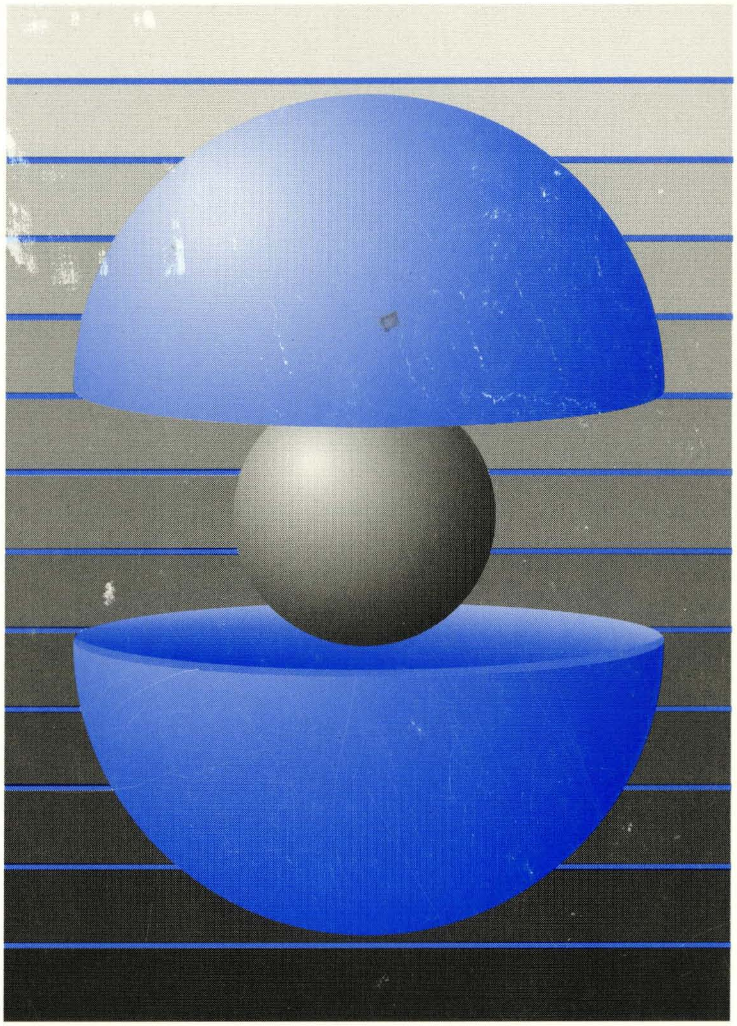
DEC OSF/1

digital

Writing Device Drivers
Volume 1: Tutorial



Part Number: AA-PUBVB-TE

DEC OSF/1

Writing Device Drivers, Volume 1: Tutorial

Order Number: AA-PUBVB-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This guide contains information needed by systems engineers who write device drivers for hardware that runs the DEC OSF/1 operating system. Included is information on driver concepts, device driver interfaces, kernel interfaces used by device drivers, kernel data structures, configuration of device drivers, and header files related to device drivers.

digital equipment corporation
Maynard, Massachusetts

Contents

About This Book

2 Developing a Device Driver

Tables

About This Book

This book discusses how to write device drivers for computer systems running the DEC OSF/1 operating system.

Audience

This book is intended for systems engineers who:

- Use standard library interfaces to develop programs in the C language
- Know the Bourne or some other UNIX-based shell
- Understand basic DEC OSF/1 concepts such as kernel, shell, process, configuration, and autoconfiguration
- Understand how to use the DEC OSF/1 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 the CPU hardware architecture, including interrupts, direct memory access (DMA) operations, and I/O

Although the book assumes a strong background in UNIX-based operating systems and C programming, it does not assume any background in device drivers. In addition, the book assumes that the audience has no source code licenses.

A secondary audience are systems engineers who need to implement a new bus or make changes to the implementation of an existing bus. Topics of interest to this audience include descriptions of the `bus` structure members.

Scope of the Book

The book is directed towards DEC OSF/1 on computer systems developed by Digital Equipment Corporation. However, the book provides information on designing drivers, on OSF-based data structures, and OSF-based kernel interfaces that would be useful to any systems engineer interested in writing UNIX-based device drivers.

The book presents a variety of examples including:

- A simple device driver that introduces the driver development process
- A character device driver operating on a TURBOchannel bus that illustrates a minimal implementation of all the TURBOchannel hardware functions

The book does not emphasize any specific types of device drivers. However, mastering the concepts and examples presented in this book would be very useful preparation for writing a variety of device drivers, including drivers for disk and tape controllers as well as more specialized drivers such as array processors.

Organization

Part 1 Overview

Part 1 contains one chapter, whose goal is to provide you with an overview of device drivers. The chapter is:

Chapter 1	Introduction to Device Drivers Provides an overview of device drivers. Read this chapter to obtain introductory information on device drivers and to understand the place of a device driver in DEC OSF/1.

Part 2 Anatomy of a Device Driver

Part 2 contains three chapters, whose combined goal is to provide enough information to allow you to write a simple DEC OSF/1 device driver. The chapters are:

Chapter 2	Developing a Device Driver Describes how to design a device driver. Read this chapter if you are not familiar with the driver development process on DEC OSF/1. Even if you have written UNIX device drivers, you may want to read the sections that describe the design issues related to loadable drivers, to CPU architectures, and to porting drivers from ULTRIX to DEC OSF/1.
Chapter 3	Analyzing the Structure of a Device Driver Analyzes the sections that make up character and block device drivers. Read this chapter if you are not familiar with the sections that make up character and block drivers on DEC OSF/1. If you are experienced writing UNIX

	drivers, you may want to read selected sections, particularly the section that describes how to set up a configure interface for loadable device drivers.
Chapter 4	Coding, Configuring, and Testing a Device Driver Using a simple example, describes how to code, configure, and test a device driver. Read this chapter if you have never written a UNIX-based driver before. If you have written a UNIX-based driver, you may want to read only selected sections.

Part 3 Hardware Environment

Part 3 contains two chapters, whose combined goal is to provide you with a view into the device drivers' hardware environment. The chapters are:

Chapter 5	Hardware-Independent Model and Device Drivers Provides an overview of the hardware-independent model and how it relates to device drivers. Read this chapter to gain an understanding of how device drivers fit into the hardware-independent model.
Chapter 6	Hardware Components and Hardware Activities Describes the hardware components and activities related to device drivers. Read this chapter to obtain an understanding or to refresh your knowledge about the individual hardware components you will work with when writing your drivers.

Part 4 Kernel Environment

Part 4 contains three chapters, whose combined goal is to provide information about the kernel environment. The chapters are:

Chapter 7	Device Autoconfiguration Discusses the events that occur during the autoconfiguration of devices, with an emphasis on how autoconfiguration relates to static and loadable device drivers. In addition, the chapter provides detailed information on the data structures related to autoconfiguration. Read this chapter if you are not familiar with autoconfiguration on DEC OSF/1. Those of you experienced in writing UNIX device drivers may want to read selected sections, especially the sections that discuss data structure members that you are not familiar with.

Chapter 8	Data Structures Used in I/O Operations Describes members of the structures used in input/output (I/O). Read this chapter if you are not familiar with the I/O-related data structures. If you are experienced with other UNIX I/O subsystems, you may want to read selected sections, especially those sections that will refresh your memory about the I/O data structures.
Chapter 9	Using Kernel Interfaces with Device Drivers Discusses the kernel interfaces most commonly used by device drivers, including those interfaces used to move data and allocate and free memory. Read this chapter if you need examples of when, how, and why you would use these kernel interfaces in device drivers.

Part 5 Device Driver Example

Part 5 contains one chapter, whose goal is to offer a more complex and challenging device driver for you to analyze. The chapter is:

Chapter 10	Writing a Character Device Driver Describes how to code a character device driver for a real device that operates on a TURBOchannel bus. Read this chapter if you want source code examples that exemplify a driver implementation for a real device. Those of you experienced in writing TURBOchannel device drivers may want to study only selected sections of the code.

Part 6 Device Driver Configuration

Part 6 contains three chapters, whose combined goal is to provide enough information to allow you to choose the driver configuration procedure most suitable for your development environment. The chapters are:

Chapter 11	Device Driver Configuration Models Provides an overview of the two configuration models: the third-party device driver configuration model and the traditional device driver configuration model. Read this chapter to obtain a general overview of the two models. The third-party driver configuration model is particularly well suited for driver developers who want to deliver DEC OSF/1 device drivers to their customers.
Chapter 12	Device Driver Configuration Syntaxes and Mechanisms Describes the syntaxes and mechanisms used to populate the files needed for device driver configuration. Read this

	chapter if you are not familiar with these syntaxes and mechanisms.
Chapter 13	Device Driver Configuration Examples
	Provides step-by-step instructions for configuring the example device drivers, using the third-party and traditional models. Read this chapter to learn how to configure the example drivers using the third-party and the traditional driver configuration models.

Part 7 Appendixes

Part 7 contains three appendixes and a glossary.

Appendix A	Summary Tables
	Presents tables that summarize the header files, kernel interfaces, data structures, and other interfaces used by device drivers.
Appendix B	Device Driver Example Source Listings
	Contains the source code listings for the examples presented in this book.
Appendix C	Device Driver Development Worksheets
	Provides worksheets for use in designing and coding a device driver.
Glossary	Glossary

Related Documentation

The printed version of the DEC OSF/1 documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

Audience	Icon	Color Code
General Users	G	Teal
System Administrators	S	Red
Network Administrators	N	Yellow
Programmers	P	Blue
Reference Page Users	R	Black

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the books in the DEC OSF/1 documentation set.

Writing device drivers is a complex task; driver writers require knowledge in a variety of areas. One way to acquire this knowledge is to have at least the following categories of documentation available:

- Hardware documentation
- Bus-specific device driver documentation
- Programming tools documentation
- System management documentation
- Porting documentation
- Reference pages

The following sections list the documentation associated with each of these categories.

Hardware Documentation

You should have available the hardware manual associated with the device for which you are writing the device driver. Also, you should have access to the manual that describes the architecture associated with the CPU that the driver operates on, for example, the *Alpha Architecture Reference Manual.*

Bus-Specific Device Driver Documentation

Writing Device Drivers, Volume 1: Tutorial is the core book for developing device drivers on DEC OSF/1. It contains information needed for developing drivers on any bus that operates on Digital platforms. *Writing Device Drivers, Volume 2: Reference* is a companion volume to the tutorial and describes, in reference (man) page style, the header files, kernel interfaces, data structures, and other interfaces used by device drivers. The following books provide information about writing device drivers for a specific bus that is beyond the scope of the core tutorial and reference:

- *Writing EISA Bus Device Drivers*

  This manual provides information for systems engineers who write device drivers for the EISA bus. The manual describes EISA bus-specific topics including EISA bus architecture and kernel interfaces used by EISA bus drivers.

- *Writing Device Drivers for the SCSI/CAM Architecture Interfaces*

  This manual provides information for systems engineers who write device drivers for the SCSI/CAM Architecture interfaces.

  The manual provides an overview of the DEC OSF/1 SCSI/CAM Architecture and describes User Agent routines, data structures, common and generic routines and macros, error handling and debugging routines. The manual includes information on configuration and installation. Examples show how programmers can define SCSI/CAM device drivers and write to the SCSI/CAM special I/O interface supplied by Digital to process special SCSI I/O commands.

  The manual also describes the SCSI/CAM Utility (SCU) used for maintenance and diagnostics of SCSI peripheral devices and the CAM subsystem.

- *Writing TURBOchannel Device Drivers*

  This manual provides information for systems engineers who write device drivers for the TURBOchannel. The manual describes TURBOchannel-specific topics, including TURBOchannel architecture and kernel interfaces used by TURBOchannel drivers.

Programming Tools Documentation

To create your device drivers, you use a number of programming development tools and should have on hand the manuals that describe how to use these tools. The following manuals provide information related to programming tools used in the DEC OSF/1 operating system environment:

- *Kernel Debugging*

  This manual provides information on debugging a kernel and analyzing a crash dump of a DEC OSF/1 operating system. The manual provides an overview of kernel debugging and crash dump analysis and describes the tools used to perform these tasks. The manual includes examples with commentary that show how to analyze a running kernel or crash dump. The manual also describes how to write a `kdbx` utility extension and how to use the various utilities for exercising disk, tape, memory, and communications devices.

  This manual is for system administrators responsible for managing the operating system and for systems programmers writing applications and device drivers for the operating system.

- *Programming Support Tools*

  This manual describes several commands and utilities in the DEC OSF/1 system, including facilities for text manipulation, macro and program generation, source file management, and software kit installation and creation.

The commands and utilities described in this manual are intended primarily for programmers, but some of them (such as `grep, awk, sed,` and the Source Code Control System (SCCS)) are useful for other users. This manual assumes that you are a moderately experienced user of UNIX systems.

- *Programmer's Guide*

  This manual describes the programming environment of the DEC OSF/1 operating system, with an emphasis on the C programming language.

  This manual is for all programmers who use the DEC OSF/1 operating system to create or maintain programs in any supported language.

System Management Documentation

Refer to the *System Administration* book for information about building a kernel and for general information on system administration. This manual describes how to configure, use, and maintain the DEC OSF/1 operating system. It includes information on general day-to-day activities and tasks, changing your system configuration, and locating and eliminating sources of trouble.

This manual is for the system administrators responsible for managing the operating system. It assumes a knowledge of operating system concepts, commands, and configurations.

Porting Documentation

Refer to the *DEC OSF/1 Migration Guide* for a discussion of the differences between the DEC OSF/1 and ULTRIX operating systems. This manual compares the DEC OSF/1 operating system to the ULTRIX operating system by describing the differences between the two systems.

This manual has three audiences, as follows:

- General users can read this manual to determine what differences exist between using an ULTRIX system and using the DEC OSF/1 system.
- System and network administrators can read this manual to determine what differences exist between ULTRIX and DEC OSF/1 system administration.
- Programmers can read this manual to determine differences in the DEC OSF/1 programming environment and the ULTRIX programming environment.

This manual assumes you are familiar with the ULTRIX operating system.

Reference Pages

The following provide reference (man) pages that are of interest to device driver writers:

- *Reference Pages Section 2*

  This section defines system calls (entries into the DEC OSF/1 kernel) that programmers use. The introduction to Section 2, `intro`(2), lists error numbers with brief descriptions of their meanings. The introduction also defines many of the terms used in this section. This section is for programmers.

- *Reference Pages Section 3*

  This section describes the routines available in DEC OSF/1 programming libraries, including the C library, Motif library, and X library. This section is for programmers. In printed format, this section is divided into volumes.

- *Reference Pages Sections 4, 5, and 7*

  - Section 4 describes the format of system files and how the files are used. The files described include assembler and link editor output, system accounting, and file system formats. This section is for programmers and system administrators.
  - Section 5 contains miscellaneous information, including ASCII character codes, mail-addressing formats, text-formatting macros, and a description of the root file system. This section is for programmers and system administrators.
  - Section 7 describes special files, related device driver functions, databases, and network support. This section is for programmers and system administrators.

- *Reference Pages Section 8*

  This section describes commands for system operation and maintenance. It is for system administrators.

Reader's Comments

Digital welcomes your comments on this or any other DEC OSF/1 manual. You can send your comments in the following ways:

- Internet electronic mail:
  `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of

each printed DEC OSF/1 manual.

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

Conventions

The following conventions are used in this book:

. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
filename	In examples, syntax descriptions, and function definitions, this typeface indicates variable values.
buf	In function definitions and syntax definitions used in driver configuration, this typeface is used to indicate names that you must type exactly as shown.
[ ]	In formal parameter declarations in function definitions and in structure declarations, brackets indicate arrays. Brackets are also used to specify ranges for device minor numbers and device special files in `stanza.loadable` file fragments. However, for the syntax definitions used in driver configuration, these brackets indicate items that are optional.
|	Vertical bars separating items that appear in the syntax definitions used in driver configuration indicate that you choose one item from among those listed.

This book uses the word kernel ‘‘interface’’ instead of kernel ‘‘routine’’ or kernel ‘‘macro’’ because, from the driver writer’s point of view, it does not matter whether the interface is a routine or a macro.

Summary of Changes and Additions

The following sections summarize the changes and additions made to each chapter for this version of the book.

Chapter 2: Developing a Device Driver

The following list summarizes the changes and additions made to each section in this chapter:

- Section 2.1.5: Providing a Description of the Device Registers

  This section recommends that device drivers use two new interfaces to access device registers: `read_io_port` and `write_io_port`.
- Section 2.4.1: Control Status Register Issues

  This section recommends that device drivers use the CSR I/O access interfaces to read from and write to device registers.
- Section 2.4.2: Input/Output Copy Operation Issues

  This section recommends that device drivers use the I/O copy interfaces to perform I/O copy operations.
- Section 2.4.3: Direct Memory Access Operation Issues

  This section recommends that device drivers use the DMA interfaces to perform direct memory access operations.
- Section 2.4.7.1: Forcing a Barrier Between Load/Store Operations

  This section uses the new interfaces `read_io_port` and `write_io_port` in the code example.
- Section 2.4.7.2: After the CPU Has Prepared a Data Buffer in Memory

  This section uses the new interface `read_io_port` in the code example.
- Section 2.4.7.3: Before Attempting to Read Any Device CSRs

  This section uses the new interface `read_io_port` in the code example.
- Section 2.4.7.4: Between Writes

  This section uses the new interface `write_io_port` in the code example.
- Section 2.6.5: Checking Kernel Interfaces

  The table in this section provides technical corrections to the remarks column for the `useracc` interface.

Chapter 3: Analyzing the Structure of a Device Driver

The following list summarizes the changes and additions made to each section in this chapter:

- Section 3.1.2.3: The devdriver.h Header File

  This section mentions two new opaque data types: `io_handle_t` and `dma_handle_t`.
- Section 3.1.4: Device Register Header File

  This section describes and provides an example of another technique for defining the registers of a device.

Chapter 4: Coding, Configuring, and Testing a Device Driver

The following list summarizes the changes and additions made to each section in this chapter:

- Section 4.1.1: The nonereg.h Header File

  This section uses the new technique for defining the registers of the `none` device.

- Section 4.1.3: Autoconfiguration Support Declarations and Definitions Section

  This section contains a rewrite of code explanation items to reflect the use of the new technique for defining the registers of the `none` device.

- Section 4.1.6.1: Implementing the noneprobe Interface

  This section contains a rewrite of code explanation items to reflect the use of the I/O handle, `read_io_port`, and `write_io_port`.

- Section 4.1.8.2: Implementing the noneclose Interface

  This section contains a rewrite of code explanation items to reflect the use of the I/O handle and `write_io_port`.

