xcmintr 0xc8
```

9.1.3.6 **Device Specification for TURBOchannel** – The following is the syntax for specifying a device that runs on the TURBOchannel:

`device` *dev* `at` *condev* `vector` *vec...*

`device`	Specifies the key word that precedes a device name and its associated logical unit number.
dev	Specifies the device's name and logical unit number. You specify the device name as a character mnemonic.
`at`	Specifies the key word that appears after the `device` key word and its associated mnemonic and logical unit number.
condev	Specifies the name and logical unit number of the adapter or controller to which the device is attached. You specify the adapter or controller name as a character mnemonic. For the TURBOchannel, the adapter mnemonic is `ibus`.
`vector`	Specifies the key word that precedes the name or names of the interrupt handlers for a device.
vec...	Specifies the name or names of the interrupt handlers for a device.

This example builds on the adapter example by showing you the adapter entries followed by a device entry for a device connected to the TURBOchannel:

```
# ibus entries for DECstation 5000 Model 200
# IO option slots
adapter         ibus0   at nexus?
adapter         ibus1   at nexus?
adapter         ibus2   at nexus?
adapter         ibus3   at nexus?
adapter         ibus4   at nexus?
adapter         ibus5   at nexus?
adapter         ibus6   at nexus?
adapter         ibus7   at nexus?
device          qac0    at ibus?        vector qacvint
```

9.1.3.7 **Disk Specification for VMEbus** – The following is the syntax for specifying a disk that runs on the VMEbus:

`disk` *dev* `at` *condev* `drive` *n*

`disk`	Specifies the key word that precedes a disk drive name and its logical unit number.
dev	Specifies the disk drive's name and logical unit number. You specify the disk drive name as a character mnemonic.
`at`	Specifies the key word that appears after the `disk` key word and its associated mnemonic and unit number.
condev	Specifies the name and logical unit number of the controller to which the disk drive is connected. You specify the controller name as a character mnemonic.
`drive`	Specifies the key word that precedes the physical unit number of the disk drive.
n	Specifies the physical unit number of the disk drive.

This example builds on previous examples by showing you the adapter entry, followed by the controller entry, followed by the disk entry for a device connected to the VMEbus:

```
adapter vba0 at nexus?
controller td0 at vba0 csr 0x8020 priority 1 vector tdintr 0x45
disk ra0 at td0 drive 0
```

9.1.3.8 **Disk Specification for TURBOchannel** – The following is the syntax for specifying a disk that runs on the TURBOchannel:

`disk` *`dev`* `at` *`condev`* `drive` *`n`*

`disk`	Specifies the key word that precedes a disk drive name and its logical unit number.
dev	Specifies the disk drive's name and logical unit number. You specify the disk drive name as a character mnemonic.
`at`	Specifies the key word that appears after the `disk` key word and its associated mnemonic and unit number.
condev	Specifies the name and logical unit number of the controller to which the disk drive is connected. You specify the controller name as a character mnemonic.
`drive`	Specifies the key word that precedes the physical unit number of the disk drive.
n	Specifies the physical unit number of the disk drive.

This example builds on previous examples by showing you the adapter entries, followed by the controller entries, followed by the disk entries for a device connected to the TURBOchannel:

```
# ibus entries for DECstation 5000 Model 200
# IO option slots
adapter         ibus0   at nexus?
adapter         ibus1   at nexus?
adapter         ibus2   at nexus?
adapter         ibus3   at nexus?
adapter         ibus4   at nexus?
adapter         ibus5   at nexus?
adapter         ibus6   at nexus?
adapter         ibus7   at nexus?
controller      asc0    at ibus?        vector ascintr
controller      asc1    at ibus?        vector ascintr
controller      asc2    at ibus?        vector ascintr
controller      asc3    at ibus?        vector ascintr
disk            rz0     at asc0         drive 0
disk            rz1     at asc0         drive 1
disk            rz2     at asc0         drive 2
disk            rz3     at asc0         drive 3
```

9.1.3.9 **Tape Specification for VMEbus** – The following is the syntax for specifying a tape that runs on the VMEbus:

`tape` *`dev`* `at` *`condev`* `drive` *`n`*

`tape`	Specifies the key word that precedes a tape drive name and its logical unit number.
dev	Specifies the tape drive's name and logical unit number. You specify the tape drive name as a character mnemonic.
`at`	Specifies the key word that appears after the `tape` key word and its associated name and logical unit number.
condev	Specifies the name and logical unit number of the controller to which the tape drive is connected. You specify the controller name as a character mnemonic.
`drive`	Specifies the key word that precedes the physical unit number of the tape drive.
n	Specifies the physical unit number of the tape drive.

This example builds on previous examples by showing you the adapter entry followed by the controller entry followed by the tape drive entry for a device connected to the VMEbus:

```
adapter vba0 at nexus?
controller xx0 at vba0 csr 0x8010 flags 0x0f priority 7 vector xxint 0xc0
tape yy0 at xx0 drive 0
```

9.1.3.10 Tape Specification for TURBOchannel – The following is the syntax for specifying a tape that runs on the TURBOchannel:

```
tape dev at condev drive n
```

`tape`	Specifies the key word that precedes a tape drive name and its logical unit number.
dev	Specifies the tape drive's name and logical unit number. You specify the tape drive name as a character mnemonic.
`at`	Specifies the key word that appears after the `tape` key word and its associated name and logical unit number.
condev	Specifies the name and logical unit number of the controller to which the tape drive is connected. You specify the controller name as a character mnemonic.
`drive`	Specifies the key word that precedes the physical unit number of the tape drive.
n	Specifies the physical unit number of the tape drive.

This example builds on previous examples by showing you the adapter entries followed by the controller entries followed by the tape drive entries for devices connected to the TURBOchannel:

```
# ibus entries for DECstation 5000 Model 200
# IO option slots
adapter         ibus0   at nexus?
adapter         ibus1   at nexus?
adapter         ibus2   at nexus?
adapter         ibus3   at nexus?
adapter         ibus4   at nexus?
adapter         ibus5   at nexus?
adapter         ibus6   at nexus?
```

```
adapter         ibus7   at nexus?
controller      asc0    at ibus?        vector ascintr
controller      asc1    at ibus?        vector ascintr
controller      asc2    at ibus?        vector ascintr
controller      asc3    at ibus?        vector ascintr
tape            tz0     at asc0         drive 0
tape            tz1     at asc0         drive 1
tape            tz2     at asc0         drive 2
tape            tz3     at asc0         drive 3
```

9.2 Installing VMEbus Device Drivers

This section assumes you are familiar with the syntax associated with modifying the appropriate system files, as discussed in previous sections of this chapter.

To add a VMEbus driver to the ULTRIX operating system, follow these steps:

1. Write the device driver. The name of the device driver source file should be in the following form, where `devname.c` represents the device name (for example, `xx.c`, `lp.c`, and `dz.c`):

   ```
   devname.c
   ```

2. Copy the device driver source files into the appropriate directory. If you are providing only the object files, then copy them into the `/sys/MIPS/BINARY` directory. If you are providing the source files, copy them into an appropriate directory. Directories are distinguished by bus type. For VMEbus drivers, copy the driver source into the `/usr/sys/io/vme` directory.

3. Make an entry in `/usr/sys/conf/mips/MACHINE`, the system configuration file, to add the device to the system configuration. `MACHINE` represents the name of the system you want to configure, for example, TIGRIS. The entry must follow the syntaxes associated with the VMEbus, as described in previous sections. For example:

   ```
   controller td0 at vba0 csr 0x8020 priority 1 vector tdintr 0x45
   device xcm0 at vba0 csr 0xa000 priority 3 vector xcmintr 0xc8
   ```

4. Add the driver source file to `/usr/sys/conf/mips/files.mips` as either `Binary` or `Notbinary`. The following example shows the addition of a driver without source code:

   ```
   io/vme/td.c     optional    td     device_driver     Binary
   ```

   The following example shows the addition of a driver with source code:

   ```
   io/vme/xcm.c     optional    xcm     device-driver     Notbinary
   ```

5. Declare the device driver entry points for your device by editing the `/usr/sys/machine/common/conf.c` file. The following shows device driver routine declarations for a VMEbus device driver:

   ```
   #include "sk.h"
   #if NSK > 0
   int     skopen(),skclose(),skmmap();
   #else
   #define skopen          nodev
   #define skclose         nodev
   #define skmmap          nodev
   #endif
   ```

   First, you include the device driver header file that was created by `config`.

The `config` command creates this header file by using the name of the controller or device that you specified in the system configuration file. In this example, the header file is `sk.h`, which indicates that the characters ''sk'' were previously specified for a memory-mapped device in the system configuration file. Next, you declare the device driver routines that were defined in the `cdevsw` or the `bdevsw` if the device constant (or constants) is greater than zero, which indicates that the device is actually in the system configuration file. The device constant was also created by `config` in the following way:

- It locates the name of the controller or device that you specified in the system configuration file.
- It converts the lowercase name to uppercase.
- It appends the uppercase name to the letter ''N.''

In this example, the device constant is `NSK`, and the `sk` routines defined in the `cdevsw` are declared to return a value of type `int`. Otherwise, if the device is not actually in the system configuration file, you declare the entry points as nodev.

6. Modify the `cdevsw` or `bdevsw` table. To do this, edit the `/usr/sys/machine/common/conf.c` file and search for `struct cdevsw` or `struct bdevsw`. Add your entries to the end of the table. The easiest way to add entries to the tables is to copy the previous entry, change the driver entry point names, and increment the comment by 1. The number in the comment is your major device number. Keep this number for use in a subsequent step. The following example shows an entry for a VMEbus device driver, along with the entry that precedes it:

```
struct cdevsw   cdevsw[] =
 {
 .
 .
 .
 {spopen,        spclose,        spread,         spwrite, /*74*/
  spioctl,        spstop,         spreset,        sp_tty,
  nodev,          nodev,          0,      0},
 .
 .
 .
/* VMEbus device driver entry points     */
 {skopen,        skclose,        nodev,          nodev, /*77*/
 nodev,        nodev,          nulldev,        0,
 asyncsel,       skmmap,          nulldev,      0},
};
```

7. Run `config` on the `MACHINE` file from the `/usr/sys/conf/mips` directory. Most of the problems you encounter here will be syntactical errors you introduced while editing the `MACHINE` file and the `files.mips` file. In the following example, `config` is run on the system called TIGRIS:

```
% config TIGRIS
```

8. Create a file system entry for your device. Use the `mknod` command:

```
% mknod /dev/sk c 77 0
```

In this example, *c* represents character (as opposed to *b* for block), 77 is the major number (the number you were told to record when you added the device to the table), and 0 is the minor number.

9. Create a new kernel by going to the `/usr/sys/MIPS/MACHINE` directory, which was created by `config`. Specify the following:

   ```
   % cd /usr/sys/MIPS/TIGRIS
   % make depend all
   ```

   Some common errors are coding errors in your driver, especially if the driver was defined as `Notbinary`. In addition, you may obtain errors from the `MACHINE` file, the `files.mips` file, and the `conf.c` file.

10. If a new kernel was built successfully, you may still want to back up the existing kernel and then place the new kernel in `/vmunix`. For example:

    ```
    % mv /vmunix /vmunix.sav
    % cp vmunix /vmunix
    ```

Use the following for specific modifications:

- To modify driver source code, start with step 9.
- To add a new device, start with step 7.
- To change csr addresses or vectors, perform steps 3, 6, and 8-10.
- To add entry points, perform steps 5, 6, and 8-10.

9.3 Installing TURBOchannel Device Drivers

To add a TURBOchannel driver to the ULTRIX operating system, follow these steps:

1. Write the device driver. The name of the device driver source file should be in the following form, where `devname.c` represents the device name (for example, `asc.c` and `dc.c`):

   ```
   devname.c
   ```

2. Copy the device driver source files into the appropriate directory. If you are providing only the object files, then copy them into the `/sys/MIPS/BINARY` directory. If you are providing the source files, copy them into an appropriate directory. Directories are distinguished by bus type. For TURBOchannel drivers, copy the driver source into the `/usr/sys/io/tc` directory.

3. Make an entry in `/usr/sys/data/tc_option_data.c`, the tc option data table. This table provides a mapping between the device name in the ROM on the hardware device module and the driver in the ULTRIX kernel. The following shows a sample `tc_option_data.c` file:

   ```
   struct tc_option tc_option [] =
   {
   /*  module         driver  intr_b4 itr_aft         adpt    */
   /*  name           name    probe   attach  type    config  */
   /*  ------         ------  ------- ------- ----    ------  */
   {   "PMAD-AA ",    "ln",    0,       1,    'D',    0},     /* Lance */
   {   "PMAZ-AA ",    "asc",   0,       1,    'C',    0},     /* SCSI */
   {   "PMAG-BA ",    "qac",   0,       0,    'D',    0},     /* QAC */
   {   "PMAG-CA ",    "ga",    0,       1,    'D',    0},     /* 2DA */
   {   "PMAG-DA ",    "gq",    0,       1,    'D',    0},     /* 3DA */
   {   "PMAG-FA ",    "???",   0,       1,    'D',    0},     /* Reserved */
   /*
    * Do not delete any table entries above this line or your system
    * will not configure properly.
    *
    * Add any new controllers or devices here.
   ```

```
 * Remember, the module name must be blank padded to 8 bytes.
 */
/*
 * Do not delete this null entry, which terminates the table or your
 * system will not configure properly.
 */
 {   "",              ""       }  /* Null terminator in the table */
};
```

The items in the tc option table have the following meanings:

module name

In this column, you specify the device name in the ROM on the hardware device. The module names in the example are seven characters in length. However, you must blank-pad the name to eight bytes.

driver name

In this column, you specify the driver name as it appears in the system configuration file.

intr_b4 probe

In this column, you specify whether the device needs interrupts enabled during execution of the `probe` routine. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled.

intr_aft attach

In this column, you specify whether the device needs interrupts enabled after the `probe` and `attach` routines complete. A value of 1 indicates that the device does not need interrupts enabled; a value of zero (0) indicates that the device needs interrupts enabled.

type

In this column, you specify the type of device: `D` (device), `C` (controller), or `A` (adapter).

adpt config

If the device type in the previous column is `A` (adapter), then you specify the name of the routine to configure the adapter.

4. Make an entry in `/usr/sys/conf/mips/MACHINE`, the system configuration file, to add the device to the system configuration. `MACHINE` represents the name of the system you want to configure, for example, TIGRIS. For example:

```
device          qac0       at ibus?    vector qacvint
```

5. Add the driver source file to `/usr/sys/conf/mips/files.mips` as either `Binary` or `Notbinary`. The following example shows the addition of a driver without source code:

```
io/tc/qac.c  optional qac device-driver Binary
```

The following example shows the addition of a driver with source code:

```
io/tc/qac.c  optional qac device-driver Notbinary
```

6. Declare the device driver entry points for your device by editing the `/usr/sys/machine/common/conf.c` file. The following example shows

device driver routine declarations for a TURBOchannel device driver:

```
#include "qac.h"
#if NQAC > 0
int     qacopen(),qacclose(),qacread(),qacwrite(),qacioctl();
int     qacstop();
#else
#define qacopen           nodev
#define qacclose          nodev
#define qacread           nodev
#define qacwrite          nodev
#define qacioctl          nodev
#define qacstop           nodev
#endif
```

First, you include the device driver header file that was created by `config`. The `config` command creates this header file by using the name of the controller or device that you specified in the system configuration file. In this example, the header file is `qac.h`, which indicates that the characters ''qac'' were previously specified for this device in the system configuration file. Next, you declare the device driver routines that were defined in the `cdevsw` or the `bdevsw` if the device constant (or constants) is greater than zero, which indicates that the device is actually in the system configuration file. The device constant was also created by `config` in the following way:

- It locates the name of the controller or device that you specified in the system configuration file.
- It converts the lowercase name to uppercase.
- It appends the uppercase name to the letter ''N.''

In this example, the device constant is `NQAC`, and the `qac` routines defined in the `cdevsw` are declared to return a value of type `int`. Otherwise, if the device is not actually in the system configuration file, you declare the entry points as nodev.

7. Modify the `cdevsw` or `bdevsw` table. To do this, edit the `/usr/sys/machine/common/conf.c` file and search for `struct cdevsw` or `struct bdevsw`. Add your entries to the end of the table. The easiest way to add entries to the tables is to copy the previous entry, change the driver entry point names, and increment the comment by 1. The number in the comment is your major device number. Keep this number for use in a subsequent step. The following example shows an entry for a TURBOchannel device driver, along with the entry that precedes it:

```
struct cdevsw    cdevsw[] =
 {
 .
 .
 .
{propen,         nulldev,          nulldev,         nulldev, /*75*/
prioctl,         nulldev,          nulldev,         0,
nodev,           nodev,            0,       0},
/* TURBOchannel qac device driver entry points      */
 {qacopen,        qacclose,         qacread,         qacwrite, /*76*/
 qacioctl,        qacstop,          nulldev,         0,
 asyncsel,        nodev,            nodev,           0},
};
```

8. Run `config` on the `MACHINE` file from the `/usr/sys/conf/mips` directory. In the following example, `config` is run on the system called

TIGRIS:

```
% config TIGRIS
```

Most of the problems you encounter here will be syntactical errors you introduced while editing the `MACHINE` file and the `files.mips` file.

9. Create a file system entry for your device. Use the `mknod` command:

   ```
   % mknod /dev/qac c 76 0
   ```

   The entry *c* represents character (as opposed to *b* for block), 76 is the major number (the number you were told to record when you added the device to the table), and zero (0) is the minor number.

10. Create a new kernel by going to the `/usr/sys/MIPS/MACHINE` directory, which was created by `config`. Specify the following:

    ```
    % cd /usr/sys/MIPS/TIGRIS
    % make depend all
    ```

    Some common errors are coding errors in your driver, especially if the driver was defined as `Notbinary`. In addition, you may obtain errors from the `MACHINE` file, the `files.mips` file, and the `conf.c` file.

11. If a new kernel was built successfully, you may still want to back up the existing kernel and then place the new kernel in `/vmunix`. For example:

    ```
    % mv /vmunix /vmunix.sav
    % cp vmunix /vmunix
    ```

Use the following for specific modifications:

- To modify driver source code, start with step 10.
- To add a new device, start with step 8.
- To change vectors, perform steps 4, 7, 10, and 11.
- To add entry points, perform steps 6, 7, 8, 10, and 11.

Part V: Example Drivers

VMEbus Device Driver Examples

10

This chapter provides the following example VMEbus device drivers:

- Memory-mapped device driver
- DMA device driver

The main purpose of these examples is to illustrate techniques and strategies for writing VMEbus device drivers. Although these examples are not working drivers, you can use them as the basis for writing working drivers. The source code for the examples is located in the `/usr/examples/devdrivers` directory, which includes:

- `vmemmap.c`

  Contains the VMEbus memory-mapped example
- `vmedma.c`

  Contains the VMEbus DMA example

10.1 Memory-Mapped Device Driver

The memory-mapped device driver example illustrates a driver that provides a memory map mechanism for a generic memory-mapped device. For convenience in reading and studying the memory-mapped device driver, the source code is divided into parts. Table 10-1 lists the parts of the memory-mapped device driver and the sections of the chapter where each is discussed.

Table 10-1: Parts of the Memory-Mapped Device Driver

Part	Section
Include Files	Section 10.1.1
Declarations	Section 10.1.2
Autoconfiguration	Section 10.1.3
Open and Close	Section 10.1.4
Memory Mapping	Section 10.1.5

10.1.1 Include Files Section

This example shows the include files section for the memory-mapped device driver:

```
/* sk.c - Memory mapped device driver                   */
/*                                                      */
/* Abstract:                                            */
/*                                                      */
/* This driver provides a memory map mechanism for a    */
/* generic memory mapped device.                        */
/*                                                      */
/* Author: Digital Equipment Corporation                */
/*                                                      */
/********************************************************/
/*        INCLUDE FILES                                 */
/*                                                      */
/********************************************************/

/* Header files required by memory mapped device driver */

#include "sk.h" /* Driver header file generated by config */ [1]
#include "../h/types.h"
#include "../h/errno.h"
#include "../machine/param.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h" /* Include for BADADDR */ [2]
#include "../io/uba/ubavar.h"
#include "../h/ioctl.h"
#include "../h/param.h"
#include "../h/buf.h"
#include "../io/vme/vbareg.h" /* VMEbus definitions */ [3]
#include "../h/vmmac.h"
```

[1] This line includes the `sk.h` file, which is the device driver header file created by `config`. This file is also included in `/usr/sys/machine/common/conf.c`, which is where you define the entry points for most device driver routines. The `sk.h` file contains #define statements for the number of sk devices configured into the system. See Section 9.2 for more information on the `conf.c` file.

[2] The `cpuconf.h` file is where the `BADADDR` macro is defined. ULTRIX device drivers use this macro to determine whether a device is present on the hardware configuration. See the `skprobe` routine in Section 10.1.3 for an example of how the memory-mapped device driver uses `BADADDR`.

[3] The `/usr/sys/io/vme/vbareg.h` header file is specific to VMEbus device drivers. It contains definitions for the different VMEbus adapters. For summary descriptions of other header files used by device drivers, see Appendix A.

10.1.2 Declarations Section

This example shows the declarations section for the memory-mapped device driver:

```
/**********************************************************/
/*         DECLARATIONS                                   */
/*                                                        */
/**********************************************************/

#define SKREGSIZE 256            /* First csr area size */ [1]
#define SKUNIT(dev) (minor(dev)) /* Device minor number */ [2]

/* Driver routines declarations */
int skprobe(),skattach(),skintr(),skmmap();

/* Array of pointers to uba_device structures */ [3]
struct uba_device *skdinfo[NSK];

/* Declare and initialize uba_driver structure */ [4]
struct uba_driver skdriver = {
        skprobe,0,skattach,0,0,
        "sk",skdinfo,0,0,0,SKREGSIZE,VMEA16D16,0,0
};

/* Device register structure */ [5]
struct sk_reg_t {
       volatile char stub_0;    /* Base address */
       volatile char V;         /* First readable, always V */
       volatile char stub_1;    /* Data is only on every other byte */
       volatile char M;         /* Second readable */
       volatile char nonused[124];
       volatile short status;
       volatile unsigned short intvec;
       volatile unsigned short reset;
       volatile char unused[2];
       volatile unsigned short start;
       volatile char nevused[2];
       volatile unsigned short skdata;
       volatile char pads[92];  /* Fills out the remainder of */
                                /* the 256 byte block */
};

/* Define a softc structure for use by the interrupt service */
/* routines, the error log routines, etc. */ [6]
struct sk_softc{
       int sk_time;       /* Timeout value*/
       int sk_expint;     /* Expecting interrupt*/
       int sk_timeout;    /* Timeout situation : true or false */
       int sk_data;       /* Value read after interrupt*/
       int intcnt;        /* Number of times interrupts may happen */
       struct sk_reg_t *sk_base; /* Pointer to sk_reg_t structure */
} sksoftc[NSK];

/* Define debug constants */ [7]
#define SK_DEBUG
#ifdef SK_DEBUG
int sk_debug = 0;
#endif SK_DEBUG
```

[1] This line defines a constant that can be used for the size of the first CSR area. The memory-mapped device driver initializes the `ud_addr1_size` member of the `uba_driver` structure with this constant.

[2] This line defines a constant that represents the device minor number. A call to the `minor` macro obtains the device minor number. This macro takes one argument: the number of the device whose associated minor device number you want to obtain. See Appendix B for a description of the `minor` macro.

[3] This line declares an array of pointers to `uba_device` structures and calls it `skdinfo`. This array is referenced by the driver's `skattach` and `skmmap` routines. The constant `NSK` represents the maximum number of sk devices for a particular hardware configuration. This number is used to size the array of pointers to `uba_device` structures. This constant was defined by `config` in `sk.h`.

[4] The `uba_driver` structure called `skdriver` is initialized to the following:

- The driver's probe routine, `skprobe`.
- The value 0, to indicate that this driver does not use a slave routine.
- The driver's attach routine, `skattach`.
- The value 0, to indicate that this driver does not use a go routine.
- The value 0, because VMEbus device drivers do not use the `ud_addr` member of the `uba_driver` structure.
- The value `sk`, which is the name of the device.
- The value `skdinfo`, which references the array of pointers to the previously declared `uba_device` structures. You index this array with the unit number as specified in the `ui_unit` member of the `uba_device` structure.
- The value 0, to indicate that there is no controller name associated with this device.
- The value 0, to indicate that this driver does not use the `uba_ctlr` structure.
- The value 0, to indicate that this driver does not want exclusive use of the buffer data paths (bdps).
- The value `SKREGSIZE`, to indicate the size in bytes of the first CSR area.
- The value `VMEA16D16`, to indicate the VME address space (A16) and data size (D16) of the first CSR area.
- The value 0, to indicate that this driver does not use the second CSR area. (This member specifies the size of the second CSR area.)
- The value 0, to indicate that this driver does not use the second CSR area. (This member specifies the address space and data size of the second CSR area.)

[5] This line defines a structure called `sk_reg_t` whose members map to the characteristics of the sk device. This structure is referenced in the autoconfiguration section of the memory-mapped driver, specifically by the `skprobe`, `skintr`, and `skmmap` routines. The members of this structure are declared using the key word `volatile` because some of its members correspond to hardware device registers for the sk device. In addition, the values stored in these members could be changed by something other than the device driver. See Section 4.2 for information on when to declare a variable or data structure as volatile.

6 This line declares an array of `softc` structures and calls it `sksoftc`. The size of the array is the value represented by the `NSK` constant. The memory-mapped device driver's `sk_softc` structure allows the interrupt service routines and the error logging routines to share data. Driver routines in the autoconfiguration and memory-mapping sections reference this structure.

7 These lines use several of the C preprocessor statements to set up conditional compilation for debugging purposes. In the sk driver, these statements are used with the `cprintf` kernel routine to print intermediate results to the console terminal.

10.1.3 Autoconfiguration Section

This example shows the autoconfiguration section for the memory-mapped device driver:

```
/***************************************************/
/*        AUTOCONFIGURATION                        */
/*                                                 */
/***************************************************/

/********* Probe Routine ****************************/
/*                                                 */
/* The skprobe routine calls the BADADDR macro to  */
/* determine that there is indeed a board at the   */
/* specified address.  If the board is present,    */
/* skprobe returns the size of the register space that */
/* the board occupies.  If the device is not present, */
/* skprobe returns 0.                              */
/*                                                 */
/***************************************************/

skprobe(unit,addr1)
int unit; /* Unit number associated with the sk device */ 1
caddr_t addr1; /* System Virtual Address for the sk device */ 2
{

        /* Pointer to device register structure */ 3
        register struct sk_reg_t *sk_reg;

        /* Pointer to sk_softc structure */ 4
        register struct sk_softc *sksc;

/* Kernel was properly configured */ 5
#ifdef SK_DEBUG
        if (sk_debug) cprintf("SK probe routine entered\n");
#endif SK_DEBUG

        /* Point to device registers */ 6
        sk_reg = (struct sk_reg_t *)addr1;

        /* Call the BADADDR macro to determine if */
        /* the device is present */ 7
        if (BADADDR ((char *) &sk_reg->V,sizeof(char)) !=0)
        {
            return (0);
         }

        /* Check the first location */ 8
        if (sk_reg->V != 'V') return(0);

        /* Call the BADADDR macro a second time to determine */
        /* if the device is present */ 9
        if (BADADDR ((char *) &sk_reg->M,sizeof(char)) !=0)
        {
             return (0);
        }

        /* Check the second location */ 10
        if (sk_reg->M != 'M') return(0);

        /* Set the pointer to the address of the sk_softc */
        /* structure array */ 11
        sksc = &sksoftc[unit];

        /* Store the base address */ 12
```

```
        sksc->sk_base = (struct sk_reg_t *) addr1;

/* Device found */ 13
#ifdef SK_DEBUG
        if (sk_debug) cprintf("SK driver found\n");
#endif SK_DEBUG

        /* Return size of register space */ 14
        return (SKREGSIZE);
}

/*********** Attach Routine *******************************/
/*                                                        */
/* The skattach routine initializes the device and its    */
/* software state.                                        */
/**********************************************************/

skattach(ui)
struct uba_device *ui; /* Pointer to uba_device structure */ 15
{
/* Attach routine code goes here */
}

/********** Interrupt Routine *****************************/
/*                                                        */
/**********************************************************/

skintr(unit)
int unit; /* Logical unit number of device */ 16
{

        /* Pointer to device register structure */ 17
        register struct sk_reg_t *sk_reg;

        /* Pointer to sk_softc structure */ 18
        register struct sk_softc *sksc;

        /* Set the device's softc structure */ 19
        sksc = &sksoftc[unit];

         /* Store the base address */ 20
        sk_reg = sksc->sk_base;

         /* Check some status word and then set it */ 21
        if (sk_reg->status < 0)
        {
              sk_reg->status = 5;

              /* Read in some data */ 22
              sksc->sk_data = sk_reg->skdata;
        }
}
```

1 This line declares a *unit* variable that is used to specify the sk device.

2 This line declares an *addr1* argument that is the System Virtual Address (SVA) that corresponds to the first CSR address that was specified in the system configuration file for the sk device. Note that the `skprobe` routine would need an *addr2* variable if the sk device used a second CSR area. For this example, this is not the case because the `ud_addr2_size` and `ud_addr2_atype` members of the `uba_driver` structure were previously initialized to 0.

3 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skprobe` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

4 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skprobe` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

5 This section is executed during debugging of the sk driver. The line calls the `cprintf` kernel routine to print the message ''SK probe routine entered'' on the terminal to indicate that the kernel was properly configured. For more information on this routine, see Appendix B.

6 This line initializes the `sk_reg` pointer to the SVA for the memory-mapped device, which is contained in the *addr1* argument. Because the data types are different, this line performs a type casting operation that converts the *addr1* argument (which is of type `caddr_t`) to be of type pointer to an `sk_reg_t` structure.

7 This line calls the `BADADDR` macro to determine if the device is present. The `BADADDR` macro takes two arguments: the address of the device whose existence you want to check and the length of the data to be checked. In this call to the macro, the address of the `V` member of the `sk_reg` pointer is passed. The length is the value returned by the `sizeof` operator, in this case the number of bytes needed to contain a value of type `char` (because the `V` member is a size `char`).

Because the first argument to the `BADADDR` macro is of type `caddr_t`, this line also performs a type casting operation that converts the type of the `V` member (which is of type `char`) to type `char *`. (The data type `caddr_t` is actually a typedef to the data type `char *`.)

If a device is present, `BADADDR` returns the value 0.

8 If a device is present, this line checks the first location. That is, if the `V` member of the `sk_reg` pointer is not equal to the character V, it is not a supported device. Therefore, the `skprobe` routine returns 0.

Some VMEbus devices have proms with an ID that usually starts with the letters VME. Thus, this line reads the prom looking for the specific value V. Your driver code may need to find a more unique string.

9 This line is identical to the one that previously called `BADADDR`, except this time the `M` member of the `sk_reg` pointer is passed. If a device is present, `BADADDR` returns the value 0.

10 If a device is present, this line checks the second location. That is, if the `M` member of the `sk_reg` pointer is not equal to the character M, it is not a supported device. Therefore, the `skprobe` routine returns 0.

11 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device.

12 This line sets the *sk_base* member of the `sksc` pointer to the base address where the device was found, which is contained in the *addr1* argument. Note that the *sk_base* member is a pointer to `sk_reg_t`, the sk device register structure. Therefore this line performs a type casting operation that converts the *addr1* argument (which is of type `caddr_t`) to be of type pointer to an `sk_reg_t` structure.

13 This line is executed during debugging of the sk driver. The line calls the `cprintf` routine to print the message ''SK driver found'' on the terminal to indicate that the `skprobe` routine was successful in finding a device. For more information on this routine, see Appendix B.

14 This line returns the size of the register space, which indicates that the sk board is present.

15 The sk device does not need an attach routine. However, this line shows that your attach routine would declare a pointer to a `uba_device` structure. The driver can send any information contained in this structure to the device.

16 This line declares a *unit* argument that is used to specify the logical unit number of the memory-mapped device that is interrupting.

17 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skintr` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

18 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skintr` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

19 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device. Note that `sksoftc` is the array of structures declared in the Declarations section and that *unit* is the index into this array.

20 This line sets the `sk_reg` pointer to the base address, which is the `sk_base` member of the `sksc` pointer.

21 If the `status` member of the `sk_reg` pointer is less than 0, then this line sets it to the value 5.

22 This line sets the `sk_data` member of the `sksc` pointer to the data contained in the `skdata` member of the `sk_reg` pointer.

10.1.4 Open and Close Section

This example shows the open and close section for the memory-mapped device driver:

```
/**********************************************************/
/*         OPEN AND CLOSE                                 */
/*                                                        */
/**********************************************************/

/********* Open Routine ***********************************/
/*                                                        */
/**********************************************************/

skopen(dev,flag)
dev_t dev; /* Major/minor device number */ [1]
int flag; /* Flags from /usr/sys/h/file.h */ [2]
{
      /* Return to the open system call */ [3]
      return (0);
}

/********* Close Routine **********************************/
/*                                                        */
/**********************************************************/

skclose(dev,flag)
dev_t dev; /* Major/minor device number */ [1]
int flag; /* Flags from /usr/sys/h/file.h */ [2]
{
      /* Return to the close system call */ [3]
      return (0);
}
```

[1] This line declares an integer variable that holds the major and minor device numbers for the memory-mapped device. The minor device number will be used in determining the logical unit number for the memory-mapped device that is to be opened or closed.

[2] This line declares an integer variable to contain flag bits from the file `/usr/sys/h/file.h`. These flags indicate whether the device is being opened for reading, writing, or both.

[3] The `skopen` routine does not do any intricate work other than to return execution to the `open` system call. Likewise, the `skclose` routine simply returns execution to the `close` system call.

10.1.5 Memory-Mapping Section

This example shows the memory-mapping section for the memory-mapped device driver:

```
/***************************************************/
/*         MEMORY MAPPING                          */
/*                                                 */
/***************************************************/

/********* Memory Mapping Routine *****************/
/*                                                 */
/* The skmmap routine is invoked by the kernel as a      */
/* result of an application calling the mmap(2) system   */
/* call.  The skmmap routine makes sure that the         */
/* specified offset into the memory mapped device's      */
/* memory is valid.  If the offset is not valid, skmmap  */
/* returns -1.  If the offset is valid, skmmap returns   */
/* the page frame number corresponding to the page at    */
/* the specified offset.                                 */
/*                                                       */
/*********************************************************/

skmmap(dev, off, prot)
dev_t dev; /* Device whose memory is to be mapped */ [1]
off_t off; /* Byte offset into device memory */ [2]
int prot;  /* Protection flag: PROT_READ or PROT_WRITE */ [3]
{
      /* Pointer to device register structure */ [4]
      register struct sk_reg_t *sk_reg;

      /* Pointer to sk_softc structure */ [5]
      register struct sk_softc *sksc;

      /* Page frame number */ [6]
      int kpfnum;

      /* Make sure that the offset into the device registers */
      /* is less than the size of the device register space. */ [7]
      if ((u_int) off >= SKREGSIZE)
        return (-1);

      /* Otherwise, set the device's sk_softc structure */ [8]
      sksc = &sksoftc [SKUNIT(dev)];

      /* and store the base address */ [9]
      sk_reg = sksc->sk_base;

      /* Find the register space of the device */ [10]
      kpfnum = vtokpfnum(sk_reg+off);
      return kpfnum;
}
```

[1] This line declares a *dev* argument that specifies the character device whose memory is to be mapped.

[2] This line declares an *off* argument that specifies the offset in bytes into the character device's memory. The offset must be a valid offset into the device memory.

[3] This line declares a *prot* argument that specifies the protection flag for the mapping. The protection flag is the bitwise inclusive OR of these valid protection flag bits defined in `/usr/sys/h/mman.h`: `PROT_READ` or

`PROT_WRITE.`

4 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skmmap` routine makes reference to this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

5 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skmmap` routine makes reference to this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

6 This line declares a *kpfnum* variable to contain the page frame number returned by the `vtokpfnum` kernel routine.

7 If the offset into the memory-mapped device's memory is greater than or equal to the size of the first CSR area, the `skmmap` routine returns –1. This value indicates an unsuccessful attempt at mapping this device's memory into the user's address space. This line also performs a type casting operation that converts the *off* argument (which is of type `off_t`) to be of type `u_int`. The reason is to ensure that you compare an unsigned quantity because the offset may be a full longword.

8 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device. Note the use of the `SKUNIT` macro to obtain the minor number associated with this sk device.

9 This line sets the `sk_reg` pointer to the base address, which is the `sk_base` member of the `sksc` pointer.

10 This line calls the `vtokpfnum` kernel routine. This routine takes one argument: the kernel virtual address whose page frame number is to be returned. In this example, this address is the result of the expression whose operands consist of the pointer to the `sk_reg_t` structure and the byte offset. Upon completing execution successfully, `vtokpfnum` sets the *kpfnum* variable to the page frame number associated with the page in the sk device's memory. See Appendix B for a description of the `vtokpfnum` kernel routine.

10.2 DMA Device Driver

The DMA device driver is a simple DMA interface that uses the 32-bit VMEbus. For convenience in reading and studying the DMA device driver, the source code is divided into parts. Table 10-2 lists the parts of the DMA device driver and the sections of the chapter where each is discussed.

Table 10-2: Parts of the DMA Device Driver

Part	Section
Include Files	Section 10.2.1
Declarations	Section 10.2.2
Autoconfiguration	Section 10.2.3
Open and Close	Section 10.2.4
Read and Write	Section 10.2.5
Strategy	Section 10.2.6
Interrupt	Section 10.2.7

10.2.1 Include Files Section

This example shows the include files section for the DMA device driver:

```
/* xx.c - DMA device driver                             */
/*                                                      */
/* Abstract:                                            */
/*                                                      */
/* This driver supports an XX device.  The XX device    */
/* is a simple DMA interface that uses the              */
/* 32-bit VMEbus.                                       */
/*                                                      */
/* Author: Digital Equipment Corporation                */
/*                                                      */
/********************************************************/
/*        INCLUDE FILES                                 */
/*                                                      */
/********************************************************/
/*                                                      */
/* Header files required by DMA device driver */
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/file.h"
#include "../h/map.h"
#include "../machine/cpu.h"
#include "../io/uba/ubavar.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h" /* Include for BADADDR */ [1]
#include "../io/vme/vbareg.h" /* VMEbus definitions */ [2]
#include "xx.h" /* Driver header file generated by config */ [3]
```

[1] The `cpuconf.h` file is where the `BADADDR` macro is defined. ULTRIX device drivers use this macro to determine whether a device is present on the hardware configuration. See the `xxprobe` routine in Section 10.2.3 for an example of how the DMA device driver uses `BADADDR`.

[2] The `/usr/sys/io/vme/vbareg.h` header file is specific to VMEbus device drivers. It contains definitions for the different VMEbus adapters. For summary descriptions of other header files used by device drivers, see Appendix A.

[3] This line includes the `xx.h` file, which is the device driver header file created by `config` for the XX device. This file is also included in `/usr/sys/machine/common/conf.c`, which is where you define the entry points for most device driver routines. The `xx.h` file contains #define statements for the number of XX devices configured into the system. See Section 9.2 for more information on the `conf.c` file.

10.2.2 Declarations Section

This example shows the declarations section for the DMA device driver:

```
/**********************************************************/
/*          DECLARATIONS                                  */
/*                                                        */
/**********************************************************/

/********* Register Structure for XX device ***************/
/*                                                        */
/**********************************************************/

struct xx_reg {
    volatile char  csr;          /* One byte control/status register */
    volatile short count;        /* Short byte count */
    volatile unsigned int addr;  /* 32-bit transfer address */
} /* Declare a register structure */ [1]

/********* Bits for csr member ****************************/
/*                                                        */
/**********************************************************/

#define  IE         0001   /* Interrupt Enable */
#define  DMA_GO     0002   /* Start DMA */
#define  RESET      0010   /* Ready for data transfer */
#define  ERROR      0020   /* Indicate error */ [2]
#define  READ       0040   /* Indicate data transfer is read */

/********* Driver Routines ********************************/
/*                                                        */
/**********************************************************/

/* Declare DMA device driver routines */
int  xxprobe(), xxopen(), xxclose(), xxread(), xxwrite(),
     xxstrategy(), xxintr();

/********* buf, uba_device, and uba_driver Structures *******/
/*                                                          */
/************************************************************/

/* Declare an array of buf structures */ [3]
struct buf xxbuf[NXX];

/* Declare an array of pointers to uba_device structures */ [4]
struct uba_device *xxdinfo[NXX];

/* Declare and initialize uba_driver structure */ [5]
struct uba_driver xxdriver = {
        xxprobe,0,0,0,0,
        "xx",xxdinfo,0,0,0,0x5,VMEA32D32,0,0
};

/********* Unit Number Compare Variable *******************/
/*                                                        */
/**********************************************************/

/* Declare and initialize unit number compare variable */ [6]
int nNXX=NXX;

/********** Softc Structure *******************************/
/*                                                        */
/**********************************************************/

/* Declare an xx_softc structure */ [7]
```

```
struct xx_softc {
        char  sc_csr;           /* A copy of csr */
        int   sc_open;          /* XXOPEN, XXCLOSE */
#define XXOPEN  1
#define XXCLOSE  0
        int   sc_error;         /* Driver specific error code */
#define EACCFAULT  200          /* Access violation */
#define ENOMAPREG  201          /* No mapping registers */
#define EBUFTOOBIG 202          /* Buffer too big */
        unsigned int vmeaddr;  /* Return for vbasetup */
        struct buf *bp;         /* To save buffer pointer */
                                /* for use by xxintr */
} xx_softc [NXX];
```

1 The XX device has a 1–byte control/status register, a 16-bit byte count, and a 32-bit transfer address. The `xx_reg` structure describes these XX device characteristics by defining these members: `csr`, `count`, and `addr`. The members of this structure are declared using the key word `volatile` because some of its members correspond to hardware device registers for the XX device. In addition, the values stored in these members could be changed by something other than the device driver (that is, the controller itself). See Section 4.2 for information on when to declare a variable or data structure as volatile.

2 The symbolic constants used to define the bits for the `csr` member of the `xx_reg` structure are used by the `xxprobe` routine.

3 This line declares an array of `buf` structures and calls it `xxbuf`. This array is referenced by the driver's `xxread` and `xxwrite` routines. Note that the constant `NXX` is used to size the array. This constant was created by the `config` command using the name of the device, in this case xx, that you specified in the system configuration file.

4 This line declares an array of pointers to `uba_device` structures and calls it `xxdinfo`. This array is referenced by the driver's `xxopen`, `xxclose`, `xxstrategy`, and `xxintr` routines. Note the use of the `NXX` constant to size the array.

5 The `uba_driver` structure called `xxdriver` is initialized to the following:

- The driver's probe routine, `xxprobe`.
- The value 0, to indicate that this driver does not use a slave routine.
- The value 0, to indicate that this driver does not use an attach routine.
- The value 0, to indicate that this driver does not use a go routine.
- The value 0, because VMEbus device drivers do not use the `ud_addr` member of the `uba_driver` structure.
- The value `xx`, to indicate the name of the device.
- The value `xxdinfo`, which references the array of pointers to the previously declared `uba_device` structures.
- The value 0, to indicate that there is no controller name associated with this device.

- The value 0, to indicate that this driver does not use the `uba_ctlr` structure.
- The value 0, to indicate that this driver does not want exclusive use of the buffer data paths (bdps).
- The value `0x5`, to indicate the size in bytes of the first CSR area.
- The value `VMEA32D32`, to indicate the address space (A32) and data size (D32) of the first CSR area.
- The value 0, to indicate that this driver does not use the second CSR area. (This member specifies the size of the second CSR area.)
- The value 0, to indicate that this driver does not use the second CSR area. (This member specifies the address space and data size of the second CSR area.)

6 The *nNXX* variable is initialized to the value of the `NXX` constant. The *nNXX* variable is used by the driver's `xxopen` routine.

7 The `xx_softc` structure allows the DMA device driver's routines to share data. The driver routines that reference this structure are `xxopen`, `xxclose`, `xxstrategy`, and `xxintr`. Again, note that the constant `NXX` is used to size the array.

10.2.3 Autoconfiguration Section

This example shows the autoconfiguration section for the DMA device driver:

```
/**********************************************************/
/*         AUTOCONFIGURATION                              */
/*                                                        */
/**********************************************************/

/********* Probe Routine **********************************/
/*                                                        */
/* The xxprobe routine is called from the ULTRIX          */
/* configuration code during the boot phase.  The xxprobe */
/* routine calls the BADADDR macro to determine           */
/* if the device is present.  If the device is present,   */
/* xxprobe returns the size of the device structure.      */
/* If the device is not present, xxprobe returns 0.       */
/**********************************************************/

xxprobe(unit, addr1)
int unit; /* Unit number for XX device */ [1]
caddr_t addr1; /* System Virtual Address for the XX device */ [2]
{
      /* Initialize pointer to an xx_reg structure */ [3]
      register struct xx_reg *reg = (struct xx_reg *) addr1;

      /* Determine if device is present */ [4]
      if (BADADDR( (caddr_t) &reg->csr, sizeof(char)) !=0)
      {
          return(0);
      }

      /* Reset the device */ [5]
      reg->csr = RESET;

      /* Assure that write to I/O space completes */ [6]
      wbflush();

      /* If device error, return 0 */ [7]
      if (reg->csr & ERROR)
      {
          return(0);
      }

      /* Otherwise, initialize the csr */ [8]
      reg->csr = 0;

      /* Assure that write to I/O space completes */ [9]
      wbflush();

      /* Return size of xx_reg structure */ [10]
      return (sizeof(struct xx_reg));
}
```

[1] This line declares a *unit* variable that is used to specify the logical unit number of the XX device.

[2] This line declares an *addr1* argument that is the System Virtual Address (SVA) that corresponds to the first CSR address that was specified in the system configuration file for the XX device. Note that the `xxprobe` routine would need an *addr2* variable if the XX device used a second CSR area. For this example, this is not the case because the `ud_addr2_size` and `ud_addr2_atype` members of the `uba_driver` structure were previously initialized to 0.

3 This line declares a pointer to an `xx_reg` structure and calls it `reg`. The `xx_reg` structure and its associated members were previously defined in the Declarations section. This line also initializes `reg` to the SVA for the XX device, which is represented by the *addr1* argument. Because the data types are different, this line performs a type casting operation that converts the *addr1* argument (which is of type `caddr_t`) to be of type pointer to an `xx_reg` structure.

4 This line calls the `BADADDR` macro to determine if the device is present. The `BADADDR` macro takes two arguments: the address of the device whose existence you want to check and the length of the data to be checked. In this call to the macro, the address of the `csr` member of the `reg` pointer is passed. The `csr` member maps to the one byte control/status register for this XX device. The length is the value returned by the `sizeof` operator, in this case the number of bytes needed to contain a value of type `char` (because the `csr` member is a size `char`).

Because the first argument to the `BADADDR` macro is of type `caddr_t`, this line also performs a type casting operation that converts the type of the `csr` member (which is of type `char`) to type `caddr_t`.

If a device is present, `BADADDR` returns the value 0.

5 This line sets the XX device's control/status register (represented by the `csr` member of the `reg` pointer) to the bit represented by the constant `RESET`. This bit instructs the device to reset itself in preparation for data transfer operations.

6 This line calls the kernel routine `wbflush` to ensure that a write to I/O space has completed. See Appendix B for detailed information on `wbflush`.

7 If the result of the bitwise AND operation produces a nonzero value (that is, the error bit is set), then `xxprobe` returns the value 0 to the configuration code to indicate that the device is broken.

8 If the result of the bitwise AND operation produces a zero value (that is the error bit is not set), then `xxprobe` initializes the device's control/status register (represented by the `csr` member of the `reg` pointer) to the value 0.

9 The `wbflush` routine is called a second time to ensure that a write to I/O space has completed.

10 The `xxprobe` routine returns to the configuration code the size of the device structure, which indicates that the device is present.

10.2.4 Open and Close Section

This example shows the open and close section for the DMA device driver:

```
/***************************************************/
/*          OPEN AND CLOSE                         */
/*                                                 */
/***************************************************/

/********* Open Routine ****************************/
/*                                                 */
/* The xxopen routine is called from the ULTRIX    */
/* spec_open routine.  The xxopen routine checks   */
/* that the device is open uniquely.  In addition, it */
/* initializes the flag variable.                  */
/***************************************************/

xxopen(dev, flag)
dev_t  dev; /* Major/minor device number */ 1
int flag;   /* Flags from /usr/sys/h/file.h */ 2
{

      /* Initialize unit to the minor device number */ 3
      register int unit = minor(dev);

      /* Initialize pointer to uba_device structure */ 4
      register struct uba_device  *devptr = xxdinfo[unit];

      /* Initialize pointer to xx_softc structure */ 5
      register struct xx_softc *sc = &xx_softc[unit];

      /* Make sure that the unit number is no more than the */
      /* system configured */ 6
      if (unit >= nNXX )
             return (EIO);

      /* Make sure the open is unique */ 7
      if (sc->sc_open == XXOPEN)
             return (EBUSY);

      /* If device is initialized, set sc_open */
      /* and return 0 to indicate success. */ 8
      if ((devptr !=0) && (devptr->ui_alive == 1))
      {
             sc->sc_open = XXOPEN;
             return(0);
       }
      /* Otherwise, return an error. */ 9
      else return(ENXIO);
}
```

1 This line declares a variable that holds the major and minor device numbers for the XX device. The minor device number will be used in determining the logical unit number for the XX device that is to be opened.

2 This line declares an integer variable to contain flag bits from the file `/usr/sys/h/file.h`. These flags indicate whether the device is being opened for reading, writing, or both.

3 This line declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

4 This line declares a pointer to a `uba_device` structure and calls it `devptr`. This line also initializes `devptr` to the `uba_device` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `uba_device` structures to determine which `uba_device` structure is associated with this XX device.

5 This line declares a pointer to an `xx_softc` structure and calls it `sc`. This line also initializes `sc` to the address of the `xx_softc` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `xx_softc` structures to determine which `xx_softc` structure is associated with this XX device.

6 If the device minor number (*unit*) is greater than or equal to the number of devices configured by the system, this line returns the error code `EIO`, which indicates an I/O error. This error code is defined in `/usr/sys/h/errno.h`.

7 If the `sc_open` member of the `sc` pointer is equal to the `XXOPEN` constant, this line returns the error code `EBUSY`, which indicates that the XX device has already been opened. This error code is defined in `/usr/sys/h/errno.h`.

8 If the `devptr` pointer is not equal to 0 and the `ui_alive` member of `devptr` is equal to 1, then the device exists. If this is the case, the `xxopen` routine sets the `sc_open` member of the `sc` structure to the open bit `XXOPEN` and returns 0 to indicate a successful open.

9 If the device does not exist, `xxopen` returns the error code `ENXIO`, which indicates that the device does not exist. This error code is defined in `/usr/sys/h/errno.h`.

```
/********* Close Routine *********************************/
/*                                                       */
/* The xxclose routine is called from the ULTRIX         */
/* spec_close routine.  The xxclose routine clears the   */
/* XXOPEN flag to allow other processes to use the       */
/* device.                                               */
/*                                                       */
/*********************************************************/

xxclose(dev, flag)
dev_t  dev; /* Major/minor device number */ 1
int flag;   /* Flags from /usr/sys/h/file.h */ 2
{

        /* Initialize unit to the minor device number */ 3
        register int unit = minor(dev);

        /* Initialize pointer to uba_device structure */ 4
        register struct uba_device  *devptr = xxdinfo[unit];

        /* Initialize pointer to xx_softc structure */ 5
        register struct xx_softc *sc = &xx_softc[unit];

        /* Initialize pointer to xx_reg structure */ 6
        struct xx_reg *reg = (struct xx_reg *) devptr->ui_addr;

        /* Turn off the open bit. */ 7
        sc->sc_open = XXCLOSE;

        /* Turn off interrupts. */ 8
        reg->csr = 0;

        /* Assure write to I/O space completes. */ 9
```

```
        wbflush();

        /* Return success. */ [10]
        return(0);
}
```

[1] This line declares a variable that holds the major and minor device numbers for the XX device. The minor device number will be used in determining the logical unit number for the XX device that is to be closed.

[2] This line declares a *flag* argument. Note that although the `xxclose` routine declares a *flag* argument, it does not use it.

[3] This line declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

[4] This line declares a pointer to a `uba_device` structure and calls it `devptr`. This line also initializes `devptr` to the `uba_device` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `uba_device` structures to determine which `uba_device` structure is associated with this XX device.

[5] This line declares a pointer to an `xx_softc` structure and calls it `sc`. This line also initializes `sc` to the address of the `xx_softc` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `xx_softc` structures to determine which `xx_softc` structure is associated with this XX device.

[6] This line declares a pointer to the `xx_reg` structure and calls it `reg`. This line also initializes `reg` to the SVA of the device's registers, which is represented by the value stored in the `ui_addr` member of the `uba_device` structure associated with this XX device.

Because the `ui_addr` member is of type `caddr_t`, this line also performs a type casting operation that converts the type of the `ui_addr` member to type `struct xx_reg *`.

[7] This line sets the `sc_open` member of the `sc` pointer to the close bit `XXCLOSE`.

[8] This line turns off interrupts by setting the device's control/status register (represented by the `csr` member of the `reg` pointer) to the value 0.

[9] This line calls the `wbflush` kernel routine to assure that a write to I/O space has completed. See Appendix B for detailed information on `wbflush`.

[10] The `xxclose` routine returns the value 0 to `spec_close`, to indicate a successful close of the XX device.

10.2.5 Read and Write Section

This example shows the read and write section for the DMA device driver:

```
/***********************************************************/
/*         READ AND WRITE                                  */
/*                                                         */
/***********************************************************/

/********* Read Routine **********************************/
/*                                                         */
/* The xxread routine is called from the ULTRIX            */
/* spec_rwgp routine.  The xxread routine will call        */
/* the ULTRIX physio routine to perform the buffer         */
/* lock, buffer check, I/O package set up.                 */
/* The physio routine calls the xxstrategy routine         */
/* to access the device.                                   */
/*                                                         */
/***********************************************************/

xxread(dev, uio)
dev_t  dev; /* Major/minor device number */ [1]
struct uio *uio; /* Pointer to uio structure */ [2]
{

      /* Initialize unit to the minor device number */ [3]
      register int unit = minor(dev);

      /* Call physio to perform buffer lock, buffer check, and */
      /* I/O package set up. */ [4]
      return (physio(xxstrategy, &xxbuf[unit], dev, B_READ, minphys, uio));
}
```

[1] This line declares a variable that holds the major and minor device numbers for the XX device. The minor device number is used to determine the logical unit number for the device on which the read operation is performed.

[2] Specifies a pointer to a `uio` structure. This structure contains the information for transferring data to and from the address space of the user's process. You typically pass this structure unchanged to the `uiomove` or `physio` routines. See Section 5.1.3 for information on the `uio` structure. For information on `uiomove`, see Appendix B.

[3] This line declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

[4] The `xxread` routine calls the `physio` kernel routine. The following values are passed to `physio`:

- The driver's strategy routine, `xxstrategy`.

  See Section 10.2.6 for a discussion of the `xxstrategy` routine.

- The address of a `buf` structure

  Note that the minor device number (*unit*) is used as an index into the array of `buf` structures to determine the buffer associated with this XX device. This buffer is a special buffer header owned exclusively by this device.

- The device minor number for the XX device
- The `B_READ` bit for the read/write flag

  This bit indicates this is a read operation.
- A `minphys` routine

  This argument is a pointer to a `minphys` routine. The `minphys` kernel routine bounds the data transfer size. You can also provide your own `minphys` routine.
- A `uio` structure

```
/********* Write Routine *********************************/
/*                                                       */
/* The xxwrite routine is called from the ULTRIX         */
/* spec_rwgp routine.  The xxwrite routine will call     */
/* the ULTRIX physio routine to perform the buffer       */
/* lock, buffer check, I/O package set up.               */
/* The physio routine calls the xxstrategy routine       */
/* to access the device.                                 */
/*                                                       */
/*********************************************************/

xxwrite(dev, uio)
dev_t  dev; /* Major/minor device number */
struct uio *uio; /* Pointer to uio structure */
{

      /* Initialize unit to the minor device number */
      register int unit = minor(dev);

      /* Call physio to perform buffer lock, buffer check, and */
      /* I/O package set up. */ [1]
      return (physio(xxstrategy, &xxbuf[unit], dev, B_WRITE, minphys, uio));
}
```

[1] The `xxwrite` routine is almost identical to the `xxread` routine. The only difference is that `xxwrite` uses the `B_WRITE` bit instead of the `B_READ` bit for the read/write flag to indicate that this is a write operation.

10.2.6 Strategy Section

This example shows the strategy section for the DMA device driver:

```
/**********************************************************/
/*         STRATEGY                                        */
/*                                                         */
/**********************************************************/
/*                                                         */
/********* Strategy Routine *******************************/
/*                                                         */
/* The xxstrategy routine is called from the ULTRIX        */
/* physio routine.  The xxstrategy routine first makes     */
/* sure that the user buffer is both readable and          */
/* writeable.  It determines if the buffer size            */
/* is larger than MAXPHYS and then initiates the I/O.      */
/**********************************************************/

xxstrategy(bp)
struct buf  *bp; /* Pointer to buf structure */ 1
{
      /**************************************************/
      /* Declare and initialize: unit variable, pointer */
      /* to uba_device structure, pointer to xx_softc   */
      /* structure, pointer to xx_reg structure, and    */
      /* csr variable.                                  */
      /************************************************/

      register int unit = minor(bp->b_dev); 2
      register struct uba_device *devptr = xxdinfo[unit]; 3
      register struct xx_softc *sc = &xx_softc[unit]; 4
      register struct xx_reg *reg = (struct xx_reg *) devptr->ui_addr; 5
      short csr; 6

       /* Determine if the user buffer is writeable */
       /* during write operations and readable      */
       /* during read operations.                   */ 7
      if (useracc(bp->b_un.b_addr, (int) bp->b_bcount,
         ((bp->b_flags & B_READ)==B_READ?B_READ:B_WRITE))
           == NULL){

              /* Access violation */ 8
              bp->b_error = EACCFAULT;

              /* A copy to sc_error */ 9
              sc->sc_error = bp->b_error;

              /* Flag the error */ 10
              bp->b_flags |= B_ERROR;

              /* Complete the I/O and return execution */
              /* to xxstrategy                         */ 11
              iodone(bp);
              return;
        }

      /* Determine if the buffer size is larger than */
      /* xxMAXPHYS */ 12
#define xxMAXPHYS (64*1024) /* Maximum DMA size for this device */
      if (bp->b_bcount > xxMAXPHYS) {

              /* Indicate error */ 13
              bp->b_error = EBUFTOOBIG;

              /* A copy to the xx_softc structure */ 14
```

```
              sc->sc_error = bp->b_error;

            bp->b_flags |= B_ERROR; [15]

              /* Complete the I/O and return execution */
              /* to xxstrategy                         */ [16]
              iodone(bp);
              return;
        }

        /* Save bp for use in interrupt routine */ [17]
        sc->bp = bp;

        /* Set up the DMA mapping registers */ [18]
        sc->vmeaddr = vbasetup (devptr->ui_vbahd, bp,
                                VME_DMA | VMEA32D32 | VME_BS_NOSWAP,
                                0);

        /* If requested mapping could not be performed */ [19]
        if (sc->vmeaddr == 0) {
                bp->b_error = ENOMAPREG;
                sc->sc_error = bp->b_error;
                bp->b_flags |= B_ERROR;
                iodone(bp);
                return;
        }