Chapter 6: Hardware Components and Hardware Activities

Section 6.2.1, How a Device Driver Accesses Device Registers, contains a rewrite to correct technical inaccuracies.

Chapter 7: Device Autoconfiguration

Section 7.4.6, The slot Member, contains a rewrite to correct technical inaccuracies.

Chapter 9: Using Kernel Interfaces with Device Drivers

The following list summarizes the changes and additions made to each section in this chapter:

- Section 9.1.3: Copying a Null-Terminated Character String

  This section contains a rewrite of the code explanation item to make it more technically accurate.

- Section 9.5.2: Printing Text to the Console and Error Logger

  This section contains a rewrite of the code explanation item to make it more technically accurate.

- Section 9.5.3: Putting a Calling Process to Sleep

  This section contains a rewrite to correct technical inaccuracies.

- Section 9.7: Input/Output (I/O) Handle-Related Interfaces

  This section introduces the I/O handle and the CSR I/O access and I/O copy interface categories.

- Section 9.7.1.1: Reading Data from a Device Register

  This section describes the new CSR I/O access interface `read_io_port`.

- Section 9.7.1.2: Writing Data to a Device Register

  This section describes the new CSR I/O access interface `write_io_port`.

- Section 9.7.2.1: Copying a Block of Memory from I/O Address Space to System Memory

  This section describes the new I/O copy interface `io_copyin`.

- Section 9.7.2.2: Copying a Block of Byte-Contiguous System Memory to I/O Address Space

  This section describes the new I/O copy interface `io_copyout`.

- Section 9.7.2.3: Copying a Memory Block of I/O Address Space to Another Memory Block of I/O Address Space

  This section describes the new I/O copy interface `io_copyio`.

- Section 9.8: DMA-Related Interfaces

  This section introduces the DMA-related interfaces.

- Section 9.8.1: DMA Handle

  This section introduces the DMA handle concept.

- Section 9.8.2: The sg_entry Structure

  This section describes the data structure returned by some of the DMA-related interfaces. This new data structure is called `sg_entry`.

- Section 9.8.3: Allocating System Resources for DMA Data Transfers

  This section describes the new DMA interface `dma_map_alloc`.

- Section 9.8.4: Loading and Setting Allocated System Resources for DMA Data Transfers

  This section describes the new DMA interface `dma_map_load`.

- Section 9.8.5: Unloading System Resources for DMA Data Transfers

  This section describes the new DMA interface `dma_map_unload`.

- Section 9.8.6: Releasing and Deallocating Resources for DMA Data Transfers

  This section describes the new DMA interface `dma_map_dealloc`.
- Section 9.8.7: Returning a Pointer to the Current Bus Address/Byte Count Pair

  This section describes the new DMA interface `dma_get_curr_sgentry`.
- Section 9.8.8: Putting a New Bus Address/Byte Count Pair into the List

  This section describes the new DMA interface `dma_put_curr_sgentry`.
- Section 9.8.9: Returning a Kernel Segment Address of a DMA Buffer

  This section describes the new DMA interface `dma_kmap_buffer`.

Chapter 10: Writing a Character Device Driver

The following list summarizes the changes and additions made to each section in this chapter:

- Section 10.2: The cbreg.h Header File

  This section contains a rewrite of code explanation items to reflect the use of the new technique for defining the registers of the `CB` device.
- Section 10.3: Include Files Section

  This section adds a define for `DEV_FUNNEL_NULL`, which the `/dev/cb` device driver uses to initialize the `d_funnel` member of the `cdevsw` structure.
- Section 10.6: Local Structure and Variable Definitions Section

  This section removes the pointer to the `CB_REGISTERS` struture as a member of the `cb_unit` structure and replaces it with an I/O handle. The associated code explanation item reflects this change.
- Section 10.8.2: Implementing the cbattach Interface

  This section shows that the example driver uses the `io_handle_t` data type to perform a type cast operation with the `CB_ADR` macro. The associated code explanation item reflects this change.
- Section 10.11.1: Implementing the cbread Interface

  This section updates the example code and its associated code explanation item to reflect the use of the `read_io_port` interface.
- Section 10.11.2: Implementing the cbwrite Interface

  This section updates the example code and its associated code explanation item to reflect the use of the `write_io_port` interface.

- Section 10.12.2.3: Converting the Buffer Virtual Address

  This section updates the example code and its associated code explanation item to reflect the use of the `vtop` interface.

- Section 10.12.2.4: Converting the 32-Bit Physical Address

  This section updates the example code and its associated code explanation item to reflect the use of the `write_io_port` interface.

- Section 10.13: Start Section

  This section updates the example code and its associated code explanation items to reflect the use of the `read_io_port` and `write_io_port` interfaces.

- Section 10.14.4: Performing an Interrupt Test

  This section updates the example code and its associated code explanation items to reflect the use of the `read_io_port` and `write_io_port` interfaces.

- Section 10.14.5: Returning a ROM Word, Updating the CSR, and Stopping Increment of the Lights

  This section updates the example code and its associated code explanation items to reflect the use of the `read_io_port` and `write_io_port` interfaces.

Chapter 13: Device Driver Configuration Examples

Section 13.1.7, Providing the Contents of the Device Driver Kit, expands on the specific contents that driver writers provide to their kit developers. Six sections describe the following file fragments and files that become the contents of the device driver kit:

- `config.file` file fragment
- `files` file fragment
- `stanza.loadable` file fragment
- `stanza.static` file fragment
- Device driver objects
- Device driver load modules

Appendix A: Summary Tables

The following list summarizes the changes and additions made to each section in this chapter:

- Section A.1: List of Header Files

  The table lists a new header file `cpu.h`.

- Section A.2: List of Kernel Support Interfaces

  The table lists the following new kernel interfaces:

  - `dma_get_curr_sgentry`
  - `dma_get_next_sgentry`
  - `dma_get_private`
  - `dma_kmap_buffer`
  - `dma_map_alloc`
  - `dma_map_dealloc`
  - `dma_map_load`
  - `dma_map_unload`
  - `dma_min_boundary`
  - `dma_put_curr_sgentry`
  - `dma_put_prev_sgentry`
  - `dma_put_private`
  - `drvr_register_shutdown`
  - `io_copyin`
  - `io_copyio`
  - `io_copyout`
  - `io_zero`
  - `IS_KSEG_VA`
  - `KSEG_TO_PHYS`
  - `PHYS_TO_KSEG`
  - `read_io_port`
  - `vtop`
  - `write_io_port`

- Section A.3: List of Global Variables that Device Drivers Use

  The table lists a new global variable `page_size`.

- Section A.4: List of Data Structures

  The table lists a new data structure `sg_entry`.

Appendix B: Device Driver Example Source Listings

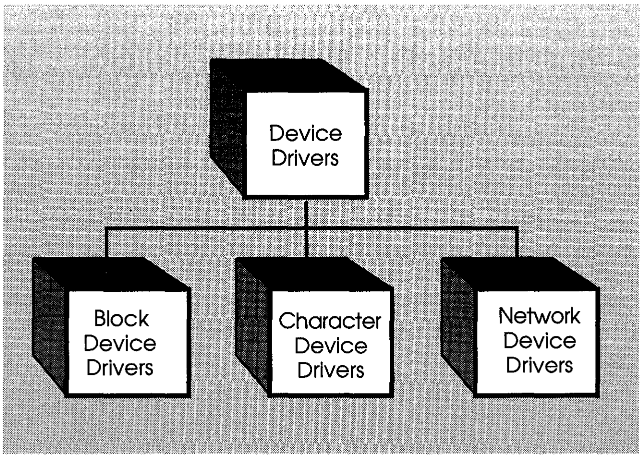
Section B.1, Source Listing for the /dev/none Device Driver, reflects changes made as a result of using the new interfaces `read_io_port` and `write_io_port`.

Glossary

The following terms now appear in the glossary:

- DMA handle
- I/O handle

Part 1 Overview



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
- Static versus loadable device drivers
- When a device driver is called
- The place of a device driver in DEC OSF/1

The chapter concludes with an example of how a device driver in DEC OSF/1 reads a single character.

1.1 Purpose of a Device Driver

The purpose of a device driver 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 easier.

1.2 Types of Device Drivers

A device driver is a software module that resides within the DEC OSF/1 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, or network controller device. In general, there is one device driver for each type of hardware device. Device drivers can be classified as:

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

The following sections briefly discuss each type.

1.2.1 Block Device Driver

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

Block device drivers are particularly well-suited for disk drives, the most common block devices. For block devices, all I/O occurs through the buffer cache.

1.2.2 Character Device Driver

A character device driver does not handle input and output through the buffer cache, so it is 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, character drivers are not limited to performing I/O one character at a time (despite the name ``character'' driver). For example, tape drivers frequently perform I/O in 10K chunks. You can also use a character device driver when it is necessary to copy data directly to or from a user process.

Because of their flexibility in handling I/O, many drivers are character drivers. 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. It 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 book does not discuss network device drivers.

1.2.4 Pseudodevice Driver

Not all device drivers control physical hardware. Such device drivers are called ``pseudodevice'' drivers. Like block and character device drivers, pseudodevice drivers make use of the device driver interfaces. Unlike block and character device drivers, pseudodevice drivers do not operate on a bus. One example of a pseudodevice driver is the pseudoterminal or `pty` terminal driver, which simulates a terminal device. The `pty` terminal driver is a character device driver typically used for remote logins.

1.3 Static Versus Loadable Device Drivers

Traditional kernels require that device drivers be installed by performing tasks that include rebuilding the kernel, shutting down the system, and rebooting. In these kinds of environments, device drivers can be viewed as static, that is, they are linked directly into the kernel at build time. Historically, this was necessary because many kernel interfaces consisted of static tables with no means of dynamic expansion. Thus, when changes are made to these device drivers, the only way to link them into the kernel is to go through the previously listed steps.

A design goal of OSF/1 was to provide cleanly architected kernel interfaces that would make it easier to add functionality to the kernel. To accomplish the task of allowing functional enhancements at kernel run time rather than at kernel build time, it was necessary for these kernel interfaces not to rely exclusively on statically configured tables. Thus, a set of kernel subsystems was defined.

A subsystem is a kernel module that defines a set of kernel framework interfaces that allow for the dynamic configuration and unconfiguration (adding and removal) of subsystem functionality. Examples of subsystems include (but are not restricted to) device drivers, file systems, and network protocols. The ability to dynamically add subsystem functionality is utilized by loadable drivers to allow the driver to be configured and unconfigured without the need for kernel rebuilds and reboots.

The DEC OSF/1 operating system embraces and builds on this portability and configurability philosophy by providing the ability for device drivers to be installed dynamically at run time without having to rebuild the kernel, shut down the system, and reboot. You must know the bus type and CPU architecture when deciding to make your device driver loadable. For DEC OSF/1, loadable drivers are not supported on the Alpha AXP architecture. However, you might want to follow the approach taken by the example drivers in this book and implement the loadable driver code now in anticipation of future support.

1.4 When a Device Driver Is Called

Figure 1-1 shows that the kernel calls a device driver 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 (I/O) 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 from devices capable of generating them.

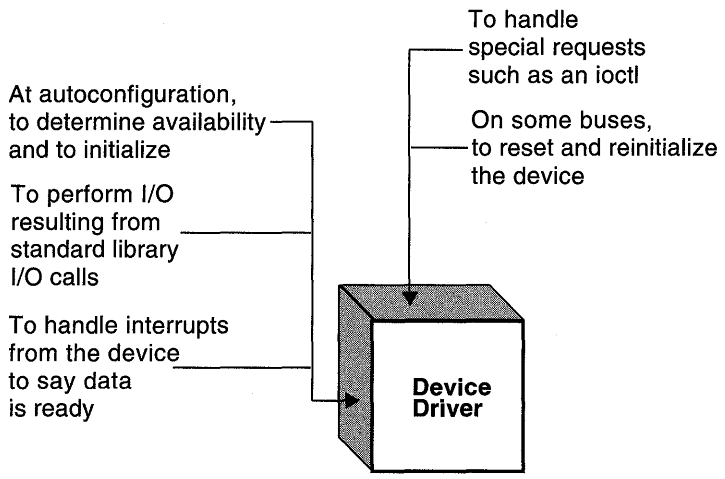
- Special requests

  The kernel calls a device driver to handle 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 1-1: When the Kernel Calls a Device Driver


To handle special requests such as an ioctl
On some buses, to reset and reinitialize the device
At autoconfiguration, to determine availability and to initialize
To perform I/O resulting from standard library I/O calls
To handle interrupts from the device to say data is ready
Device Driver

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 but from activities that occur at boot time.

1.4.1 Place of a Device Driver in DEC OSF/1

Figure 1-2 shows the place of a device driver in DEC OSF/1 relative to the device:

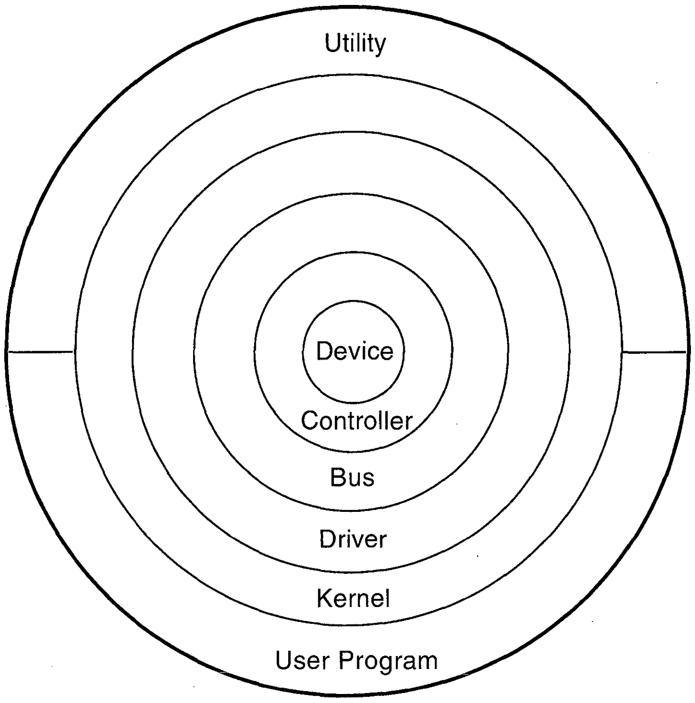
- User program or utility

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

- Kernel

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

Figure 1-2: Place of a Device Driver in DEC OSF/1



- Device driver

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

- 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. A controller connects to a bus.

- Peripheral device

  A peripheral device is a device that can be connected to a controller. Disk and tape drives are examples of peripheral devices that can be connected to the controller. Other devices (for example, the network) may be integral to the controller.

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

1.4.2 User Program or Utility

User programs, or utilities, 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` interface.

1.4.3 Kernel

The kernel makes requests to a device driver to perform operations on a particular device. Some of these requests result directly from user program requests. 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 result from activities performed by the kernel. At boot time, for example, the kernel calls the driver's `probe` interface.

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

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

1.4.4 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 controller interfaces. On DEC OSF/1, you can configure more device drivers than there are physical devices configured into the hardware system. At boot time, the autoconfiguration software 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. Similarly, when a driver is dynamically loaded, the kernel performs the configuration sequence for each instance of the physical device.

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. This is the case when the write request is segmented into several partial transfers at the driver level.

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

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

1.4.5 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 TURBOchannel and the VMEbus, for example, can be classified as having OPENbus architectures.

Device driver writers must understand the bus that the device is connected to. This book covers topics that all driver writers need to know regardless of the bus. However, the TURBOchannel is the bus used for describing the implementation of the `/dev/none` and `/dev/cb` example drivers.

1.4.6 Device Controller

A device controller is the hardware interface between the computer and a peripheral device. Sometimes a controller handles several devices. In other cases, a controller is integral to the device.

1.4.7 Peripheral Devices

A peripheral device is hardware, such as a disk controller, 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
- A disk or tape drive

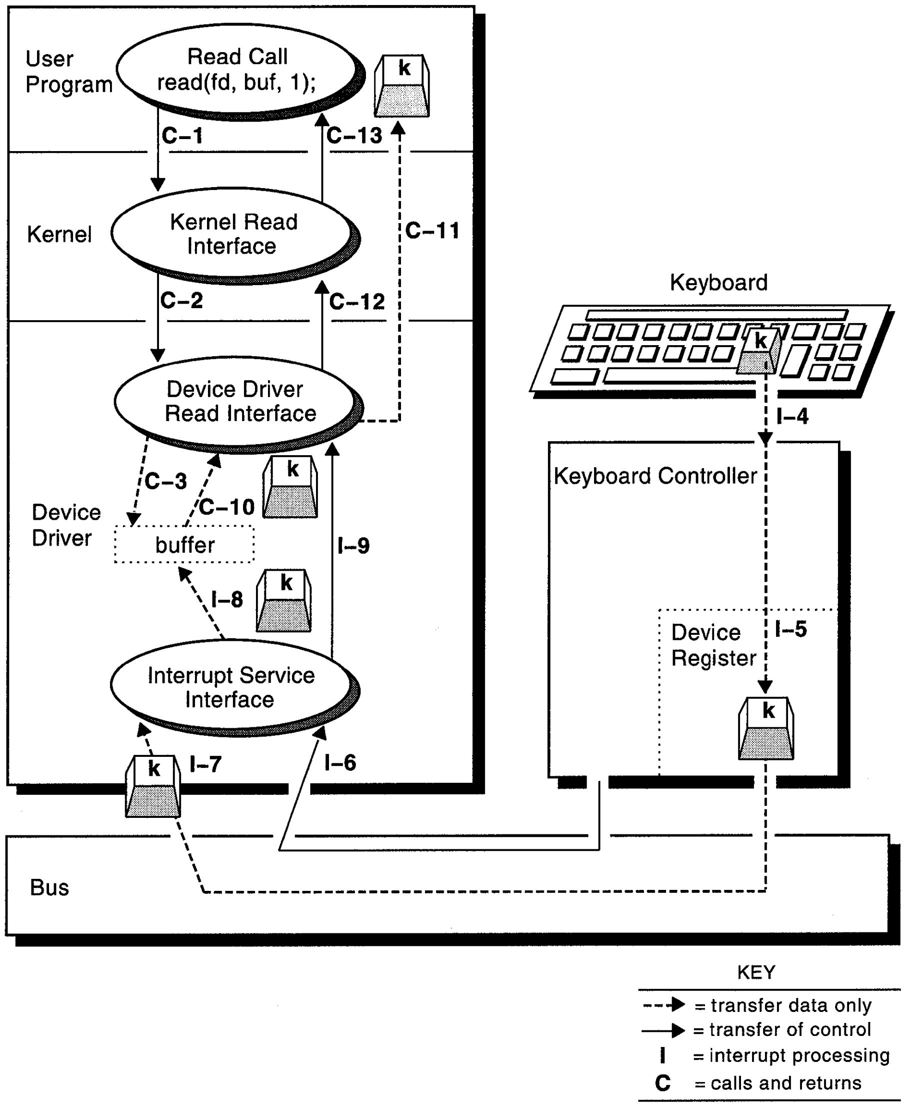
1.5 Example of Reading a Character

This section provides an example of how DEC OSF/1 processes a read request of a single character in raw mode from a terminal. (Raw mode returns single characters.) Although the example takes a simplified view of character processing, it does illustrate how control can pass from a user program to the kernel to the device driver. It also shows that interrupt processing occurs asynchronously from other device driver activity. Figure 1-3 summarizes the flow of control between a user program, the kernel, the device driver, and the hardware. The figure shows the following sequence of events:

- A read request is made to the device driver (C-1 to C-3).
- The character is captured by the hardware (I-4 and I-5).
- The interrupt is generated (I-6).
- The interrupt service interface handles the interrupt (I-7 to I-9).
- The character is returned (C-10 to C-13).

Figure 1-3 provides a snapshot of the processing that occurs in the reading of a single character. The following sections elaborate on this sequence.

Figure 1-3: Simple Character Driver Interrupt Example



1.5.1 A Read Request Is Made to the Device Driver

A user program issues a `read` system call (C-1). The figure shows that the `read` system call passes three arguments: a file descriptor (the `fd` argument), the character pointer to where the information is stored (the `buf` argument), and an integer (the value 1) that tells the driver's `read` interface how many bytes to read. The calling sequence is blocked inside the device driver's `read` interface because the buffer where the data is stored is empty, indicating that there are currently no characters available to satisfy the read. The kernel's `read` interface makes a request of the device driver's `read` interface to perform a read of the character based on the arguments passed by the `read` system call (C-2). Essentially, the driver `read` interface is waiting for a character to be typed at the terminal's keyboard. The currently blocked process that caused the kernel to call the driver's `read` interface is not running in the CPU (C-3).

1.5.2 The Character Is Captured by the Hardware

Later, a user types the letter ''k'' on the terminal keyboard (I-4). The letter is stored in the device's data register (I-5).

1.5.3 The Interrupt Is Generated

When the user types a key, the console keyboard controller alters some signals on the bus. This action notifies the CPU that something has changed inside the console keyboard controller. This condition causes the CPU to immediately start running the console keyboard controller's interrupt service interface (I-6). The state of the interrupted process (either some other process or the idle loop) is saved so that the process can be returned to its original state as though it were never interrupted in the first place.

1.5.4 The Interrupt Service Interface Handles the Interrupt

The console device driver's interrupt service interface first checks the state of the driver and notices that a pending read operation exists for the original process. The console device driver manipulates the controller hardware by way of the bus hardware in order to obtain the value of the character that was typed. This character value was stored somewhere inside the console controller's hardware (I-7). In this case, the value 107 (the ASCII representation for the ''k'' character) is stored. The interrupt service interface stores this character value into a buffer that is in a location known to the rest of the console driver interfaces (I-8). It then awakens the original, currently sleeping, process so that it is ready to run again (I-9). The interrupt service interface returns, in effect restoring the interrupted process (not the original process yet) so that it may continue where it left off.

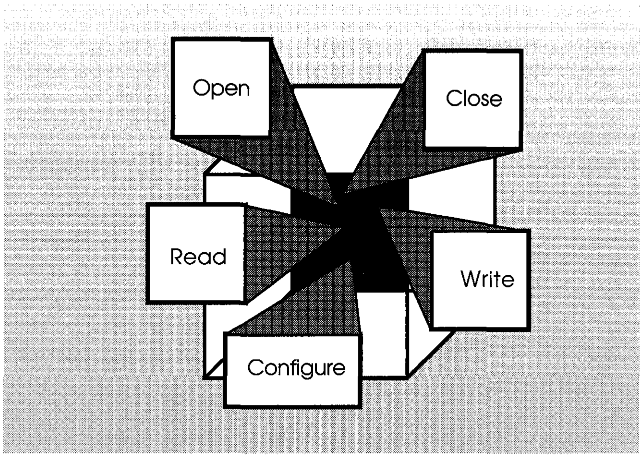
1.5.5 The Character Is Returned

Later, the kernel's process scheduler notices that the original process is ready to run, and so allows it to run. After the original process resumes running (after the location where it was first blocked), it knows which buffer to look at to obtain the typed character (C-10). It removes the character from this buffer and puts it into the user's address space (C-11). The device driver's `read` interface returns control to the kernel's `read` interface (C-12). The kernel `read` interface returns control to the user program that previously initiated the read request (C-13).

1.5.6 Summary of the Example

Although this example presents a somewhat simplified view of character processing, it does illustrate how control passes from a user program to the kernel to the device driver. It also shows clearly that interrupt processing occurs asynchronously from other device driver activity.

Part 2 Anatomy of a Device Driver



Developing a Device Driver 2

This chapter discusses how you develop a device driver. Included is a simple example called `/dev/none`, which is written by a fictitious third-party driver company called EasyDriver Incorporated. The chapter uses worksheets to illustrate the kinds of information you need to gather to make developing the device driver easier. Appendix C provides these same worksheets for use in developing your own device drivers. If you use the worksheets for your driver development, consider organizing them in a device driver project binder. This will make them available to systems engineers who need to maintain the driver.

Specifically, the chapter describes the following tasks associated with developing any device driver:

- Gathering information
- Designing the device driver
- Determining the structure allocation technique
- Understanding differences in CPU architectures
- Creating a device driver development environment

The chapter concludes with a section on porting device drivers. This task is only for device driver writers who want to port device drivers from ULTRIX to DEC OSF/1 systems.

2.1 Gathering Information

The first task in writing a device driver is to gather pertinent information about the host system and the device for which you are writing the driver. You need to:

- Specify information about the host system
- Identify the conventions used in writing the driver
- Specify characteristics of the device
- Describe device usage
- Provide a description of the device registers
- Identify support in writing the driver

The following sections describe how you would fill out the worksheets provided for each of these tasks, using the /dev/none device driver as an example.

2.1.1 Specifying Information About the Host System

Figure 2-1 and Figure 2-2 show the host system information associated with the /dev/none driver. As the worksheets show, you gather the following information about the host system:

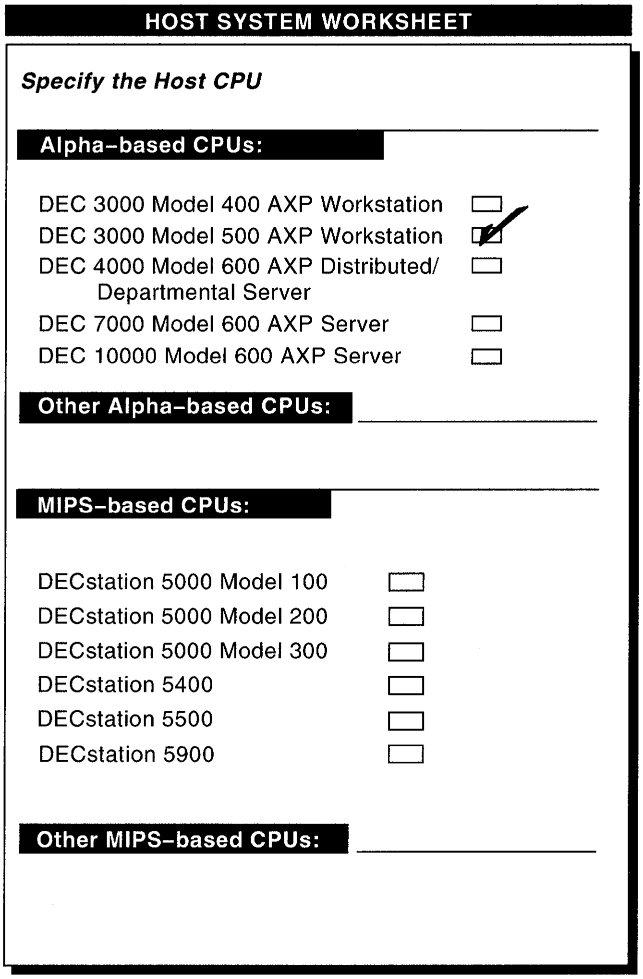
- The host CPU or CPUs your driver operates on
- The operating system or systems your driver operates on
- The bus or buses that your driver connects to

2.1.1.1 Specifying the Host CPU or CPUs on Which Your Driver Operates

The /dev/none driver will be developed for use on a DEC 3000 Model 500 AXP Workstation.

The goal should always be to write any device driver to operate on more than one CPU hardware platform, so you should be aware of any differences presented by different CPU architectures. By identifying the hardware platforms you want the driver to operate on, you can address any driver design decisions related to the CPU hardware architecture. This identification can help you determine whether one driver, with appropriate conditional compilation statements, can handle the CPU hardware platforms you want the driver to operate on. You might decide that it would be easier to write two device drivers. Section 2.4 discusses some device driver design issues that are affected by CPU architectures.

Figure 2-1: Host System Worksheet for /dev/none



HOST SYSTEM WORKSHEET

Specify the Host CPU

Alpha-based CPUs:

DEC 3000 Model 400 AXP Workstation	☐
DEC 3000 Model 500 AXP Workstation	✓
DEC 4000 Model 600 AXP Distributed/ Departmental Server	☐
DEC 7000 Model 600 AXP Server	☐
DEC 10000 Model 600 AXP Server	☐

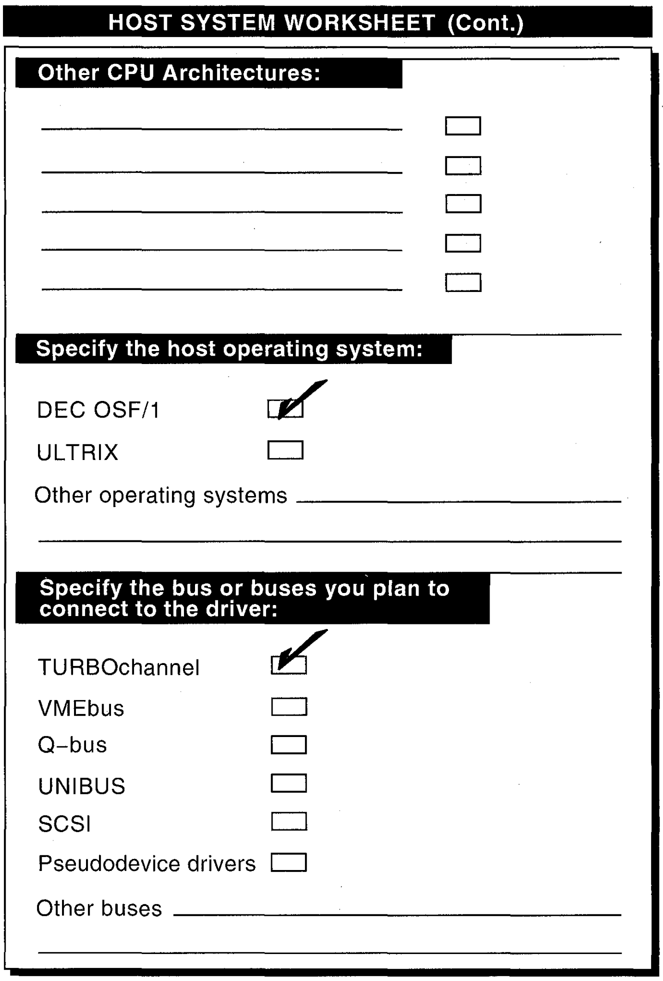
Other Alpha-based CPUs: ____________

MIPS-based CPUs:

DECstation 5000 Model 100	☐
DECstation 5000 Model 200	☐
DECstation 5000 Model 300	☐
DECstation 5400	☐
DECstation 5500	☐
DECstation 5900	☐

Other MIPS-based CPUs: ____________

Figure 2-2: Host System Worksheet for /dev/none (Cont.)



HOST SYSTEM WORKSHEET (Cont.)

Other CPU Architectures:

____________	☐
____________	☐
____________	☐
____________	☐
____________	☐

Specify the host operating system:

DEC OSF/1	✓
ULTRIX	☐

Other operating systems ____________

Specify the bus or buses you plan to connect to the driver:

TURBOchannel	✓
VMEbus	☐
Q-bus	☐
UNIBUS	☐
SCSI	☐
Pseudodevice drivers	☐

Other buses ____________

2.1.1.2 Specifying the Operating System or Systems on Which Your Driver Operates

The `/dev/none` driver will be developed for use on the DEC OSF/1 operating system. This is an important consideration because data structures and kernel interfaces differ between operating systems. For example, device drivers developed for DEC OSF/1 systems might initialize a `driver` structure, while drivers developed for other versions of UNIX systems would initialize a different structure. This identification can help you determine the amount of work that is involved in porting an existing device driver from some other UNIX operating system to the DEC OSF/1 operating system. Section 2.6 provides information to help you understand the tasks involved in porting from ULTRIX to DEC OSF/1 systems.

The host CPU types you select may dictate which host operating system you can use. For example, VAX-based CPUs do not support the DEC OSF/1 operating system. When you choose a CPU type, ensure that it supports the chosen operating system.

2.1.1.3 Specifying the Bus or Buses to Which Your Driver Connects

You must identify 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 task is not applicable for drivers that operate on other buses. For example purposes, the `/dev/none` driver will be developed for use on the TURBOchannel bus.

You must know the bus type and CPU architecture when deciding to make your device driver loadable. For DEC OSF/1, loadable drivers are not supported on the Alpha AXP architecture. However, you might want to follow the approach taken by the example drivers in this book and implement the loadable driver code now in anticipation of future support.

Note that the worksheet provides a space for pseudodevice drivers. A pseudodevice driver, such as the `pty` terminal driver, is structured like any other driver. The difference is that a pseudodevice driver does not operate on a bus. This book does not specifically address pseudodevice drivers.

2.1.2 Identifying the Standards Used in Writing the Driver

Figure 2-3 and Figure 2-4 show the device driver conventions worksheet for the `/dev/none` driver. As the worksheets show:

- You specify a naming scheme
- You choose an approach for writing comments and documentation

2.1.2.1 Specifying a Naming Scheme

The `/dev/none` driver uses the name `none` as the prefix for device driver interface names. Device driver interfaces written for DEC OSF/1 can use the following naming conventions:

- A prefix that represents the name of some device. In this example, the device is called `none`; therefore, each driver interface begins with that prefix.
- The name of the interface, for example, `open` and `close`. Thus, the example driver has interface names such as `noneopen` and `noneclose`.

The /dev/none driver uses the name none as the prefix for data structures internal to the device driver. These structures include the device register structure and, possibly, a data structure to store driver-specific information.

The /dev/none driver uses the prefix DN for device driver constant names. The prefix matches the first two characters in the driver name, /dev/none. These constants can represent values or macros. For example, the constant DN_SIZE might represent the size of the device register area.

The previously described naming schemes are recommendations, not requirements. The one naming requirement you must follow concerns the name of the configure interface, which for the /dev/none driver is none_configure. This interface is the configuration entry point called when the driver is dynamically loaded. For the configure interface, the underscore character (_) must follow the driver's name. This underscore character in the name is a requirement of the configuration process for loadable drivers and is the OSF convention.

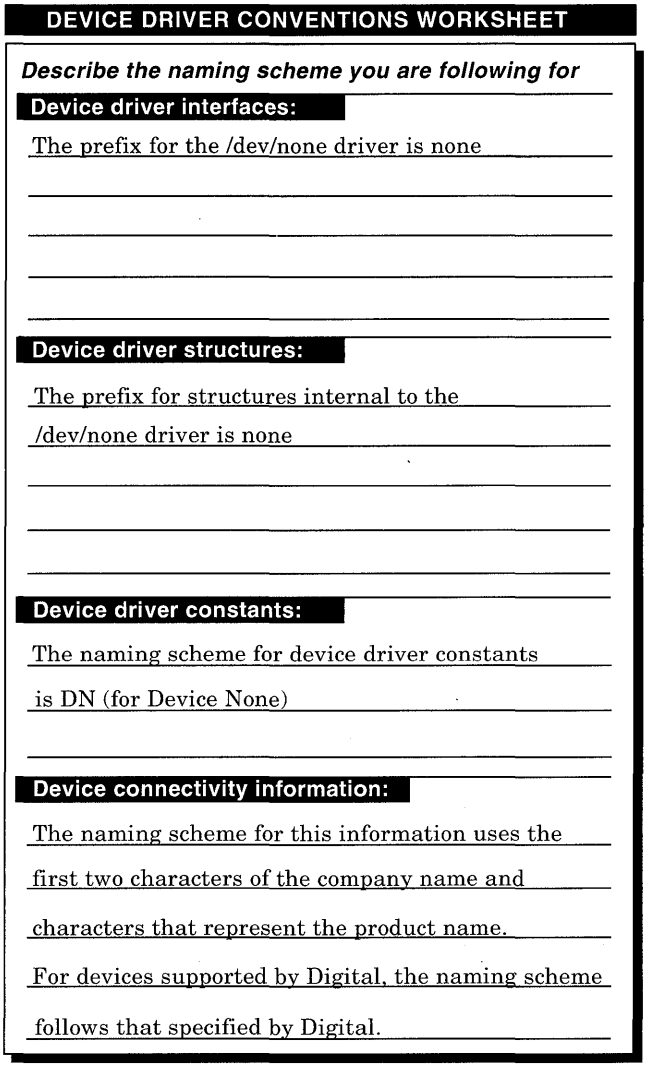
Before choosing a naming scheme, you have to make sure that these names do not conflict with other driver interface and structure names. To help you determine what names are currently used by the system, run the nm command on the kernel image file. This image file is usually called /vmunix. If you follow the third-party device driver configuration model described in Chapter 11, you will want to be particularly careful about choosing a naming scheme to ensure that it does not conflict with other third-party driver vendors.

If you follow the third-party device driver configuration model, you also need to choose a naming scheme for specifying device connectivity information in the following files: system configuration file and the stanza.static file fragment for static drivers and the stanza.loadable file fragment for loadable drivers. A naming scheme is discussed along with these files in Chapter 12. However, a brief discussion here can prepare you for a better understanding of the naming scheme when you encounter it in Chapter 12.

Third-party driver writers may need to specify bus, controller, and device information in the previously mentioned files. If you are supporting Digital devices, you specify valid strings listed in the *System Administration* guide. These strings represent the buses, controllers, and devices supported by Digital.