         /* If requested mapping could be performed, */
         /* set up the device for transfer.          */ [20]
        reg->addr = sc->vmeaddr;
        reg->count = bp->b_bcount;
        if (bp->b_flags & B_READ)
                csr = READ | IE;
        else
                csr = IE;
        reg->csr = csr | DMA_GO;
        wbflush();
}
```

[1] This line declares a pointer to a `buf` structure and calls it `bp`. The `xxstrategy` routine uses these members of the `buf` structure: `b_dev`, `b_addr`, `b_bcount`, `b_flags`, and `b_error`. See Section 5.1.1 for descriptions of these members of the `buf` structure.

[2] This line declares a variable that holds the major and minor device numbers for the XX device. The device minor number is obtained by calling the `minor` macro. Note that the device number passed to `minor` is the `b_dev` member of the `buf` structure pointed to by `bp`. See Appendix B for a description of the `minor` macro.

[3] This line declares a pointer to a `uba_device` structure and calls it `devptr`. This line also initializes `devptr` to the `uba_device` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `uba_device` structures to determine which `uba_device` structure is associated with this XX device.

[4] This line declares a pointer to an `xx_softc` structure and calls it `sc`. This line also initializes `sc` to the address of the `xx_softc` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `xx_softc` structures to determine which `xx_softc` structure is associated with this XX device.

5 This line declares a pointer to an `xx_reg` structure and calls it `reg`. The `xx_reg` structure and its associated members were previously defined in the Declarations section.

This line also initializes the `reg` pointer to the System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file. This SVA is stored in the `ui_addr` member of the `devptr` pointer.

Because the data types are different, this line performs a type casting operation that converts the `ui_addr` member (which is of type `caddr_t`) to be of type pointer to an `xx_reg` structure.

6 This line declares a variable called *csr*, which stores read, write, and enable interrupts status information.

7 This line calls the `useracc` kernel routine, which determines read or write access to a user segment. The `xxstrategy` routine passes the following to `useracc`:

- The address of the user segment

  This address is the value of the `b_addr` member of the `bp` pointer.

- The size of the user segment

  This size is the value of the `b_bcount` member of the `bp` pointer. Because the second argument to the `useracc` routine is of type `int`, this line also performs a type casting operation that converts the type of the `b_bcount` member (which is of type `long`) to type `int`.

- The read/write access flag

  This flag specifies the desired access, either `B_READ` or `B_WRITE`. Note that this line uses the conditional expression operator to determine the value that gets set for `b_flags`. If the user does not have the appropriate access (that is, read or write), this value is NULL and items 8-12 get executed. Otherwise, the `xxstrategy` routine determines if the buffer size is larger than `MAXPHYS` (item 13).

  You must use `B_READ` to test the `b_flags` member because `B_WRITE` is equal to zero (0).

8 This line sets the `b_error` member of the `bp` pointer to `EACCFAULT`, which indicates there was an access violation when attempting to access the user segment (that is, `b_flags` is equal to NULL).

9 This line copies `EACCFAULT` to the `sc_error` member of the `sc` pointer.

10 This line sets the `B_ERROR` flag in the `b_flags` member of the `buf` structure pointed to by `bp`. This indicates an error occurred on this buffer.

11 This line completes the I/O operation by calling the `iodone` kernel routine. This routine takes a pointer to a `buf` structure as an argument. See Appendix B for a description of this routine. After the I/O completes, `iodone` returns control to `xxstrategy` which in turn returns to the ULTRIX `physio` routine. The `physio` routine was called from the DMA driver's `xxread` or `xxwrite` routine.

12 This line checks the buffer size (the `b_bcount` member of the `bp` pointer) to determine if it is greater than `xxMAXPHYS`. The xxMAXPHYS constant is

defined as 64 * 1024. It represents the maximum DMA for this device.

13 If the buffer size is greater than `xxMAXPHYS`, this line sets the `b_error` member of the `bp` pointer to `EBUFTOOBIG`.

14 This line copies the error `EBUFTOOBIG` to the `sc_error` member of the `sc` pointer.

15 This line sets the `B_ERROR` flag in the `b_flags` member of the `buf` structure pointed to by `bp`. This indicates an error occurred on this buffer.

16 This line completes the I/O operation by calling the `iodone` kernel routine. This routine takes a pointer to a `buf` structure as an argument. After the I/O completes, `iodone` returns control to `xxstrategy` which in turn returns to the ULTRIX `physio` routine. The `physio` routine was called from the DMA driver's `xxread` or `xxwrite` routine.

17 This line saves the pointer to the `buf` structure used by this device. Note that the pointer to the `xx_softc` structure contains as a member a pointer to a `buf` structure.

18 This line calls the `vbasetup` kernel routine, which allocates and sets up the DMA mapping registers. The `vbasetup` routine takes four arguments:

- A pointer to a `vba_hd` strucuture

  In this case, the `ui_vbahd` member of the pointer to the `uba_device` structure gets passed. This member is a back pointer to the `vba_hd` structure associated with this XX device.

- A pointer to a `buf` structure

- VMEbus flags bits

  The flags argument is the bitwise inclusive OR of a valid bit representing the address space and the data size and bits representing other characteristics. In this example, the ORed bits have the following meanings:

Flags Bits	**Meaning**
`VME_DMA`	Specifies the need for DMA access.
`VMEA32D32`	Specifies a request for the 32-bit address space and the 32-bit data size.
`VME_BS_NOSWAP`	Specifies no byte swapping.

- A VMEbus address

  This argument specifies an address in the appropriate DMA space (the A24 or the A32 DMA space). In this example, the value 0 is passed, which indicates that the `vbasetup` routine uses the next available VMEbus address in the A24 or A32 DMA space.

The return value is stored in the `vmeaddr` member of the `sc` pointer.

19 If `vbasetup` could not perform the requested mapping of the DMA mapping registers, it returns 0 to the `vmeaddr` member of the `sc` pointer. The `xxstrategy` routine does the following:

- It sets the `b_error` member of the `bp` structure to the constant `ENOMAPREG`.
- It sets the `sc_error` member of the `sc` pointer to the constant `ENOMAPREG`.
- It sets the `B_ERROR` flag in the `b_flags` member of the `buf` structure pointed to by `bp`. This indicates an error occurred on this buffer.
- It calls the `iodone` kernel routine to complete the I/O operation. This routine takes a pointer to a `buf` structure as an argument. After the I/O completes, `iodone` returns control to `xxstrategy` which in turn returns to the ULTRIX `physio` routine. The `physio` routine was called from the DMA driver's `xxread` or `xxwrite` routine.

20 If `vbasetup` returned a VMEbus address that is mapped to the buffer, then it set up and allocated the DMA mapping registers. The `xxstrategy` routine does the following:

- It sets the XX device's transfer address to the VMEbus address mapped to the buffer. The XX device's transfer address is represented by the `addr` member of the `xx_reg` structure pointed to by `reg`.
- It sets the XX device's byte count register to the size of the requested transfer, in bytes. The XX device's byte count register is represented by the `count` member of the `xx_reg` structure pointed to by `reg`. The size of the requested transfer was stored in the `b_bcount` member of the `buf` structure pointed to by `bp`.
- If the request is for a read operation, the `READ` and `IE` flags are set in the *csr* variable.
- Otherwise, the request is a write and the `IE` flag is set in the *csr* variable.
- It sets the device's control/status register (represented by the `csr` member of the `xx_reg` structure pointed to by `reg`) to the bitwise inclusive OR of the value in *csr* and the bits represented by the `DMA_GO` constant.
- It calls the `wbflush` kernel routine to ensure that writes to I/O space have completed. See Appendix B for detailed information on `wbflush`.

10.2.7 Interrupt Section

This example shows the interrupt section for the DMA device driver:

```
/***************************************************/
/*        INTERRUPT                                */
/*                                                 */
/***************************************************/
/*                                                 */
/********* Interrupt Routine ***********************/
/*                                                 */
/*                                                 */
/* The xxintr routine is the interrupt service routine    */
/* for the XX device.  It releases VMEbus mapping         */
/* registers and flushes the cache if the operation was   */
/* a DMA read.  It then calls iodone to finish the I/O.   */
/***************************************************/

xxintr(unit)
int  unit; /* Logical unit number for device */ [1]
{
      /**********************************************/
      /* Declare and initialize: pointer to uba_device  */
      /* structure, pointer to xx_softc structure,      */
      /* and pointer to xx_reg structure.  Declare      */
      /* pointer to buf structure.                      */
      /**********************************************/

        register struct uba_device *devptr = xxdinfo[unit]; [2]
        register struct xx_softc *sc = &xx_softc[unit]; [3]
        /* Pointer to xx_softc structure */ [4]
        register struct xx_reg *reg =
        (struct xx_reg *) devptr->ui_addr;
        struct buf *bp; [5]

        /* Retrieve saved buf pointer */ [6]
        bp = sc->bp;

        /* If error bit set, error occurred*/ [7]
        if (reg->csr & ERROR) {
                bp->b_error = EIO;
                bp->b_flags |= B_ERROR;
        }

        /* Record the number of bytes remaining */ [8]
        bp->b_resid = reg->count;

        /* Release the mapping registers. */ [9]
        vbarelse(devptr->ui_vbahd, sc->vmeaddr);

        /* If the operation was a read, then it is necessary */
        /* to flush the data cache to ensure that the next */
        /* access will get the newly read data. */ [10]
        if (bp->b_flags & B_READ) bufflush(bp);
        iodone(bp);
}
```

[1] This line declares a *unit* variable that specifies the logical unit number for this XX device that is interrupting. This logical unit number was previously specified in the system configuration file.

[2] This line declares a pointer to a `uba_device` structure and calls it `devptr`. This line also initializes `devptr` to the `uba_device` structure associated with this XX device. The minor device number (*unit*) is used as an index into

the array of `uba_device` structures to determine which `uba_device` structure is associated with this XX device.

3 This line declares a pointer to an `xx_softc` structure and calls it `sc`. This line also initializes `sc` to the address of the `xx_softc` structure associated with this XX device. The minor device number (*unit*) is used as an index into the array of `xx_softc` structures to determine which `xx_softc` structure is associated with this XX device.

4 This line declares a pointer to an `xx_reg` structure and calls it `reg`. The `xx_reg` structure and its associated members were previously defined in the Declarations section.

This line also initializes the `reg` pointer to the System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file. This SVA is stored in the `ui_addr` member of the `devptr` pointer.

Because the data types are different, this line performs a type casting operation that converts the `ui_addr` member (which is of type `caddr_t`) to be of type pointer to an `xx_reg` structure.

5 This line declares a pointer to a `buf` structure and calls it `bp`. The `xxintr` routine uses these members of the `buf` structure: `b_error`, `b_flags`, and `b_resid`. See Section 5.1.1 for descriptions of these members of the `buf` structure.

6 This line retrieves the pointer to the `buf` structure that was saved in the Strategy section. It does this by setting the `bp` pointer to the pointer to the `buf` structure member in the `xx_softc` structure associated with this XX device.

7 If the error bit in the device csr is set, then an error occurred on the transfer. The `xxintr` routine:

- Sets the `b_error` member of the `buf` structure to the error code `EIO`. This code indicates that there was an I/O error.
- Sets the `B_ERROR` flag in the `b_flags` member of the `buf` structure pointed to by `bp`. This indicates an error occurred on this buffer.

8 This line sets the `b_resid` member of the `bp` pointer to the byte count register of the XX device, which is represented by the `count` member of the `reg` pointer. This indicates the data (in bytes) not transferred because of the I/O error.

9 This line calls the `vbarelse` kernel routine to release the resources on the VMEbus adapter registers. The `vbarelse` routine takes two arguments:

- A `vba_hd` structure

  This structure contains the VMEbus adapter number on which the mapping registers were allocated in a call to `vbasetup` in the Strategy section. Note that the `ui_vbahd` member of the pointer to the `uba_device` structure is a back pointer to the `vba_hd` structure associated with this XX device.

- The VMEbus address

  This is the value returned in the previous call to the `vbasetup` routine in the Strategy section. Note that this value was stored in the `vmeaddr` member of the pointer to the `xx_softc` structure.

10 If the transfer was a read, the `xxintr` routine: calls the `bufflush` kernel routine to flush the processor data cache after a read operation. The `xxintr` routine calls the `iodone` kernel routine to indicate that the I/O is complete. This routine takes a pointer to the `buf` structure. This routine is called for all data transfers. After completion, `iodone` returns control back to `xxintr`.

TURBOchannel Device Driver Examples 11

This chapter provides the following example TURBOchannel device drivers:

- qac device driver
- Memory-mapped device driver

The source code for the examples is located in the `/usr/examples/devdrivers` directory, which includes `tcmmap.c`. This source file contains the TURBOchannel memory-mapped example.

11.1 qac Device Driver

For convenience in reading the qac device driver, the source code is divided into parts. Table 11-1 lists the parts of the qac device driver and the section of the chapter where each appears.

Table 11-1: Parts of the qac Device Driver

Part	Section
Include Files	Section 11.1.1
Declarations	Section 11.1.2
Autoconfiguration	Section 11.1.3
Open and Close	Section 11.1.4
Read and Write	Section 11.1.5
ioctl	Section 11.1.6
Interrupt	Section 11.1.7
Start	Section 11.1.8
Stop	Section 11.1.9

Table 11-1: (continued)

Part	Section
Parameter	Section 11.1.10
Break on and break off	Section 11.1.11

11.1.1 Include Files

This example shows the include files section for the qac device driver:

```
/* qac.c -                                                    */
/*                                                            */
/* Abstract:                                                  */
/*                                                            */
/* This driver supports a QAC device.                         */
/*                                                            */
/* Author: Digital Equipment Corporation                      */
/*                                                            */
/************************************************************/
/*       INCLUDE FILES                                        */
/*                                                            */
/************************************************************/
/*                                                            */
/* Header files required by qac device driver */

#include "qac.h" /* Driver header file generated by config */ [1]
#include "../machine/pte.h"
#include "../h/param.h"
#include "../h/systm.h"
#include "../h/ioctl.h"
#include "../h/tty.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/proc.h"
#include "../h/map.h"
#include "../h/buf.h"
#include "../h/vm.h"
#include "../h/conf.h"
#include "../h/file.h"
#include "../h/uio.h"
#include "../h/kernel.h"
#include "../h/devio.h"
#include "../../machine/common/cpuconf.h"
#include "../h/exec.h"
#include "../h/kmalloc.h"
#include "../io/uba/ubavar.h"    /* auto-config headers */
#include "../io/tc/qacreg.h"    /* qac definitions */ [2]
#include "../machine/cpu.h"
#include "../io/tc/tc.h" [3]
```

[1] This line includes the `qac.h` file, which is the device driver header file created by `config`. This file is also included in `/usr/sys/machine/common/conf.c`, which is where you define the entry points for most device driver routines. The `qac.h` file contains #define statements for the number of qac devices configured into the system. See Section 9.1.1 for information on the `conf.c` file.

[2] The `/usr/sys/io/tc/qacreg.h` header file is specific to the qac device driver. It contains definitions for use by the different structures referenced by the qac driver. For summary descriptions of other header files used by device drivers, see Appendix A.

[3] The `/usr/sys/io/tc/tc.h` header file is specific to TURBOchannel device drivers. It contains definitions and routine declarations needed by TURBOchannel device drivers.

11.1.2 Declarations

This example shows the declarations section for the qac device driver:

```
/**********************************************************/
/*         DECLARATIONS                                   */
/*                                                        */
/**********************************************************/
/*                                                        */
/********* Register Structure for QAC device **************/
/*                                                        */
/**********************************************************/
/*                                                        */
typedef unsigned short uhword;
typedef unsigned int uword;
/* Device register structure */ 1
typedef volatile struct {
    uhword   csr;          /* DZ control Status Register */
    uhword   pad0;         /* Set in qacattach */
    uword    pad1;         /* Read in qacint */
    union {
        uhword  rbuf_ro;   /* data/status buffer read in qac_rint */
        uhword  lpr_wo;    /* Sets line characteristics */
    } r1;                  /* Set in qacparam */
    uhword   pad2;
    uword    pad3;
    uhword   tcr;          /* Enable/Disable output interrupts by line */
    uhword   pad4;         /* Set in qac_tint and qacstart */
    uword    pad5;
    union {
        uhword  msr_ro;    /* Not referenced */
        uhword  tdr_wo;    /* Sets line break by line */
    } r3;                  /* Set in qac_tint, qacbreakon, qacbreakoff */
    uhword   pad6;
    uword    pad7;
} DZ_REGISTERS; /* Registers are aligned on double word boundaries */

/*************************************************************/
/********* Driver routines declarations **********************/
/*                                                           */
/*************************************************************/

int qacstart(), qacbaudrate();
int ttrstrt();
int qacprobe(), qacattach();    /* forward decl for qacdriver table */

u_short qacstd[] = { 0 };       /* stub for uba csr address */
struct uba_device *qacinfo[1];  /* storage for uba device structure */

/********* Declare and initialize uba_driver structure ****/ 2
/*                                                        */
/**********************************************************/

struct  uba_driver qacdriver =
        { qacprobe, 0, qacattach, 0, qacstd, "qac", qacinfo };

/********* Additional Structures **************************/
/*                                                        */
/**********************************************************/

/* Device unit structure */ 3
struct qac_unit {
    int  attached;      /* An attach was done for this unit */
    int  adapter;       /* TC slot number of this unit */
    int  brk;           /* Force line break flags */
```

```
    DZ_REGISTERS  *dz; /* Where the dz is */
} qac_unit [NQACOPT];

/* Generic tty driver flags */ 4
struct tty qac_tty[NQACOPT * NQACLINE];

/* Declare qac_pdma structure */ 5
struct qac_pdma {
    char *p_mem;         /* Pseudo DMA transmitter head */
    char *p_end;         /* Pseudo DMA transmitter tail */
} qac_pdma [NQACOPT * NQACLINE];

/* Structure to define valid DZ baud rates */ 6
int qac_speeds[16] = {
        /* B0  */       DZ_LPR_SC_9600,
        /* B50 */       DZ_LPR_SC_50,
        /* B75 */       DZ_LPR_SC_75,
        /* B110 */      DZ_LPR_SC_110,

        /* B134 */      DZ_LPR_SC_135,
        /* B150 */      DZ_LPR_SC_150,
        /* B200 */      -1,
        /* B300 */      DZ_LPR_SC_300,

        /* B600 */      DZ_LPR_SC_600,
        /* B1200 */     DZ_LPR_SC_1200,
        /* B1800 */     DZ_LPR_SC_1800,
        /* B2400 */     DZ_LPR_SC_2400,

        /* B4800 */     DZ_LPR_SC_4800,
        /* B9600 */     DZ_LPR_SC_9600,
        /* B19200 */    DZ_LPR_SC_19200,
        /* B38400 */    -1};

/* Define debug constants */ 7
#define QAC_DEBUG
#ifdef QAC_DEBUG
int qac_debug = 0;
#endif QAC_DEBUG
```

1 This line defines a structure called `DZ_REGISTERS` whose members map to the registers of the QAC device. This structure is declared using the key word `volatile` because some of its members correspond to hardware device registers for the QAC device. In addition, the values stored in these members could be changed by something other than the device driver.

2 The `uba_driver` structure called `qacdriver` is initialized to the following:

- The driver's probe routine, `qacprobe`.
- The value 0, to indicate that this driver does not use a slave routine.
- The driver's attach routine, `qacattach`.
- The value 0, to indicate that this driver does not use a go routine.
- The device's CSR address, represented by this previously defined array.
- The value `qac`, which is the name of the device.
- The value `qacinfo`, which references the array of pointers to the previously declared `uba_device` structures. You index this array with the unit number as specified in the `ui_unit` member of the `uba_device` structure.

3 This line declares an array of structures called `qac_unit`. The size of the array is represented by the constant `NQACOPT`, which is defined in `/usr/sys/io/tc/qacreg.h`. The constant indicates the number of qac option boards configured into the system.

4 This line declares an array of tty structures called `qac_tty`. The size of the array is represented by the result of the expression of the constants `NQACOPT` and `NQACLINE`. As stated previously, `NQACOPT` represents the number of TURBOchannel option slots associated with this qac device. The `NQACLINE` constant represents the number of lines per DZ.

5 This line declares an array of structures called `qac_pdma`. Like the previously declared array of structures, this structure's array size is the result of the expression of the two constants `NQACOPT` and `NQACLINE`.

6 This line declares a structure called `qac_speeds`, which is initialized to the constants that represent the valid DZ baud rates.

7 These lines use several of the C preprocessor statements to set up conditional compilation for debugging purposes. In the qac driver, these statements are used with the `printf` kernel routine to print intermediate results to the console terminal and to the error logger.

11.1.3 Autoconfiguration Section

This example shows the autoconfiguration section for the qac device driver:

```
/***********************************************************/
/*         AUTOCONFIGURATION                               */
/*                                                         */
/***********************************************************/

/********* Probe Routine ***********************************/
/*                                                         */
/* The qacprobe routine is called only after the           */
/* TURBOchannel initialization code verifies that the      */
/* device is present.  Therefore, qacprobe assumes that    */
/* the device is okay.                                     */
/***********************************************************/

qacprobe(vbaddr, unit)
char *vbaddr;            /* Virtual base address of slot */ [1]
struct uba_device *unit;  /* uba_device structure for this */
                         /* ui->ui_unit */ [2]
{
        return(1);       /* Assume that the device is okay */
                         /* because the TC ROM probe worked */
}

/********* Attach Routine **********************************/
/*                                                         */
/* The qacattach routine initializes the qac_unit          */
/* structure and also initializes the csr for the scan     */
/* and interrupt enable.                                   */
/***********************************************************/

qacattach(ui)
struct uba_device *ui; /* uba_device structure for */
                       /* this unit */
{
        struct qac_unit *qp; /* Pointer to qac_unit structure */ [3]
        int i; /* [Mark, Larry: Why is this declared? It's not used. */

        qp = &qac_unit[ui->ui_unit];           /* Pick unit structure */ [4]
        qp->attached = 1;                      /* Mark unit attached */ [5]
        qp->adapter = ui->ui_adpt;             /* Save adapter address */ [6]
        qp->dz = DZ_ADR(ui->ui_addr);          /* Calculate device */
                                               /* register address */ [7]
        qp->dz->csr = DZ_CSR_TIE |
                      DZ_CSR_RIE | DZ_CSR_MSE; /* Enable scan */
                                               /* and interrupts */ [8]
}
```

[1] This line declares an argument that is used to specify the System Virtual Address (SVA) control/status registers for the qac device.

[2] This line declares a pointer to a `uba_device` structure. None of the members of this structure are used by the `qacprobe` routine, because the TURBOchannel initialization code already verified that the device was present. Therefore, `qacprobe` simply returns the value 1.

However, the `qacattach` routine references some of the members of the `uba_device` structure. This structure contains such information as the logical unit number of the device, whether the device is functional, the bus number the device resides on, the address of the control/status registers, and so forth. The driver can send any information contained in this structure to the device. See

Section 5.1.6 for a description of the `uba_device` structure.

[3] This line declares a pointer to a `qac_unit` structure and calls it `qp`. This structure was previously defined in the Declarations section.

[4] This line sets `qp` to the address of the `qac_unit` structure associated with this qac device. The `ui_unit` member of the `uba_device` structure pointed to by `ui` holds the unit number of this qac device. Thus, this member is used as an index into the array of `qac_unit` structures associated with this qac device.

[5] This line indicates that this qac device was attached by setting the `attached` member of the `qac_unit` structure pointed to by `qp` to the value 1.

[6] This line sets the `adapter` member of the `qac_unit` structure pointed to by `qp` to the adapter number associated with this qac device. The adapter number is obtained from the `ui_adpt` member of the `uba_device` structure pointed to by `ui`.

[7] This line calls the `DZ_ADDR` macro, which uses the device's System Virtual Address (SVA) stored in `ui_addr` to calculate this qac device's register address. The `DZ_ADDR` macro is defined in `/usr/sys/io/tc/qacreg.h`. The line also sets the `dz` member of the `qac_unit` structure pointed to by `qp` to this SVA. Note that the `dz` member is a pointer to a `DZ_REGISTERS` structure defined in `qac_unit`.

The SVA is obtained from the `ui_addr` member of the `uba_device` structure pointed to by `ui`.

[8] This line sets the following bits in the `csr` member of the `DZ_REGISTERS` structure pointed to by `dz`:

- The transmit interrupt bit, `DZ_CSR_TIE`
- The receiver interrupt bit, `DZ_CSR_RIE`
- The master scan enable bit, `DZ_CSR_MSE`

11.1.4 Open and Close Section

This example shows the open and close device section for the qac device driver:

```
/***************************************************/
/*          OPEN AND CLOSE                         */
/*                                                 */
/***************************************************/

/********* Open Routine ****************************/
/*                                                 */
/* The qacopen routine checks for the validity and for    */
/* the availability of the device.  It calls the generic  */
/* tty driver open routine to set up the tty structure.   */
/* It also calls qacparam to set up the device.           */
/***************************************************/

qacopen(dev, flag)
dev_t dev; /* Major/minor device number */ 1
int flag; /* Flags from /usr/sys/h/file.h */ 2
{
        int n = minor(dev); /* Get minor device number */ 3
        struct tty *tp; /* Pointer to tty structure */ 4
        int s; /* Return value for spltty */ 5

        /* Is minor ok and is  device attached */ 6
        if ((n > NQACOPT * NQACLINE) || !qac_unit[QU(n)].attached)
                return(ENXIO);

        /* Pick tty structure */ 7
        tp = &qac_tty[n];

        /* Is the line busy? */ 8
        if ((tp->t_state & TS_XCLUDE) && (u.u_uid != 0))
                return(EBUSY);

        /* Set the t_addr member */ 9
        tp->t_addr = (caddr_t)tp;

        /* Pass start routine name */
        tp->t_oproc = qacstart; 10

        /* Set up the tty structure */ 11
        tty_def_open(tp, dev, flag, 1 << (QL(n)));

        /* Set up the line */ 12
        qacparam(n);

        if ((flag & O_NOCTTY) && (u.u_procp->p_progenv == A_POSIX)) 13
        {
                s = spltty();
                tp->t_state |= TS_ONOCTTY;
                splx(s);
        }

/* Return value of open call for line discipline */ 14
        return((*linesw[tp->t_line].l_open)(dev, tp));
}
```

1 This line declares an integer argument that holds the major and minor device numbers for the qac device. The minor device number is used to determine the logical unit number for the qac device that is to be opened or closed.

[2] This line declares an integer argument that contains flag bits from the file `/usr/sys/h/file.h`. These flag bits indicate whether the device is being opened for reading, writing, or both. The flag bits also indicate whether the terminal becomes the process's controlling terminal.

[3] This line declares an *n* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

[4] This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

[5] This line declares a variable that holds the value returned by a call to the `spltty` kernel routine and passed as an argument to the `splx` kernel routine.

[6] This line returns the error constant `ENXIO` (no such device or address) if the device minor number for this qac device is not valid or if the device is not attached. Note that this line calls the `QU` macro, which is defined in `/usr/sys/io/tc/qacreg.h`. In this case, `QU` takes the minor device number as an argument and uses it to determine the DZ line number associated with this qac device.

[7] This line sets `tp` to the address of the `tty` structure associated with this qac device. The minor device number is used as an index into the `qac_tty` array of `tty` structures to obtain the `tty` structure associated with this qac device.

[8] This line returns the error constant `EBUSY` (mount device busy) if the exclusive use flag constant (`TS_XCLUDE`) is set and the effective user id (uid) is not equal to zero. The effective uid is obtained from the `u_uid` member of the `user` structure. A uid of 0 indicates the superuser.

[9] This line sets the `t_addr` member of the `tty` structure pointed to by `tp` to the address of the `tty` structure associated with this qac device.

Because the `t_addr` member is of type `caddr_t`, this line also performs a type casting operation that converts the type of the `tty` structure pointed to by `tp` to the type `caddr_t`.

[10] This line sets the `t_oproc` member of the `tty` structure pointed to by `tp` to the qac driver's start routine, `qacstart`.

[11] This line calls the `tty_def_open` routine, which is a generic routine used to open a tty. The following arguments are passed:

- The `tty` structure pointed to by `tp`, which was set to the address of the `tty` stucture associated with this qac device in item 7.
- The device minor number for this qac device.
- The flag argument, whose value was specified on the configuration line.
- The line number associated with this qac device. The `QL` macro, defined in `/usr/sys/io/tc/qacreg.h`, uses the device minor number to calculate the line number associated with this qac device. The bit position indicates the line number.

[12] This line calls the `qacparam` routine and passes to it the minor device number associated with this qac device. See Section 11.1.10 for a discussion of the `qacparam` routine.

13 If the `O_NOCTTY` error bit is set in the *flag* argument and the programming environment mode of the `p_progenv` member of the `proc` structure is equal to the constant `A_POSIX` (an IEEE P1003.1-compliant process), the `qacopen` routine:

- Calls the `spltty` kernel routine. This routine sets the processor interrupt mask to block all device interrupts. See Appendix B for more information on the `spltty` routine.
- Sets the `TS_ONOCTTY` bit in the `t_state` member of the `tty` structure pointed to by `tp`. The `TS_ONOCTTY` bit indicates not to get the controlling tty structure on an open.
- Calls the `splx` kernel routine, passing to it the value returned by the previous call to `spltty`. The `splx` routine restores the processor interrupt mask to its previous value. See Appendix B for more information on the `splx` routine.

14 This line calls the open routine for the line discipline and returns the value. Note that the `open` routine is accessed through the `linesw` table, which is defined in `/usr/sys/h/conf.h`. The arguments passed to this routine are the device minor number for this qac device and the `tty` structure pointed to by `tp`. The routine pointed to by `linesw` is used to set generic terminal driver attributes in the associated `tty` structure. One such attribute is the assignment of a controlling terminal to the process group.