If you are supporting non-Digital devices, you can select any string other than those already chosen by Digital to represent the device. However, without an exclusive naming scheme, your choices could conflict with other third-party driver vendors. To avoid these conflicts, you can select a string that includes the vendor and product names and, possibly, the version and release numbers. This type of naming scheme minimizes the potential for name conflicts. For example, the driver writers at EasyDriver Incorporated might specify edgd for an internally developed device.

Figure 2-3: Device Driver Conventions Worksheet for /dev/none



DEVICE DRIVER CONVENTIONS WORKSHEET

Describe the naming scheme you are following for

Device driver interfaces:

The prefix for the /dev/none driver is none

Device driver structures:

The prefix for structures internal to the /dev/none driver is none

Device driver constants:

The naming scheme for device driver constants is DN (for Device None)

Device connectivity information:

The naming scheme for this information uses the first two characters of the company name and characters that represent the product name. For devices supported by Digital, the naming scheme follows that specified by Digital.

2.1.2.2 Choosing an Approach for Writing Comments and Documentation

The `/dev/none` driver takes two approaches to supplying comments in the driver code. In the first approach, the `/dev/none` driver contains no inline comments. Instead, the following convention is used:

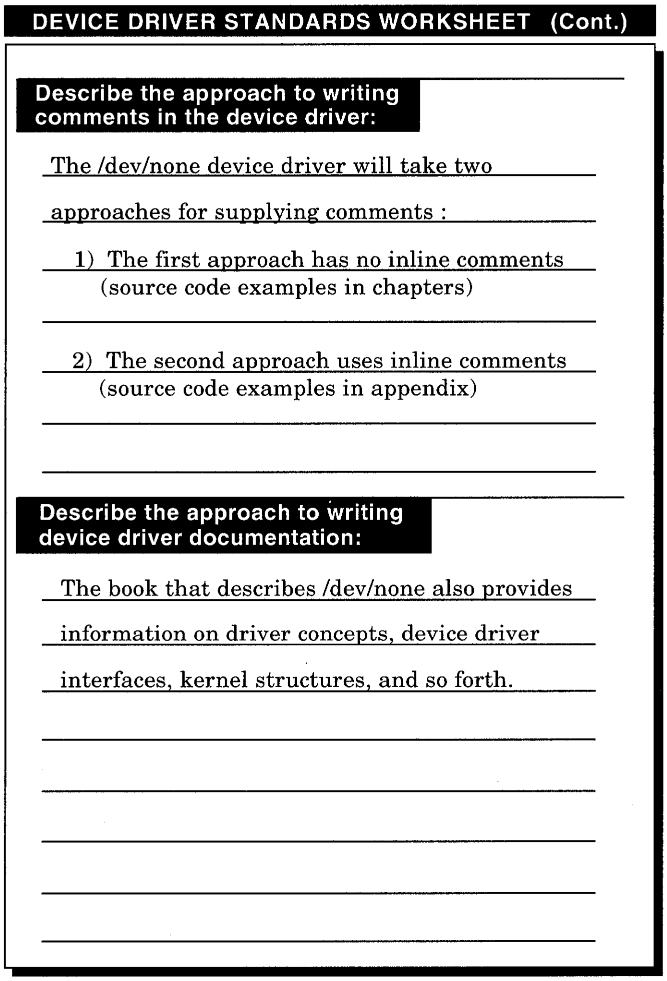
```
int unit = minor(dev); [1]
```

[1] A number appears after a line of code in the `/dev/none` device driver example. Following the example, a corresponding number appears that contains an explanation of the associated line or lines. The device driver examples in Chapter 4 and Chapter 10 use the first approach to make the source code easier to read.

In the second approach, the `/dev/none` device driver supplies appropriate inline comments. The source code listings in Appendix B use the second approach.

In addition to providing background information and detailed explanations of the `/dev/none` driver, this book also provides information on device driver concepts, kernel interfaces, data structures, and so forth. Your approach to writing device driver documentation may be different.

Figure 2-4: Device Driver Conventions Worksheet for /dev/none (Cont.)



DEVICE DRIVER STANDARDS WORKSHEET (Cont.)

Describe the approach to writing comments in the device driver:

The /dev/none device driver will take two approaches for supplying comments :

1) The first approach has no inline comments (source code examples in chapters)

2) The second approach uses inline comments (source code examples in appendix)

Describe the approach to writing device driver documentation:

The book that describes /dev/none also provides information on driver concepts, device driver interfaces, kernel structures, and so forth.

2.1.3 Specifying Characteristics of the Device

Figure 2-5 and Figure 2-6 show the device characteristics for the `none` device associated with the `/dev/none` driver.

As the worksheets show, you specify the following characteristics associated with the device:

- Whether the device is capable of block I/O
- Whether the device will support a file system
- Whether the device supports byte stream access
- Actions to be taken on interrupts
- How the device should be reset
- Other device characteristics

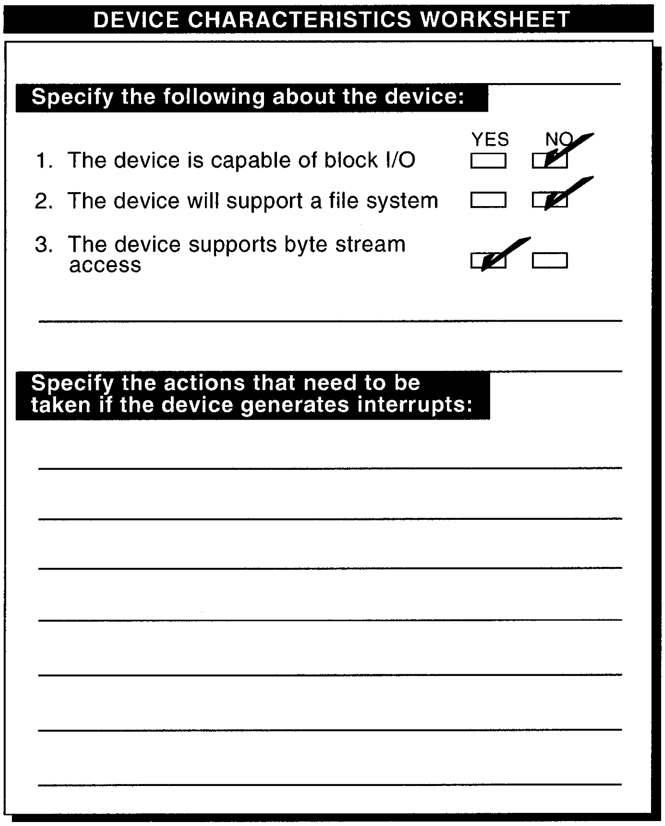
2.1.3.1 Specifying Whether the Device Is Capable of Block I/O

If the device is capable of block I/O, then you would write a block device driver. A block device is one that stores data on its media in a standard way. For example, most disk drives store data in disk sectors (typically 512 bytes). Tape drives sometimes store data in a standard size tape record.

Typically, block devices are random access devices (that is, disks) because the file system does not always perform I/O to sequential disk sectors. Tape devices are typically sequential access devices and, therefore, not suitable for using as a block device.

The `none` device is not capable of handling blocks of data so the No box on the worksheet is marked.

Figure 2-5: Device Characteristics Worksheet for the none Device



DEVICE CHARACTERISTICS WORKSHEET

Specify the following about the device:

	YES	NO
1. The device is capable of block I/O		✓
2. The device will support a file system		✓
3. The device supports byte stream access	✓	

Specify the actions that need to be taken if the device generates interrupts:

2.1.3.2 Specifying Whether the Device Supports a File System

Most block devices can support file systems. If a block device supports a file system, it must be able to map between file system blocks and the underlying structure on the device. In DEC OSF/1, this mapping is accomplished through partition tables.

The `none` device is not capable of supporting file systems, so the No box on the worksheet is marked.

2.1.3.3 Specifying Whether the Device Supports Byte Stream Access

Most devices support byte stream access. This access can be viewed as sequentially accessing data through the device. For example, a sequence of characters typed at a terminal constitutes a byte stream. Most block devices can also be accessed in this manner. When a block device is accessed as a stream of bytes, the access is typically called ''raw'' access. When accessed this way, the data on the block device is accessed sequentially without any underlying structure being placed on the data (for example, disk sectors).

For the `none` device, the Yes box on the worksheet is marked.

2.1.3.4 Specifying Actions to Take on Interrupts

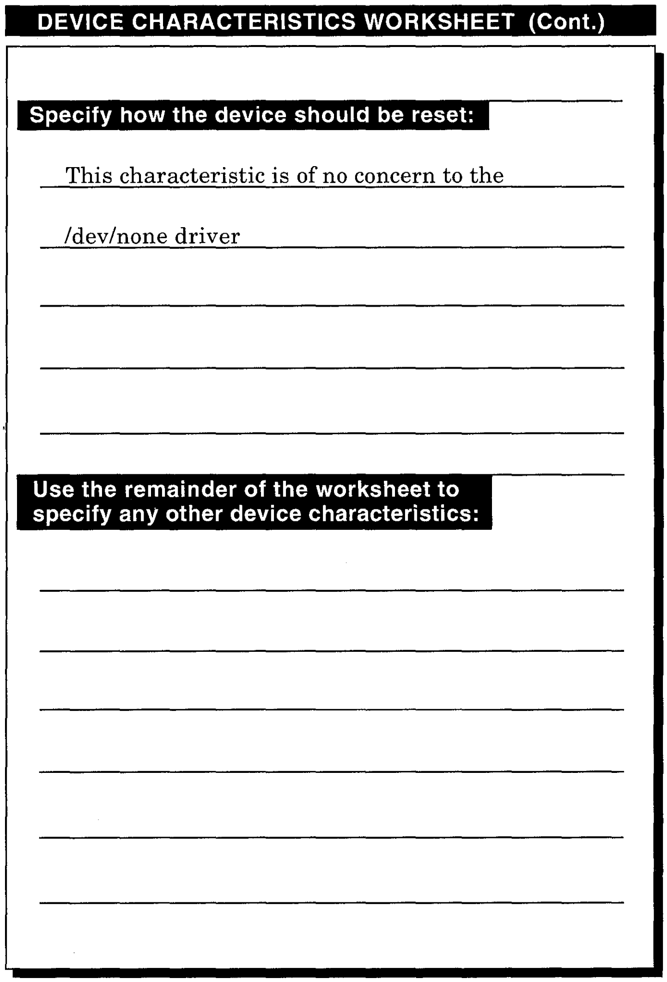
Use this space to summarize what the driver interrupt interfaces will do when the device generates an interrupt. For example, a terminal type character driver's interrupt service interface (ISI) can receive a character that was typed on a user's keyboard. Typically, the ISI must determine the source of the interrupt, respond to the interrupt (for example, by reading in the data), and perform the appropriate actions to cause the interrupt to be dismissed.

Some other issues concerning interrupts are:

- Locking out interrupts when performing vulnerable operations
- Not locking out interrupts for an extended period of time
- Queueing the data so that handling of it can be interrupted

Because the `none` device has no underlying physical hardware, it cannot generate interrupts. Therefore, this part of the worksheet is left blank.

Figure 2-6: Device Characteristics Worksheet for the none Device (Cont.)



DEVICE CHARACTERISTICS WORKSHEET (Cont.)

Specify how the device should be reset:

This characteristic is of no concern to the /dev/none driver

Use the remainder of the worksheet to specify any other device characteristics:

2.1.3.5 Specifying How to Reset the Device

If the bus that the device controller is connected to supports the reset function, the device driver must be able to stop all current work and place the device connected to the controller in a known, quiescent state.

For example purposes, the `none` device is connected to the TURBOchannel bus. To keep the example driver simple, the reset function will not be implemented as part of the `/dev/none` device driver. Thus, this characteristic is of no concern to the `/dev/none` device driver.

2.1.3.6 Specifying Other Device Characteristics

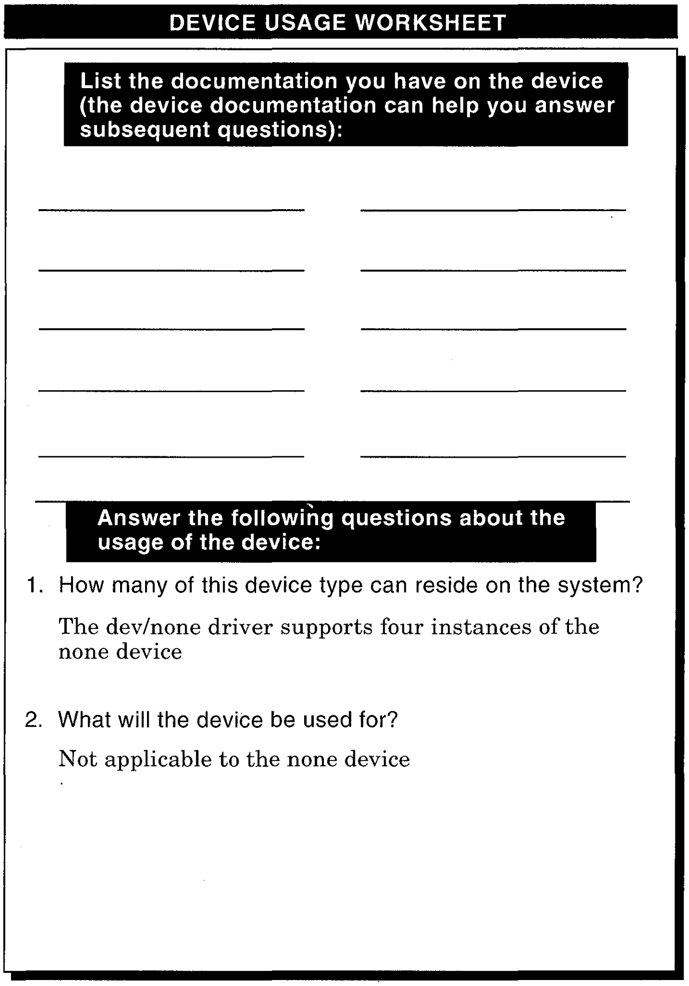
Use this space to identify other characteristics of the device that might influence how you design your device driver.

2.1.4 Describing Device Usage

Figure 2-7 shows the device usage information for the `none` device associated with the `/dev/none` driver. As the worksheet shows, you gather the following information about device usage:

- The documentation you have on the device
- The number of instances of this device type that can reside on the system
- The purpose of the device

Figure 2-7: Device Usage Worksheet for the none Device



DEVICE USAGE WORKSHEET

List the documentation you have on the device (the device documentation can help you answer subsequent questions):

Answer the following questions about the usage of the device:

1. How many of this device type can reside on the system?

   The dev/none driver supports four instances of the none device

2. What will the device be used for?

   Not applicable to the none device

2.1.4.1 Listing the Device Documentation

For a real device, you should have on hand the manual for the device that is supplied by the manufacturer. The none device is a fictitious device; therefore, this section of the worksheet is left blank.

2.1.4.2 Specifying the Number of Device Types to Reside on the System

The number of devices that can be supported has a direct effect on the design of the driver. If only one will be supported, the driver need not worry about determining which device is being accessed. If a small number (for example, 2 – 5) is supported, the driver can use simple data structures and indexing to keep track of device access. If a greater number is to be supported, the driver must use more sophisticated methods to keep track of which device is being accessed.

The /dev/none driver is written to accommodate more than one instance. It will allocate fixed storage for four instances of the none device.

2.1.4.3 Describing the Purpose of the Device

For this device usage item, enter a short description of the purpose of the device. For example, the purpose of most disk devices is to provide storage for user data and files. The purpose of most terminal devices is to provide a means for interacting with users of the system.

This item is not applicable to the none device because it is a fictitious device.

2.1.5 Providing a Description of the Device Registers

Figure 2-8 and Figure 2-9 show the device register information for the /dev/none driver. As the worksheets show, you gather the following information about the device registers:

- A description or sketch of the layout of the device registers

  The manual supplied with the device would most likely have the following:

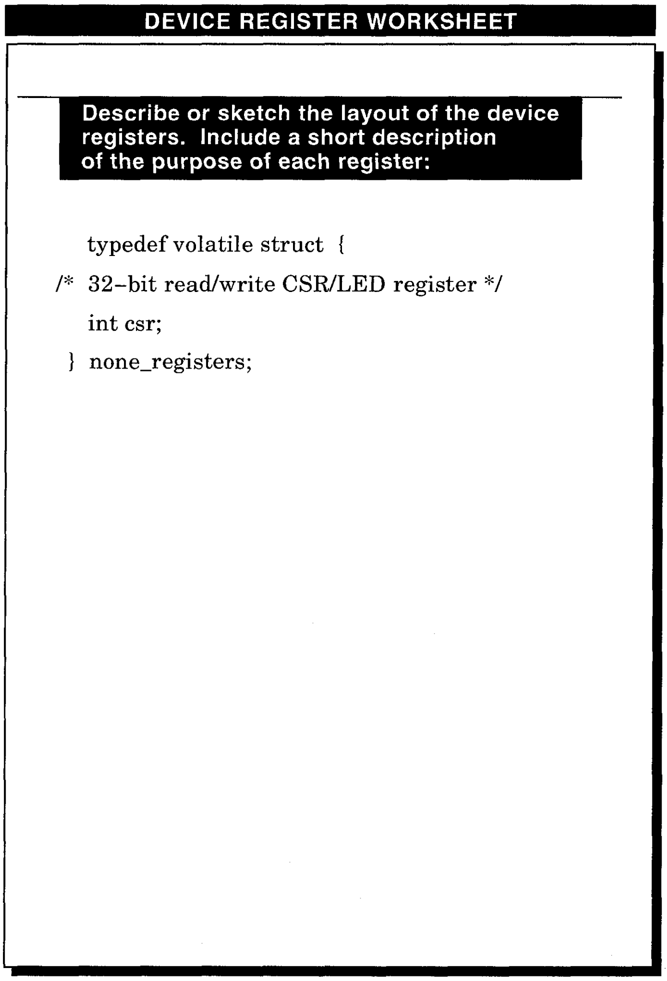
  – The layout of the registers and their offset

  – How the registers are used

  The worksheet shows a structure definition of the device register for the none device along with a comment as to its function. In previous versions of DEC OSF/1, device drivers directly accessed the device registers. With this version of DEC OSF/1, Digital recommends that device drivers access the device registers by calling the read_io_port and write_io_port interfaces. The /dev/none driver now accesses the device registers for the `none` device by calling these interfaces. See Section 9.7.1 for more information on `read_io_port` and `write_io_port`.

- A mapping of the device register with the memory address

Figure 2-8: Device Register Worksheet for /dev/none

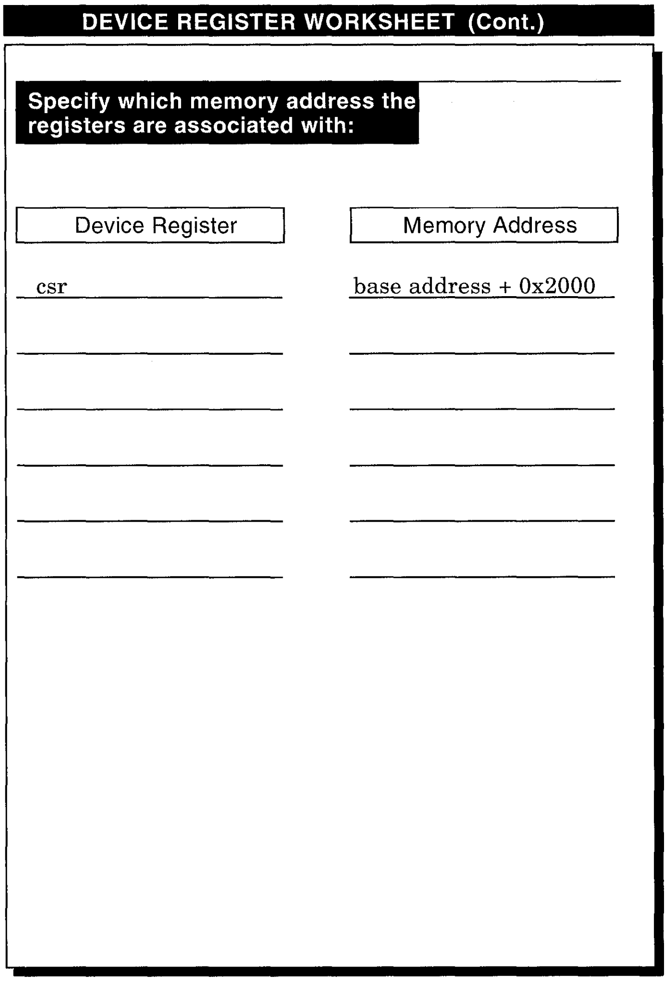


DEVICE REGISTER WORKSHEET

Describe or sketch the layout of the device registers. Include a short description of the purpose of each register:

```
   typedef volatile struct {
/* 32-bit read/write CSR/LED register */
   int csr;
 } none_registers;
```

Figure 2-9: Device Register Worksheet for /dev/none (Cont.)



DEVICE REGISTER WORKSHEET (Cont.)

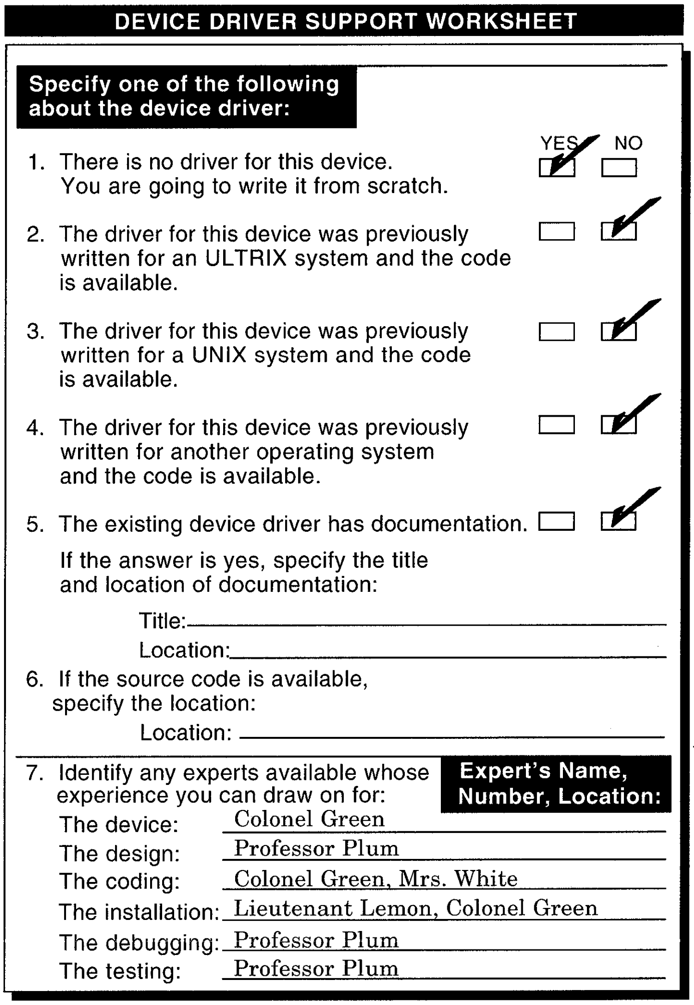
Specify which memory address the registers are associated with:

Device Register	Memory Address
csr	base address + 0x2000

2.1.6 Identifying Support in Writing the Driver

Figure 2-10 shows the support associated with writing the `/dev/none` driver.

Figure 2-10: Device Driver Support Worksheet for /dev/none



DEVICE DRIVER SUPPORT WORKSHEET

Specify one of the following about the device driver:

	YES	NO
1. There is no driver for this device. You are going to write it from scratch.	✓	☐
2. The driver for this device was previously written for an ULTRIX system and the code is available.	☐	✓
3. The driver for this device was previously written for a UNIX system and the code is available.	☐	✓
4. The driver for this device was previously written for another operating system and the code is available.	☐	✓
5. The existing device driver has documentation.	☐	✓

If the answer is yes, specify the title and location of documentation:

Title: __________

Location: __________

6. If the source code is available, specify the location:

Location: __________

7. Identify any experts available whose experience you can draw on for:

	Expert's Name, Number, Location:
The device:	Colonel Green
The design:	Professor Plum
The coding:	Colonel Green, Mrs. White
The installation:	Lieutenant Lemon, Colonel Green
The debugging:	Professor Plum
The testing:	Professor Plum

As the worksheet shows, you gather the following information about device driver support:

- Whether you are writing the driver from scratch

  Answering this question can help determine the amount of time you need to spend writing the driver. Most device drivers are written by modifying an existing device driver that contains similar functionality. However, if source code for an existing driver is not available, it may take you more time to develop the driver. The `/dev/none` driver will be developed from scratch; therefore, the Yes box on the worksheet is marked.

- Whether the driver for the device was written previously for an ULTRIX system

  If the source code is available, you can update the device driver to reflect the DEC OSF/1 operating system. Section 2.6 provides information to help you understand the tasks involved in porting from ULTRIX to DEC OSF/1. In this case, no source code is available so the No box on the worksheet is marked.

- Whether the driver for the device was written previously for a UNIX system

  If the source code is available, you can begin updating the device driver by identifying areas that are different from DEC OSF/1. Other versions of UNIX would probably have different data structures and some of the kernel interfaces would probably behave differently and expect different arguments. In this case, no source code is available so the No box on the worksheet is marked.

- Whether the driver for the device was written previously for another operating system

  If the source code is available, you must study the differences between writing a device driver on that operating system and on the DEC OSF/1 operating system. This would probably be more difficult than the previous two situations. For the `/dev/none` device driver, the No box is marked.

- Whether the existing driver has documentation

  If the existing driver has documentation, specify the title and where it is located. Look for chapters on data structures and the use of kernel interfaces. These chapters might help in the porting task. For the `/dev/none` device driver, the No box is marked.

- Location of the existing driver source code

  This information is important. It allows systems engineers added to the project to locate the source code quickly. This item is left blank for the `/dev/none` device driver.

- Experts available

  Writing complex device drivers is always easier when you have access not only to documentation but also to other device driver experts. The worksheet shows the experts who helped in writing the `/dev/none` device driver. Identifying your experts will make it easier for any future driver writers who work on your driver projects.

2.2 Designing the Device Driver

After you gather information about the host system and the device, your next task is to design the device driver. You need to:

- Specify the type of device driver
- Identify device driver entry points

The following sections describe how you would fill out the worksheets provided for each of the above tasks, using the `/dev/none` device driver as an example.

2.2.1 Specifying the Device Driver Type

Figure 2-11 shows the types of device drivers you can write on a DEC OSF/1 operating system. As the worksheet shows, you identify your driver as one of the following:

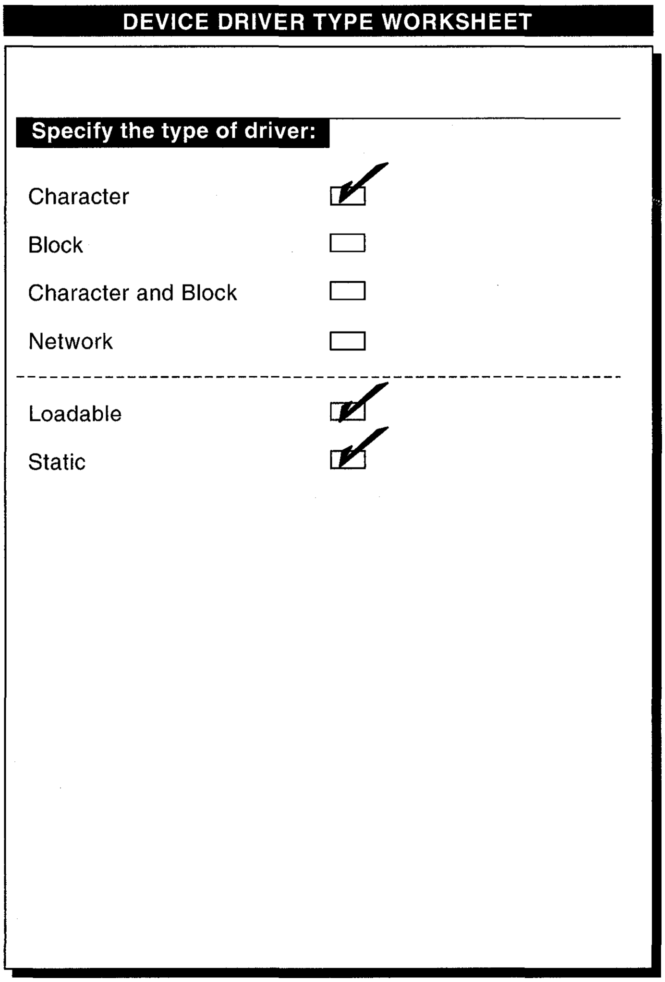
- Block
- Character
- Block and character
- Network

When using the buffer cache to perform I/O on blocks of data, you use block device drivers. Otherwise, you use character device drivers. Most device drivers are character drivers. Some device drivers are both character and block drivers. The network driver is another type of device driver. This book does not discuss network device drivers. The Character box is marked for the `/dev/none` device driver because it has no requirements to handle blocks of data.

The worksheet also shows that these types of device drivers can be both loadable and static. When a driver is both loadable and static, the system manager decides at installation time whether to configure the driver as loadable or static. By designing and writing the driver to be both loadable and static, you offer customers maximum flexibility. Even though loadable drivers are not currently supported on Alpha AXP systems, you may want to design and implement the loadable driver-specific code.

The Loadable and Static boxes are marked for the `/dev/none` device driver.

Figure 2-11: Device Driver Type Worksheet for /dev/none



DEVICE DRIVER TYPE WORKSHEET

Specify the type of driver:

Character	✓
Block	☐
Character and Block	☐
Network	☐
Loadable	✓
Static	✓

2.2.2 Identifying Device Driver Entry Points

Figure 2-12 shows the device driver entry points for the /dev/none device driver. The worksheet is divided into parts because the possible entry points vary depending on whether the driver is a block, character, or network driver. Because the /dev/none driver is a character driver, the worksheet shows the following entry points:

- noneprobe

  This interface will determine if the device exists.

- nonecattach

  This interface performs initialization for the none controller.

- none_configure

  This interface is the configuration entry point called when the driver is dynamically loaded. For the configure interface, the underscore character (_) must follow the driver's name. This underscore character in the name is a requirement of the configuration process for loadable drivers and is the OSF convention.

- noneopen

  This interface will turn on the open flag.

- noneclose

  This interface will turn off the open flag.

- noneread

  This interface will return an EOF (end-of-file).

- nonewrite

  This interface will add to the count the number of characters written.

- noneioctl

  This interface will handle the following special requests:

  - Reinitialize the count to zero
  - Return the current count

- noneintr

  This interface is a stub for future development

Chapter 3 shows you how to set up the interfaces associated with character and block device drivers. Chapter 4 explains how the above interfaces are implemented for the /dev/none driver.

You may want to describe the goals of the interfaces for your driver in a more formal device driver development specification. Consider adding this specification along with the worksheets to the device driver project binder.

Figure 2-12: Device Driver Entry Points Worksheet for /dev/none

DEVICE DRIVER ENTRY POINTS WORKSHEET

Block driver entry points:

Entry point:	Name:
probe	
slave	
cattach	
dattach	
configure	
open	
close	
strategy	
ioctl	
interrupt	
psize	
dump	

Character driver entry points:

Entry point:	Name:
probe	noneprobe
slave	
cattach	nonecattach (stub)
dattach	
configure	none_configure
open	noneopen
close	noneclose
strategy	
ioctl	noneioctl
stop	
reset	
read	noneread
write	nonewrite
mmap	
interrupt	noneintr (stub)

2.3 Determining the Structure Allocation Technique

Another design consideration is the technique you plan to use for allocating data structures. Generally, there are two techniques you can follow: dynamic allocation and static allocation. Dynamic allocation is the recommended method; however, static allocation is discussed in this book because many existing drivers allocate their data structures statically. Table 2-1 lists the structure allocation techniques discussed in this book along with some guidelines to help you choose the best method for your device drivers. Sections that provide examples of each follow the table.

Table 2-1: Structure Allocation Technique Guidelines

Technique	Guidelines for Using
Static allocation model 1	Use this technique if you do not plan to implement loadable device drivers now or in the future.
Static allocation model 2	Use this technique if you: • Plan to implement loadable drivers now or in the future • Know that the maximum number of devices is five or less • Know that the driver does not use numerous data structures
Dynamic allocation	Use this technique if you: • Plan to implement loadable drivers now or in the future • Know that the maximum number of devices is greater than five • Know that the driver uses numerous data structures

2.3.1 Static Allocation Technique Model 1

The static allocation technique model 1 lets you statically allocate data structures by using the compile time variable. The following code fragment uses the `none_softc` structure associated with the `/dev/none` device driver to illustrate the static allocation technique model 1:

```
  •
  •
  •
/**************************************************
 * softc structure used to compare static and     *
 * dynamic allocation techniques                  *
 **************************************************/
```

```
struct none_softc {
        int sc_openf;
        int sc_count;
        int sc_state;
}; 1
   •
   •
   •
/*******************************************
 * Declarations using the static technique    * 2
 *******************************************/
struct controller *noneinfo[NNONE];
struct none_softc none_softc[NNONE];
   •
   •
   •
```

1 These lines define a structure called `none_softc`, which is used to share data between the different `/dev/none` device driver interfaces. The members of this data structure are not important to the discussion of the static allocation technique model 1.

2 These declarations appear in the driver source `none.c`. The declarations show that the main characteristic of this structure allocation technique is the use of the compile time variable to size the structure arrays. Thus, in the example, the array of pointers to `controller` structures and the `none_softc` structure array use the compile time variable `NNONE`. Note that in the `none_softc` structure array there is one structure for each instance of the device as specified in the system configuration file.

The `config` program defines the compile time variable for statically configured drivers and stores it in the device driver header file. In the example, the device driver header file created by `config` would contain the following entry if there were four devices on the system:

```
#define NNONE 4
```

Section 3.1.1 provides additional details about the device driver header file and the compile time variable. The major drawback to statically allocating the data structures is that the number of configured devices is not available when loadable drivers are built. If you want your drivers to be both static and loadable, you need to dynamically allocate the data structures or statically allocate enough data structures to match the maximum configuration.

2.3.2 Static Allocation Technique Model 2

The static allocation technique model 2 lets you statically allocate enough data structures to accommodate the maximum configuration. To make access to the data structures as easy as possible, you statically declare an array of pointers to the data structures. The only difference between the static allocation techniques is that static allocation technique model 1 uses the compile time variable created by `config` and static allocation technique model 2 uses a constant that defines the maximum number of devices.

The following code fragment illustrates the static allocation model 2 technique:

```
/*********************************************
 * structure declarations to illustrate static *
 * allocation technique model 2               *
 *********************************************/

   •
   •
   •
#define MAX_NONE 4 /* Define maximum number of */
                   /* devices */ [1]
/*******************************************
 * Use the constant to size the arrays      * [2]
 *******************************************/
struct controller *noneinfo[MAX_NONE];
struct none_softc none_softc[MAX_NONE];
   •
   •
   •
```

[1] If you knew that your device driver supported at most four instances of some device (for example, four controllers), you would first declare a constant to represent this maxium number. Thus, the example defines a `MAX_NONE` constant and assigns it the maximum value 4.

You would probably want to declare this constant in a *name*`_data.c` file. See Section 3.1.5 for a description of the *name*`_data.c` file.

[2] The next two lines use the `MAX_NONE` constant to size the respective arrays. In the first line, `MAX_NONE` sizes the array of pointers to `controller` structures, and in the second line it sizes an array of `none_softc` structures.

The disadvantage of the static allocation model 2 technique is that, if you have only one instance of the device on the system, you are wasting three of the four declared `none_softc` data structures. If the number of these data structures does not exceed five and if they contain no more than a reasonable number of data members, the static allocation model 2 technique technique is acceptable.

2.3.3 Dynamic Allocation Technique

The previously described static allocation technique model 2 is suitable for device drivers that support a small number of devices (five or less). Some device drivers, however, support more than five devices and declare a significant number of data structures that might contain a large amount of data. For example, the Digital Storage Architecture (DSA) subsystem can support up to 256 disks and at least 16 controllers. Clearly, it would not be desirable to always allocate all the required data structures because for most configurations there are not nearly that many devices present. Statically allocating the maximum number of data structures would waste too much space. The DSA subsystem and systems like it are good examples of where dynamic configuration of the required data structures would be beneficial.