```
/********* Close Routine **********************************/
/*                                                        */
/* The qacclose routine shuts down the line.              */
/**********************************************************/

qacclose(dev, flag)
dev_t dev; /* Major/minor device number */ 1
int flag; /* Flags from /usr/sys/h/file.h */ 2
{
        /* Initialize n to the minor device number 3
        int n = minor(dev);

        /* Declare pointer to tty structure */ 4
        struct tty *tp;

        /* Initialize pointer to tty structure */ 5
        tp = &qac_tty[n];

        /* Call close routine for line discipline */ 6
        if (tp->t_line)
                (*linesw[tp->t_line].l_close)(tp);

        /* Call ttyclose for line discipline */ 7
        ttyclose(tp);
        tty_def_close(tp); /* Call ttydef_close */ 8
        qacbreakoff(n); /* Call qacbreakoff */ 9
        return(0); /* Return */ 10
}
```

1 This line declares an integer argument that holds the major and minor device numbers for the qac device. The minor device number is used to determine the logical unit number for the qac device that is to be opened or closed.

2 This line declares an integer argument that contains flag bits from the file `/usr/sys/h/file.h`. These flag bits indicate whether the device is being opened for reading, writing, or both. Note that `qacclose` does not use this argument.

3 This line declares an *n* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

4 This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

5 This line sets `tp` to the address of the `tty` structure associated with this qac device. The minor device number is used as an index into the `qac_tty` array of `tty` structures to obtain the `tty` structure associated with this qac device.

6 If a line discipline for this qac device was stored in the `t_line` member of the `tty` structure pointed to by `tp`, then call the close routine. Note that the close routine is accessed through the `linesw` table, which is defined in `/usr/sys/h/conf.h`. The argument passed to this routine is the `tty` structure pointed to by `tp`.

7 These lines call `ttyclose`, passing to it the `tty` structure pointed to by `tp`. The `ttyclose` routine is found in /sys/sys/tty.c. Before completing a close on this line, `ttyclose` waits for all pending output to drain. This routine also disassociates this terminal line as the controlling terminal for this process.

8 This line calls `tty_def_close`, passing to it the `tty` structure pointed to by `tp`. The `tty_def_close` routine is also found in /sys/sys/tty.c. This generic terminal driver routine is called to clear many of the terminal attributes that are stored in the different members of the `tty` structure associated with this device.

9 This line calls `qacbreakoff`, passing to it the device minor number for this qac device. See Section 11.1.11 for a description of the `qacbreakoff` routine.

10 Upon completion, `qacclose` returns to the `close` system call.

11.1.5 Read and Write Section

This example shows the read and write section for the qac device driver:

```
/**********************************************************/
/*         READ AND WRITE                                 */
/*                                                        */
/**********************************************************/

/********* Read Routine ***********************************/
/*                                                        */
/* The qacread routine calls the read routine for the     */
/* line discipline.                                       */
/**********************************************************/

qacread(dev, uio)
dev_t dev;       /* Major/minor device number */ [1]
struct uio *uio; /* Pointer to uio structure */ [2]
{
        struct tty *tp; /* Pointer to tty structure */ [3]

        tp = &qac_tty[minor(dev)]; /* Pick tty structure */ [4]

        /* Call read routine for line discipline */ [5]
        return((*linesw[tp->t_line].l_read)(tp, uio));
}

/********* Write Routine **********************************/
/*                                                        */
/* The qacwrite routine calls the write routine for the   */
/* line discipline.                                       */
/**********************************************************/

qacwrite(dev, uio)
dev_t dev; /* Major/minor device number */ [1]
struct uio *uio; /* Pointer to uio structure */ [2]
{
        struct tty *tp; /* Pointer to tty structure */ [3]

        tp = &qac_tty[minor(dev)]; /* Pick tty structure */ [4]

        /* Call write routine for line discipline */ [5]
        return((*linesw[tp->t_line].l_write)(tp, uio));
}
```

[1] This line declares an argument that holds the major and minor device numbers for the qac device. The minor device number will be used in determining the logical unit number for the device on which the read or write operation will be performed.

[2] Specifies a pointer to a `uio` structure. This structure contains the information for transferring data to and from the address space of the user's process. You typically pass this structure unchanged to the `uiomove` or `physio` routines. In this example, the `uio` structure gets passed to the read and write routines for the line discipline. See Section 5.1.3 for information on the `uio` structure.

[3] This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

4 This line sets `tp` to the address of the `tty` structure associated with this qac device. The minor device number is used as an index into the `qac_tty` array of `tty` structures to obtain the `tty` structure associated with this qac device.

5 This line calls the read or write routine for the line discipline. Note that in both cases, the read and write routines are accessed through the `linesw` table, which is defined in `/usr/sys/h/conf.h`. The arguments passed to these routines are the `tty` structure pointed to by `tp` and the `uio` structure pointed to by `uio`.

The `read` and `write` driver routines are called in response to a user-level `read` or `write` system call. The `read` driver routine returns the requested number of characters to the user. If there are no characters available or if another condition exists that prohibits the read request to be satisfied, the `read` driver routine returns an error.

The `l_read` routine also performs character processing based on the setting of terminal attributes in the `tty` structure associated with this device. The `l_write` routine returns an error condition if the write request cannot be performed. If the write can be performed, the driver's `qacstart` routine is called to transfer the characters from the user's data structure to the terminal driver's output queue. The driver can perform processing on the characters based on the setting of terminal attributes in the `tty` structure associated with this device.

11.1.6 ioctl Section

This example shows the ioctl section for the qac device driver:

```
/***************************************************************/
/*          ioctl                                              */
/*                                                             */
/***************************************************************/

/********* ioctl Routine **********************************/
/*                                                             */
/* The qacioctl routine implements standard tty ioctl         */
/* calls, mostly through calls to the qacparam routine.       */
/***************************************************************/

qacioctl(dev, cmd, data, flag)
dev_t dev;      /* Major/minor device number */ 1
int cmd;        /* The ioctl command */ 2
caddr_t data;   /* ioctl command-specified data */ 3
int flag;       /* Access mode of the device */ 4
{
        struct tty *tp;         /* Pointer to tty structure */ 5
        int n = minor(dev);     /* Get minor device number */ 6
        int error;              /* To hold return values */ 7
        struct devget *devget; /* Pointer to devget structure */ 8

        /* Pick tty structure */ 9
        tp = &qac_tty[n];

        /* Call to ioctl routine */ 10
        error = (*linesw[tp->t_line].l_ioctl)(tp, cmd, data, flag);

        /* Return error or call ttioctl */ 11
        if (error >= 0)
                return(error);
        error = ttioctl(tp, cmd, data, flag);

        /* Evaluate cmd and call qacparam */ 12
        if (error >= 0)
        {
                switch (cmd)
                {
                case TCSANOW:                     /* POSIX termios */
                case TCSADRAIN:                   /* POSIX termios */
                case TCSADFLUSH:                  /* POSIX termios */
                case TCSETA:                      /* SVID termio */
                case TCSETAW:                     /* SVID termio */
                case TCSETAF:                     /* SVID termio */
                case TIOCSETP:                    /* Berkeley sgttyb */
                case TIOCSETN:                    /* Berkeley sgttyb */
                case TIOCLBIS:                    /* Berkeley lmode */
                case TIOCLBIC:                    /* Berkeley lmode */
                case TIOCLSET:                    /* Berkeley lmode */
                case TIOCLGET:                    /* Berkeley lmode */
                        qacparam(n);
                        break;
                }
                return(error);
        }
        /* Evaluate cmd if erro < 0 */ 13
        switch (cmd)
        {
        /* Call qacbreakon */ 14
        case TIOCSBRK:
                qacbreakon(n);
```

```
                break;
        /* Call qacbreakoff */ 15
        case TIOCCBRK:
                qacbreakoff(n);
                break;
        /* Fill in devget structure and perform other tasks */ 16
        case DEVIOCGET:
                devget = (struct devget *)data;
                bzero(devget, sizeof(struct devget));
                devget->category = DEV_TERMINAL;       /* terminal cat.*/
                devget->bus = DEV_NB;                    /* NO bus */
                bcopy(DEV_VS_SLU, devget->interface,
                      strlen(DEV_VS_SLU));               /* interface */
                bcopy(DEV_UNKNOWN, devget->device,
                      strlen(DEV_UNKNOWN));              /* terminal */
                devget->adpt_num = qac_unit[QU(n)].adapter;
                devget->nexus_num = 0;                   /* fake nexus 0 */
                devget->bus_num = 0;                     /* NO bus */
                devget->ctlr_num = QU(n);                /* cntlr number */
                devget->slave_num = QL(n);               /* line number */
                bcopy("qac", devget->dev_name, 4);       /* Ultrix "qac" */
                devget->unit_num = QL(n);                /* dc line? */
                devget->soft_count = 0;                  /* soft err cnt */
                devget->hard_count = 0;                  /* hard err cnt */
                devget->stat = 0;                        /* status */
                devget->category_stat = DEV_MODEM;       /* cat. stat. */
               break;

        /* Check program environment and return */ 17
        default:
                if (u.u_procp->p_progenv == A_POSIX)
                        return (EINVAL);
                return (ENOTTY);
        }
        /* Return 0 */ 18
        return(0);
}
```

1 This line declares an argument that holds the major and minor device numbers for the qac device. The minor device number is used to determine the logical unit number for the qac device on which the ioctl is to be performed.

2 This line declares a variable to contain the ioctl command as specified in `/usr/sys/h/ioctl.h` or in another include file defined by the device driver writer.

3 This line declares a pointer to ioctl command-specified data that is to be passed to the device driver or filled in by the device driver. The size of this data cannot exceed 128 bytes.

4 This line declares a variable that holds the access mode of the device. The access modes are represented by flag constants defined in `/usr/sys/h/file.h`.

5 This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

6 This line declares an *n* variable and initializes it to the device minor number. Note the use of the `minor` macro to obtain the device minor number. See Appendix B for more information on the `minor` macro.

7 This line declares a variable to hold the values returned by the ioctl routine for the line discipline and the `ttioctl` routine.

8 This line declares a pointer to a `devget` structure and calls it `devget`. This structure contains such information as the general class of the device, the communications bus type, generic device status values, and so forth. This structure is defined in `/usr/sys/h/devio.h`.

9 This line sets `tp` to the address of the `tty` structure associated with this qac device. The minor device number is used as an index into the `qac_tty` array of `tty` structures to obtain the `tty` structure associated with this qac device.

10 This line calls the ioctl routine for the line discipline. The line discipline ioctl routine handles ioctl calls that are specific to the line discipline in use. Note that the ioctl routine is accessed through the `linesw` table, which is defined in `/usr/sys/h/conf.h`. The specific line discipline for this qac device is accessed through the `t_line` member of the `tty` structure pointed to by `tp`. The following are the arguments passed to the ioctl routine for the line discipline: the `tty` structure pointed to by `tp`, the *cmd* argument, the *data* argument, and the *flag* argument.

11 If the value in *error* is greater than or equal to zero (0), `qacioctl` returns this error. By returning the error condition, the ioctl system call relays the error state to the user level program. Otherwise, it calls the `ttioctl` routine, passing to it the same arguments it passed to the ioctl routine for the line discipline. The `ttioctl` routine is called to handle generic terminal driver ioctls that are not specific to the qac device.

12 If *error* is greater than or equal to 0, `ttioctl`, the generic terminal driver ioctl routine, completed without error. In this case, there are a number of specific ioctls that affect terminal attributes represented by the underlying qac hardware. For this class of ioctls, the `qacparam` routine is called to set these hardware attributes. The success status from `ttioctl` is returned to the upper level ioctl system call code to allow the system call to complete with a success status. See Section 11.1.11 for a description of `qacparam`.

13 If *error* is less than 0, the ioctl might not be one of the generic terminal driver ioctls handled by `ttioctl`. For example, the ioctl might be specific to the qac device, or the ioctl relates closely to the qac hardware. The particular ioctl that is specified in the *cmd* argument is compared against a list of qac-related ioctls. If the ioctl command in *cmd* matches one in the list, appropriate action is taken.

14 If *cmd* is the set break bit macro (`TIOCSBRK`), `qacioctl` calls the `qacbreakon` routine. See Section 11.1.11 for a description of `qacbreakon`.

15 If *cmd* is the clear break bit macro (`TIOCCBRK`), `qacioctl` calls the `qacbreakoff` routine. See Section 11.1.11 for a description of `qacbreakoff`.

16 If *cmd* is the get device information macro (`DEVIOCGET`), `qacioctl` calls the `bzero` and `bcopy` kernel routines and fills in different members of the `devget` structure pointed to by `devget`. This ioctl is called to obtain device-specific information in a generic `devget` data structure. See Appendix B for information on `DEVIOCGET`.

17 If *cmd* has none of the previous values, `qacioctl` determines if the `p_progenv` member of the `proc` structure pointed to by `u_procp` is equal to `A_POSIX` (an IEEE P1003.1-compliant process). If so, `qacioctl` returns the error constant `EINVAL` (invalid argument). Otherwise, it returns `ENOTTY` (not a typewriter).

This is done to allow the ioctl system call to return with an error status indicating that the specified ioctl is not implemented or is not relevant to the qac device.

18 A value of zero (0) is returned to allow the ioctl system call to return successful status to the user level program.

11.1.7 Interrupt Section

This example shows the interrupt section for the qac device driver:

```
/***************************************************/
/*        Interrupt                                */
/*                                                 */
/***************************************************/

/********* Interrupt Routines ***************************/
/*                                                 */
/* The qacint routine vectors control to qac_rint and    */
/* qac_int.                                        */
/***************************************************/

qacint(ctlr)
int ctlr; /* Unit number of controller */ 1
{
        struct qac_unit *qp; /* Pointer to qac_unit structure */ 2
        int csr;             /* DZ control status register */ 3

        qp = &qac_unit[ctlr]; /* Pick unit structure */ 4
        csr = qp->dz->csr;    /* Set the DZ control status register */ 5

        /* Call qac_rint */ 6
        if (csr & DZ_CSR_RDONE)
                qac_rint(qp);
        /* Call qac_tint */ 7
        if (csr & DZ_CSR_TRDY)
                qac_tint(qp, csr);
}
```

1 This line declares a variable that holds the logical unit number of the controller that is interrupting. This logical unit number was previously specified in the system configuration file. The logical unit number is used as an index into the qac driver's data structures to obtain per device information. See Section 9.1.3.4 for information on how to specify a controller's name and logical unit number in the system configuration file.

2 This line declares a pointer to a `qac_unit` structure and calls it `qp`. This structure was previously defined in the Declarations section.

3 This line declares a variable that holds a local copy of the DZ control status register.

4 This line sets `qp` to the address of the `qac_unit` structure associated with this qac device. Note that the address of the structure is obtained by referencing the logical unit number of the controller associated with this qac device.

5 This line sets the *csr* variable to the DZ control status register, which is obtained from the `csr` member of the `DZ_REGISTERS` structure pointed to by `dz`. (This pointer is a member of the `qac_unit` structure pointed to by `qp`).

6 If the receiver interrupt occurred bit (`DZ_CSR_RDONE`) is set, `qacint` calls `qac_rint` passing to it the `qac_unit` structure pointed to by `qp` for this qac device.

7 If the transmit interrupt occurred bit (`DZ_CSR_TRDY`) is set, `qacint` calls `qac_tint` passing to it:

- The `qac_unit` structure pointed to by `qp` for this qac device.
- The DZ control status register associated with this device

```
/*******************************************************/
/*                                                     */
/* The qac_rint processes incoming characters          */
/*******************************************************/
qac_rint(qp)
struct qac_unit *qp;
{
        int data;
        int line;
        struct tty *tp;
        int iflag;

        /* Device driver spins as long as charcaters available */ 1
        while ((data = qp->dz->rbuf) & DZ_RBUF_DVAL)
        {

                /* Examine relevant bits in data */ 2
                line = DZ_RBUF_RL(data);

                /* Locate the relevant tty structure */ 3
                tp = &qac_tty[(qp - qac_unit) * NQACLINE + line];

                /* Discard the character */ 4
                if ((tp->t_state & TS_ISOPEN) == 0)
                {
                        wakeup((caddr_t)&tp->t_rawq);
                        continue;
                }

                /* Set iflag to the termio flag */ 5
                iflag = tp->t_iflag;

                /* Indicate that receive silo overflowed */ 6
                if (data & DZ_RBUF_OERR)
                {
                        printf("qac%d: input silo overflow0, qp - qac_unit);
                        continue;
                }
                /* Indicate framing error occurred */ 7
                else if (data & DZ_RBUF_FERR)
                {
                        data = 0;

                        if (iflag & IGNBRK)
                                continue;
                        else if (iflag & BRKINT)
                        {
                                if (((tp->t_lflag_ext & PRAW) == 0) ||
                                    (tp->t_line == TERMIODISC))
                                {
                                        ttyflush(tp, FREAD | FWRITE);
                                        gsignal(tp->t_pgrp, SIGINT);
                                        continue;
                                }
                        }
                        else if (iflag & PARMRK)
                        {
                                (*linesw[tp->t_line].l_rint)(0377, tp);
                                (*linesw[tp->t_line].l_rint)(0, tp);
```

```
                        }
                }
                /* Indicate parity error occurred */ 8
                else if (data & DZ_RBUF_PERR)
                {

                        if (iflag & INPCK)
                        {
                                if (iflag & IGNPAR)
                                        continue;
                                else if (iflag & PARMRK)
                                {
                                        (*linesw[tp->t_line].l_rint)(0377, tp);
                                        (*linesw[tp->t_line].l_rint)(0, tp);
                                }
                                else
                                        data = 0;
                        }
                }

                /* 8-bit character isolated from data var */ 9
                data = DZ_RBUF_DATA(data);
                /* Received character stripped to 7 bits */ 10
                if (iflag & ISTRIP)
                        data &= 0177;
                else if ((data == 0377) && (tp->t_line == TERMIODISC) &&
                        (iflag & PARMRK))
                        (*linesw[tp->t_line].l_rint)(0377, tp);

                /* Pass character to line discipline */ 11
                (*linesw[tp->t_line].l_rint)(DZ_RBUF_DATA(data), tp);
        }
}
```

1 The device driver spins in this while loop as long as there are characters available. Because it takes time to process each character, it is possible that another character will become available from the qac device while processing the current character. By reading the `qp->dz->rbuf` address, the driver removes the character from the qac device and assigns it to the *data* argument.

The driver checks *data* to determine if the bit specified by `DZ_RBUF_DVAL` is set. If the bit is set, a valid character has just been read from the qac device. The driver exits from this routine when there are no longer any valid characters available.

2 Because the qac hardware supports more than one terminal line, the `qac_rint` routine examines the relevant bits in *data* to determine which line the input character is associated with.

3 Based on the terminal line number, the `qac_rint` routine locates the device's associated `tty` structure.

4 If the terminal line is not presently in use, the continue statement causes the character to be discarded.

5 The `qac_rint` routine sets *iflag* to be a register local copy of the terminal driver termio input modes flag, `t_iflag`.

6 If the `DZ_RBUF_OERR` bit is set, the receive silo has overflowed. This indicates that the receive interrupts are not being serviced fast enough to keep pace with the input rate of the qac device.

7 If the `DZ_RBUF_FERR` bit is set, a framing error occurred. This typically indicates that a break condition was detected on this line. Based on the setting of various terminal attributes, `qac_rint` performs appropriate processing to break the condition.

8 If the `DZ_RBUF_PERR` bit is set, a parity error occurred. Based on the setting of various terminal attributes, `qac_rint` performs appropriate processing to handle the parity error.

9 The 8-bit character is isolated from the *data* variable.

10 Based on the setting of various terminal attributes, `qac_rint` might strip the received character to 7 bits.

11 The `qac_rint` routines passes the character to the line discipline specific input routine. This routine typically performs any character processing specified in the terminal attributes prior to passing the character on to the user level process.

```
/*******************************************************/
/*                                                     */
/* The qac_tint processes outgoing characters          */
/*******************************************************/
qac_tint(qp, csr)
struct qac_unit *qp;
int csr;
{
        int n;
        struct tty *tp;
        struct qac_pdma *pd; /* Pointer to qac_pdma structure */ 1

        /* Set n to the appropriate index */ 2
        n = (qp - qac_unit) * NQACLINE + DZ_CSR_TL(csr);
        /* Assign pointer to relevant structure */ 3
        pd = &qac_pdma[n];

        /* Start break condition */ 4
        if (pd->p_mem != pd->p_end)
        {
                qp->dz->tdr = qp->brk | (unsigned char)(*pd->p_mem++);
        }
        /* Previously initiated transmissions completed */ 5
        else
        {
                tp = &qac_tty[n];
                tp->t_state &= ~TS_BUSY;
                if (tp->t_state & TS_FLUSH)
                        tp->t_state &= ~TS_FLUSH;
                else
                {
                        /* Remove properly transmitted characters */ 6
                        ndflush(&tp->t_outq, pd->p_mem-tp->t_outq.c_cf);
                       pd->p_end = pd->p_mem = tp->t_outq.c_cf;
                }
                /* Call line discipline specific start routine */ 7
                if (tp->t_line)
                        (*linesw[tp->t_line].l_start)(tp);
                else
                        /* Call qacstart to commence next transmission */ 8
                        qacstart(tp);

                /* Disable transmitter interrupts */ 9
                if ((tp->t_state & TS_BUSY) == 0)
                        qp->dz->tcr &= ~DZ_TCR_ENA(QL(n));
        }
}
```

1. Each terminal line on a qac device has an associated `qac_pdma` structure that is used to store characters waiting to be transmitted.
2. This line sets the variable *n* to the appropriate index for this qac line within its associated `qac_pdma` structure.
3. Based on the index assigned to *n*, this line assigns a pointer to the relevant `qac_pdma` structure.
4. If there are characters waiting to be output, start a break condition on this line by setting the appropriate bit in the `tdr` register.
5. There are no more characters waiting to be transmitted. This indicates that all previously initiated transmissions have now completed.
6. This line removes the characters that were properly transmitted on the qac device from the terminal driver's output queue.
7. If a line discipline-specific start routine is available, call it to commence the next transmission.
8. Call the `qacstart` routine to begin the next transmission, if there are additional characters waiting to be output.
9. If the device is not currently busy transmitting, disable transmitter interrupts because there is no present need to be notified of transmitter completion.

11.1.8 Start Section

This example shows the start section for the qac device driver:

```
/**********************************************************/
/*         Start                                          */
/*                                                        */
/**********************************************************/

/********* Start Routine **********************************/
/*                                                        */
/* The qacstart routine starts output on a terminal.      */
/*                                                        */
/**********************************************************/

qacstart(tp)
struct tty *tp; /* Pointer to tty structure */ 1
{
        int n;               /* Holds minor device number */
        int s;               /* Stores return value for spltty */
        int cc;              /* Return value for ndqb */
        struct qac_pdma *pd; /* Pointer to qac_pdma structure */ 2

        n = minor(tp->t_dev); /* Get minor device number */ 3

        s = spltty(); /* Block all device interrupts */ 4

        /* If bits set, continue execution at out */ 5
        if (tp->t_state & (TS_TIMEOUT | TS_BUSY | TS_TTSTOP))
                goto out;

        /* Otherwise, check linked list queue */ 6
        if (tp->t_outq.c_cc <= TTLOWAT(tp))
```

```
        {
                /* Is TS_ASLEEP bit set? */
                /* If so, flip the bits  */
                /* and call wakeup.      */ [7]
                if (tp->t_state & TS_ASLEEP)
                {
                        tp->t_state &= ~TS_ASLEEP;
                        wakeup((caddr_t)&tp->t_outq);
                }
                /* Otherwise, check proc structure       */ [8]
                /* If condition is true, call selwakeup, */
                /* set the proc structure to 0, and      */
                /* flip the bits in t_state              */
                if (tp->t_wsel)
                {
                        selwakeup(tp->t_wsel, tp->t_state & TS_WCOLL);
                        tp->t_wsel = 0;
                        tp->t_state &= ~TS_WCOLL;
                }
        }

        /* Otherwise, check that linked list queue equals 0 */ [9]
        if (tp->t_outq.c_cc == 0)
                goto out;

        /* Determine number of characters awaiting output */ [10]
        if ((tp->t_lflag_ext & PRAW) || (tp->t_oflag_ext & PLITOUT) ||
            ((tp->t_oflag & OPOST) == 0))
                cc = ndqb(&tp->t_outq, 0);
        else
        {
                cc = ndqb(&tp->t_outq, DELAY_FLAG);
                if (cc == 0) {
                        cc = getc(&tp->t_outq);
                        timeout(ttrstrt, (caddr_t)tp, (cc&0x7f) + 6);
                        tp->t_state |= TS_TIMEOUT;
                        goto out;
                }
        }
        /* Initiate actual character transmission */ [11]
        tp->t_state |= TS_BUSY;
        pd = &qac_pdma[n];
        pd->p_end = pd->p_mem = tp->t_outq.c_cf;
        pd->p_end += cc;
        /* Enable transmit interrupts */ [12]
        qac_unit[QU(n)].dz->tcr |= DZ_TCR_ENA(QL(n));

out:
        splx(s); /* Restore spl level */ [13]
}
```

[1] This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

[2] This line declares a pointer to a `qac_pdma` structure and calls it `pd`. This structure was previously defined in the Declarations section.

[3] This line initializes *n* to the device minor number associated with this qac device. The device minor number is obtained from the `t_dev` member of the `tty` structure pointed to by `tp` for this qac device.

4 This line calls the `spltty` kernel routine, which blocks all device interrupts. After it completes execution, `spltty` returns the current spl level (that is, the spl level prior to its being called). See Appendix B for more information on the `spltty` routine.

5 If the `t_state` member of the `tty` structure pointed to by `tp` for this qac device is set to the `TS_TIMEOUT` or `TS_BUSY` or `TS_TTSTOP` bit, execution continues at label out. The bit settings, defined in `/usr/sys/h/tty.h`, have the following meanings:

Bit Value	**Meaning**
`TS_TIMEOUT`	Delay execution; timeout in progress
`TS_BUSY`	A previous transmission is currently in progress
`TS_TTSTOP`	Output stopped when user pressed ^S

6 If the bits in the previous line were not set, check the linked list queue of characters to determine if it is less than or equal to the value returned by the `TTLOWAT` macro. This macro indicates if there is presently room in the output queue to accept additional characters. This linked list queue of characters is defined by the `clist` structure defined in `/usr/sys/h/tty.h`.

7 If the previous line is true, this line checks if the `TS_ASLEEP` bit is set in the `t_state` member of the `tty` structure pointed to by `tp` for this qac device. If `TS_ASLEEP` is set, this terminal line was previously blocked on output because there were already too many characters in the output queue. Now that the number of characters in the output queue is below an acceptable level the `TS_ASLEEP` flag can be cleared. This bit is defined in `/usr/sys/h/tty.h`.

After clearing the bit, this line calls the `wakeup` kernel routine. This routine takes one argument: the address on which the wakeup is to be issued. In this case, this address is that of the linked list queue of characters. This line also performs a type casting operation because the data type expected by `wakeup` is of type `caddr_t` and the data type of the linked list queue is of type `clist`. See Appendix B for more information on the `wakeup` routine.

8 If the `TS_ASLEEP` bit in the previous condition statement was not set, check the `proc` structure pointed to by `t_wsel` to determine if a select system call was previously issued on this line. If the condition is true:

- Call the `selwakeup` kernel routine

  This routine wakes up a select blocked process and takes two arguments. The first argument is a pointer to a `proc` structure. In this case the pointer is accessed through the `t_wsel` member of the `tty` structure pointed to by `tp` for this qac device. The second argument is a value that indicates whether more than one process is blocked on this file descriptor. In this case, the value is obtained from setting the `t_state` member to the `TS_WCOLL` bit. This bit, defined in `/usr/sys/h/tty.h`, indicates a collision on a write select, indicating that there is more than one process that issued a select system call on this line.

- Clear the `t_wsel` member and the `TS_WCOLL` bit in the `t_state` to indicate that the pending select was serviced.

9 If `cc` is zero (0), it indicates that there are no more characters waiting to be output. Thus, a jump to the label out is done to exit this routine.

10 In a manner determined by the terminal attributes, the number of characters awaiting to be output is determined.

11 The actual character transmission is initiated by setting the appropriate pointers, state fields, and character counts.

12 Transmit interrupts are enabled for this line so that the driver knows when transmission completes.

13 This line calls the `splx` kernel routine. This routine takes as an argument the interrupt mask returned in a previous call to one of the `spl` routines, in this case `spltty`. The `splx` routine restores the processor to the interrupt mask specified in the argument.

11.1.9 Stop Section

This example shows the stop section for the qac device driver:

```
/**********************************************************/
/*         Stop                                           */
/*                                                        */
/**********************************************************/

/********* Stop Routine ***********************************/
/*                                                        */
/* The qacstop routine suspends transmission on a        */
/* specified line.                                        */
/*                                                        */
/**********************************************************/

qacstop(tp, flag)
struct tty *tp; /* Pointer to tty structure */ [1]
int flag ;      /* Output flag */ [2]
{
        int n; /* Holds device minor number */
        int s; /* Return from spltty */
        struct qac_pdma *pd; /* Pointer to qac_pdma structure */ [3]

        n = tp - qac_tty; /* Get device minor number */ [4]
        pd = &qac_pdma[n]; /* Pick qac_pdma structure */ [5]
        s = spltty(); /* Set processor interrupt mask */ [6]
        if (tp->t_state & TS_BUSY) /* If TS_BUSY bit set */ [7]
        {
                pd->p_end = pd->p_mem; /* Set p_end member */
                if ((tp->t_state & TS_TTSTOP) == 0) /* If TS_TSTTSTOP */
                                                    /* bit set */
                        tp->t_state |= TS_FLUSH;    /* Set TS_FLUSH bit */
        }
        splx(s); /* Restore interrupt mask */ [8]
}
```

[1] This line declares a pointer to a `tty` structure and calls it `tp`. This structure contains information such as state information about the hardware terminal line, input and output queues, the line discipline number, and so forth. This structure is defined in `/usr/sys/h/tty.h`.

[2] This line declares a variable that specifies whether the output is to be flushed or suspended. ULTRIX device drivers do not use this argument.

[3] This line declares a pointer to a `qac_pdma` structure and calls it `pd`. This structure was previously defined in the Declarations section.

[4] The *n* variable is assigned to the index of this line within the `qac_pdma` structure.

[5] This line sets `pd` to the address of the `qac_pdma` structure associated with this qac device. The minor device number is used as an index into the array of `qac_pdma` structures to obtain the `qac_pdma` structure associated with this qac device.

[6] This line calls the `spltty` kernel routine to block all device interrupts. This routine returns the current spl level.

[7] This line performs a conditional test based on the `TS_BUSY` bit, which indicates that output of characters is in progress. If the bit is set in the `t_state` member of the `tty` structure pointed to by `tp`, `qacstop`:

- Sets the pseudo DMA transmitter tail (the `p_end` member of the `qac_pdma` structure pointed to by `pd`) to the pseudo DMA transmitter head (the `p_mem` member of the `qac_pdma` structure pointed to by `pd`).
- Performs a conditional test on setting the `TS_TTSTOP` bit, which indicates that output of characters was stopped by the user pressing ^S. If this bit is set in the `t_state` member of the `tty` structure pointed to by `tp`, then set `t_state` to the `TS_FLUSH` bit. This bit indicates that the output queue has been flushed during DMA.

8 This line calls the `splx` kernel routine passing to it the value returned in a previous call to `spltty`. The `splx` routine returns the processor interrupt mask to the previous spl level.

11.1.10 Parameter Section

This example shows the parameter section for the qac device driver:

```
/**********************************************************/
/*         Parameter                                      */
/*                                                        */
/**********************************************************/

/********* Parameter Routine ******************************/
/*                                                        */
/* The qacparam routine sets the hardware attribute for   */
/* this line to the values specified in the terminal      */
/* attributes from the associated tty structure.          */
/*                                                        */
/**********************************************************/

qacparam(n)
int n;
{
        struct tty *tp;
        int param;

        tp = &qac_tty[n];

        /* Set baud rate */ [1]
        param = qac_speeds[tp->t_cflag & CBAUD] | DZ_LPR_LINE(QL(n));

        if ((tp->t_line != TERMIODISC) && ((tp->t_cflag_ext & CBAUD) == B110))
        /* Set number of stop bits */
                tp->t_cflag |= CSTOPB;

        if (tp->t_cflag & CREAD)
                /* Enable receive interrupts */
                param |= DZ_LPR_RXENA;

        if (tp->t_cflag & CSTOPB)
        /* Set number of stop bits */
                param |= DZ_LPR_STOP;

        if (tp->t_cflag & PARENB)
                /* Enable parity detection */ [2]
                param |= DZ_LPR_PARENA;

        if (tp->t_cflag & PARODD)
                /* Set parity to odd or even */
                param |= DZ_LPR_ODDPAR;

        switch (tp->t_cflag & CSIZE)
        {
        /* Set number of data bits to 5, 6, 7, or 8 */
        case CS5: param |= DZ_LPR_CHAR_5; break;
        case CS6: param |= DZ_LPR_CHAR_6; break;
        case CS7: param |= DZ_LPR_CHAR_7; break;
        case CS8: param |= DZ_LPR_CHAR_8; break;
        }

        /* Write specified line parameters */ [3]
        qac_unit[QU(n)].dz->lpr = param;
}
```

[1] This line sets the the baud rate of this terminal line in accordance with the values of the terminal attributes.

[2] This line enables parity detection on input and generation on output.

[3] This line writes the specified line parameters from a local copy to the actual hardware register.

11.1.11 Break On and Break Off Section

This example shows the break on and break off section for the qac device driver:

```
/**********************************************************/
/*         Break On and Break Off                         */
/*                                                        */
/**********************************************************/

/********* Break On Routine *******************************/
/*                                                        */
/* The qacbreakon routine turns on line breaks.           */
/* This routine is called by qacioctl.                    */
/**********************************************************/

qacbreakon(n)
int n; /* Major/minor device number */ [1]
{
        struct qac_unit *qp; /* Pointer to qac_unit structure */ [2]

        qp = &qac_unit[QU(n)]; /* Pick qac_unit structure */ [3]

        /* Set line breaks for flag */ [4]
        qp->brk |= DZ_TDR_BRK(QL(n));
        /* tdr<15:8> byte write */ [5]
        *(char*)(((int)&qp->dz->tdr) + 5) = qp->brk >> 8;
}

/********* Break Off Routine ******************************/
/*                                                        */
/* The qacbreakoff routine turns off line breaks.         */
/* This routine is called by qacioctl.                    */
/**********************************************************/

qacbreakoff(n)
int n; /* Major/minor device number */ [1]
{
        struct qac_unit *qp; /* Pointer to qac_unit structure */ [2]

        qp = &qac_unit[QU(n)]; /* Pick qac_unit structure */ [3]

        /* Set line breaks for flag */ [4]
        qp->brk &= ~DZ_TDR_BRK(QL(n));
        /* tdr<15:8> byte write */ [5]
        *(char*)(((int)&qp->dz->tdr) + 5) = qp->brk >> 8;
}
```

[1] This line declares an integer variable that holds the major/minor device number for this qac device. The `qacioctl` routine passes the minor device number for this qac device to `qacbreakon` and `qacbreakoff`. See Section 11.1.6 for a description of `qacioctl`.

[2] This line declares a pointer to a `qac_unit` structure and calls it `qp`. This structure was previously defined in the Declarations section.

[3] This line sets `qp` to the address of the `qac_unit` structure associated with this qac device. Note that the `QU` macro is used to calculate the DZ unit number for this qac device. The minor device number is passed to this macro. The `QU` macro is defined in `/usr/sys/io/tc/qacreg.h`.

[4] Both routines set the line breaks flag as follows:

- The `qacbreakon` routine calls the `DZ_TDR_BRK` and `QU` macros to calculate the bits to OR in the `brk` member of the `qac_unit` structure pointed to by `qp`.
- The `qacbreakoff` routine calls the same macros to calculate the bits to AND in the `brk` member of the `qac_unit` structure pointed to by `qp`. Note, however, that `qacbreakoff` uses the ones complement operator to flip the bits calculated by the macros.

5 Each line of the qac device has an associated break bit in the `tdr` register. This statement sets or clears the individual break bit corresponding to this line while leaving the break bits of the other line unchanged.

11.2 Memory-Mapped Device Driver

The memory-mapped device driver example illustrates a driver that provides a memory map mechanism for a generic memory-mapped device. For convenience in reading and studying the memory-mapped device driver, the source code is divided into parts. Table 11-2 lists the parts of the memory-mapped device driver and the sections of the chapter where each is discussed.

Table 11-2: Parts of the Memory-Mapped Device Driver

Part	Section
Include Files	Section 11.2.1
Declarations	Section 11.2.2
Autoconfiguration	Section 11.2.3
Open and Close	Section 11.2.4
Memory Mapping	Section 11.2.5

11.2.1 Include Files Section

This example shows the include files section for the memory-mapped device driver:

```
/* sk.c - Memory mapped device driver                   */
/*                                                      */
/* Abstract:                                            */
/*                                                      */
/* This driver provides a memory map mechanism for a    */
/* generic memory mapped device.                        */
/*                                                      */
/* Author: Digital Equipment Corporation                */
/*                                                      */
/********************************************************/
/*        INCLUDE FILES                                 */
/*                                                      */
/********************************************************/

/* Header files required by memory mapped device driver */

#include "sk.h" /* Driver header file generated by config */ 1
#include "../h/types.h"
#include "../h/errno.h"
#include "../machine/param.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h" /* Include for BADADDR */ 2
#include "../io/uba/ubavar.h"
#include "../h/ioctl.h"
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/vmmac.h"
#include "../io/tc/tc.h" /* TURBOchannel definitions */ 3
```

1. This line includes the `sk.h` file, which is the device driver header file created by `config`. This file is also included in `/usr/sys/machine/common/conf.c`, which is where you define the entry points for most device driver routines. The `sk.h` file contains #define statements for the number of sk devices configured into the system. See Section 9.2 for more information on the `conf.c` file.

2. The `cpuconf.h` file is where the `BADADDR` macro is defined. ULTRIX device drivers use this macro to determine whether a device is present on the hardware configuration. See the `skprobe` routine in Section 10.1.3 for an example of how the memory-mapped device driver uses `BADADDR`.

3. The `/usr/sys/io/tc/tc.h` header file is specific to TURBOchannel device drivers. For summary descriptions of other header files used by device drivers, see Appendix A.

11.2.2 Declarations Section

This example shows the declarations section for the memory-mapped device driver:

```
/**********************************************************/
/*         DECLARATIONS                                   */
/*                                                        */
/**********************************************************/

#define SKREGSIZE 256            /* First csr area size */ 1
#define SKUNIT(dev) (minor(dev)) /* Device minor number */ 2

/* Driver routines declarations */
int skprobe(),skattach(),skintr(),skmmap();

/* Array of pointers to uba_device structures */ 3
struct uba_device *skdinfo[NSK];

/* Declare and initialize uba_driver structure */ 4
struct uba_driver skdriver = {
        skprobe,0,skattach,0,0,
        "sk",skdinfo,0,0,0
};

/* Device register structure */ 5
struct sk_reg_t {
        volatile char stub_0;    /* Base address */
        volatile char T;         /* First readable, always T */
        volatile char stub_1;    /* Data is only on every other byte */
        volatile char C;         /* Second readable */
        volatile char nonused[124];
        volatile short status;
        volatile unsigned short intvec;
        volatile unsigned short reset;
        volatile char unused[2];
        volatile unsigned short start;
        volatile char nevused[2];
        volatile unsigned short skdata;
        volatile char pads[92];  /* Fills out the remainder of */
                                 /* the 256 byte block */
};

/* Define a softc structure for use by the interrupt service */
/* routines, the error log routines, etc. */ 6
struct sk_softc{
        int sk_time;       /* Timeout value*/
        int sk_expint;     /* Expecting interrupt*/
        int sk_timeout;    /* Timeout situation : true or false */
        int sk_data;       /* Value read after interrupt*/
        int intcnt;        /* Number of times interrupts may happen */
        struct sk_reg_t *sk_base; /* Pointer to sk_reg_t structure */
} sksoftc[NSK];

/* Define debug constants */ 7
#define SK_DEBUG
#ifdef SK_DEBUG
int sk_debug = 0;
#endif SK_DEBUG
```

1 This line defines a constant that can be used for the size of the first CSR area. The memory-mapped device driver initializes the `ud_addr1_size` member of the `uba_driver` structure with this constant.

2 This line defines a constant that represents the device minor number. A call to the `minor` macro obtains the device minor number. This macro takes one argument: the number of the device whose associated minor device number you want to obtain. See Appendix B for a description of the `minor` macro.

3 This line declares an array of pointers to `uba_device` structures and calls it `skdinfo`. This array is referenced by the driver's `skattach` and `skmmap` routines. The constant `NSK` represents the maximum number of sk devices for a particular hardware configuration. This number is used to size the array of pointers to `uba_device` structures. This constant was defined by `config` in `sk.h`.

4 The `uba_driver` structure called `skdriver` is initialized to the following:

- The driver's probe routine, `skprobe`.
- The value 0, to indicate that this driver does not use a slave routine.
- The driver's attach routine, `skattach`.
- The value 0, to indicate that this driver does not use a go routine.
- The value 0, because VMEbus device drivers do not use the `ud_addr` member of the `uba_driver` structure.
- The value `sk`, which is the name of the device.
- The value `skdinfo`, which references the array of pointers to the previously declared `uba_device` structures. You index this array with the unit number as specified in the `ui_unit` member of the `uba_device` structure.
- The value 0, to indicate that there is no controller name associated with this device.
- The value 0, to indicate that this driver does not use the `uba_ctlr` structure.
- The value 0, to indicate that this driver does not want exclusive use of the buffer data paths (bdps).

5 This line defines a structure called `sk_reg_t` whose members map to the characteristics of the sk device. This structure is referenced in the autoconfiguration section of the memory-mapped driver, specifically by the `skprobe`, `skintr`, and `skmmap` routines. The members of this structure are declared using the key word `volatile` because some of its members correspond to hardware device registers for the sk device. In addition, the values stored in these members could be changed by something other than the device driver. See Section 4.2 for information on when to declare a variable or data structure as volatile.

6 This line declares an array of `softc` structures and calls it `sksoftc`. The size of the array is the value represented by the `NSK` constant. The memory-mapped device driver's `sk_softc` structure allows the interrupt service routines and the error logging routines to share data. Driver routines in the autoconfiguration and memory-mapping sections reference this structure.

7 These lines use several of the C preprocessor statements to set up conditional compilation for debugging purposes. In the sk driver, these statements are used with the `cprintf` kernel routine to print intermediate results to the console terminal.

11.2.3 Autoconfiguration Section

This example shows the autoconfiguration section for the memory-mapped device driver:

```
/***************************************************/
/*        AUTOCONFIGURATION                        */
/*                                                 */
/***************************************************/

/********* Probe Routine *********************************/
/*                                                 */
/* The skprobe routine calls the BADADDR macro to        */
/* determine that there is indeed a board at the         */
/* specified address.  If the board is present,          */
/* skprobe returns the size of the register space that   */
/* the board occupies.  If the device is not present,    */
/* skprobe returns 0.                                    */
/*                                                 */
/***************************************************/

skprobe(addr1,unit)
caddr_t addr1; /* System Virtual Address for the sk device */ 1
int unit; /* Unit number associated with the sk device */ 2
{

        /* Pointer to device register structure */ 3
        register struct sk_reg_t *sk_reg;

        /* Pointer to sk_softc structure */ 4
        register struct sk_softc *sksc;

/* Kernel was properly configured */ 5
#ifdef SK_DEBUG
        if (sk_debug) cprintf("SK probe routine entered\n");
#endif SK_DEBUG

        /* Point to device registers */ 6
        sk_reg = (struct sk_reg_t *)addr1;

        /* Call the BADADDR macro to determine if */
        /* the device is present */ 7
        if (BADADDR ((char *) &sk_reg->T,sizeof(char)) !=0)
        {
            return (0);
         }

        /* Check the first location */ 8
        if (sk_reg->T != 'T') return(0);

        /* Call the BADADDR macro a second time to determine */
        /* if the device is present */ 9
        if (BADADDR ((char *) &sk_reg->C,sizeof(char)) !=0)
        {
             return (0);
        }

        /* Check the second location */ 10
        if (sk_reg->C != 'C') return(0);

        /* Set the pointer to the address of the sk_softc */
        /* structure array */ 11
        sksc = &sksoftc[unit];

        /* Store the base address */ 12
```

```
        sksc->sk_base = (struct sk_reg_t *) addr1;

/* Device found */ 13
#ifdef SK_DEBUG
        if (sk_debug) cprintf("SK driver found\n");
#endif SK_DEBUG

        /* Return size of register space */ 14
        return (SKREGSIZE);
}

/*********** Attach Routine *******************************/
/*                                                        */
/* The skattach routine initializes the device and its    */
/* software state.                                        */
/**********************************************************/

skattach(ui)
struct uba_device *ui; /* Pointer to uba_device structure */ 15
{
/* Attach routine code goes here */
}

/********** Interrupt Routine *****************************/
/*                                                        */
/**********************************************************/

skintr(unit)
int unit; /* Logical unit number of device */ 16
{

        /* Pointer to device register structure */ 17
        register struct sk_reg_t *sk_reg;

        /* Pointer to sk_softc structure */ 18
        register struct sk_softc *sksc;

        /* Set the device's softc structure */ 19
        sksc = &sksoftc[unit];

         /* Store the base address */ 20
        sk_reg = sksc->sk_base;

         /* Check some status word and then set it */ 21
        if (sk_reg->status < 0)
        {
              sk_reg->status = 5;

              /* Read in some data */ 22
              sksc->sk_data = sk_reg->skdata;
        }
}
```

1 This line declares an *addr1* argument that is the System Virtual Address (SVA) that corresponds to the base slot address.

2 This line declares a *unit* variable that is used to specify the sk device.

3 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skprobe` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

4 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skprobe` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

5 This section is executed during debugging of the sk driver. The line calls the `cprintf` kernel routine to print the message ''SK probe routine entered'' on the terminal to indicate that the kernel was properly configured. For more information on this routine, see Appendix B.

6 This line initializes the `sk_reg` pointer to the SVA for the memory-mapped device, which is contained in the *addr1* argument. Because the data types are different, this line performs a type casting operation that converts the *addr1* argument (which is of type `caddr_t`) to be of type pointer to an `sk_reg_t` structure.

7 This line calls the `BADADDR` macro to determine if the device is present. The `BADADDR` macro takes two arguments: the address of the device whose existence you want to check and the length of the data to be checked. In this call to the macro, the address of the `T` member of the `sk_reg` pointer is passed. The length is the value returned by the `sizeof` operator, in this case the number of bytes needed to contain a value of type `char` (because the `T` member is a size `char`).

Because the first argument to the `BADADDR` macro is of type `caddr_t`, this line also performs a type casting operation that converts the type of the `T` member (which is of type `char`) to type `char *`. (The data type `caddr_t` is actually a typedef to the data type `char *`.)

If a device is present, `BADADDR` returns the value 0.

8 If a device is present, this line checks the first location. That is, if the `T` member of the `sk_reg` pointer is not equal to the character V, it is not a supported device. Therefore, the `skprobe` routine returns 0.

Some TURBOchannel devices have proms with an ID that usually starts with the letters TC. Thus, this line reads the prom looking for the specific value `T`. Your driver code may need to find a more unique string.

9 This line is identical to the one that previously called `BADADDR`, except this time the `C` member of the `sk_reg` pointer is passed. If a device is present, `BADADDR` returns the value 0.

10 If a device is present, this line checks the second location. That is, if the `C` member of the `sk_reg` pointer is not equal to the character `C`, it is not a supported device. Therefore, the `skprobe` routine returns 0.

11 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device.

12 This line sets the *sk_base* member of the `sksc` pointer to the base address where the device was found, which is contained in the *addr1* argument. Note that the *sk_base* member is a pointer to `sk_reg_t`, the sk device register structure. Therefore this line performs a type casting operation that converts the *addr1* argument (which is of type `caddr_t`) to be of type pointer to an `sk_reg_t` structure.

13 This line is executed during debugging of the sk driver. The line calls the `cprintf` routine to print the message ''SK driver found'' on the terminal to indicate that the `skprobe` routine was successful in finding a device. For

more information on this routine, see Appendix B.

14 This line returns the size of the register space, which indicates that the sk board is present.

15 The sk device does not need an attach routine. However, this line shows that your attach routine would declare a pointer to a `uba_device` structure. The driver can send any information contained in this structure to the device.

16 This line declares a *unit* argument that is used to specify the logical unit number of the memory-mapped device that is interrupting.

17 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skintr` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

18 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skintr` routine makes several references to members of this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

19 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device. Note that `sksoftc` is the array of structures declared in the Declarations section and that *unit* is the index into this array.

20 This line sets the `sk_reg` pointer to the base address, which is the `sk_base` member of the `sksc` pointer.

21 If the `status` member of the `sk_reg` pointer is less than 0, then this line sets it to the value 5.

22 This line sets the `sk_data` member of the `sksc` pointer to the data contained in the `skdata` member of the `sk_reg` pointer.

11.2.4 Open and Close Section

This example shows the open and close section for the memory-mapped device driver:

```
/***********************************************************/
/*         OPEN AND CLOSE                                  */
/*                                                         */
/***********************************************************/

/********* Open Routine ************************************/
/*                                                         */
/***********************************************************/

skopen(dev,flag)
dev_t dev; /* Major/minor device number */ [1]
int flag; /* Flags from /usr/sys/h/file.h */ [2]
{
      /* Return to the open system call */ [3]
      return (0);
}

/********* Close Routine ***********************************/
/*                                                         */
/***********************************************************/

skclose(dev,flag)
dev_t dev; /* Major/minor device number */ [1]
int flag; /* Flags from /usr/sys/h/file.h */ [2]
{
      /* Return to the close system call */ [3]
      return (0);
}
```

[1] This line declares an integer variable that holds the major and minor device numbers for the memory-mapped device. The minor device number will be used in determining the logical unit number for the memory-mapped device that is to be opened or closed.

[2] This line declares an integer variable to contain flag bits from the file `/usr/sys/h/file.h`. These flags indicate whether the device is being opened for reading, writing, or both.

[3] The `skopen` routine does not do any intricate work other than to return execution to the `open` system call. Likewise, the `skclose` routine simply returns execution to the `close` system call.

11.2.5 Memory-Mapping Section

This example shows the memory-mapping section for the memory-mapped device driver:

```
/***********************************************************/
/*         MEMORY MAPPING                                  */
/*                                                         */
/***********************************************************/

/********* Memory Mapping Routine **************************/
/*                                                         */
/* The skmmap routine is invoked by the kernel as a        */
/* result of an application calling the mmap(2) system    */
/* call.  The skmmap routine makes sure that the           */
/* specified offset into the memory mapped device's        */
/* memory is valid.  If the offset is not valid, skmmap   */
/* returns -1.  If the offset is valid, skmmap returns    */
/* the page frame number corresponding to the page at     */
/* the specified offset.                                   */
/*                                                         */
/***********************************************************/

skmmap(dev, off, prot)
dev_t dev; /* Device whose memory is to be mapped */ [1]
off_t off; /* Byte offset into device memory */ [2]
int prot;  /* Protection flag: PROT_READ or PROT_WRITE */ [3]
{
      /* Pointer to device register structure */ [4]
      register struct sk_reg_t *sk_reg;

      /* Pointer to sk_softc structure */ [5]
      register struct sk_softc *sksc;

      /* Page frame number */ [6]
      int kpfnum;

      /* Make sure that the offset into the device registers */
      /* is less than the size of the device register space. */ [7]
      if ((u_int) off >= SKREGSIZE)
        return (-1);

      /* Otherwise, set the device's sk_softc structure */ [8]
      sksc = &sksoftc [SKUNIT(dev)];

      /* and store the base address */ [9]
      sk_reg = sksc->sk_base;

      /* Find the register space of the device */ [10]
      kpfnum = vtokpfnum(sk_reg+off);
      return kpfnum;
}
```

[1] This line declares a *dev* argument that specifies the character device whose memory is to be mapped.

[2] This line declares an *off* argument that specifies the offset in bytes into the character device's memory. The offset must be a valid offset into the device memory.

[3] This line declares a *prot* argument that specifies the protection flag for the mapping. The protection flag is the bitwise inclusive OR of these valid protection flag bits defined in `/usr/sys/h/mman.h`: `PROT_READ` or

`PROT_WRITE`.

4 This line declares a pointer to the `sk_reg_t` structure and calls it `sk_reg`. The `skmmap` routine makes reference to this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

5 This line declares a pointer to the `sk_softc` structure and calls it `sksc`. The `skmmap` routine makes reference to this structure. This structure and its associated members were previously defined in the Declarations section of the memory-mapped driver.

6 This line declares a *kpfnum* variable to contain the page frame number returned by the `vtokpfnum` kernel routine.

7 If the offset into the memory-mapped device's memory is greater than or equal to the size of the first CSR area, the `skmmap` routine returns –1. This value indicates an unsuccessful attempt at mapping this device's memory into the user's address space. This line also performs a type casting operation that converts the *off* argument (which is of type `off_t`) to be of type `u_int`. The reason is to ensure that you compare an unsigned quantity because the offset may be a full longword.

8 This line sets the `sksc` pointer to the address of the `sk_softc` structure associated with this sk device. Note the use of the `SKUNIT` macro to obtain the minor number associated with this sk device.

9 This line sets the `sk_reg` pointer to the base address, which is the `sk_base` member of the `sksc` pointer.

10 This line calls the `vtokpfnum` kernel routine. This routine takes one argument: the kernel virtual address whose page frame number is to be returned. In this example, this address is the result of the expression whose operands consist of the pointer to the `sk_reg_t` structure and the byte offset. Upon completing execution successfully, `vtokpfnum` sets the *kpfnum* variable to the page frame number associated with the page in the sk device's memory. See Appendix B for a description of the `vtokpfnum` kernel routine.

Part VI: Porting Issues

Porting VMEbus Device Drivers 12

The VMEbus is an industry standard bus that operates in a variety of hardware platforms. This chapter discusses the tasks you need to perform when porting VMEbus device drivers from other hardware platforms to Digital hardware, focusing on drivers written for the Sun Microsystems platform, which is a BSD UNIX derivative. Porting from hardware platforms that use a derivative of System V may be more difficult.

This chapter makes reference to the way the operating system for Sun Microsystems performs certain tasks. These references are based on an understanding of Sun Microsystems UNIX Versions 3.1 to 4.0.3. Because Digital is not in a position to fully understand the mechanisms or future plans for Sun Microsystems hardware and software, there can be no guarantee as to the accuracy of these references.

Table 12-1 lists the tasks associated with porting VMEbus device drivers and the section where each is discussed.

Table 12-1: Tasks Associated with Porting VMEbus Device Drivers

Task	Section
Writing test suites	Section 12.1
Checking header files	Section 12.2
Reviewing device driver installation	Section 12.3
Checking driver routines	Section 12.4
Checking data structures	Section 12.5
Comparing DMA mechanisms	Section 12.6
Testing for device access	Section 12.7
Checking the design of a device driver	Section 12.8
Setting interrupt priority levels	Section 12.9
Performing byte swapping operations	Section 12.10
Comparing memory mapping	Section 12.11

12.1 Writing Test Suites

Porting a device driver from one company's hardware platform to the Digital platform requires that you understand the hardware device and the associated driver you want to port. One way to learn about the hardware device and its associated driver is to run a test suite, if it exists, on the machine you are porting from (the source machine). If the test suite does not exist, you need to write a full test suite for that device on the source machine. For example, if you port a device driver from a Sun Microsystems machine, write the full test suite on the Sun Microsystems machine.

To successfully port drivers from one company's hardware platform to Digital hardware requires you to write tests for all the tasks performed by the driver. Write the test suite so that only minimal changes are necessary when you move it to the system you are porting to (the target machine). The test suite represents a cross-section of your users, and they should not have to modify their applications to work with the ported driver. You need to have both the source machine and the target machine on a network or make them accessible through a common interface, such as SCSI.

After writing the test suite on the source machine, move the driver and the test suite to the target machine. Move only the `.c` and the `.h` files that were created for the driver. Do not copy any system files, because the system files on the source machine will probably not be compatible on the target machine.

12.2 Checking Header Files

Although you may be porting a driver from a hardware platform that is a BSD UNIX derivative, do not assume that all of the header files are the same as those used in ULTRIX. Check the header files contained in the driver you want to port with those used in ULTRIX device drivers. See Section 4.1.1 for the minimal header files needed by VMEbus drivers. See Appendix A for short descriptions of the header files related to device drivers.

12.3 Reviewing Device Driver Installation

The actual steps of installing a driver may vary on the source machine and the target machine. When installing a device driver on the ULTRIX operating system, follow the steps described in Chapter 9.

12.4 Checking Driver Routines

The `probe` routine and the `attach` routine may execute differently on the source machine and the target machine. This section discusses some of the differences you need to consider.

A device driver's `probe` routine is called by the system during the boot phase. For VMEbus drivers on ULTRIX, the `probe` routine can take three arguments: *ctrl*, *addr1*, and *addr2*. You should check the `probe` routine from the source machine because the order and number of arguments may be different. An important task performed by the `probe` routine is to access each controller to see whether it is actually present on the system. Because it is not certain whether the device is on the system at the time it is invoked, it is necessary to recover from bus errors when attempting to access device registers. On the Sun Microsystems platform, the `peek` and `poke` routines perform the task of recovering from bus errors. On the Digital platform, the `BADADDR` macro performs a similar task. Note that `BADADDR` does not actually pass data from or to the device. If the probe of the device is successful, the device exists and the `probe` routine must return the size of the I/O space in bytes. If the access to the device is unsuccessful, the `probe` routine returns zero (0), indicating that the device is not present. See Section 10.1.3 for an example of the `BADADDR` macro used with the `skprobe` routine. See Section 10.2.3 for an example of the `BADADDR` macro used with the `xxprobe` routine.

For VMEbus drivers on ULTRIX, the `attach` routine takes a pointer to a `uba_device` structure. You should check the `attach` routine from the source machine because the argument or arguments it takes may be different.

12.5 Checking Data Structures

Data structures are an important area to consider when porting device drivers from other platforms. In general, structure and member names used by the source machine are not the same as those used by the target machine. However, in many cases the members of the source machine perform similar tasks to those performed on the target machine.

This section describes some of the differences between the structures used by Sun Microsystems device drivers and those used by ULTRIX VMEbus device drivers. Specifically, you need to check these data structures:

- `uba_driver`
- `uba_device`
- `uba_ctlr`

12.5.1 Checking the uba_driver Structure

A VMEbus device driver on ULTRIX must declare and initialize a `uba_driver` structure. Likewise, a comparable structure called `mb_driver` must be declared and initialized for a device driver in the Sun Microsystems environment. Table 12-2 compares the members of the `uba_driver` structure with the members of the `mb_driver` structure.

Table 12-2: Comparison of the uba_driver and mb_driver Structures

uba_driver Member	mb_driver Member	Comments
`ud_probe`	`mdr_probe`	Both specify a pointer to the driver's `probe` routine.
`ud_slave`	`mdr_slave`	Both specify a pointer to a `slave` routine located within the device driver.
`ud_attach`	`mdr_attach`	Both specify a pointer to an `attach` routine located within the device driver.
`ud_dgo`	`mdr_go`	Both specify a pointer to a `go` routine located within the device driver. This routine is not used by VMEbus and TURBOchannel device drivers.

Table 12-2: (continued)

uba_driver Member	mb_driver Member	Comments
`ud_addr`	N/A	The `ud_addr` member stores the device's CSR address. This member is not used by VMEbus and TURBOchannel device drivers. The `mb_driver` structure does not have a member that corresponds to `ud_addr`.
N/A	`mdr_done`	The `uba_driver` structure does not have a member that corresponds to `mdr_done`. This member points to a `done` routine located within the device driver.
N/A	`mdr_intr`	This member specifies a pointer to a polling interrupt routine located within the device driver and is specific to Multibus machines. Because Digital does not support Multibus machines, this member is not needed; thus, there is no corresponding member in the `uba_driver` structure.
`ud_dname`	`mdr_dname`	Both specify the name of the device.
`ud_dinfo`	`mdr_dinfo`	The `ud_dinfo` member specifies an array of pointers to `uba_device` structures accessed by this device driver. This array is indexed with the unit number, as specified in the `ui_unit` member of the `uba_device` structure. The `mdr_dinfo` member specifies backpointers to `mbdinit` structures.
`ud_mname`	`mdr_cname`	Both specify the name of the controller.
`ud_minfo`	`mdr_cinfo`	The `ud_minfo` member specifies an array of pointers to `uba_ctlr` structures accessed by this device driver. This array is indexed with the controller number as specified in the `um_ctlr` member of the `uba_ctlr` structure. The `mdr_cinfo` member specifies backpointers to `mbcinit` structures.

Table 12-2: (continued)

uba_driver Member	mb_driver Member	Comments
`ud_xclu`	`mdr_flags`	The `ud_xclu` member specifies the driver's need to exclusively use buffer data paths (bdps). This member is not used by VMEbus device drivers. The `mdr_flags` member specifies several flags, one of which indicates that the device needs exclusive use of the Main Bus.
`ud_addr1_size`	`mdr_size`	The `ud_addr1_size` member specifies the size in bytes of the first CSR area. This area is usually the control status register of the device. The `mdr_size` member specifies the amount of memory in bytes needed by the device.
`ud_addr1_atype`	N/A	The `ud_addr1_atype` member specifies the address space and data size of the first CSR area. The `mb_driver` structure does not have a member that corresponds to `ud_addr1_atype`.
`ud_addr2_size`	N/A	The `ud_addr2_size` member specifies the size in bytes of the second CSR area. This area is usually the data area and is used with devices that have two separate CSR areas. The `mb_driver` structure does not have a member that corresponds to `ud_addr2_size`.
`ud_addr2_atype`	N/A	The `ud_addr2_atype` member specifies the address space and data size of the second CSR area. The `mb_driver` structure does not have a member that corresponds to `ud_addr2_atype`.
N/A	`mdr_link`	The `uba_driver` structure does not have a member that corresponds to `mdr_link`. This member specifies an interrupt routine linked list, which is used by the Sun Microsystems autoconfiguration procedure.

12.5.2 Checking the uba_device and uba_ctlr Structures

To account for devices that do not have controllers, BSD UNIX specifies many similar members in both the device and controller structures. This tradition has been continued by both Sun Microsystems and Digital. In ULTRIX, these structures are called `uba_device` and `uba_ctlr`. In the Sun Microsystems platform, the corresponding structures are `mb_device` and `mb_ctlr`. The most important differences between the Sun Microsystems device and controller structures and the corresponding ULTRIX structures relate to the architecture of their respective machines.

One architectural difference is that the Digital processors support several buses in addition to the VMEbus. Because the VMEbus on Digital machines is only one of several supported buses, the `uba_device` and `uba_ctlr` structures have members that identify the nexus (`ui_nexus` and `um_nexus`), the remote controller number (`ui_rctlr` and `um_rctlr`), and the adapter (`ui_adpt` and `um_adpt`). The device driver writer may need to use these members only when debugging a system crash.

Another architectural difference is the presence of the `ui_bus_priority` member in the `uba_device` structure and the `um_bus_priority` member in the `uba_ctlr` structure. These members specify the configured VMEbus priority level of the device. You should not confuse these members with the `ui_priority` and `um_priority` members, which the driver should use to reference the system priority level of the VMEbus device. The `ui_bus_priority` and `um_bus_priority` members should be used for informational purposes only.

A third architectural difference involves two separate I/O spaces. To accommodate these two I/O spaces, the `uba_device` structure provides the `ui_addr` and `ui_addr2` members and the `uba_ctlr` structure provides the `um_addr` and `um_addr2` members. These members allow you to access a device that occupies two separate I/O spaces with one device driver. Therefore, there are instances where a device that required two drivers on the Sun Microsystems platform only requires one on the Digital platform. However, you can still use two drivers on the Digital platform, if you want.

The interrupt members are slightly different in syntax: Sun Microsystems uses a vector structure defined in the `mb_ctlr` structure to store some of this information. (This vector structure is not used in the `mb_device` structure.) If you use this information, you have to change your code to remove this intermediate reference. The following maps the members of the vector structure to the corresponding members in the `uba_device` and `uba_ctlr` structures.

uba_device Member	uba_ctlr Member	mb_ctlr Member
`ui_intr`	`um_intr`	`v_func`
`ui_ivnum`	`um_ivnum`	`v_vec`
N/A	N/A	`v_ptr`

12.6 Comparing Direct Memory Access Mechanisms

When comparing the Direct Memory Access mechanisms for Digital and Sun Microsystems, you need to consider:

- Underlying mapping mechanisms
- Methods for allocating DMA space
- Maximum DMA

12.6.1 Underlying Mapping Mechanisms

To understand the porting issues involving DMA, you need to understand the underlying mapping mechanisms on the Sun Microsystems platform and the Digital platform. On Sun Microsystems VME implementations, the first 1MB of VME address space (0–1MB) is hard mapped into 1MB of system address space. Any DMA transfer must be performed to buffers in that 1MB of system space.

On ULTRIX systems, there is no hard mapping of VMEbus address space to system space. The mapping is performed by using Page Map Registers (PMRs). Each PMR maps one system page. A PMR can map to any system address, including those in a user process. Therefore, the management of buffers is entirely separated from the mapping operation.

12.6.2 Methods for Allocating DMA Space

The differences in the underlying mapping mechanisms require alternative ways for allocating DMA space. The following code fragments are examples of DMA I/O. The first fragment is for a Sun Microsystems system and the second is for an ULTRIX system.