In this model of dynamically allocating the data structures, you need to:

- Determine the maximum configuration
- Statically declare an array of pointers to the data structures
- Allocate the data structures
- Access the members of the dynamically allocated data structures
- Free up the dynamically allocated memory

2.3.3.1 Determining the Maximum Configuration

The first task is to determine the maximum number of instances of the device that can exist on your system. In the example, suppose that there can be a maximum of 16 `none` devices. Thus, there is one driver that handles, at most, 16 instances of the device. In this case, you could define a constant similar to the following in the *name*`_data.c` file:

```
#define MAX_NONE 16
```

2.3.3.2 Statically Declaring an Array of Pointers to the Data Structures

Your next task is to statically declare an array of pointers to the data structures that will be used by the device driver. The example declares an array of pointers to `controller` and `none_softc` data structures as follows:

```
struct controller *noneinfo[MAX_NONE];
struct none_softc *none_softc[MAX_NONE];
```

These declarations are pointers to the data structures, not the data structures themselves. Although this approach is not required, it is chosen for the example because having the static array of pointers makes access to the data structures easy.

2.3.3.3 Allocating the Data Structures

The next task is to allocate the data structures in the Autoconfiguration Support Section of the device driver. This section is where you implement the driver's `probe`, `attach`, and `slave` interfaces. The following example shows how to allocate the data structures with the probe interface.

The driver's probe interface is called for each instance of the device that is actually present on the system. This makes the probe interface one place to dynamically allocate the data structures, as in the following example:

```
noneprobe(unit, ctlr)
   caddr_t unit; /* Unit number of device */ [1]
   struct controller *ctlr;
{
   if (unit > MAX_NONE) /* Is unit greater than 16? */ [2]
           return(0);
   •
   •
   •
/**********************************************
 * Allocate the softc structure associated     *
 * with this instance                          * [3]
 **********************************************/
   if (none_softc[unit] == (struct none_softc *)NULL) {
           none_softc[unit] = kalloc(sizeof(struct none_softc));
           if (none_softc[unit] == (struct none_softc *)NULL) {
                   return(0);
           }
   }
   return(sizeof(struct none_reg));
}
```

[1] The device driver has been passed the unit number as a parameter to the noneprobe interface. This *unit* parameter tells you which instance of the device you are referring to.

[2] This if statement determines if there are more than 16 controllers. If there are, noneprobe returns the value zero (0) to indicate an error. Otherwise, it allocates the data structures.

[3] Now you allocate the data structures associated with this instance. The first if statement is a test to make sure that the noneprobe interface has not already been called for this unit.

You then dynamically allocate the memory to accommodate the none_softc data structures by calling the kalloc interface and passing to it the number of bytes to allocate.

The second if statement checks the return value from kalloc. The return value is NULL if there is no memory available to be dynamically allocated for this instance of the driver. The driver will not be operational without its data structures. In this case, return the value zero (0) to indicate that the driver's probe interface failed.

2.3.3.4 Accessing the Members of the Dynamically Allocated Data Structures

After you dynamically allocate the data structures in the `probe` interface, you need to correctly access them. When using the compile time variable to size the arrays, the declaration looked like this:

```
struct none_softc none_softc[NNONE];
```

The declaration for dynamically allocating the data structures looks like this:

```
struct none_softc *none_softc[MAX_NONE];
```

The difference is that one declares the actual data structures while the other declares pointers to an array of the data structures. Thus, to access the data structures, you must reference the data structures as an array of pointers:

```
   •
   •
   •
/**********************************************
 * Accessing statically allocated data        *
 * structures                                 *
 **********************************************/
struct controller *ctlr = noneinfo[unit];
struct none_softc *sc = &none_softc[unit];
   •
   •
   •
/**********************************************
 * Accessing dynamically allocated data       *
 * structures                                 *
 **********************************************/
struct controller *ctlr = noneinfo[unit]; /* No change from */
                                          /* static allocation */
struct none_softc *sc = none_softc[unit]; /* Different from static */
                                          /* allocation */
```

The reference to the `none_softc` structures uses the address operator in the static case. In the dynamic case, the address operator is not used.

2.3.3.5 Freeing up the Dynamically Allocated Memory

Loadable drivers have a `ctlr_unattach` or a `dev_unattach` interface. When the driver is unloaded, these interfaces are called for each instance of the controller or device present on the system. When these interfaces are called, they should free up the dynamically allocated memory. For the `/dev/none` device driver, the `ctlr_unattach` interface frees up memory as follows:

```
   •
   •
   •
if (unit < MAX_NONE) {
/*********************************************
 * Free up the dynamically allocated memory    * [1]
 *********************************************/
if (none_softc[unit] != (struct none_softc *)NULL) {
    kfree(none_softc[unit], sizeof(struct none_softc));
/*********************************************
 * Set the array element to NULL              * [2]
 *********************************************/
    none_softc[unit] = (struct none_softc *)NULL;
    }
   •
   •
   •
}
```

[1] This example of a `ctlr_unattach` interface uses *`unit`* to refer to the specific instance of the driver `/dev/none`. The `if` statement is included as a safety check. Note that the `kfree` interface is used to free the memory previously allocated in a call to `kalloc`.

[2] After the dynamically allocated memory has been freed, the following line sets the `none_softc` array to NULL. This ensures that there will be no dangling references to the memory.

2.4 Understanding CPU Issues That Influence Device Driver Design

Whenever possible, you should design a device driver so that it can accommodate peripheral devices that operate on more then one CPU architecture. You need to consider the following issues to make your drivers portable across CPU architectures. The discussion centers around Alpha AXP and MIPS CPU platforms, but the topics may be applicable to other CPU architectures:

- Control status register (CSR) access issues
- Input/output (I/O) copy operation issues
- Direct memory access (DMA) operation issues
- Memory mapping issues
- 64-bit versus 32-bit issues
- Loadable driver issues

- Memory barriers

2.4.1 Control Status Register Issues

Many device drivers based on UNIX access a device's control status register (CSR) addresses directly through a device register structure. This method involves declaring a device register structure that describes the device's characteristics, which include a device's control status register. After declaring the device register structure, the driver accesses the device's CSR addresses through the member that maps to it. In the following code fragment, the device driver accesses the `none` device's CSR address through the `status` member of the device register structure:

```
   •
   •
   •
/**********************************************
 * Device register structure                  *
 **********************************************/
typedef volatile struct {
  int status;
  int error;
}some_registers;
   •
   •
   •
/**********************************************
 * noneprobe interface                        *
 **********************************************/
noneprobe(vbaddr, ctlr)
caddr_t vbaddr; /* System virtual address for */
                /* the none device*/
struct controller *ctlr; /* controller structure */
                         /* for this unit */
/**********************************************
 * Initialize pointer to none_registers       *
 * structure                                  *
 **********************************************/
register some_registers *reg = (struct some_registers *) vbaddr;
   •
   •
   •
/**********************************************
 * If device error, return 0                  *
 **********************************************/
if (reg->status & SOME_ERROR)
{
  return(0);
}
/**********************************************
 * Otherwise, initialize the csr              *
 **********************************************/
reg->status = 0;
```

There are some CPU architectures that do not allow you to access the device CSR addresses directly. If you want to write your device driver to operate on both types of CPU architectures, you can write one device driver with the appropriate conditional compilation statements. You can also avoid the potentially confusing proliferation of conditional compilation statements by using the CSR I/O access kernel interfaces provided by DEC OSF/1 to read from and write to the device's CSR addresses. Because the CSR I/O access interfaces are designed to be CPU hardware independent, their use not only simplifies the readability of the driver, but also makes the driver more portable across different CPU architectures and different CPU types within the same architecture. Section 9.7.1 shows you how to use the CSR I/O access kernel interfaces to read from and write to the device's CSR addresses.

2.4.2 Input/Output Copy Operation Issues

Input/output (I/O) copy operations can differ markedly from one device driver to another because of the differences in CPU architectures. Using the current techniques for performing I/O copy operations, you would probably not be able to write one device driver that operates on more than one CPU architecture or more than one CPU type within the same architecture. To overcome this lack of portability when performing I/O copy operations, DEC OSF/1 provides generic kernel interfaces to the system-level interfaces required by device drivers to perform an I/O copy operation. Because these I/O copy interfaces are designed to be CPU hardware independent, their use makes the driver more portable across different CPU architectures and more than one CPU type within the same architecture. Section 9.7.2 shows you how to call these I/O copy operation interfaces.

2.4.3 Direct Memory Access Operation Issues

Direct memory access (DMA) operations can differ markedly from one device driver to another because of the DMA hardware support features for buses on Alpha AXP systems and because of the diversity of the buses themselves. Using the current techniques for performing DMA, you would probably not be able to write one device driver that operates on more than one CPU architecture or more than one CPU type within the same architecture. To overcome this lack of portability when it comes to performing DMA operations, DEC OSF/1 provides generic kernel interfaces to the system-level interfaces required by device drivers to perform a DMA operation. These generic interfaces are typically called ''mapping interfaces.'' This is because their historical background is to acquire the hardware and software resources needed to map contiguous I/O bus addresses and accesses into discontiguous system memory addresses and accesses. Because these interfaces are designed to be CPU hardware independent, their use makes the driver more portable across different CPU architectures and more than one CPU type within the same architecture.

Section 9.8 shows you how to use these mapping interfaces to achieve device driver portability across different CPU architectures.

2.4.4 Memory Mapping Issues

Many device drivers based on UNIX provide a memory map section to handle applications that make use of the `mmap` system call. An application calls `mmap` to map a character device's memory into user address space. Some CPU architectures, including some Alpha AXP CPUs, do not support an application's use of the `mmap` system call. If your device driver operates only on CPUs that support the `mmap` feature, you can continue writing a memory map section. If, however, you want the device driver to operate on CPUs that do not support the `mmap` feature, you should design the device driver so that it uses something other than a memory map section.

2.4.5 64-Bit Versus 32-Bit Issues

This section describes issues related to declaring data types for 32-bit and 64-bit CPU architectures. By paying careful attention to data types, you can make your device drivers work on both 32-bit and 64-bit systems. Table 2-2 lists the C compiler data types and bit sizes for the MIPS 32-bit and the Alpha AXP 64-bit CPUs.

Table 2-2: C Compiler Data Types and Bit Sizes

C Type	MIPS 32-Bit Data Size	Alpha AXP 64-Bit Data Size
short	16 bits	16 bits
int	32 bits	32 bits
long	32 bits	64 bits
* (pointer)	32 bits	64 bits
long long	64 bits	64 bits
char	8 bits	8 bits

The following sections describe some common declaration situations:

- Declaring 32-bit variables
- Declaring 32-bit and 64-bit variables
- Declaring arguments to C interfaces (functions)
- Declaring register variables

- Performing bit operations on constants
- Using NULL and zero (0) values
- Modifying type `char`
- Declaring bit fields
- Using `printf` formats
- Using `mb` and `wbflush`
- Using the compiler keyword volatile

Note

The `/usr/sys/include/io/common/iotypes.h` file defines constants used for 64-bit conversions. See *Writing Device Drivers, Volume 2: Reference* for a description of the contents of this file.

2.4.5.1 Declaring 32-Bit Variables

Declare any variable that you want to be 32 bits in size as type `int`, not type `long`. The size of variables declared as type `int` is 32 bits on both the 32-bit MIPS systems and the 64-bit Alpha AXP systems.

Look at any variables declared as type `int` in your existing device drivers to determine if they hold an address. On Alpha AXP systems, `sizeof (int)` is not equal to `sizeof (char *)`.

In your existing device drivers, also look at any variable declared as type `long`. If it must be 32 bits in size, you have to change the variable declaration to type `int`.

2.4.5.2 Declaring 32-Bit and 64-Bit Variables

If a variable should be 32 bits in size on a 32-bit MIPS system and 64 bits in size on a 64-bit Alpha AXP system, declare it as type `long`.

2.4.5.3 Declaring Arguments to C Functions

Watch out for arguments to C interfaces (functions) where the argument is not explicitly declared and typed. You should explicitly declare the formal parameters to C interfaces; otherwise, their sizes may not match up with the calling program. The default size is type `int`, which truncates 64-bit addresses.

2.4.5.4 Declaring Register Variables

When you declare variables with the `register` keyword, the compiler defaults its size to that of type `int`. For example:

```
register somevariable;
```

Remember that these variable declarations also default to type `int`. For example:

```
unsigned somevariable;
```

Thus, if you want the variable to be 32 bits in size on both the 32-bit MIPS and 64-bit Alpha AXP systems, the above declarations are correct. However, if you want the variable to be 32 bits in size on a 32-bit MIPS system and 64 bits in size on a 64-bit Alpha AXP system, declare the variables explicitly, using the type `long`.

2.4.5.5 Performing Bit Operations on Constants

By default, constants are 32-bit quantities. When you perform shift or bit operations on constants, the compiler gives 32-bit results. If you want a 64-bit result, you must follow the constant with an `L`. Otherwise, you get a 32-bit result. For example, the following is a left shift operation that uses the `L`:

```
long foo, bar;
foo = 1L << bar;
```

2.4.5.6 Using NULL and Zero Values

Using the value zero (0) where you should use the value NULL means that you get a 32-bit constant. On Alpha AXP systems, this usage could mean the value zero (0) in the low 32 bits and indeterminate bit values in the high 32 bits. Using NULL from the `types.h` file allows you to obtain the correct value for both the MIPS and Alpha AXP CPUs.

2.4.5.7 Modifying Type char

Modifying a variable declared as type `char` is not atomic on Alpha AXP systems. You will get a load of 32 or 64 bits and then byte operations to extract, mask, and shift the byte, followed by a store of 32 or 64 bits.

2.4.5.8 Declaring Bit Fields

Bit fields declared as type `int` on Alpha AXP systems generate a load/store of longword (32 bits). Bit fields declared as type `long` on Alpha AXP systems generate a load/store of quadword (64 bits).

2.4.5.9 Using printf Formats

The `printf` formats %d and %x will print 32 bits of data. To obtain 64 bits of data, use %ld and %lx.

2.4.5.10 Using mb and wbflush

Device drivers used the `wbflush` interface in ULTRIX on MIPS CPUs. You can continue to call `wbflush` because in DEC OSF/1 it is aliased to the `mb` interface for Alpha AXP CPUs. The remainder of this section discusses when to call the `mb` interface on Alpha AXP CPUs.

In most situations that would require a cache flush on other CPU architectures, call the `mb` (memory barrier) interface on DEC OSF/1 Alpha AXP. The reason is not that `mb` is equivalent to a cache flush (as it is not). Rather, a common reason for doing a cache flush is to make data written by the host CPU available in main memory for access by the DMA device or to access from the host CPU data that was put in main memory by a DMA device. In each case, on an Alpha AXP CPU you should synchronize with that event by using a memory barrier.

A call to `mb` is occasionally needed even where a call to `wbflush` was not needed. In general, a memory barrier causes loads/stores to be serialized (not out-of-order), empties memory pipelines and write buffers, and assures that the data cache is coherent.

You should use the `mb` interface to synchronize DMA buffers. Use it before the host releases the buffer to the device and before the host accesses a buffer filled by the device.

Alpha AXP CPUs do not guarantee to preserve write ordering, so memory barriers are required between multiple writes to I/O registers where order is important. The same is also true for read ordering.

Use the memory barrier to prevent writes from being collapsed in the write buffer, that is, to prevent bytes, shorts, and ints from being merged into one 64-bit write.

Alpha AXP CPUs require that data caches be transparent. Because there is no way to explicitly flush the data cache on an Alpha AXP platform, you need not call `mb` before or after. The following code fragment illustrates the use of a memory barrier:

```
•
•
•
      bcopy (data, DMA_buffer, nbytes);
      mb();
      device->csr = GO;
      mb();
•
•
```

```
    •
```

Another example is presented in the following code fragment:

```
    •
    •
    •
device_intr()
{
     mb();
     bcopy (DMA_buffer, data, nbytes);
    /* If we need to update a device register, do: */
     mb();
     device->csr = DONE;
     mb();
}
```

Another way to look at this issue is to recognize that Alpha AXP CPUs maintain cache coherency for you. However, Alpha AXP CPUs are free to do the cache coherency in any manner and time. The events that cause you to want to read buffers, or the events you want to trigger to release a buffer you have written, are not guaranteed to occur at a time consistent with when the hardware maintains cache coherency. You need the memory barrier to achieve this synchronization.

2.4.5.11 Using the Compiler Keyword volatile

The `volatile` keyword prevents compiler optimizations from being performed on data structures and variables; such actions could result in unexpected behavior. The following example shows the use of the `volatile` keyword on a device register structure:

```
typedef volatile struct {
    unsigned adder;
    unsigned pad1;
    unsigned data;
    unsigned pad2;
    unsigned csr;
    unsigned pad3;
    unsigned test;
    unsigned pad4;
} CB_REGISTERS;
```

The following variables or data structures should be declared as volatile by 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

The purpose of using the `volatile` keyword on the example data structure is to prevent compiler optimizations from being performed on it; such actions could result in unexpected behavior.

2.4.6 Loadable Driver Issues

Although some CPUs do not support loadable drivers, you can implement device drivers that are both loadable and static. In fact, you might want to implement the loadable sections of the device driver now in anticipation that the CPU will eventually support loadable drivers. Chapter 10 shows how to implement a character device driver that is both loadable and static. This driver is implemented in such a way that the differences between the loadable and static versions of the driver are identified at run time and not at compile time.

2.4.7 Memory Barriers

The Alpha AXP architecture, unlike traditional CPU architectures, does not guarantee read/write ordering. That is, the memory subsystem is free to complete read and write operations in any order that is optimal, without regard for the order in which they were issued. Read/write ordering is not the same as cache coherency, which is handled separately and is not an issue. The Alpha AXP architecture also contains a write buffer (as do many high-performance RISC CPUs, including the MIPS R3000). This write buffer can coalesce multiple writes to identical or adjacent addresses into a single write, effectively losing earlier write requests. Similarly, multiple reads to the same identical or adjacent addresses can be coalesced into a single read.

This coalescing has implications for multiprocessor systems, as well as systems with off-board I/O or DMA engines that can read or modify memory asynchronously or that can require multiple writes to actually issue multiple data items. The `mb` (memory barrier) interface guarantees ordering of operations. The `mb` interface is derived from the `MB` instruction, which is described in the *Alpha Architecture Reference Manual.*

The `mb` interface is a superset of the `wbflush` interface used on MIPS CPUs. For compatibility, `wbflush` is aliased to `mb` in DEC OSF/1 Alpha AXP.

You call `mb` in a device driver under the following circumstances:

- To force a barrier between load/store operations
- After the CPU has prepared a data buffer in memory and before the device driver tries to perform a DMA out of the buffer
- Before attempting to read any device CSRs after taking a device interrupt
- Between writes

Each of these is briefly discussed in the following sections.

Note

> Device drivers and the DEC OSF/1 operating system are the primary users of the mb interface. However, some user programs, such as a graphics program that directly maps the frame buffer and manipulates registers, might need to call mb. The DEC OSF/1 operating system does not provide a C library interface for mb. User programs that require use of mb should use the following asm construct:

```
#include <c_asm.h>

asm ("mb");
```

2.4.7.1 Forcing a Barrier Between Load/Store Operations

You can call the mb interface to force a barrier between load/store operations. This call ensures that all previous load/store operations access memory or I/O space before any subsequent load/store operations. The following call to mb ensures that the first register is physically written before the load attempts to read the second register. The call assumes that `device` is an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). You can perform standard C mathematical operations on the I/O handle. For example, this code fragment adds the I/O handle and general register 1 and general register 2 in the calls to `write_io_port` and `read_io_port`.

```
   •
   •
   •
#define csr 0   /* Command/Status register */
#define reg1 8  /* General register 1 */
#define reg2 16 /* General register 2 */

io_handle_t device;
   •
   •
   •
write_io_port(device + reg1, 8, 0, value); [1]
mb (); [2]
next_value = read_io_port(device + reg2, 8, 0); [3]
   •
   •
   •
```

[1] Writes the first value to general register 1 by calling the `write_io_port` interface. In previous versions of this book, this code fragment directly accessed the device registers. The code fragment now follows the recommendation of accessing the device registers by calling

write_io_port. See Section 9.7.1.2 for a discussion of write_io_port.

[2] Calls the mb interface to ensure that the write of the value is completed.

[3] Reads the new value from general register 2 by calling the read_io_port interface. In previous versions of this book, this code fragment directly accessed the device registers. The code fragment now follows the recommendation of accessing the device registers by calling the read_io_port interface. See Section 9.7.1.1 for a discussion of read_io_port.

2.4.7.2 After the CPU Has Prepared a Data Buffer in Memory

You call the mb interface after the CPU has prepared a data buffer in memory and before the device driver tries to perform a DMA out of the buffer. You also call mb in device drivers that perform a DMA into memory and before using the data in the DMA buffer. The following calls to mb ensure that data is available (out of memory pipelines/write buffers) and that the data cache is coherent. The call assumes that device is an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). You can perform standard C mathematical operations on the I/O handle. For example, this code fragment adds the I/O handle and the command/status register in the call to read_io_port.

```
   •
   •
   •
#define csr 0    /* Command/Status register */
#define reg1 8   /* General register 1 */
#define reg2 16 /* General register 2 */

io_handle_t device;
   •
   •
   •
bcopy (data, dma_buf); [1]
mb (); [2]
write_io_port(device + csr, 8, 0, START_DMA); [3]

/* or */ [4]

if (read_io_port(device + csr, 8, 0) | DMA_DONE) {
    mb ();
    bcopy (dma_buf, data);
}
   •
   •
   •
```

[1] Writes the data into the DMA buffer.

[2] Calls the mb interface to ensure that the write of the data is completed.

[3] Issues the start command to the device.

[4] This sequence of code presents another way to accomplish the same thing. In previous versions of this book, this code fragment directly accessed the device registers. The code fragment now follows the recommendation of accessing the device registers by calling the `read_io_port` interface. See Section 9.7.1.1 for a discussion of `read_io_port`.

If the DMA is finished:

- Calls the mb interface to ensure that the buffer is correct
- Gets the data from the DMA buffer

2.4.7.3 Before Attempting to Read Any Device CSRs

You call the mb interface before attempting to read any device CSRs after taking a device interrupt. The call assumes that `device` is an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). You can perform standard C mathematical operations on the I/O handle. For example, this code fragment adds the I/O handle and the command/status register in the call to `read_io_port`.

```
   •
   •
   •
#define csr 0    /* Command/Status register */
#define reg1 8   /* General register 1 */
#define reg2 16 /* General register 2 */

io_handle_t device;
   •
   •
   •
device_intr()
{
      mb(); [1]
      stat = read_io_port(device + csr, 8, 0); [2]

       /* or */ [3]

       mb();
       bcopy (dma_buf, data);
}
   •
   •
   •
```

[1] Calls the mb interface to ensure that the device CSR write completes.

[2] Reads the status from the device. In previous versions of this book, this code fragment directly accessed the device registers. The code fragment now follows the recommendation of accessing the device registers by calling the `read_io_port` interface. See Section 9.7.1.1 for a discussion of `read_io_port`.

[3] This sequence of code presents another way to accomplish the same thing. It calls the `mb` interface to ensure that the buffer is correct. It then gets the data from the DMA buffer by calling the `bcopy` interface.

Note

DEC OSF/1 on Alpha AXP provides a memory barrier in the interrupt stream before calling any device interrupt service interfaces. Thus, a call to `mb` is not strictly necessary in the device interrupt case. For performance reasons in the device interrupt case, you can omit the call to `mb`.

2.4.7.4 Between Writes

You call the `mb` interface between writes if you do not want a write buffer to collapse the writes (merge bytes/shorts/ints/quads or reorder). The call assumes that `device` is an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). You can perform standard C mathematical operations on the I/O handle. For example, this code fragment adds the I/O handle and general register 1 and general register 2 in the calls to `write_io_port`.

```
   •
   •
   •
#define csr 0   /* Command/Status register */
#define reg1 8  /* General register 1 */
#define reg2 16 /* General register 2 */

io_handle_t device;
   •
   •
   •
        *ptr = value; [1]
        mb (); [2]
        *(ptr+1) = value2; [3]

/* or */ [4]

        write_io_port(device + reg1, 8, 0, value);
        mb ();
        write_io_port(device + reg2, 8, 0, value2);
   •
   •
   •
```

1 Writes the first location.

2 Calls the mb interface to force a write out of the write buffer.

3 Writes the second location.

4 This sequence of code illustrates an example more specifically tailored to device drivers. Note that this use of mb is exactly equivalent to wbflush.

In previous versions of this book, this code fragment directly accessed the device registers. The code fragment now follows the recommendation of accessing the device registers by calling the write_io_port interface. See Section 9.7.1.2 for a discussion of write_io_port. This sequence:

- Writes the first location by calling write_io_port.
- Calls mb to force a write out of the write buffer
- Writes the second location by calling write_io_port a second time.

Note

The *Alpha Architecture Reference Manual* (1992 edition) has a technical error in the description of the MB instruction. It specifies that MB is needed only on multiprocessor systems. This statement is not accurate. The MB instruction must be used in any system to guarantee correctly ordered access to I/O registers or memory that can be accessed via off-board DMA. All such off-board I/O and DMA engines are considered ''processors'' in the Alpha AXP's definition of multiprocessor.

2.5 Creating a Device Driver Development and Kitting Environment

When you are ready to write your driver, you will probably want to design a logical directory structure to hold the different files associated with the driver. Chapter 11 discusses the device driver configuration models and provides examples of driver development environments suitable for the third-party and traditional device driver configuration models.

2.6 Porting ULTRIX Device Drivers to the DEC OSF/1 Operating System

This section discusses the tasks you need to perform when porting device drivers from the ULTRIX operating system (running on Digital hardware) to the DEC OSF/1 operating system (also running on Digital hardware). The section does not discuss how to port drivers running on other UNIX operating systems, such as System V, or running on other hardware platforms, such as Sun Microsystems. Specifically, you need to:

- Write test suites
- Check header files
- Review device driver configuration
- Check driver interfaces
- Check kernel interfaces
- Check data structures

These tasks are discussed in the following sections.

2.6.1 Writing Test Suites

Porting a device driver 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 and the operating system you are porting from (the source machine and the source operating system). If the test suite does not exist, you need to write a full test suite for that device on the source machine and the source operating system. For example, if you port a device driver written for a Digital CPU running the ULTRIX operating system, write the full test suite on that Digital CPU.

Write the test suite so that only minimal changes are necessary when you move it to the Digital CPU running the DEC OSF/1 operating system you are porting to (the target machine and the target operating system). 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 source operating system and the target machine and target operating system on a network or make them accessible through a common interface, such as the Small Computer System Interface (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 header or binary executable files because these files on the source machine will probably not be compatible on the target machine.

2.6.2 Checking Header Files

Check the header files in the driver you want to port with those in the DEC OSF/1 device drivers. Section 3.1 provides information on the header files used in DEC OSF/1, including those header files related to loadable device drivers. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the header files most frequently used by DEC OSF/1 device drivers.

The following example summarizes the differences in the way header files are included in device drivers on ULTRIX and DEC OSF/1 systems:

```
/* Header Files Included in ULTRIX */ [1]
#include "../h/types.h"

/* Header Files Included in DEC OSF/1 */ [1]
#include <sys/types.h>
```

[1] This example shows that drivers written for DEC OSF/1 use left (<) and right (>) angle brackets, instead of the begin (") and end (") quotes used in ULTRIX pathnames. Note also that the location of the file has changed for DEC OSF/1.

2.6.3 Reviewing Device Driver Configuration

Device driver configuration on ULTRIX systems follows what this book refers to as the traditional model. The DEC OSF/1 operating system supports both the traditional model and the third-party device driver configuration model, as described in Chapter 11. Chapter 13 gives examples on configuring drivers by using both the third-party and the traditional models.

2.6.4 Checking Driver Interfaces

You need to check the driver interfaces used in ULTRIX device drivers and to compare them with those used in DEC OSF/1 device drivers. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the driver interfaces used by DEC OSF/1 device drivers. Use this information to compare the interface's behavior, number and type of arguments, return values, and so forth with its associated ULTRIX driver interface.

2.6.5 Checking Kernel Interfaces

You need to check the kernel interfaces used in ULTRIX device drivers and to compare them with those used in DEC OSF/1 device drivers. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the kernel interfaces used by DEC OSF/1 device drivers. Use this information to compare the interface's behavior, number and type of arguments, return values, and so forth with its associated ULTRIX kernel interface. Table 2-3 lists some of these differences.

Table 2-3: Highlights of Differences Between DEC OSF/1 and ULTRIX Kernel Interfaces

Kernel Interface	Remarks
`BADADDR`	Device drivers written for ULTRIX systems use the `BADADDR` interface in the autoconfiguration section to determine if a device is present on the system. You can use `BADADDR` in device drivers that are configured statically on DEC OSF/1. However, you cannot use `BADADDR` in device drivers that are configured as loadable on DEC OSF/1. Loadable device drivers cannot call the `BADADDR` interface because it is usable only in the early stages of system booting. Loadable device drivers are loaded during the multiboot stage. If your driver is both loadable and static, you can declare a variable and use it to control any differences in the tasks performed by the loadable and static drivers. Thus, the static driver can still call `BADADDR`. Section 4.1.6.1 shows how `noneprobe` uses such a variable called *`none_is_dynamic`*.
`bufflush`	ULTRIX BSD device drivers written for platforms based on MIPS use the `bufflush` interface. This interface is not used for platforms based on Alpha AXP. Therefore, delete this interface from your device driver.
`KM_ALLOC`	Replace calls to `KM_ALLOC` with calls to `kalloc`. Note that `kalloc` does not zero out memory.
`KM_FREE`	Replace calls to `KM_FREE` with calls to `kfree`.
`printf` interfaces	ULTRIX device drivers can call `cprintf`, `mprintf`, `printf`, and `uprintf`. DEC OSF/1 device drivers can call `printf` and `uprintf`.
`selwakeup`	The `selwakeup` interface is not used in DEC OSF/1. Replace calls to `selwakeup` with calls to `select_wakeup`. Note that the formal parameters for the two interfaces are different.

Table 2-3: (continued)

Kernel Interface	Remarks
useracc	The useracc interface is obsolete on DEC OSF/1. If you called useracc with vslock, replace both interfaces with a call to vm_map_pageable. Typically, the useracc interface was called by the driver's *xx*strategy interface. In most cases, the driver's *xx*strategy interface would be called indirectly from the physio interface (in response to read and write system calls or file-system access). In DEC OSF/1, the physio interface verifies access permissions to the user buffer and locks down the memory. For this reason, in existing drivers that may have historically called useracc, it is no longer necessary to perform such a check and subsequent locking of memory with vslock. The general rule is that any interface called from the physio interface (that is, the driver's *xx*strategy interface) should not call useracc or its replacement vm_map_pageable because the physio interface performs those functions. If you called useracc prior to accessing user data, replace it with calls to copyin and copyout to access user space. Calling useracc simply verifies access permissions to the specified memory at the time the useracc interface is called. It is possible that immediately after the useracc interface has returned back to the driver that the corresponding memory could be invalid. The memory could be invalid if it was swapped out or otherwise remapped by the virtual memory management portion of the kernel. Therefore, a driver should not assume that the memory region whose access permissions are verified through a call to useracc is persistent. In DEC OSF/1, calls to useracc are either unnecessary (as previously explained) or should be replaced by direct calls to vm_map_pageable. This interface performs functions for both verifying access permissions and for locking down the memory so that it cannot be invalidated until it is unlocked by subsequent calls to vm_map_pageable for this memory reigon.
vslock	The vslock interface is obsolete on DEC OSF/1. Therefore, replace calls to vslock with calls to vm_map_pageable.
vsunlock	The vsunlock interface is obsolete on DEC OSF/1. Therefore, replace calls to vsunlock with calls to vm_map_pageable.

2.6.6 Checking Data Structures

You need to check the data structures used in ULTRIX device drivers and compare them to those used in DEC OSF/1 device drivers. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the data structures used by DEC OSF/1 device drivers. Use this information to compare the data structure's members with its associated ULTRIX data structure. Table 2-4 lists some of these differences.

Table 2-4: Highlights of Differences Between DEC OSF/1 and ULTRIX Data Structures

Data Structure	Remarks
`uba_ctlr`	Replace references to the `uba_ctlr` structure and its associated members with references to the `controller` structure and its associated members.
`uba_device`	Replace references to the `uba_device` structure and its associated members with references to the `device` structure and its associated members. Make sure that the reference is to a slave, for example, a disk or tape drive. If the reference is to a controller, reference the `controller` structure, not the `device` structure.
`uba_driver`	Replace references to the `uba_driver` structure and its associated members with references to the `driver` structure and its associated members.

Analyzing the Structure of a Device Driver 3

Before implementing the sample device driver discussed in the previous chapter, you need to understand the sections of a DEC OSF/1 device driver. Analyzing the sections of a device driver gives you the opportunity to learn how to set up the device driver interfaces in preparation for writing your own device drivers. This chapter mentions some structures (for example, the `device` structure) you may not be familiar with yet. However, to learn how to set up the device driver interfaces, you do not need an intimate understanding of the structures. Chapter 7 discusses these structures.

The sections that make up a DEC OSF/1 device driver differ depending on whether the driver is a block, character, or network driver. Figure 3-1 illustrates the sections that a character device driver can contain and the possible sections for a block device driver. Device drivers are not required to use all of the sections and more complex drivers can have additional sections.

Both types of drivers contain:

- An include files section
- A declarations section
- An autoconfiguration support section
- A configure section (only for loadable drivers)
- An open and close device section
- An ioctl section
- An interrupt section

The block device driver can also contain a strategy section, a `psize` section, and a dump section.

The character device driver contains the following sections not contained in a block device driver:

- A read and write device section
- A reset section
- A stop section
- A select section
- A memory map section (only for CPUs that include map registers)

Each device driver section is described in the following sections. For convenience in referring to the names for the driver interfaces, the chapter uses the prefix *xx*. For example, `xx`*probe*...

Figure 3-1: Sections of a Character Device Driver and a Block Device Driver

Character Device Driver

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

Block Device Driver

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

3.1 Include Files Section

Data structures and constant values are defined in header files that you include in the include files section of the driver source code. The number and types of header files you specify in the include files section vary, depending on such things as what structures, constants, and kernel interfaces your device driver references. You need to be familiar with:

- The device driver header file
- The common driver header files
- The loadable driver header files
- The device register header file
- The `name_data.c` file

The following sections describe these categories of header files. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the header files most frequently used by DEC OSF/1 device drivers.

3.1.1 Device Driver Header File

The device driver header file contains `#define` statements for as many devices as are configured into the system. This file is generated by the `config` program during static configuration of the device driver. This file need not be included if you configure the driver as a loadable driver.

The `config` program creates the name for this file by using the name of the controller or device that you specify in the system configuration file. For example, if you specified `rd` as the device in the system configuration file, `config` creates a header file called `rd.h`. The following example shows the possible contents for the `rd.h` device driver header file:

```
#define NRD 1
```

The `config` program creates a compile time variable, in this example `NRD`, which defines how many devices exist on the system. This variable is set to the value zero (0) to indicate that no devices exist on the system. It is set to a specific number, for example 4, to indicate that four devices exist on the system. Many existing ULTRIX-based drivers use this compile-time variable in device driver code to refer to the number of this type of device on the system. This usage most frequently occurs in structure array declarations and in condition (for example, `if` and `while`) statements.