```
/********************************************************************/
/*                      SUN MICROSYSTEMS                            */
/********************************************************************/

 .
 .
 .
/********************************************************************/
/*           DECLARE ARGUMENTS [1]                   */
/********************************************************************/
/* Declare the arguments passed to kernel routines.  Also, declare  */
/* the variables to contain values returned by these kernel         */
/* routines.                                                        */
/********************************************************************/
        struct cmd {
               int dma_addr;
                   } *cmd_buf;
     struct buf *bp; /* pointer to buf structure */
     struct mb_device *md; /* pointer to mb_device structure */
     int info; /* return from mbsetup */
        struct reg {
                int cmd_addr;
                int start;
                   } regptr *reg; /* pointer to reg structure */
/********************************************************************/

/********************************************************************/
/*           ALLOCATE COMMAND BUFFER [2]             */
```

```
/*******************************************************************/
/* Allocate the command buffer by calling rmalloc and map the      */
/* data buffer by calling mbsetup.                                 */
/*******************************************************************/
     cmd_buf = (struct *cmd)rmalloc(iopbmap, size)
     info = mbsetup(md->md_hd, bp, 0);
/*******************************************************************/

/*******************************************************************/
/*           ACCESS DEVICE REGISTERS 3                */
/*******************************************************************/
/* Access the device registers, obtain the VMEbus address to be    */
/* used, and start the DMA transfer.                               */
/*******************************************************************/
     regptr = (struct reg*) md->md_addr;
     cmd_buf->dma_addr = MBI_ADDR(info);
     regptr->cmd_addr = &cmd_buf - DVMA;
     regptr->start = 1; /* Start DMA */
 .
 .
 .
/* Code for I/O completion */
 .
 .
 .
/*******************************************************************/

/*******************************************************************/
/*           RECYCLE MAP RESOURCES 4                  */
/*******************************************************************/
/* Recycle the previously allocated map resources and free the     */
/* the main bus resources.                                         */
/*******************************************************************/
     rmfree(iopbmap, size, (long)cmd_buf);
     mbrelse(md->md_hd, &info);
/*******************************************************************/

 .
 .
 .
```

ULTRIX

```
/*******************************************************************/
/*******************************************************************/
/*******************************************************************/
/*******************************************************************/
/*******************************************************************/
/*******************************************************************/

/*******************************************************************/
/*           DECLARE ARGUMENTS 1                      */
/*******************************************************************/
/* Declare the arguments passed to kernel routines.  Also, declare */
/* the variables to contain values returned by these kernel        */
/* routines.                                                       */
*/                                                                 */
/*******************************************************************/
        struct cmd {
                int dma_addr;
                   } cmd_buf; /* command buffer */
     struct  buf *bp; /* pointer to buf structure */
     struct    uba_ctlr *um; /* pointer to uba_ctlr structure */
     unsigned int vme_cmd_addr; /* return from vballoc */
     unsigned int vme_data_addr; /* return from vbasetup */
```

```
        struct reg {
                int cmd_addr;
                int start;
                   } regptr *reg; /* pointer to reg structure */
/********************************************************************/

/********************************************************************/
/*           ALLOCATE DMA SPACE [2]                    */
/********************************************************************/
/* Allocate the DMA space and then set up the mapping registers     */
/* for DMA transfer by calling the vballoc and/or the         */
/* vbasetup routine.                                          */
/********************************************************************/

     vme_cmd_addr =
           vballoc(um->um_vbahd, &cmd_buf, sizeof(cmd_buf),
                      VME_DMA|VMEA16D16|VME_BS_NOSWAP);
     vme_data_addr = vbasetup(um->um_vba_hd, bp,
                                VME_DMA|VMEA16D16|VME_BS_NOSWAP);
/********************************************************************/

/********************************************************************/
/*           ACCESS DEVICE REGISTERS [3]               */
/********************************************************************/
/* Access the device registers, obtain the VMEbus address to be     */
/* used, and start the DMA transfer.                                */
/********************************************************************/
     regptr = (struct reg*) um->um_addr;
     cmd_buf->data_addr = vme_data_addr;
     regptr->dma_addr = vme_cmd_addr;
     regptr->start = 1; /* Start DMA */
  .
  .
  .
/* Code for I/O completion */
  .
  .
  .
/********************************************************************/

/********************************************************************/
/*           RELEASE RESOURCES [4]                     */
/********************************************************************/
/* Release the previously allocated resources on the VME adapter.   */
/********************************************************************/

     vbarelse(vhp, vme_cmd_addr);
     vbarelse(vhp, vme_data_addr);
/********************************************************************/

  .
  .
  .
```

[1] In the Sun Microsystems environment, *cmd_buf* specifies a command buffer allocated by `rmalloc`. In the ULTRIX environment, *cmd_buf* specifies the virtual address of a command buffer passed to `vballoc`. Note that *cmd_buf* is a pointer to a structure in the Sun Microsystems environment and a structure of type `cmd` in the ULTRIX environment.

In both the Sun Microsystems and ULTRIX environments, there is a pointer to a `buf` structure. In the ULTRIX environment, the `buf` structure is typically passed in from the driver's `strategy` routine. You pass this `buf` structure to the `vbasetup` routine.

In the Sun Microsystems environment, the `mbsetup` and `mbrelse` routines take a pointer to an `mb_device` structure.

In ULTRIX, when you write a device driver routine that calls `vballoc` or `vbasetup`, you declare a pointer to a `uba_ctlr` structure. You then pass to `vballoc` or `vbasetup` the `um_vbahd` member of the `uba_ctlr` structure or the `ui_vbahd` member of the `uba_device` structure.

In the Sun Microsystems environment, *info* stores the virtual address returned by `mbsetup`. You pass *info* to the `MBI_ADDR` macro to obtain the VMEbus address. In the ULTRIX environment, *vme_cmd_addr* stores the value returned by `vballoc` and *vme_data_addr* stores the value returned by `vbasetup`. The value returned by these routines is a VMEbus address that is mapped to the buffer.

In both environments, you declare a pointer to a `reg` structure. This structure has two members: `cmd_addr` and `start`. The `reg` structure represents the characteristics of the hardware device.

2 In the Sun Microsystems environment, your driver calls `rmalloc` to allocate command, data, or miscellaneous buffers. For large buffers, Sun Microsystems recommends that the driver allocate a `buf` structure and call the `mbsetup` routine to allocate the buffer from the DMA space. To perform DMA to user space, the pages must be mapped onto the 1MB of DMA space by using the `mbsetup` routine. In the ULTRIX environment, you can allocate the DMA space and then set up the mapping registers for DMA transfer by calling the `vballoc` and the `vbasetup` routines or both.

The differences in allocating DMA space can be further elaborated by discussing the arguments passed to the respective routines. The `rmalloc` routine takes two arguments. The first argument is a resource map, which in this example is a preinitialized `rmalloc` map called `iopbmap`. The second argument is the size of the address map to allocate. Note that the return type for `rmalloc` is type cast to `struct cmd *` because the return type for `rmalloc` is of type `long` and the type for the *cmd_buf* argument is of type `struct cmd *`. The `mbsetup` routine takes three arguments. The first argument is a pointer to an `mb_hd` structure. The second argument is the `buf` structure. The last argument is a *flag*, which in this example is the value zero (0).

The `vballoc` and `vbasetup` routines take similar arguments. The primary difference between the two routines is that `vbasetup` takes a pointer to a `buf` structure as an argument, while `vballoc` takes an address and the number of bytes as arguments. You would use `vbasetup` when a `buf` structure is provided to the driver. All file system I/O and most user I/O occur using a `buf` structure. You would use the `vballoc` routine for driver-initiated I/O, for example, device command packets. Each of these routines returns a VMEbus address that is mapped to the buffer. If the requested mapping could not be performed, each of these routines returns a value of zero (0).

3 In the Sun Microsystems environment, *regptr* is set to the base address of the device, that is, its control/status registers. The base address of the device is represented by the value stored in the `md_addr` member of the `mb_device` structure. In the ULTRIX environment, *regptr* is set to the the System Virtual Address (SVA) corresponding to the CSR specified in the system configuration file. This SVA is stored in the `um_addr` member of the `uba_ctlr` structure.

In the Sun Microsystems environment, you call the `MBI_ADDR` macro to obtain the VMEbus address. The `MBI_ADDR` macro subtracts the offset of the DMA space to obtain the VMEbus address to be used. The argument you pass to this macro is the value returned in a previous call to `mbsetup`. In the ULTRIX environment, you set the `data_addr` member of the `cmd_buf` structure to the VMEbus address that is mapped to the buffer (*vme_data_addr*). This address was returned in a previous call to `vbasetup`.

In the Sun Microsystems environment, you set the `cmd_addr` member to the result of the expression `&cmd_buf - DVMA`, which provides the VMEbus address for the command buffer.

In the ULTRIX environment, you set the `cmd_addr` member to the VMEbus address that is mapped to the buffer (*vme_data_addr*). This address was returned in a previous call to `vballoc`.

Finally, in both environments, the `regptr->start=1;` line starts the Direct Memory Access transfer.

4 In the Sun Microsystems environment, you call `rmfree` to recycle the map resource allocated in a previous call to `rmalloc`. You also call `mbrelse` to release the Main Bus DVMA resources allocated in a previous call to `mbsetup`. In the ULTRIX environment, you call `vbarelse` to release resources on the VMEbus adapter registers, which were allocated in a previous call to `vballoc` or `vbasetup`.

The differences in releasing resources can be further elaborated by discussing the arguments passed to these routines. The `rmfree` routine takes three arguments. The first argument is a pointer to the resource map allocated in a previous call to `rmalloc`. In this case, the allocated map was the preinitialized `rmalloc` map called `iopbmap`. The second argument is the size of the address map that was allocated. The third argument is the address at which the allocated map begins. Note that the third argument is the value returned by `rmalloc`. The argument is type cast as a `long` to satisfy the data type required by the `rmfree` routine.

The `mbrelse` routine takes two arguments. The first argument is the pointer to the `mb_device` structure. The second argument is the address of the integer (in this case *info*) returned in a previous call to `mbsetup`.

The `vbarelse` routine takes two arguments. The first argument is a pointer to a `vba_hd` structure, which contains the VMEbus adapter number on which mapping registers were allocated in a previous call to `vballoc` and/or `vbasetup`. The second argument specifies the VMEbus address, which is the value returned in a previous call to `vballoc` and/or `vbasetup`.

12.6.3 Maximum DMA

Because all DMA on Sun Microsystems systems must go to the low 1MB of VMEbus address space, the maximum DMA is 1MB. On ULTRIX systems, the maximum DMA is limited by the number of Page Map Registers and the size of a system page. This may vary among Digital VMEbus adapters. On the DECstation 5000 Model 200, the maximum DMA is 128MB in the A32 space.

12.7 Testing for Device Access

Because various bus error conditions may be created when attempting to access a device that is not present or is not functional, it is necessary to use special routines that protect from those error conditions. These routines are typically called once, in the `probe` routine of the driver. It is also advisable to use these routines when logging device registers after a fatal device error has occurred. This will ensure that the system does not crash because the device is no longer functional.

On Sun Microsystems systems, the `peek` and `poke` family of routines is used. On ULTRIX, the equivalent is the `BADADDR` macro. Because the `BADADDR` macro takes a size as one of its arguments, multiple functions are not needed. Note that ULTRIX has no equivalent to the Sun Microsystems `poke` routine. You must call the `BADADDR` macro prior to any read or write that is to be protected.

12.8 Checking the Design of a Device Driver

As central processing unit performance dramatically increases with each generation of CPUs, certain design inconsistencies may begin to appear. For example, race conditions that did not show up on a slower machine may become magnified on a faster machine. Architectural differences used to achieve these improvements can mean that certain failures that are fatal on some machines are not fatal on others. Some of the improvements may be caches, buffers, and compiler optimizations.

These differences should not affect a well-written driver or application. They simply amplify a problem that already existed but may not have been detected.

One area where a well written driver may still need modification is when operating systems are upgraded or migrated. It has been Digital's software policy to always provide a compatible migration path. This is not guaranteed if your software uses undocumented or unsupported features of the operating system.

12.9 Setting Interrupt Priority Levels

You may be porting a device driver from a machine that uses all seven VMEbus interrupt priority levels. Digital machines can map these seven VMEbus levels to a smaller number of system levels. The DECstation 5000 Model 200 has one system level. Note that the VME adapters choose the device to be serviced based on the VMEbus priority level. The level mapping takes place after the VMEbus priority arbitration completes. For example, a VMEbus level 7 will be processed before a VMEbus level 1; however, both levels may be mapped to the same internal system level (as on the DECstation 5000 Model 200).

12.10 Performing Byte Swapping Operations

Many hardware devices use the big endian model of byte ordering. Digital devices use the little endian model. The mechanisms provided to accomplish the byte swapping are explained in Section 2.2.3.

12.11 Comparing Memory Mapping

The device driver you are porting may have implemented a memory mapping routine. You should compare the memory mapping routine interface of the driver you are porting with that implemented on ULTRIX, which takes three arguments: *device*, *off*, and *prot*. See Section 4.12 for a discussion of the memory map section of a device driver.

Porting TURBOchannel Device Drivers 13

This chapter presents guidelines for porting device drivers from other Digital buses (specifically, the UNIBUS and the Q-bus) to the TURBOchannel. These guidelines are actually summaries of topics described in more detail in other chapters; these summaries are included here as a convenient checklist before porting a driver to the TURBOchannel.

Consider the following when porting device drivers from other Digital buses to the TURBOchannel:

- Structure the driver for a TURBOchannel device like a driver for a UNIBUS or Q-bus device.
- Make sure that the TURBOchannel driver defines a `uba_driver` structure with all necessary information filled in.
- Include the appropriate header files. In general, TURBOchannel drivers include many of the same header files that appear in UNIBUS or Q-bus drivers. However, TURBOchannel drivers must also include `tc.h`.
- Use the routine `wbflush` to assure that a write to I/O space has completed.
- Explicitly flush the processor data cache by calling `bufflush`, if the device performs DMA-to-host memory. Call this routine after the DMA is complete but before releasing the buffer to the system.
- Call `tc_enable_option` to enable a device's interrupt line to the processor. Call `tc_disable_option` to disable a device's interrupt line to the processor.
- Add a TURBOchannel driver to the ULTRIX operating system using the steps described in Chapter 9. Make sure to add the appropriate entry to the `tc_option` table in `/usr/sys/data/tc_option_data.c`.

The `tc_option` table and the system configuration file provide a flexible mechanism for adding third-party devices and device drivers. This table allows third-party device driver writers to map additional device names with their associated names in the system configuration file. Third-party or customer device drivers must conform to standard ULTRIX operating system conventions. For instance, drivers must have a `uba_driver` structure with the name of the device `probe` routine, `attach` routine, device name, and so forth. The qac, for example, has a `uba_driver` structure that looks like this:

```
 .
 .
 .
struct  uba_driver qacdriver =
        { qacprobe, 0, qacattach, 0, qacstd, "qac", qacinfo };
 .
 .
 .
```

The corresponding entry in the system configuration file looks like this:

```
device          qac0        at ibus?    vector qacvint
```

Part VII: Appendixes

Header Files Related to Device Drivers A

Table A-1 lists the header files related to device drivers, along with a short description of their contents. For convenience, the path name is included with the file and the files are listed in alphabetical order. Note, however, that device drivers should include header files that use the relative path name instead of the explicit path name. For example, although `buf.h` resides in `/usr/sys/h/buf.h`, device drivers should include it as `"../h/buf.h"`.

Table A-1: Header Files Related to Device Drivers

Header File	Contents
`/usr/sys/h/buf.h`	Defines the `buf` structure used to pass I/O requests to the `strategy` routine of a block driver.
`/usr/sys/h/clist.h`	Defines the `cblock` structure used to hold `clist` data.
`/usr/sys/h/conf.h`	Defines the `bdevsw` (block device switch), `cdevsw` (character device switch), and `linesw` (tty control line switch) structures. This file is included in the source file `/usr/sys/machine/common/conf.c`.
`/usr/sys/machine/common/cpuconf.h`	Defines a variety of macros, constants, and structures used by the system. The `BADADDR` macro, which is of interest to VMEbus device driver writers, is defined in this file.
`/usr/sys/h/devio.h`	Defines common structures and definitions for device drivers and `ioctl`.
`/usr/sys/h/dir.h`	Defines structures and macros that operate on directories.
`/usr/sys/h/errno.h`	Defines the error codes returned to a user process by a driver. The codes `EIO`, `ENXIO`, `EACCES`, `EBUSY`, `ENODEV`, and `EINVAL` are used by driver routines.
`/usr/sys/h/file.h`	Defines I/O mode flags supplied by user programs to `open` and `fcntl` system calls.
`/usr/sys/h/inode.h`	Defines values associated with the generic file system.
`/usr/sys/h/ioctl.h`	Defines commands for `ioctl` routines in different drivers.

Table A-1: (continued)

Header File	Contents
`/usr/sys/h/kernel.h`	Defines global variables used by the kernel.
`/usr/sys/h/map.h`	Defines structures associated with resource allocation maps.
`/usr/sys/h/mbuf.h`	Defines constants related to memory allocation and macros used for type conversion.
`/usr/sys/h/mtio.h`	Defines commands and structures for magnetic tape operations.
`/usr/sys/h/param.h`	Defines constants and macros used by the ULTRIX kernel.
`/usr/sys/h/proc.h`	Defines the `proc` structure, which defines a user process. This file is not usually included by device driver source files.
`/usr/sys/h/systm.h`	Defines global variables, such as the number of entries in the block switch and the number of character switch entries. It also defines the structure of the system-entry table.
`/usr/sys/io/tc/tc.h`	Contains definitions and routine declarations needed by TURBOchannel drivers.
`/usr/sys/h/time.h`	Contains structures and symbolic names used b time-related routines and macros.
`/usr/sys/h/tty.h`	Defines parameters and structures associated with interactive terminals; also defines the `clist` structure. This file can be included by any device driver that uses the `clist` structure.
`/usr/sys/h/types.h`	Defines system data types and major and minor device macros.
`/usr/sys/h/uio.h`	Contains the definition of the `uio` structure, some symbolic names, and an enumerated data type that can be assigned the value `UIO_READ` or `UIO_WRITE`.
`/usr/sys/h/user.h`	Defines the `user` structure that describes a user process and passes information about I/O requests to device drivers.
`/usr/sys/io/vme/vbareg.h`	Contains definitions for the VMEbus adapter.
`/usr/sys/h/vm.h`	Contains a sequence of include statements that includes all of the virtual memory-related files. Including this file is a quicker way of including all of the virtual memory-related files.
`/usr/sys/h/vmmac.h`	Contains defintions for the `vtokpfnum` kernel routine.

Kernel Support Routines B

This appendix describes:

- The kernel I/O support routines (and macros) used by device drivers
- Special files used by device drivers
- Global variables used by device drivers

B.1 Kernel Support Routines

Table B-1 summarizes the kernel routines discussed in this appendix. Following the table are descriptions of each routine, presented in alphabetical order.

Of particular interest to VMEbus device driver writers are:

- `bufflush`
- `vballoc`
- `vba_get_vmeaddr`
- `vbarelse`
- `vbasetup`
- `swap_lw_bytes`
- `swap_word_bytes`
- `swap_words`
- `vme_rmw`
- `vtokpfnum`
- `wbflush`

TURBOchannel device driver writers will be interested in:

- `bufflush`
- `tc_disable_option`
- `tc_enable_option`
- `vtokpfnum`
- `wbflush`

Note

The following lists the header files most frequently used by any device driver, including VMEbus and TURBOchannel device drivers:

```
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/uio.h"
#include "../../machine/common/cpuconf.h"
```

Table B-1: Summary Description for Kernel I/O Support Routines

Kernel Routine	Summary Description
`BADADDR`	checks read accessibility of addressed data
`bcmp`	compares byte strings
`bcopy`	copies a byte string
`bzero`	zeros a byte string
`bufflush`	flushes the processor data cache
`copyin`	copies data from user space to kernel space
`copyout`	copies data from kernel space to user space
`cprintf`	writes text only to the console
`DELAY`	delays the calling routine a specified number of microseconds
`fubyte`	fetches a byte from user space
`fuword`	fetches a word from user space (See `fubyte`)
`getnewbuf`	returns a pointer to a `buf` structure previously found on a free list
`gsignal`	sends a signal to a process group
`insque`	adds an element to the queue
`iodone`	indicates that I/O is complete
`KM_ALLOC`	allows dynamic allocation of kernel virtual memory
`KM_FREE`	deallocates (frees) the allocated kernel virtual memory
`log_vme_ctlr_error`	logs VMEbus controller errors into the errorlog file
`log_vme_device_error`	logs VMEbus device errors into the errorlog file
`major`	gets the device major number
`makedev`	makes a device number
`minor`	gets the device minor number
`minphys`	bounds the data transfer size
`mprintf`	logs a message to the error logger (See `cprintf`)
`panic`	causes a system crash
`physio`	implements raw I/O

Table B-1: (continued)

Kernel Routine	Summary Description
`printf`	prints text to the console and the error logger (See `cprintf`)
`psignal`	sends a signal to a process
`remque`	removes an element from the queue
`selwakeup`	wakes up a select blocked process
`sleep`	puts a calling process to sleep
`spl5`	sets the Interrupt Priority Level (IPL) field of the Processor Status Longword (PSL) to the level indicated by the routine name
`spl6`	sets the IPL field of the PSL to the level indicated by the routine name
`spl7`	sets the IPL field of the PSL to the level indicated by the routine name
`splbio`	blocks against all I/O interrupts
`splextreme`	blocks against all but halt interrupts
`splimp`	blocks against network device interrupts
`spltty`	blocks against terminal device interrupts
`splx`	resets the hardware interrupt priority to the level specified by the argument
`strcmp`	compares two strings
`strncmp`	compares two strings, using a specified number of characters
`strlen`	computes the length of a string
`subyte`	stores a byte into user space
`suser`	determines if the current process is superuser
`suword`	stores a word into user space (See `subyte`)
`svtophy`	returns the physical address
`swap_lw_bytes`	performs a long word byte swap
`swap_word_bytes`	performs a short word byte swap (See `swap_lw_bytes`)
`swap_words`	performs a word byte swap (See `swap_lw_bytes`)
`tc_disable_option`	disables a device's interrupt line to the processor
`tc_enable_option`	enables a device's interrupt line to the processor
`tc_module_name`	determines the name of a specific option module
`timeout`	initializes a callout queue element
`uiomove`	moves data between user and system virtual space
`untimeout`	removes the scheduled routine from the callout queues

Table B-1: (continued)

Kernel Routine	Summary Description
`uprintf`	nonsleeping kernel printf function (See `cprintf`)
`useracc`	determines read or write access to a user segment
`uvtophy`	returns the physical address
`vballoc`	allocate and set up the DMA mapping registers
`vba_get_vmeaddr`	obtains the VMEbus address
`vbarelse`	releases the resources (map registers) used to map the specified VMEbus address
`vbasetup`	allocate and set up the DMA mapping registers (See `vballoc`)
`vme_rmw`	performs a read-modify-write
`vslock`	locks a virtual segment
`vsunlock`	unlocks a virtual segment
`vtokpfnum`	obtains the page frame number
`wakeup`	wakes up all processes sleeping on a specified address
`wbflush`	ensures a write to I/O space has completed

Name

BADADDR — checks read accessibility of addressed data

Syntax

#include "../../machine/common/cpuconf.h"

BADADDR(*addr*, *length*)
caddr_t *addr*;
int *length*;

Arguments

addr	Specifies the address of the data to be checked for read accessibility.
length	Specifies the length in bytes of the data to be to be checked. Valid values are 1, 2, and 4.

Description

The `BADADDR` macro generates a call to a machine- and model-dependent routine that does a read access check of the data at the supplied address and dismisses any machine check exception that may result from the attempted access.

Return Value

The `BADADDR` macro returns zero (0) if the data is accessible and nonzero if the data is not accessible.

Name

bcmp — compares byte strings

Syntax

unsigned int bcmp(*string1*, *string2*, *length*)
caddr_t *string1*;
caddr_t *string2*;
unsigned int *length*;

Arguments

string1	Specifies the first string to be compared.
string2	Specifies the second string to be compared.
length	Specifies the length in bytes of the data to be compared.

Description

The `bcmp` routine compares byte string *string1* with byte string *string2*. Each string is assumed to be *length* bytes long.

Return Value

The `bcmp` routine returns zero (0) if the compared strings are identical and nonzero if the compared strings are not identical.

See Also

`bcopy, bzero`

Name

bcopy — copies a byte string

Syntax

void bcopy(*string1*, *string2*, *length*)
caddr_t *string1*;
caddr_t *string2*;
unsigned int *length*;

Arguments

string1	Specifies the source string.
string2	Specifies the destination string.
length	Specifies the length in bytes of the data to be copied.

Description

The `bcopy` routine copies *length* bytes from byte string *string1* to byte string *string2*.

Return Value

None.

See Also

`bcmp, bzero`

Name

bzero — zeros a byte string

Syntax

void bzero(*string1*, *length*)
caddr_t *string1*;
unsigned int *length*;

Arguments

string1	Specifies the string to be zeroed.
length	Specifies the length in bytes of the data to be zeroed.

Description

The `bzero` routine places zeros (ASCII null bytes) in *string1*. The value in *string1* is assumed to be *length* bytes long.

Return Value

None.

See Also

`bcmp`, `bcopy`

Name

bufflush — flushes the processor data cache

Syntax

bufflush(*bp*)
struct buf **bp*;

Arguments

bp	Specifies a pointer to a `buf` structure.

Description

The `bufflush` routine flushes the processor data cache. A device driver must explicitly flush the processor data cache if the device performs DMA-to-host-memory. The reason for this is that there is no hardware cache coherency mechanism on some RISC processors. For example, the 5800 systems support hardware cache coherency, while the DECsystem 5400 and DECsystem 5000 Model 200 systems do not.

Return Value

None.

See Also

`wbflush`

Name

copyin — copies data from user space to kernel space

Syntax

int copyin(*user_addr*, *kern_addr*, *nbytes*)
caddr_t *user_addr*;
caddr_t *kern_addr*;
unsigned int *nbytes*;

Arguments

user_addr	Specifies the virtual address in user space to copy the data from.
kern_addr	Specifies the virtual address in kernel space to copy the data to.
nbytes	Specifies the number of bytes of data to copy from user space to kernel space.

Description

The `copyin` routine copies data from user space to kernel space.

Return Value

Upon successful completion, `copyin` returns a value of zero (0). Otherwise, `copyin` can return the following errors:

Error	**Meaning**
`EFAULT`	The *user_addr* argument points outside of the allocated address space.
`EFAULT`	The *nbytes* argument is negative.

See Also

`copyout`

Name

copyout — copies data from kernel space to user space

Syntax

int copyout(*kern_addr*, *user_addr*, *nbytes*)
caddr_t *kern_addr*;
caddr_t *user_addr*;
unsigned int *nbytes*;

Arguments

kern_addr	Specifies the virtual address in kernel space to copy the data from.
user_addr	Specifies the virtual address in user space to copy the data to.
nbytes	Specifies the number of bytes of data to copy from kernel space to user space.

Description

The `copyout` routine copies data from kernel space to user space.

Return Value

Upon successful completion, `copyout` returns the value zero (0). Otherwise, `copyout` can return the following errors:

Error	**Meaning**
`EFAULT`	The *user_addr* argument points outside of the allocated address space.
`EFAULT`	The *nbytes* argument is negative.

See Also

`copyin`

Name

cprintf, mprintf, printf, uprintf — write text to some output device

Syntax

cprintf(*format*, *var_arglist*)
char **format*;
va_dcl *var_arglist*;

mprintf(*format*, *var_arglist*)
char **format*;
va_dcl *var_arglist*;

printf(*format*, *var_arglist*)
char **format*;
va_dcl *var_arglist*;

uprintf(*format*, *var_arglist*)
char **format*;
va_dcl *var_arglist*;

Arguments

format	Specifies a pointer to a string that contains two types of objects. One object is ordinary characters such as ''hello, world,'' which are copied to the output stream. The other object is a conversion specification such as %d. Each conversion specification causes the routines described here to convert and print for the next argument in the variable argument list (*var_arglist*).
var_arglist	Specifies the argument list.

Description

The `cprintf` routine prints only to the console terminal. You generally call this routine to report information when there is a problem with the error logging mechanism or to perform debugging.

The `mprintf` routine logs all text to the kernel error log file. This usually happens during hardware failures that are considered soft and corrected.

The `uprintf` routine prints to the current user's terminal. This routine guarantees not to sleep, thereby allowing it to be called by interrupt routines. It does not perform any space checking, so you do not want to use this routine to print verbose messages. The `uprintf` routine does not log messages to the error logger.

The `printf` routine prints diagnostic information directly on the console terminal, and it writes ASCII text to the error logger. Because `printf` is not interrupt driven, all system activities are suspended when you call it.

The `cprintf`, `mprintf`, `printf`, and `uprintf` routines are scaled-down versions of the C library routines. All of these routines support the following formats that device driver writers will find particularly useful:

b Allows decoding of error registers.

The following illustrates the format of the `printf` routine with the %**b** conversion character:

```
printf("reg=%b\n", regval, "<base><arg>*");
```

In this case, base and arg are defined as:

<base>	Is the output base expressed as a control character. For example, \10 gives octal and \20 gives hexadecimal.
<arg>	Is a sequence of characters. The first character gives the bit number to be inspected (origin 1). The second and subsequent characters (up to a control character, that is, a character <=32) give the name of the register.

The following illustrates a call to `printf`:

```
printf("reg=%b\n", 3, "\10\2BITTWO\1BITONE\n");
```

This example would produce this output:

```
reg=2<BITTWO,BITONE>
```

The following illustrates the format of the `printf` routine with the %**r** and %**R** conversion characters:

```
    printf("%r R", val, reg_desc);
```

r Allows formatted printing of bit fields. This code outputs a string of the format:

```
"<bit field descriptions>"
```

R Allows formatted printing of bit fields. This code outputs a string of the format:

```
"0x%x<bit field descriptions>"
```

You describe the individual bit fields by using a `reg_desc` structure. To describe multiple bit fields within a single word, you can declare multiple `reg_desc` structures. The `reg_desc` structure is defined as follows:

```
struct reg_desc {
  unsigned rd_mask;          /* mask to extract field */
  int rd_shift;             /* shift for extracted   */
                          /* value, - >>, + <<     */
  char *rd_name;             /* field name           */
  char *rd_format;           /* format to print field */
  struct reg_values *rd_values; /* symbolic names of    */
                          /* values              */
};
```

`rd_mask`	Specifies an appropriate mask to isolate the bit field within a word ANDed with the *val* argument.
`rd_shift`	Specifies a shift amount to be done to the isolated bit field. The shift is done before printing the isolated bit field with the `rd_format` member and before searching for symbolic value names in the `rd_values` member.

`rd_name`	If non-NULL, specifies a bit field name to label any output from `rd_format` or searching `rd_values`. If neither `rd_format` nor `rd_values` is non-NULL, `rd_name` is printed only if the isolated bit field is non-NULL.
`rd_format`	If non-NULL, specifies that the shifted bit field value is printed using this format.
`rd_values`	If non-NULL, specifies a pointer to a table that matches numeric values with symbolic names. The routine searches the `rd_values` member, and it prints the symbolic name if it finds a match. If it does not find a match, it prints "???".

The following is a sample `reg_desc` entry:

```
struct reg_desc dsc[] = {
      /* mask      shift      name   format  values */
      { VPNMASK,     0,       "VA",  "0x%x", NULL },
      { PIDMASK,  PIDSHIFT,   "PID",  "%d",   NULL },
      { 0,            0,      NULL,   NULL,   NULL },
};
```

The `cprintf`, `mprintf`, `printf`, and `uprintf` routines also accept a field number, zero filling to length. For example:

```
printf(" %8x\n",regval);
```

The maximum field size is 11.

Return Value

None.

Name

DELAY — delays the calling routine a specified number of microseconds

Syntax

DELAY(*n*)
int *n*;

Arguments

n	Specifies the number of microseconds for the calling process to sleep.

Description

The `DELAY` macro delays the calling routine a specified number of microseconds. `DELAY` spins, waiting for the specified number of microseconds to pass before continuing execution. For example, the following code would result in a 10000 microsecond delay:

```
.
.
.
        DELAY(10000);
.
.
.
```

The range of delays is system-dependent, due to its relation to the granularity of the system clock. The system defines the number of clock ticks per second in the `hz` variable. Specifying any value smaller than 1/hz to the `DELAY` macro results in an unpredictable delay. For any delay value, the actual delay may vary by plus or minus one clock tick.

Usage of the `DELAY` macro is discouraged. The reason for this is that the processor will be consumed for the specified time interval. Consequently, the processor is unavailable to service other processes. In cases where device drivers need timing mechanisms, the `sleep` and `timeout` routines should be used instead of the `DELAY` macro. The most common usage of the `DELAY` macro is in the system boot path. Usage of `DELAY` in the boot path is often acceptable, because there are no other processes in contention for the processor.

Return Value

None.

See Also

`hz`

Name

fubyte, fuword — fetch a byte or a word from user space

Syntax

int fubyte(*user_addr*)
caddr_t* *user_addr*;

int fuword(*user_addr*)
caddr_t* *user_addr*;

Arguments

user_addr	Specifies the user virtual address from which `fubyte` obtains a byte or `fuword` obtains a word.

Description

The `fubyte` routine fetches a byte from user space at the virtual address specified by the *user_addr* argument. The `fuword` routine fetches a word from user space at the virtual address specified by the *user_addr* argument.

Return Value

These routines return a –1 if the current user does not have write access to the specified user virtual address (*user_addr*). Otherwise, these routines return the value at the location, either a byte or a word.

Note

A user of `fuword` will not be able to distinguish between a value of –1 at an accessible user virtual address and an inaccessible address.

See Also

`copyin, copyout, subyte`

Name

getnewbuf — returns a pointer to a `buf` structure previously found on a free list

Syntax

struct buf * getnewbuf()

Arguments

None.

Description

The `getnewbuf` routine returns a pointer to a `buf` structure previously found on a free list. The routine searches the AGE list first for the `buf` structure. If the routine does not find it on the AGE list, it searches the LRU list.

Return Value

The `getnewbuf` routine returns a pointer to a `buf` structure previously found on a free list.

Name

gsignal — sends a signal to a process group

Syntax

gsignal(*pgroup*)
int *pgroup*;
int *signal*;

Arguments

pgroup	Specifies the process group to which you want to send a specified signal.
signal	Specifies the signal that you want to send to the specified process group. You can specify any of the signals defined in `/usr/sys/h/signal.h`.

Description

The `gsignal` routine sends a signal to a process group, invoking `psignal` for each process that is a member of the specified process group.

Return Value

None.

See Also

`psignal`

Name

insque, remque — manipulate the queue

Syntax

```
struct generic_qheader {
                struct generic_qheader *q_forw;
                struct generic_qheader *q_back;
               };

int insque(elem, pred)
struct generic_qheader *elem;
struct generic_qheader *pred;

int remque(elem)
struct generic_qheader *elem;
```

Arguments

elem	Specifies the address of the queue header that contains the element to be manipulated.
pred	Specifies the address of the queue header that contains the element to precede the one specified by *elem* in the queue.

Description

The `insque` routine adds the element specified by the *elem* argument to the queue. The routine inserts *elem* in the next position after *pred* in the queue.

The `remque` routine removes the element specified by the *elem* argument from the queue it is currently in.

Queues are built from doubly linked lists. Each element is linked into the queue through a queue header. Queue headers are all of the generic form `struct generic_qheader`. A given element may have multiple queue headers. This allows each element to be simultaneously linked onto multiple queues.

Any driver routine that manipulates these queues must call an appropriate `spl` routine to ensure that the spl level is high enough to block out any interrupts for other device drivers that may access these queues.

Return Value

None.

Name

iodone — indicates that I/O is complete

Syntax

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

Arguments

bp	Specifies a pointer to a `buf` structure.

Description

The `iodone` routine indicates that I/O is complete and reschedules the process that initiated the I/O.

Return Value

None.

Name

KM_ALLOC — allows dynamic allocation of kernel virtual memory

Syntax

#include <sys/kmalloc.h>

KM_ALLOC(*addr*, *cast*, *nbytes*, *type*, *flags*)
addr; /* pointer to user defined type */
cast; /* user defined type */
unsigned long *nbytes*;
long *type*;
long *flags*;

Arguments

addr	Specifies the pointer to the memory. The data type for this argument is defined by the user.
cast	Specifies that this argument will be cast to the data type of the resulting pointer, to avoid compiler warnings.
nbytes	Specifies the number of bytes to allocate.
type	Specifies the type of memory allocation and used only for statistics. The types of memory allocation are represented by the constants defined in `/usr/sys/h/kmalloc.h`. The constant `KM_DEVBUF` is normally used by device drivers to allocate and free memory.
flags	Specifies the type of memory allocation. These flags are the bitwise inclusive OR of these valid flags bits defined in `/usr/sys/h/kmalloc.h`. Flags bits that are of interest to device driver writers are `KM_NOARG`, `KM_NOWAIT`, `KM_CLEAR`, and `KM_CONTIG`.

Description

The `KM_ALLOC` macro allows dynamic allocation of kernel virtual memory. Device drivers should use `KM_ALLOC` instead of `km_alloc` to allocate temporary storage space.

You can set *flags* to the following:

Value	Meaning
`KM_NOARG`	No special requirements are placed on the memory being allocated. The process could go to sleep, if the requested amount of memory is not available.
`KM_NOWAIT`	The process should not sleep, if the requested amount of memory is not available.

Value	Meaning
`KM_CLEAR`	The memory allocated should be zeroed.
`KM_CONTIG`	The physical memory allocated must be contiguous.
`KM_CALL`	A flag that indicates whether to call the `km_alloc` kernel routine. If you set this flag, the `KM_ALLOC` macro calls the `km_alloc` kernel routine to allocate the memory. If you do not set this flag, the code to perform the memory allocation is expanded in the driver. This results in higher performance, because no subroutine call is involved; however, it results in a larger kernel image. You should use the `KM_CALL` flag whenever performance is not an issue, that is, during startup and initialization. You should not use the `KM_CALL` for any memory allocations in performance-sensitive code regions.

Return Value

None.

See Also

`KM_FREE`

Name

KM_FREE — deallocates (frees) the allocated kernel virtual memory

Syntax

#include <kmalloc.h>

KM_FREE(*addr*, *type*)
addr; /* type of pointer is user defined */
long *type*;

Arguments

addr	Specifies the pointer to the memory to be freed. You must have previously set this pointer in a call to `KM_ALLOC`. The data type for this argument is defined by the user.
type	Specifies the type of memory allocation and used only for statistics. The types of memory allocation are represented by the constants defined in `/usr/sys/h/kmalloc.h`. The constant `KM_DEVBUF` is normally used by device drivers to allocate and free memory.

Description

The `KM_FREE` macro deallocates (frees) the allocated kernel virtual memory, which was allocated in a previous call to `KM_ALLOC`.

Return Value

None.

See Also

`KM_ALLOC`

Name

log_vme_ctlr_error — logs VMEbus controller errors into the errorlog file

Syntax

log_vme_ctlr_error(*text*, *vhp*, *devptr*)
char * *text*;
struct vba_hd * *vhp*;
struct uba_ctlr * *devptr*;

Arguments

text	Specifies an ASCII error message supplied by the device driver.
vhp	Specifies a pointer to a `vba_hd` structure. When you write a driver routine that calls `log_vme_ctlr_error`, you declare a pointer to a `uba_ctlr` structure. You then pass the `um_vbahd` member of the `uba_ctlr` structure as the second argument to this routine.
devptr	Specifies a pointer to a `uba_ctlr` structure.

Description

The `log_vme_ctlr_error` routine logs VMEbus controller errors into the errorlog file. This routine allocates a message packet that includes the ASCII text supplied by the driver, controller information, and the VMEbus adapter registers.

Return Value

None.

See Also

`log_vme_device_error`

Name

log_vme_device_error — logs VMEbus device errors into the errorlog file

Syntax

log_vme_device_error(*text*, *vhp*, *devptr*)
char * *text*;
struct vba_hd * *vhp*;
struct uba_device * *devptr*;

Arguments

text	Specifies an ASCII error message supplied by the device driver.
vhp	Specifies a pointer to a `vba_hd` structure. When you write a driver routine that calls `log_vme_device_error`, you declare a pointer to a `uba_device` structure. You then pass the `ui_vbahd` member of the `uba_device` structure as the second argument to this routine.
devptr	Specifies a pointer to a `uba_device` structure.

Description

The `log_vme_device_error` routine logs VMEbus device errors into the errorlog file. This routine allocates a message packet that includes the ASCII text supplied by the driver, device information, and the VMEbus adapter registers.

Return Value

None.

See Also

`log_vme_ctlr_error`

Name

major — gets the device major number

Syntax

#include "../h/types.h"

major(*device*)
dev_t *device*;

Arguments

device	Specifies the number of the device for which the `major` macro will obtain the major device number.

Description

The `major` macro gets the device major number associated with the device specified by the *device* argument.

Return Value

None.

See Also

`minor, makedev`

Name

makedev — makes a device number

Syntax

makedev(*major*, *minor*)
int *major*;
int *minor*;

Arguments

major	Specifies the major number for the device.
minor	Specifies the minor number for the device.

Description

The `makedev` macro makes a device number of type `dev_t` based on the numbers specified for the *major* and *minor* arguments. This macro is defined in `/usr/sys/h/types.h`.

Return Value

None.

See Also

`major`, `minor`

Name

minor — gets the device minor number

Syntax

#include "../h/types.h"

minor(*device*)
dev_t *device*;

Arguments

device	Specifies the number of the device for which the `minor` macro will obtain the minor device number.

Description

The `minor` macro gets the device minor number associated with the device specified by the *device* argument.

Return Value

None.

See Also

`major, makedev`

Name

minphys — bounds the data transfer size

Syntax

unsigned int minphys(*bp*)
struct buf * *bp*;

Arguments

bp	Specifies a pointer to a `buf` structure.

Description

The `minphys` routine bounds the data transfer size by checking the `b_bcount` member of the `buf` structure. If the `b_bcount` member is greater than 64 * 1024, `minphys` sets `b_bcount` to 64 * 1024.

Return Value

The `minphys` routine does not return a value. However, it may change the contents of the `b_bcount` member of the `buf` structure.

See Also

`physio`

Name

panic — causes a system crash

Syntax

panic(*message*)
char * *message*;

Arguments

message	Specifies the message you want the `panic` routine to print on the console.

Description

The `panic` routine is called to cause a system crash, usually because of fatal errors. It prints the message, the contents of useful registers (for example, sp, fp, pc, and so forth), the interrupt stack, and the kernel stack to the console and error logger. After printing the message, `panic` reboots the system.

Return Value

None.

See Also

`printf`

Name

physio — implements raw I/O

Syntax

physio(*strategy*, *bp*, *device*, *rwflag*, *mincnt*, *uio*)
int (**strategy*)();
register struct buf **bp*;
dev_t *device*;
int *rwflag*;
unsigned int (**mincnt*)();
struct uio **uio*;

Arguments

strategy	Specifies the device driver's strategy routine for the device.
bp	Specifies a pointer to a `buf` structure. This structure contains information such as binary status flags, the major/minor device numbers, the address of the associated buffer, and so forth. Note that this buffer is always a special buffer header owned exclusively by the device for handling I/O requests.
device	Specifies the device number.
rwflag	Specifies the read/write flag.
mincnt	Specifies a pointer to a `minphys` routine.
uio	Specifies a pointer to a `uio` structure.

Description

The `physio` routine implements raw I/O. This routine maps the request directly into the user buffer, without using `bcopy`.

Return Value

None.

See Also

`vslock, vsunlock, minphys`

Name

psignal — sends a signal to a process

Syntax

psignal(*process*, *signal*)
struct proc **process*;
int *signal*;

Arguments

process	Specifies a pointer to a `proc` structure.
signal	Specifies the signal that you want to send to the specified process. You can specify any of the signals defined in `/usr/sys/h/signal.h`.

Description

The `psignal` routine posts a signal to the specified process. The posting of a signal causes that signal to be added to the set of pending signals for the specified process. Depending on the state of the process and the state of the process's signals, this signal may be ignored, masked, caught by a tracing parent, or caught by the actual target process. If the signal is to be delivered to the target process, `psignal` examines and modifies the process state to prepare the execution of the appropriate signal handler.

Return Value

None.

See Also

`gsignal`

Name

selwakeup — wakes up a select blocked process

Syntax

selwakeup(*process*, *collision*)
register struct proc **process*;
int *collision*;

Arguments

process	Specifies a pointer to a `proc` structure. This is typically a pointer to a process that has issued a select which blocked. For example, a process can select waiting for input, which causes the process to block until input becomes available.
collision	Specifies whether more than one process is blocked on this file descriptor.

Description

The `selwakeup` routine wakes up a select blocked process. This routine is used to notify a process that the condition causing the process to be blocked has changed. This allows the `select` system call to return.

It is possible to have more than one process blocked on the same file descriptor. When the blocking condition is met, `selwakeup` is called to allow the blocked process to proceed. The `selwakeup` routine examines the *collision* argument to determine if more than one process is blocked on this file descriptor. If you pass the value 0 to *collision*, then only the process pointed to by the *process* argument will be placed in a runnable state to allow the process to unblock. If you pass a nonzero value to *collision*, then there is more than one process to be unblocked. In this case, you notify the other processes by issuing a wakeup on a common address that the blocked processes would be sleeping on.

Return Value

None.

See Also

`wakeup`

Name

sleep — puts a calling process to sleep

Syntax

sleep(*channel*, *pri*)
caddr_t *channel*;
int *pri*;

Arguments

channel	Specifies a unique address associated with the calling process to be put to sleep.
pri	Specifies the priority of the calling process upon waking.

Description

The `sleep` routine puts a calling process to sleep on the address specified by the *channel* argument. This address should be unique to prevent unnecessary wake/sleep cycles. Upon waking, the calling process has the priority you specified in the *pri* argument. If the numerical value of *pri* is less than `PZERO` (which has the value 25), signals are queued but the sleeping process will not be waked.

The `sleep` and `wakeup` pair of routines block and then wake a process. Generally, device drivers call these routines to wait for the transfer to complete interrupt from the device. That is, the `write` routine of the device driver sleeps on the address of a known location, and the device's interrupt service routine wakes the process when the device interrupts. It is the responsibility of the waked process to check if the condition for which it was sleeping has been removed.

Generic priorities (for example, `PZERO` and `PUSER`) are defined in `/usr/sys/h/param.h`. Device driver writers can define their own priorities.

Return Value

None.

See Also

`wakeup`

Name

spl5, spl6, spl7 splbio, splextreme, splimp, spltty, splx— blocks against all I/O interrupts

Syntax

int spl5()

int spl6()

int spl7()

int splbio()

int splextreme()

int splimp()

int spltty()

int splx(*old_spl*)
int *old_spl*;

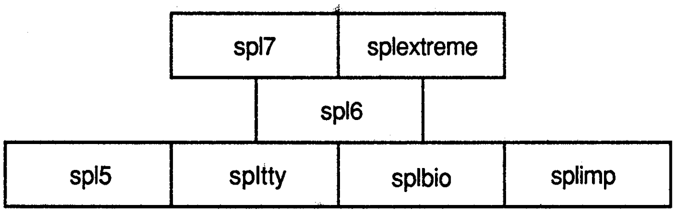
Arguments

old_spl	Specifies the interrupt mask to restore the processor to. This value was returned from a previous call to one of these `spl` routines: `spl5`, `spl6`, `spl7`, `splbio`, `splextreme`, `splimp`, or `spltty`.

Description

The `spl5`, `spl6`, `spl7`, `splbio`, `splextreme`, `splimp`, and `spltty` routines set the processor interrupt mask to an appropriate value. The hierarchical relationship between these values is shown in Figure B-1.

Figure B-1: spl Hierarchical Relationships



ZK-0211U-R

As the figure shows, `spl7` and `splextreme` occupy identical positions at the top of the hierarchy. Those routines at the top of the hierarchy can block all interrupts.

The `spl6` routine is next in the hierarchy. The `spl5`, `splbio`, `splimp`, and `spltty` routines are last on the hierarchy, and they set the processor interrupt mask to identical values. The `spl` routines with character names are preferred over the ones with numbers in their names. The `spl5`, `spl6`, and `spl7` routines are provided for compatibility with older versions of the ULTRIX operating system.

Setting the processor interrupt mask to `splextreme` blocks all interrupts. This includes device (`spl5`) and clock interrupts (`spl6`), as well as those internal-based interrupts reporting the existence of abnormal conditions (`spl7`). This setting is only used with extreme care in highly validated code where interrupts of any type cannot be tolerated. Most device drivers do not have such sections of code.

Setting the processor interrupt mask to `spl6` blocks all interrupts except for those internal-based interrupts reporting the existence of abnormal conditions. This includes device- and clock-generated interrupts. This setting, too, is used with extreme care, as the blocking of clock-generated interrupts can result in degradation of all timer-based functions. Most device drivers need never block interrupts to this extent.

Setting the processor interrupt mask to `spltty`, `splbio`, or `splimp` blocks all device interrupts. These settings are used frequently by device drivers. The processor is also set to one of these interrupt masks prior to invocation of any device interrupt service routine.

The `splx` routine restores the processor interrupt mask to its previous value. This value must have been obtained from a previous call to `splbio`, `splextreme`, `splimp`, `spltty`, `spl5`, `spl6`, or `spl7`.

The `spl` routines allow the creation of critical sections. A critical section is a special segment of code that contains a guarantee as to what kinds of interrupts can occur. You create critical sections by calling `splbio`, `splextreme`, `splimp`, or `spltty`. You terminate these critical sections by calling `splx`. These `spl` routines are used by device drivers mainly to synchronize accesses to data structures.

Device drivers often access a variety of data structures from within their interrupt service routines. They also access these same data structures from other places, including from within routines invoked in process context. Interruptions must be prohibited while such accesses are made to prevent possible data structure corruption. Critical sections represent the mechanism used by drivers to accomplish this requirement.

Note

> Not all data structure accesses need be synchonized. Only accesses to dynamically changing fields require protection through use of critical sections.

The following example illustrates how a device driver protects a criticial section of code:

```
  .
  .
  .
old_spl = splbio(); /* Set interrupt mask to block all I/O interrupts.  */
  .
  .
  .
/* criticial code section */
  .
  .
  .
```

```
(void) splx(old_spl); /* Restore interrupt mask to previous value. */
 .
 .
 .
```

Return Value

These routines return the current spl level.

Name

strcmp — compares two strings

Syntax

strcmp(*string1*, *string2*)
char **string1*;
char **string2*;

Arguments

string1	Specifies the string to be compared with *string2*.
string2	Specifies the string to be compared with *string1*.

Description

The `strcmp` routine compares *string1* with *string2* and returns an integer that is greater than, equal to, or less than zero (0), according to whether *string1* is lexicographically greater than, equal to, or less than *string2*. A string is an array of characters terminated by a NULL character.

Return Value

This routine returns an integer that is greater than, equal to, or less than zero (0), depending on the results of the comparison between *string1* and *string2*.

See Also

`strncmp`, `strlen`

Name

strncmp — compares two strings, using a specified number of characters

Syntax

strncmp(*string1*, *string2*, *n*)
char **string1*;
char **string2*;
int *n*;

Arguments

string1	Specifies the string to be compared with *string2*.
string2	Specifies the string to be compared with *string1*.
n	Specifies the maximum number of characters that can be compared.

Description

The `strncmp` routine compares two strings, using a specified number of characters. A string is an array of characters terminated by a NULL character. The `strncmp` routine compares *string1* with *string2*, comparing at most *n* characters. It returns an integer that is greater than, equal to, or less than zero (0), according to whether *string1* is lexicographically greater than, equal to, or less than *string2*.

Return Value

This routine returns an integer that is greater than, equal to, or less than zero (0), depending on the results of the comparison between *string1* and *string2*.

See Also

`strcmp`, `strlen`

Name

strlen — performs string operations

Syntax

int strlen(*string1*)
caddr_t**string1*;

Arguments

string1	Specifies the address of a string (arrays of characters terminated by a null character).

Description

The `strlen` routine determines the number of characters in the *string1* argument, not including the terminating null character.

Return Value

This routine returns the number of characters in the *string1* argument, not including the terminating null character.

See Also

`strcmp, strncmp`

Name

subyte, suword — store a byte or a word into user space

Syntax

int subyte(*user_addr*, *value*)
caddr_t* *user_addr*;
unsigned long *value*;

int suword(*user_addr*, *value*)
caddr_t* *user_addr*;
unsigned long *value*;

Arguments

user_addr	Specifies the user virtual address that is to be set to the specified value: a byte or a word.
value	Specifies the value (a byte or a word) that will be stored at the specified user virtual address.

Description

The `subyte` routine stores a byte into user space at the virtual address specified by the *user_addr* argument. The `suword` routine stores a word into user space at the virtual address specified by the *user_addr* argument.

You must specify an address that is within the virtual address space of the current process. The virtual address must also be write accessible by the process. Protection against inaccessible address faults is provided while storage occurs.

Return Value

These routines return a –1 if the current user does not have write access to the specified user virtual address (*user_addr*). Otherwise, these routines return a value other than –1.

See Also

`copyin, copyout, fubyte`

Name

swap_lw_bytes, swap_word_bytes, swap_words — perform byte swapping operations

Syntax

unsigned int swap_lw_bytes(*buffer*)
unsigned int *buffer*;

unsigned int swap_word_bytes(*buffer*)
unsigned int *buffer*;

unsigned int swap_words(*buffer*)
unsigned int *buffer*;

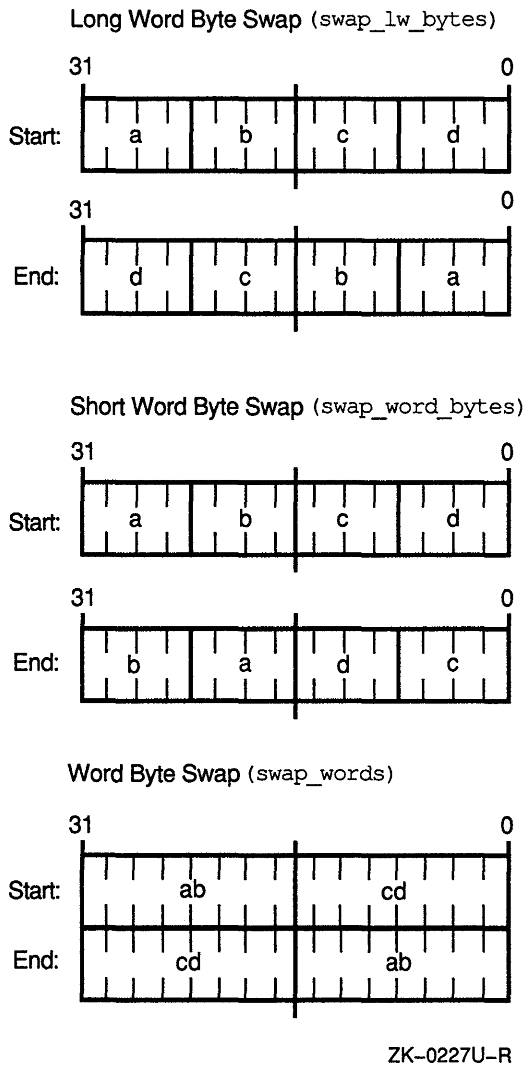
Arguments

buffer	Specifies a 32-bit (4 bytes) quantity.

Description

The `swap_lw_bytes` routine performs a long word byte swap. The `swap_word_bytes` routine performs a short word byte swap. The `swap_words` routine performs a word byte swap. Although the VMEbus does not specify any particular byte ordering, many devices use a big endian model. Because Digital devices support the little endian model of byte ordering, there is a need for these byte swapping routines. These routines perform the same type of byte swapping as that provided by the VMEbus adapter hardware. Figure B-2 compares and contrasts the byte swapping performed by these routines. For the purposes of the following discussion, a long word is equal to 4 bytes; a short word is equal to 2 bytes; and 1 byte is equal to 8 bits. The `swap_lw_bytes` routine takes the 32-bit quantity specified by the *buffer* argument and swaps all four bytes. The `swap_word_bytes` routine takes the 32-bit quantity specified by the *buffer* argument and swaps the individual bytes that make up each word of the 32-bit quantity. The `swap_words` routine takes the 32-bit quantity specified by the *buffer* argument and swaps the two 16-bit words.

Figure B-2: VMEbus Byte Swapping



Return Value

These routines return the result of the byte swapping.

Name

suser — determines if the current process is superuser

Syntax

int suser()

Arguments

None.

Description

The `suser` routine determines if the current process is superuser. The superuser is identified by a user identification number of 0.

Device drivers often need to restrict certain operations to superusers only. The need to limit operations is always device-specific. The `suser` routine provides the verification tool for enforcing such restrictions.

The ASU bit flag is set in the `u_acflag` member of the `user` structure whenever the current process is superuser. The `u_acflag` member is a user area field that contains accounting flags for the current process. By setting this bit flag, the information that the current process used its superuser privileges while executing is saved in the accounting record generated on process termination.

Return Value

This routine returns a value of 1 if the current process is superuser. Otherwise, the routine returns a value of zero (0).

Side Effects

The `EPERM` error value is set in the `u_error` member of the `user` structure whenever the current process is not superuser. This user field contains the error code automatically returned to the current process in the *errno* external integer following failure of a system call. By setting this value and disallowing the operation, the current process will be able to ascertain the appropriate reason (''not owner'') why its request was denied.

Name

svtophy — returns the physical address

Syntax

u_long svtophy(*kern_addr*)
caddr_t *kern_addr*;

Arguments

kern_addr	Specifies the kernel virtual address.

Description

The `svtophy` macro returns the physical address associated with the kernel virtual address you specified in the *kern_addr* argument.

Return Value

This routine returns the physical address associated with the specified virtual address.

See Also

`uvtophy`

Name

tc_disable_option — disables a device's interrupt line to the processor

Syntax

tc_disable_option(*ui*)
struct uba_device **ui*;

Arguments

ui	Specifies a pointer to a `uba_device` structure or a `uba_ctlr` structure. Note that the function definition shows a pointer to a `uba_device` structure.

Description

The `tc_disable_option` routine disables a device's interrupt line to the processor. A device driver uses this routine only if the device must have its interrupts alternately enabled and disabled during configuration or during operation.

Return Value

None.

See Also

`tc_enable_option`

Name

tc_enable_option — enables a device's interrupt line to the processor

Syntax

tc_enable_option(*ui*)
struct uba_device **ui*;

Arguments

ui	Specifies a pointer to a `uba_device` structure or a `uba_ctlr` structure. Note that the function definition shows a pointer to a `uba_device` structure.

Description

The `tc_enable_option` routine enables a device's interrupt line to the processor. A device driver uses this routine only if the device must have its interrupts enabled during configuration. The ULTRIX kernel automatically enables the device's interrupts after configuration, depending on what you specified in the `tc_option` data table.

Return Value

None.

See Also

`tc_disable_option`

Name

tc_module_name — determines the name of a specific option module

Syntax

tc_module_name(*ui*, *cp*)
struct uba_device **ui*;
char cp [TC_ROMNAMLEN];

Arguments

ui	Specifies a pointer to a `uba_device` structure or a `uba_ctlr` structure. Note that the function definition shows a pointer to a `uba_device` structure.
cp	Specifies a character array to be filled in by `tc_module_name`.

Description

The `tc_module_name` routine fills in the character array *cp* with the ASCII string of the TURBOchannel option's module name referred to by the pointer to the `uba_device` or `uba_ctlr` structure.

Return Value

This routine returns a value of –1 if it was unable to use the *cp* pointer you passed.

Name

timeout — initializes a callout queue element

Syntax

timeout(*function*, *argument*, *time*)
int (**function*) ();
caddr_t *argument*;
register int *time*;

Arguments

function	Specifies the address of the routine to be called.
argument	Specifies a single argument passed to the called routine.
time	Specifies the amount of time to delay before calling the specified routine. Time is expressed as *time* (in seconds) * `hz`.

Description

The `timeout` routine initializes a callout queue element to make it easy to execute the specified routine at the time specified in the *time* argument. Callout routines are often used for infrequent polling or error handling. The routine you specified will be called on the interrupt stack (not in processor context) as dispatched from the `softclock` routine. The `timeout` routine places a callout structure on the callout queue. The `hardclock` routine decrements the front elements' *time_till_due* until the specified routine is dispatched a few milliseconds after the time specified in the *time* argument. The granularity of the time delay is dependent on the hardware. For example, the granularity on a DECstation 3100 is approximately 4 milliseconds.

The global variable `hz` contains the number of clock ticks per second. This variable is a second's worth of clock ticks. Thus, if you wanted a four-minute timeout, you would pass 4 * 60 * hz as the third argument to `timeout`:

```
.
.
.
timeout(lptout, (caddr_t)dev, 4 * 60 * hz);
.
.
.
```

Return Value

None.

See Also

`untimeout`

Name

uiomove — moves data between user and system virtual space

Syntax

#include <sys/uio.h>

uiomove(*kern_buf*, *nbytes*, *rwflag*, *uio*)
register caddr_t *kern_buf*;
int *nbytes*;
enum uio_rw *rwflag*;
struct uio **uio*;

Arguments

kern_buf	Specifies a pointer to the kernel buffer in system virtual space.
nbytes	Specifies the number of bytes of data to be moved.
rwflag	Specifies whether data is to be moved from or to user space. Each operation is represented by the appropriate enumerated data type: `UIO_READ` (move data to user virtual space) or `UIO_WRITE` (move data from user virtual space).
uio	Specifies a pointer to the `uio` structure. This structure describes the current position within a logical user buffer in user virtual space. See Section 5.1.3 for a description of the `uio` structure.

Description

The `uiomove` routine moves data between user and system virtual space. Data may be moved in either direction. Accessibility to the logical user buffer is verified before the move is made. Accessibility to the kernel buffer is always assumed.

The kernel buffer must always be of sufficient size. It cannot be less than the number of bytes requested to be moved. Data corruption or a system panic may result if this is ever the case.

The size of the logical user buffer as described by the `uio` structure may be less than, equal to, or greater than the number of bytes requested. The number of bytes actually moved is truncated whenever this size is not sufficient to fulfill a request. In all other cases, only the bytes requested are moved.

Normally there is no need for device drivers to set up `uio` structures or worry about their composition or content. The `uio` structures are usually set up external to drivers. Their addresses are passed in through the `cdevsw` as arguments to driver read and write routines. The user logical buffers they describe are accessed only by routines external to the driver, for example, `uiomove`. The external `uio` structures are quite often updated by such accesses.

The `uiomove` routine always updates the `uio` structure to reflect the number of bytes actually moved. The structure continues to describe the current position within the logical user buffer. The structure members that are subject to change are listed in the Side Effects section. See Section 5.1.3 for a description of the `uio` structure.

Return Value

A zero (0) is returned whenever the user virtual space described by the `uio` structure is accessible and the data is successfully moved. Otherwise, an `EFAULT` error value is returned. This indicates an inability to fully access the user virtual space from within the context of the current process. A partial move may have occurred before the logical user buffer became inaccessible. The `uio` structure is appropriately updated to reflect such partial moves.

The `EFAULT` return value is suitable for placement in the `u_error` member of the `user` structure. Following failure of a system call, this member contains the error code automatically returned in *errno* to the current process. Device drivers should explicitly set this value when it is returned and disallow the requested operation. This allows the current process to determine the appropriate reason (''bad address'') why its request could not be satisfied.

Side Effects

The following members of the `uio` structure may be updated:

Value	Meaning
`uio_iov`	The address of the current logical buffer segment
`uio_iovcnt`	The number of remaining logical buffer segments
`uio_resid`	The size of the remaining logical buffer
`uio_offset`	The current offset into the full logical buffer.

The following members of the logical buffer segment descriptor vector (`uio_iov`) may be updated:

Value	Meaning
`iov_base`	Specifies the address of the current byte within the current logical buffer segment.
`iov_len`	Specifies the remaining size of the current segment.

Note

The `uiomove` routine can also be used to move data solely within system virtual space. In such cases, `uiomove` continues to specify a pointer to a `uio` structure. However, in this circumstance, the structure describes a logical buffer in system virtual space. See Section 5.1.3 for an explanation of the structure.

See Also

`copyin`, `copyout`, `fubyte`, `subyte`

Name

untimeout — removes the scheduled routine from the callout queues

Syntax

untimeout(*function*, *argument*)
int (**function*) ();
caddr_t *argument*;

Arguments

function	Specifies the address of the routine to be removed from the callout queues.
argument	Specifies a single argument passed to the called routine.

Description

The `untimeout` routine removes the scheduled routine from the callout queues. The specified routine was placed on the callout queues in a previous call to `timeout`. The *argument* parameter must match the *argument* parameter provided in the previous call to `timeout`.

Return Value

None.

See Also

`timeout`

Name

useracc — determines read or write access to a user segment

Syntax

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

int useracc(user_addr, nbytes, rwflag)
caddr_t user_addr;
int nbytes;
int rwflag;
```

Arguments

user_addr	Specifies the address of the user segment.
nbytes	Specifies the size of the user segment.
rwflag	Specifies the desired access, either `B_READ` or `B_WRITE`.

Description

The `useracc` routine determines read or write access to a user segment. A user segment is a representation of a portion of the virtual address space of the current process. The examination is made within the context of the current process. It also verifies existence of the user segment within the virtual address space of the current process.

You can set the *rwflag* argument to the following values:

Value	Meaning
`B_READ`	Segment is checked for read access
`B_WRITE`	Segment is checked for write access

Return Value

Return Value	Meaning
0	Access is not allowed. User segment is nonexistent within the process address space.
1	Access is allowed.

Name

uvtophy — returns the physical address

Syntax

u_long uvtophy(*process*, *addr*)
struct proc **process*;
int *addr*;

Arguments

process	Specifies a pointer to a `proc` structure.
addr	Specifies the user address that corresponds to the physical address that you want.

Description

The `uvtophy` routine returns the physical address associated with the user virtual address you specified in the *addr* argument. You must lock the page in main memory (by calling `vslock`) prior to calling `uvtophy`.

Return Value

This routine returns the physical address associated with the specified user virtual address.

See Also

`svtophy`

Name

vballoc, vbasetup — allocate and set up the DMA mapping registers

Syntax

unsigned int vballoc(*vhp*, *addr*, *bcnt*, *flags*, *vme_addr*)
struct vba_hd * *vhp*;
caddr_t *addr*;
int *bcnt*;
int *flags*;
long *vme_addr*;

unsigned int vbasetup(*vhp*, *bp*, *flags*, *vme_addr*)
struct vba_hd * *vhp*;
struct buf **bp*;
long *flags*;
long *vme_addr*;

Arguments

vhp	Specifies a pointer to a `vba_hd` structure. When you write a driver routine that calls `vballoc` or `vbasetup`, you declare a pointer to a `uba_ctlr` or a `uba_device` structure. You then pass the `um_vbahd` member of the `uba_ctlr` structure or the `ui_vbahd` member of the `uba_device` structure as the first argument to these routines.
addr	Specifies the beginning virtual address.
bp	Specifies a pointer to a `buf` structure.
bcnt	Specifies the byte count (size) of the address space you want to allocate.
flags	Specifies the bitwise inclusive OR of a valid bit representing the address space and data size and bits representing other characteristics. A table of the valid bits appears in the Description section.
vme_addr	Specifies an address in the appropriate DMA space (the A24 or the A32 DMA space). You can specify the value 0 or some specific address from the DMA space.

If you specify 0 and the `asc` member of the `vbadata` structure is set to `VME_MAP_LOW` (the default), `vballoc` or `vbasetup` uses the next available VMEbus address in the A24 or the A32 DMA space. If you specify some specific address and `asc` is set to `VME_MAP_LOW`, these routines attempt to allocate space at that VMEbus address. The address must be on a 4K boundary.

If you set the `asc` member of the `vbadata` structure to the constant `VME_MAP_HIGH` and specify the value 0, `vballoc` or `vbasetup` selects an address in the second gigabyte of the VMEbus address space for the mapping of the DMA PMRs for this adapter. If you specify some specific address and set `asc` to `VME_MAP_HIGH`, these routines attempt to allocate space at that address offset by one gigabyte.

For example, if you pass the address 010000000 to *vme_addr* and set `asc` to `VME_MAP_HIGH`, `vballoc` or `vbasetup` returns an address of 410000000 if space is available. See Section 2.3.1 for the figure that illustrates this address space.

Description

The `vballoc` and `vbasetup` routines allocate and set up the DMA mapping registers. On ULTRIX systems, there is no hard mapping of VMEbus address space to system space. The mapping is performed by using Page Map Registers (PMRs). Each PMR maps one system page. A PMR can map to any system address, including those in a user process. Therefore, the management of buffers is entirely separated from the mapping operation.

You can use the `vballoc` and the `vbasetup` routines or both to allocate and set up the DMA mapping registers. The primary difference between the two routines is that `vbasetup` takes a pointer to a `buf` structure as an argument, while `vballoc` takes an address and the number of bytes as arguments. You would use `vbasetup` when a `buf` structure is provided to the driver. All file system I/O and most user I/O occur using a `buf` structure. You would use the `vballoc` routine for driver-initiated I/O, for example, device command packets. Each of these routines returns a VMEbus address that is mapped to the buffer. If the requested mapping could not be performed, each of these routines returns a value of zero (0).

As stated previously, each of these routines can specify the bitwise inclusive OR of the valid bit representing the address space and data size and the bits representing other characteristics. The following describes the valid flags bits:

Flags Bits	Meaning
`VME_DMA`	Specifies the need for DMA access.
`VME_RESERV`	Specifies the reserve VMEbus address space.
`VME_CANTWAIT`	Specifies the driver's need to have the VMEbus address space now. If this flag is not set and the resources needed to perform the mapping are not available, the process is put to sleep until resources become available.
`VME_BS_NOSWAP`	Specifies no byte swapping.
`VME_BS_BYTE`	Specifies byte swapping in bytes.
`VME_BS_WORD`	Specifies byte swapping in words.
`VME_BS_LWORD`	Specifies byte swapping in long words.
`VMEA16D08`	Specifies a request for the 16-bit address space and the 8-bit data size.
`VMEA16D16`	Specifies a request for the 16-bit address space and the 16-bit data size.
`VMEA16D32`	Specifies a request for the 16-bit address space and the 32-bit data size.

Flags Bits	Meaning
`VMEA24D08`	Specifies a request for the 24-bit address space and the 8-bit data size.
`VMEA24D16`	Specifies a request for the 24-bit address space and the 16-bit data size.
`VMEA24D32`	Specifies a request for the 24-bit address space and the 32-bit data size.
`VMEA32D08`	Specifies a request for the 32-bit address space and the 8-bit data size.
`VMEA32D16`	Specifies a request for the 32-bit address space and the 16-bit data size.
`VMEA32D32`	Specifies a request for the 32-bit address space and the 32-bit data size.

Return Value

Each of these routines returns a VMEbus address that is mapped to the buffer. If the requested mapping could not be performed, each of these routines returns a value of zero (0).

See Also

`vbarelse`

Name

vba_get_vmeaddr — obtains the VMEbus address

Syntax

u_long vba_get_vmeaddr(*vhp*, *addr*)
struct vba_hd * *vhp*;
u_long *addr*;

Arguments

vhp	Specifies a pointer to the `vba_hd` structure. When you write a driver routine that calls `vba_get_vmeaddr`, you declare a pointer to a `uba_ctlr` or a `uba_device` structure. You then pass the `um_vbahd` member of the `uba_ctlr` structure or the `ui_vbahd` member of the `uba_device` structure as the first argument to this routine.
addr	Specifies the System Virtual Address (SVA) for the device. This address is one of the two *addr* arguments that are passed to the driver's `probe` routine.

Description

The `vba_get_vmeaddr` routine obtains the VMEbus address corresponding to the SVA you passed in the *addr* argument. You typically call this routine to retrieve the VMEbus used in device-to-device DMA.

Return Value

This routine returns the VMEbus address corresponding to the SVA passed to this routine.

Name

vbarelse — releases the resources (map registers) used to map the specified VMEbus address

Syntax

vbarelse(*vhp*, *vme_addr*)
struct vba_hd * *vhp*;
int *vme_addr*;

Arguments

vhp	Specifies the `vba_hd` structure, which contains the VMEbus adapter number on which mapping registers were allocated in a previous call to `vballoc` or `vbasetup`.
vme_addr	Specifies the VMEbus address, which is the value returned in a previous call to `vballoc` or `vbasetup`.

Description

The `vbarelse` routine releases resources on the VMEbus adapter and then unblocks any process waiting for these resources.

Return Value

None.

See Also

`vballoc, vbasetup`

Name

vme_rmw — performs a read-modify-write

Syntax

int vme_rmw(*vhp*, *address_ptr*, *data*, *mask*)
struct vba_hd **vhp*;
u_int **address_ptr*;
u_int *data*;
u_int *mask*;

Arguments

vhp	Specifies a pointer to a `vba_hd` structure.
address_ptr	Specifies a pointer to the data that you want to modify.
data	Specifies the new data to be written.
mask	Specifies which bit or bits to check for locked data.

Description

The `vme_rmw` routine performs a read-modify-write to VMEbus memory using the VMEbus adapter. This routine is an interlock primitive that can be used by device driver writers to suit the needs of their individual drivers. This routine emulates a hardware read-modify-write cycle; therefore, you can use it to lock a portion of memory, read some specified data that resides in that portion of memory, and modify (write) that portion of memory with new data.

The *address_ptr* argument is the SVA of the VMEbus memory that a device driver writer wants to modify. This address will be based on the SVA passed to the `probe` routine. The *mask* argument specifies which bits to check in order to determine if the data is locked. If the mask indicates that the data is not locked, `vme_rmw` writes the new data specified by the *data* argument to this memory location and returns a success value of zero (0). As the write occurs, the VMEbus adapter emulates a read-modify-write to the VMEbus memory, while `vme_rmw` blocks interrupts, thus ensuring that no other process can access the locked data. If the mask indicates that the data is locked, `vme_rmw` keeps the existing data in the memory location and returns a failure value of –1.

There are numerous strategies for a device driver to control the modifying of data. For example, a device driver could provide an `ioctl` routine that uses `vme_rmw` to perform the transfer of new data to VME memory in blocks of 1024 words.

Return Value

This routine returns a value of zero (0) upon successfully completing the read-modify-write. It returns –1 upon failure due to the fact that the data was locked.

Name

vslock — locks a virtual segment

Syntax

int vslock(*user_addr*, *nbytes*)
caddr_t *user_addr*;
int *nbytes*;

Arguments

user_addr	Specifies the virtual address of the segment to lock. You must supply this value to `vsunlock` when the segment is unlocked.
nbytes	Specifies the size of the virtual segment in bytes. You must supply this value to `vsunlock` when the segment is unlocked.

Description

The `vslock` routine locks a virtual segment. A virtual segment is a representation of a portion of the address space of the current process. You must lock this address space by calling `vslock` prior to utilizing this segment as a source or target of a DMA I/O operation. This guarantees the physical presence of the segment and its contents within memory during satisfaction of the I/O request. Virtual segments locked by `vslock` must be unlocked by `vsunlock` within the context of the same process following completion of the I/O request.

Prior to invocation of this routine, the device driver must verify that the virtual segment:

- Is within the current process's virtual address space
- Is accessible by the current process in the required fashion

You should use the `useracc` routine to perform these checks. Unpredictable results may occur if these checks are not made. These results include (but are not limited to) data corruption and system panics.

A request to lock a virtual segment may not be immediately satisfied at the time it is made for the following reasons:

- Not all segment contents may be physically resident
- The segment or some portion of it may already be locked

Segment contents may not be memory resident for a variety of reasons. They may never have been resident or they may have been moved to secondary storage because the physical page frames they currently occupy are required for other purposes. Such movements occur for a given page frame only if its contents have been modified. Otherwise, its contents already exist on secondary storage and the associated page frame can be immediately reused if needed.

Many instances exist where the physical page frames occupied by the segment or some subset of them are already locked. One example of such a situation occurs when multiple processes attempt to simultaneously lock virtual segments corresponding to the same shared memory region. Only the first process will be able to obtain the lock. Another example occurs when one process manipulates two virtual segments possessing a common page frame and the frame spans the boundary between the two segments. Locking one segment prevents locking of the other. This situation can be avoided by aligning nonoverlapping virtual segments on cluster boundaries. For this reason, such alignment is highly recommended.

When a request to lock a virtual segment cannot be immediately satisfied, the requesting process is put to sleep. This allows other processes access to the CPU. The sleeping process is not allowed to receive signals while it is sleeping. It is awakened each time the contents of one of the virtual segment's physical page frames is paged in. Awakening also takes place each time a lock on one of the virtual segment's physical page frames is released. This sleep/wakeup cycle continues until such time as all requirements for locking the virtual segment have been satisfied by the current process.

Return Value

None.

See Also

`useracc, vsunlock`

Name

vsunlock — unlocks a virtual segment

Syntax

#include <sys/buf.h>

int vsunlock(*user_addr*, *nbytes*, *rwflag*)
caddr_t *user_addr*;
int *nbytes*;
int *rwflag*;

Arguments

addr	Specifies the virtual address of the segment to unlock. You previously specified this value to `vslock` when the segment was originally locked.
nbytes	Specifies the size of the virtual segment in bytes. You previously specified this value to `vslock` when the segment was originally locked.
rwflag	Specifies the read/write flag. Specifies the type of I/O the virtual segment was subject to while locked. You can set the read/write flag, *rwflag*, to one of the following:

Value	**Meaning**
`B_READ`	The segment was modified. You should specify `B_READ` whenever a virtual segment was both accessed and modified (that is, reading from a device writes memory) while locked.
`B_WRITE`	The segment was accessed only.

Description

The `vsunlock` routine unlocks a virtual segment. A virtual segment is a representation of a portion of the address space of the current process. You must lock this address space by calling `vslock` prior to utilizing this segment as a source or target of a DMA I/O operation. This guarantees the physical presence of the segment and its contents within memory during satisfaction of the I/O request.

Unlocking a virtual segment invalidates the guarantee of physical presence. It allows the contents of the segment to be moved to secondary storage whenever the physical page frames they currently occupy are required for other purposes. Note that segment contents need be saved only if they were modified while the segment was locked. Otherwise, they already exist on secondary storage and the associated page frames can be immediately reused. You must set the *rwflag* argument to `B_READ` in the call to `vslock` to indicate modification of the virtual segment and to direct saving of virtual segment contents as needed.

Unlocking a virtual segment wakes up all processes sleeping on any of the physical page frames currently assigned to the segment. One example of such a situation occurs when multiple processes attempt to simultaneously lock virtual segments corresponding to the same shared memory region. The first process to make the attempt obtains the lock. The second process must wait until such time as the first process releases its lock. In the interim, it is put to sleep.

The same situation can also develop with just one process manipulating two virtual segments. However, the two segments must possess one page frame in common: the one which spans the boundary between the two segments. Aligning nonoverlapping virtual segments on cluster boundaries prevents this arrangement from existing and, for this reason, is highly recommended.

Attempting to unlock a virtual segment not locked by `vslock` results in the following system panic:

```
MUNLOCK: dup page unlock
```

Virtual segments must always be unlocked within the context of the process in which they were originally locked.

Return Value

None.

See Also

`vslock`

Name

vtokpfnum — obtains the page frame number

Syntax

#include <sys/vm.h>

u_int vtokpfnum(*kern_addr*)
caddr_t *kern_addr*;

Arguments

kern_addr	Specifies the kernel virtual address whose page frame number is to be returned.

Description

The `vtokpfnum` routine obtains the page frame number for the page in the character device's memory that was mapped at the kernel virtual address (*kern_addr*).

Return Value

The `vtokpfnum` routine always returns the page frame number. If *kern_addr* is not a kernel virtual address or if the kernel virtual address is not valid, the page frame number returned by `vtokpfnum` is undefined. There is no error return.

Name

wakeup — wakes up all processes sleeping on a specified address

Syntax

int wakeup(*channel*)
caddr_t *channel*;

Arguments

channel	Specifies the address on which the wakeup is to be issued.

Description

The `wakeup` routine wakes up all processes sleeping on a specified address. The routine wakes these processes on the address specified by the *channel* argument. All processes sleeping on this address are awakened and made scheduable according to the priorities they specified when they went to sleep. It is possible for no processes to be sleeping on the channel at the time the wakeup is issued. This may occur for a variety of reasons and does not represent an error condition.

The sleep and wakeup pair of routines block and unblock a process. Generally, a device driver issues these routines on behalf of a process requesting I/O while a transfer is in progress. That is, a process requesting I/O is put to sleep on an address associated with the request by the appropriate device driver routine. When the transfer asynchronously completes, the device driver interrupt service routine issues a wakeup on the address associated with the completed request. This makes the relevant process scheduable. The process resumes execution within the relevant device driver routine at the point immediately following the request to sleep. The driver on behalf of the process can then determine whether the condition for which it was sleeping, in this example completion of an I/O request, has been removed. If so, it can continue on to complete the I/O request. Otherwise, the appropriate driver routine can decide to put the process back to sleep to await removal of the indicated condition.

The ULTRIX kernel issues a wakeup on the global variable `lbolt` each second. This provides device drivers with a convenient method for waiting on behalf of a process for the occurrence of a specific event. They need only sleep on `lbolt`, releasing the CPU for use by other processes. After one second or so, the kernel timer maintenance routines issue the `lbolt` wakeup, making the relevant process scheduable. In due time, the process resumes execution within the relevant device driver routine at the point immediately following the request to sleep on `lbolt`. The driver on behalf of the process can then determine whether the specific event occurred. If not, this procedure can be repeated as many times as is necessary until the desired event takes place.

Return Value

None.

See Also

`sleep`, `lbolt`

Name

wbflush — ensures a write to I/O space has completed

Syntax

wbflush()

Arguments

None.

Description

The `wbflush` routine ensures a write to I/O space has completed. Whenever a device driver writes to I/O space, the write may be intermittently delayed through the imposition of a hardware-dependent system write buffer. Subsequent reads of that location will not wait for a delayed write to complete. Either the original or the new value may be obtained. Subsequent writes of that location may replace the previous value, either in I/O space or in the system write buffer, if its writing had been delayed. In this case, the previous value would never have been actually written to I/O space.

Whether a given write to I/O space is delayed and how long this period is depends upon the existence of a system write buffer, its size, and its content. In general, delayed writes are not a problem. Device drivers need not call `wbflush` except in the following special situations:

- The write causes a state change in the device, and the change is indicated by a subsequent device-induced change in the value of the location being written by the device driver. This situation normally exists only during initialization of certain devices.
- The value being written is permanently consumed by the act of writing it. This situation exists only for certain specific devices, including some terminal devices.

Return Value

None.

B.2 ioctl commands

Table B-2 summarizes the special file discussed in this appendix. Following the table is a description of the special file.

Table B-2: Summary Description for Special Files

Special File	Summary Description
`DEVIOCGET`	obtains information about a device

Name

DEVIOCGET — obtains information about a device

Syntax

```
#include <sys/devio.h>
#include <sysioctl.h>
```

Description

The `DEVIOCGET` ioctl request obtains information about a device. This request obtains generic device information by polling the underlying device driver. `DEVIOCGET` uses the following structure defined in `/usr/sys/h/devio.h`:

```
struct devget {
        short  category;
        short  bus;
        char   interface[DEV_SIZE];
        char   adpt_num;
        short  nexus_num;
        short  bus_num;
        short  ctlr_num;
        short  rctlr_num;
        short  slave_num;
        char   dev_name[DEV_SIZE];
        short  unit_num;
        unsigned soft_count;
        unsigned hard_count;
        long   stat;
        long   category_stat; };
```

The following describes the meaning of the members of this structure:

`category`	Specifies the general class of the device. This member can be set to one of these values: `DEV_TAPE` (tape category), `DEV_DISK` (disk category), `DEV_TERMINAL` (terminal category), `DEV_PRINTER` (printer category), or `DEV_SPECIAL` (special category).
`bus`	Specifies the communications bus type. For example, for XMI devices this member would be set to the value `DEV_XMI`.
`interface`	Specifies a string of up to eight characters that identifies the controller interface type.
`adpt_num`	This member is set to the bus adapter number.
`nexus_num`	This member is set to the particular node or nexus number the device represents. This node or nexus number is the specific node on this adapter.
`bus_num`	This member is set to the bus number that the device controller resides on.
`ctlr_num`	This member is set to the specific controller number for the controller of this device. This number is the specific controller number on this bus.

`rctlr_num`	This member is set to the remote controller number.
`slave_num`	This member is set to the device unit number. For a disk device, this unit number is the physical device unit number. For a terminal device, this number is the terminal line number.
`dev_name`	This member is set to the device name type, which is a string of up to eight characters. Usually this device name type is the name as it appears in device autoconfiguration messages.
`unit_num`	This member is set to the kernel configuration representation of a device unit number. The value in this member is frequently the same as the `slave_num` member. The difference is that `slave_num` represents the physical unit number, while the `unit_num` member represesents a logical unit number representation for the device.
`soft_count`	This member is set to a driver counter of soft (noncritical) errors.
`hard_count`	This member is set to a driver counter of hardware errors.
`stat`	This member is set to the device status. This member is used primarily to represent drive status for tape devices. Some examples of drive status include: the drive is at the bottom of tape, or the drive is write protected.
`category_stat`	This member is set to generic device status values, which are defined in `/usr/sys/h/devio.h`. The values are organized according to these device types: tapes, disks, and communications devices.

The following example prints out the device type and unit number:

```
  .
  .
  .
struct devget dev_st; 1
if (ioctl (fd, DEVIOCGET, &dev_st) < 0) {
          printf ("DEVIOCGET failed\n");
          exit(1);
} 2
printf ("Device type = %s\n",dev_st.device); 3
printf ("Unit number = %d\n",dev_st.unit_num); 4
```

The following numbered items explain the preceding example:

1 This line declares a structure of type `devget`.

2 This line is a test to determine whether the call to ioctl succeeds or fails. Note that fd is an open file descriptor for the associated device special file.

3 This line obtains the device number.

4 This line obtains the unit number.

B.3 Global Variables Used by Device Drivers

Table B-3 summarizes the global variables used by device drivers. Following the table are descriptions of each global variable, presented in alphabetical order.

Table B-3: Summary Description for Global Variables

Global Variable	Summary Description
`cpu`	provides a unique logical family identifier of the processor type of the running system
`hz`	variable to store number of clock ticks per second
`lbolt`	periodic wakeup mechanism

Name

cpu — provides a unique logical family identifier of the processor type of the running system

Description

The `cpu` global variable provides a unique logical family identifier of the processor type of the running system. The logical system name may represent a single processor or a family of processor types. For example, the constant `DS_5000` represents the DECstation 5000 Model 200. The defined system names appear in the file `/usr/sys/machine/common/cpuconf.h`.

This global variable is used to conditionally execute processor-specific code. For example, the following code fragment calls a system-specific initialization routine for the DECstation 5000 Model 200 processor:

```
  .
  .
  .
if (cpu == DS_5000 {
        init_5000
}
  .
  .
  .
```

Name

hz — variable to store number of clock ticks per second

Description

The `hz` global variable is set to the number of clock ticks per second. The value is useful for timing purposes. For example, if a device driver wants to schedule a routine to be run in two seconds, the following call could be used:

```
  .
  .
  .
timeout(lptout, (caddr_t)dev, 2*hz);
  .
  .
  .
```

Name

lbolt — periodic wakeup mechanism

Description

The `lbolt` global variable is used as a periodic wakeup mechanism. Wakeups are done on the `lbolt` variable once per second. For example, if a driver was polling for an event once per second, the following code could be used:

```
  .
  .
  .
sleep((caddr_t)&lbolt, DNPRI);
  .
  .
  .
```

Summary of Device Driver Routines C

Table C-1 summarizes the routines used by VMEbus and TURBOchannel device drivers. The table has the following columns:

- Routine

  This column lists the driver routine name.

- Structure/file

  This column lists the structure (or file) where you define the driver routine entry point.

- Character

  An X appears in this column if the routine is applicable to a character device. Otherwise, an N/A (not applicable) appears.

- Block

  An X appears in this column if the routine is applicable to a block device. Otherwise, an N/A (not applicable) appears.

For convenience, the routines appear in alphabetical order.

Note

> The `psize` routine is no longer used. It has been superseded by driver ioctl calls that are used to obtain disk geometry information. Previously, the routine determined the location on the disk where ULTRIX should perform a dump.
>
> ULTRIX supports dumping only to disks that it can boot from. In most cases, ULTRIX uses dump routines located in the console subsystem. Because ULTRIX does not support booting from a VMEbus disk, dumping to disk is not used in a VMEbus device.

Table C-1: Summary of Device Driver Routines

Routine	Structure/File	Character	Block
`attach`	`uba_driver`	X	X
`close`	`cdevsw` `bdevsw`	X	X
`interrupt`	system configuration file	X	X
`ioctl`	`cdevsw` `bdevsw`	X	X

Table C-1: (continued)

Routine	Structure/File	Character	Block
`mmap`	`cdevsw`	X	N/A
`open`	`cdevsw` `bdevsw`	X	X
`probe`	`uba_driver`	X	X
`read`	`cdevsw`	X	N/A
`reset`	`cdevsw`	X	N/A
`select`	`cdevsw`	X	N/A
`slave`	`uba_driver`	X	X
`stop`	`cdevsw`	X	N/A
`strategy`	`cdevsw` `bdevsw`	X	X
`write`	`cdevsw`	X	N/A

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-baud modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	**Call**	**Contact**
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital Subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal*	————	SSB Order Processing - WMO/E15 *or* Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

---

* For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

ULTRIX
Guide to Writing and Porting VMEbus
and TURBOchannel Device Drivers
AA-PGL5A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? ______________________________

____

____

What do you like best about this manual? ______________________________

____

____

What do you like least about this manual? ______________________________

____

____

Please list errors you have found in this manual:

Page	Description
________	______________________________
________	______________________________
________	______________________________
________	______________________________
________	______________________________

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? ____________

Name/Title ______________________________ Dept. ____________
Company ______________________________ Date ____________
Mailing Address ______________________________
______________________________ Email ____________________ Phone ____________

------- Do Not Tear – Fold Here and Tape --

digital™

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-2698

------- Do Not Tear – Fold Here --

Cut
Along
Dotte
Line