3.1.2 Common Driver Header Files

The following example lists the header files most frequently used by device drivers:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <io/common/devdriver.h>
#include <sys/uio.h>
#include <machine/cpu.h>
```

The example shows that device drivers should not use explicit pathnames. Using angle brackets (< and >) means you will not have to make changes to your device driver if the file path changes.

The following sections contain brief descriptions of the previously listed common driver header files.

3.1.2.1 The types.h Header File

The header file `/usr/sys/include/sys/types.h` defines system data types used to declare members in the data structures referenced by device drivers. Table 3-1 lists the system data types most frequently used by device drivers.

Table 3-1: System Data Types Frequently Used by Device Drivers

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

The `/usr/sys/include/sys/types.h` header file includes the file `/mach/machine/vm_types.h`. This file defines the data type `vm_offset_t`, which driver writers should use when addresses are treated as arithmetic quantities (that is, as ints and longs). The `vm_offset_t` data type is defined as `unsigned long` on Alpha AXP and as `unsigned` int on MIPS.

3.1.2.2 The errno.h Header File

The header file /usr/sys/include/sys/errno.h defines the error codes returned to a user process by a device driver. Examples of these error codes include EINVAL (invalid argument), ENODEV (no such device), and EIO (I/O error).

3.1.2.3 The devdriver.h Header File

The header file /usr/sys/include/io/common/devdriver.h defines structures, constants, data types, and external interfaces used by device drivers and the autoconfiguration software. Two opaque data types that you can use to make your device drivers more portable are io_handle_t and dma_handle_t. See Section 9.7 for a discussion of the I/O handle.

3.1.2.4 The uio.h Header File

The header file /usr/sys/include/sys/uio.h contains the definition of the uio structure. The kernel sets up and uses the uio structures to read and write data. Character device drivers include this file because they may reference the uio structure.

3.1.3 Loadable Driver Header Files

You need to include the following files related to loadable device drivers:

```
#include <sys/conf.h>
#include <sys/sysconfig.h>
```

The following sections contain brief descriptions of these files.

3.1.3.1 The conf.h Header File

The header file /usr/sys/include/sys/conf.h defines the bdevsw (block device switch) and cdevsw (character device switch) tables. This file should be included by loadable block and character device drivers because these drivers define and then initialize entry points for the bdevsw and cdevsw tables. In the case of loadable device drivers, the driver then passes the initialized structure to the bdevsw_add or cdevsw_add kernel interfaces. Section 8.2 describes the device switch tables.

3.1.3.2 The sysconfig.h Header File

The header file `/usr/sys/include/sys/sysconfig.h` defines operation codes and data structures used in loadable device driver configuration. The operation codes define the action to be performed by the driver configure interface. Examples of the operation types include configure, unconfigure, and query. This file also defines many of the constants that are shared between the device driver method and the drivers themselves. Within this file also appears the declaration of the data structure that is passed to and returned from the driver's configure interface. Section 3.4 shows how to set up the configure interface.

3.1.4 Device Register Header File

The device register header file contains any public declarations used by the device driver. This file usually contains the device register structure associated with the device. A device register structure is a C structure whose members map to the registers of some device. These registers are often referred to as the device's control status register or CSR addresses. The device driver writer creates the device register header file.

The following example shows a device register structure contained in a device register header file for a device driver written for a TURBOchannel test board:

```
typedef volatile struct {
    unsigned adder;
    unsigned pad1;
    unsigned data;
    unsigned pad2;
    unsigned csr;
    unsigned pad3;
    unsigned test;
    unsigned pad4;
} CB_REGISTERS;
```

The example shows the use of the compiler keyword `volatile`. Section 2.4.5.11 provides guidelines for which variables and data strucures device drivers should declare as volatile.

The device register structure is most often used in device drivers that directly access a device's CSR addresses. There are some CPU architectures that do not allow you to access the device CSR addresses directly. If you want to write your device driver to operate on both types of CPU architectures, you can write one device driver with the appropriate conditional compilation statements. You can also avoid the potentially confusing proliferation of conditional compilation statements by using the CSR I/O access kernel interfaces provided by DEC OSF/1 to read from and write to the device's CSR addresses. Because the CSR I/O access interfaces are designed to be CPU hardware independent, their use not only simplifies the readability of the driver, but also makes the driver more portable across different CPU architectures and different CPU types within the same architecture. Section 9.7.1 shows you how to use the CSR I/O access kernel interfaces to read from and write to the device's CSR addresses.

This technique of calling kernel interfaces to access a device's CSR addresses also requires defining the registers of a device, usually through constants that map to these device registers. The following example shows the device register definitions contained in a device register header file for a device driver that uses the kernel interfaces to access a device's CSR addresses:

```
/***************************************************
 * Define offsets to nvram device registers       *
 ***************************************************/
#define ENVRAM_CSR      0xc00  /* CSR */
#define ENVRAM_BAT      0xc04  /* Battery Disconnect */
#define ENVRAM_HIBASE   0xc08  /* Ext. Mem Config */
#define ENVRAM_CONFIG   0xc0c  /* EISA config reg */
#define ENVRAM_ID       0xc80  /* EISA ID reg */
#define ENVRAM_CTRL     0xc84  /* EISA control */
#define ENVRAM_DMA0     0xc88  /* DMA addr reg 0 */
#define ENVRAM_DMA1     0xc8c  /* DMA addr reg 1 */
```

3.1.5 The name_data.c File

The *name*`_data.c` file provides a convenient place to size the data structures and data structure arrays that device drivers use. In addition, the file can contain definitions that third-party driver writers might want their customers to change. This file is particularly convenient for third-party driver writers who do not want to ship device driver sources. The device driver writer creates the *name*`_data.c` file.

The *name* argument is usually based on the device name. For example, the `none` device's *name*`_data.c` file is called `none_data.c`. The `CB` device's *name*`_data.c` file is called `cb_data.c`. The `edgd` device's *name*`_data.c` file would be called `edgd_data.c`.

The *name*`_data.c` files supplied by Digital Equipment Corporation are found in the `/usr/sys/data` directory. The following example illustrates some of the *name*`_data.c` files contained in the `/usr/sys/data` directory for DEC OSF/1:

```
audit_data.c          ga_data.c           np_data.c
autoconf_data.c       gvp_data.c          pmap_data.c
binlog_data.c         gw_screen_data.c    presto_data.c
cam_data.c            gx_data.c           scc_data.c
cam_special_data.c    if_fta_data.c       scs_data.c
ci_data.c             if_fza_data.c       sysap_data.c
cippd_data.c          if_ln_data.c        tc_option_data.c
```

Third-party device driver writers can place their *name*`_data.c` file in a products directory with all of the other files they plan to ship to customers.

Figure 11-2 shows a sample directory structure for the fictitious driver development company, EasyDriver Incorporated.

3.2 Declarations Section

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

- Definitions of symbolic names
- Variable declarations
- Structure declarations
- Declarations of driver interfaces

The following example shows the declarations section for a device driver. The example provides declarations and initializations that would be provided in any device driver:

```
/* Definitions of symbolic names */
define MAX_XFR 4
   •
   •
   •
/* Variable declarations */
extern int hz;
   •
   •
   •
/* Structure declarations */
struct controller *cbinfo[NCB];
   •
   •
   •
/* Declarations of driver interfaces */
int cbstart(), cbprobe(), cbattach(), cbstrategy(), cbminphys();
   •
   •
   •
```

Note the compile time variable NCB is used to size the array of pointers to `controller` structures. This usage is an example of statically allocating data structures. Section 2.3 provides information to help you choose an appropriate data structure allocation technique.

3.3 Autoconfiguration Support Section

When DEC OSF/1 boots, the kernel determines what devices are connected to the computer. After finding a device, the kernel initializes it so that the device can be used at a later time. The probe interface determines if a particular device is present and the attach interface initializes the device.

The system performs a functionally equivalent process when a loadable driver is configured. A loadable driver, like the static driver, has a probe, attach, and possibly a slave interface. From the device driver writer's point of view, these interfaces are the same for static and loadable drivers.

The autoconfiguration support section of a device driver contains the code that implements these interfaces and the section applies to both character and block device drivers. It can contain:

- A probe interface
- A slave interface
- An attach interface

For loadable drivers, the autoconfiguration support section also contains a controller unattach or a device unattach interface, which is called when the driver is unloaded. You define the entry point for each of these interfaces in the `driver` structure. Section 7.6 describes the `driver` structure. The following sections show you how to set up each of these interfaces.

3.3.1 Setting Up the Probe Interface

The way you set up a probe interface depends on the bus on which the driver operates. The following code fragment shows you how to set up a probe interface for a driver that operates on a TURBOchannel bus:

```
xxprobe(addr, ctlr)
caddr_t addr;                /* System virtual address */
struct controller *ctlr;     /* Pointer to controller */
                             /* structure */
{
       /* Variable and structure declarations */
    •
    •
    •
       /* Code to perform necessary checks */
    •
    •
    •
}
```

The code fragment declares the two arguments associated with a probe interface that operates on a TURBOchannel bus. To learn what arguments you would specify for other buses, see the bus-specific device driver manual. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the probe interface.

Section 4.1.6.1 shows you how to implement a probe interface for the `/dev/none` device driver. Section 10.8.1 shows you how to implement a probe interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*probe interface.

3.3.2 Setting Up the Slave Interface

A device driver's slave interface is called only for a controller that has slave devices connected to it. This interface is called once for each slave attached to the controller. The way you set up a slave interface depends on the bus on which the driver operates. The following code fragment shows you how to set up a slave interface for a driver that operates on a TURBOchannel bus:

```
xxslave(device, addr)
struct device *device;  /* Pointer to device structure */
caddr_t addr;           /* System virtual address */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to check that the device is valid */
   •
   •
   •
}
```

The code fragment declares the two arguments associated with a slave interface that operates on a TURBOchannel bus. To learn what arguments you would specify for other buses, see the bus-specific device driver manual. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the slave interface. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*slave interface.

3.3.3 Setting Up the Attach Interface

A device driver's attach interface establishes communication with the device. There are two attach interfaces: an *xx*cattach interface for controller-specific initialization (called once for each controller) and an *xx*dattach interface for device-specific initialization (called once for each slave device connected to a controller). The following code fragment shows you how to set up an *xx*dattach interface:

```
xxdattach(device)
struct *device; /* Pointer to device structure */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to perform tasks associated with establishing */
```

```
        /* communication with the device */
    •
    •
    •
}
```

The code fragment declares the argument associated with a device attach interface. If the controller attach interface were used, you would be passed a pointer to a `controller` structure. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the device attach interface.

Section 4.1.6.2 shows you how to implement an attach interface for the `/dev/none` device driver. Section 10.8.2 shows you how to implement an attach interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with the *xx*`cattach` and *xx*`dattach` interfaces.

3.3.4 Setting Up the Controller Unattach Interface

A device driver's controller unattach interface removes the specified `controller` structure from the list of controllers it handles. The following code fragment shows you how to set up a controller unattach interface:

```
xxctrl_unattach(bus_struct, ctlr_struct)
struct bus *bus_struct;          /* Pointer to bus structure */
struct controller *ctlr_struct; /* Pointer to controller */
                                /* structure */
{
        /* Variable and structure declarations */
    •
    •
    •
        /* Code to perform controller unattach tasks */
    •
    •
    •
}
```

The code fragment declares the two arguments associated with a controller unattach interface. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the controller unattach interface.

Section 4.1.6.3 shows how to implement a controller unattach interface for the `/dev/none` device driver. Section 10.8.3 shows how to implement a controller unattach interface for a character device driver that operates on a TURBOchannel bus.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*`ctrl_unattach` interface.

3.3.5 Setting Up the Device Unattach Interface

A device driver's device unattach interface removes the specified `device` structure from the list of devices it handles. The following code fragment shows you how to set up a device unattach interface:

```
xxdev_unattach(ctlr_struct, dev_struct)
struct controller *ctlr_struct;  /* Pointer to controller */
                                 /* structure */
struct device *dev_struct;       /* Pointer to device */
                                 /* structure */
{
      /* Variable and structure declarations */
   •
   •
   •
      /* Code to perform device unattach tasks */
   •
   •
   •
}
```

The code fragment declares the two arguments associated with a device unattach interface that operates on a TURBOchannel bus. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the device unattach interface.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*`dev_unattach` interface.

3.4 Configure Section

The configure section applies to the loadable versions of both character and block device drivers and it contains a configure interface. A device driver's configure interface is called as a result of a user-level request to dynamically load a device driver. The following code fragment shows you how to set up a configure interface:

```
xx_configure(optype, indata, indatalen, outdata, outdatalen)
sysconfig_op_t optype;       /* Configure operation */
device_config_t *indata;     /* Input data structure */
size_t indatalen;            /* Size of input data */
                             /* structure */
```

```
device_config_t *outdata;      /* Output data structure */
size_t outdatalen;             /* Output data structure */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to perform configure tasks */
   •
   •
   •
}
```

The code fragment declares the five arguments associated with a configure interface. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the configure interface.

Section 4.1.7.1 shows you how to set up the configure interface for the `/dev/none` device driver. Section 10.9.1 shows you how to implement a configure interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*`_configure` interface.

3.5 Open and Close Device Section

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

- An open interface
- A close interface

You specify the entry for the driver's open and close interfaces in the `bdevsw` for block device drivers and the `cdevsw` for character device drivers. Section 8.2 describes the device switch tables. The following sections discuss how to set up each of these interfaces.

3.5.1 Setting Up the Open Interface

A device driver's open interface is called as the result of an `open` system call. The following code fragment shows you how to set up an open interface:

```
xxopen(dev, flag, format)
dev_t dev;  /* Major/minor device number */
int flag;   /* Flags from /usr/sys/h/file.h */
int format; /* Format of special device */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to open the device */
   •
   •
   •
}
```

The code fragment declares the three arguments associated with an open interface that operates on any bus. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the open interface.

Section 4.1.8.1 shows you how to implement an open interface for the /dev/none device driver. Section 10.10.1 shows you how to implement an open interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*open interface.

3.5.2 Setting Up the Close Interface

The open interface is called every time that any user initiates an action that invokes the open system call. The close interface, however, is called only when the last user initiates an action that closes the device. The reason for this difference is to allow the driver to take some special action when there is no work left to perform. The following code fragment shows you how to set up a close interface:

```
xxclose(dev, flag, format)
dev_t dev;  /* Major/minor device number */
int flag;   /* Flags from /usr/sys/h/file.h */
int format; /* Format of special device */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to correctly close the device */
   •
   •
   •
}
```

The code fragment declares the three arguments associated with a close interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the close interface.

Section 4.1.8.2 shows you how to implement a close interface for the `/dev/none` device driver. Section 10.10.2 shows you how to implement a close interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an `xxclose` interface.

3.6 Read and Write Device Section

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

- A read interface
- A write interface

You specify the entry for the driver's read and write interfaces in the `cdevsw` table. Section 8.2.1 describes the `cdevsw` table. The following sections discuss how to set up both of these interfaces.

3.6.1 Setting Up the Read Interface

The read interface is called from the I/O system as the result of a `read` system call. The following code fragment shows you how to set up a read interface:

```
xxread(dev, uio)
dev_t dev;       /* Major/minor device number */
struct uio *uio; /* Pointer to uio structure */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to read data from the device */
   •
   •
   •
}
```

The code fragment declares the two arguments associated with a read interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the read interface.

Section 4.1.9.1 shows you how to implement a read interface for the /dev/none device driver. Section 10.11.1 shows you how to implement a read interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*read interface.

3.6.2 Setting Up the Write Interface

The write interface is called from the I/O system as the result of a write system call. The following code fragment shows you how to set up a write interface:

```
xxwrite(dev, uio)
dev_t dev;        /* Major/minor device number */
struct uio *uio; /* Pointer to uio structure */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to write data to the device */
}
```

The code fragment declares the two arguments associated with a write interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the write interface.

Section 4.1.9.2 shows you how to implement a write interface for the /dev/none device driver. Section 10.11.2 shows you how to implement a write interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*write interface.

3.7 The ioctl Section

The ioctl interface typically performs all device-related operations other than read or write operations. A device driver's ioctl interface 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 interface. You specify the entry for the driver's ioctl interface in the cdevsw table for character drivers and in the bdevsw table for block device drivers. Section 8.2 describes the device switch tables. The following code fragment shows you how to set up an ioctl interface:

```
xxioctl(dev, cmd, data, flag)
dev_t dev;             /* Major/minor device number */
unsigned int cmd;      /* The ioctl command */
caddr_t data;          /* ioctl command-specified data */
int flag;              /* Access mode of the device */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to perform device-related operations */
   •
   •
   •
}
```

The code fragment declares the four arguments associated with an `ioctl` interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the `ioctl` interface.

Section 4.1.11 shows you how to implement an `ioctl` interface for the `/dev/none` device driver. Section 10.14 shows you how to implement an `ioctl` interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an *`xxioctl`* interface.

3.8 Strategy Section

The strategy section applies to block device drivers and contains a `strategy` interface. However, character device drivers can also contain a `strategy` interface that is called by the character driver's `read` and `write` interfaces.

The strategy interface performs block I/O for block devices and initiates read and write operations for character devices. You specify the entry point for the `strategy` interface in the `bdevsw` table. Section 8.2.2 describes the `bdevsw` table. For character drivers, you do not specify the entry point for the `strategy` interface because it is called only by the character driver's `read` and `write` interfaces. There is no entry point defined in the `cdevsw` table. The following code fragment shows you how to set up a `strategy` interface:

```
xxstrategy(bp)
struct buf *bp; /* Pointer to buf structure */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to perform block I/O or read/write operations */
   •
   •
   •
}
```

The code fragment declares the argument associated with a strategy interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the strategy interface.

Section 10.12 shows you how to implement a strategy interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an `xxstrategy` interface.

3.9 Stop Section

The stop section applies only to character device drivers and it contains a stop interface. The stop interface is used by terminal device drivers to suspend transmission on a specified line. You specify the entry for a driver's stop interface in the `cdevsw` table. Section 8.2.1 describes the `cdevsw` table.

The following code fragment shows you how to set up a stop interface:

```
xxstop(tp, flag)
struct tty *tp; /* Pointer to tty structure */
int flag;       /* Output flag */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to suspend transmission on the specified line */
   •
   •
   •
}
```

The code fragment declares the two arguments associated with a stop interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the stop interface.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*`stop` interface.

3.10 Reset Section

The reset section applies only to character device drivers and it contains a reset interface. The reset interface is used to force a device reset to place the device in a known state after a bus reset. You specify the entry for a driver's reset interface in the `cdevsw` table. Section 8.2.1 describes the `cdevsw` table. The following code fragment shows you how to set up a reset interface:

```
xxreset(busnum)
int busnum; /* Logical unit number of bus */
{
        /* Variable and structure declarations */
    •
    •
    •
        /* Code to force the device to reset */
    •
    •
    •
}
```

The code fragment declares the argument associated with a reset interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the reset interface.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*`reset` interface.

3.11 Interrupt Section

The interrupt section applies to both character and block device drivers and it contains an interrupt service interface (ISI). The interrupt service interface is called as a result of a hardware interrupt. For static drivers, you specify the entry point for the interrupt service interface in the system configuration file. Section 12.2 describes the system configuration file. For loadable drivers, you specify the entry point for the interrupt service interface through the `handler_add` and `handler_enable` kernel interfaces.

The following code fragment shows you how to set up an interrupt service interface:

```
xxintr(parameter)
caddr_t parameter; /* Logical controller number */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to handle a hardware interrupt */
   •
   •
   •
}
```

The code fragment declares the argument associated with an interrupt service interface that operates on any bus. The code fragment also sets up the sections where you declare local variables and structures and where you write the code to implement the interrupt service interface.

Section 10.16 shows you how to implement an interrupt service interface for a character device driver that operates on a TURBOchannel bus. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an `xxintr` interface. It also provides information on the handler interfaces.

3.12 Select Section

The select section applies only to character device drivers and it contains a select interface. A device driver's select interface is called to determine whether data is available for reading and whether space is available for writing data. You specify the entry point for a driver's select interface in the `cdevsw` table. Section 8.2.1 describes the `cdevsw` table. Because the `/dev/none` and `/dev/cb` device drivers do not implement a select section, the following code fragment not only shows you how to set up a select interface, but it also illustrates some typical tasks. For example purposes, the example refers to some `xx` device.

```
xxselect(dev, events, revents, scanning)
 dev_t dev;        1
 short *events;    2
 short *revents;   3
 int scanning;     4
{
        int nread; 5
        struct xxdevstruct *xxdevice; 6

        xxdevice = xxdevs[minor(dev)]; 7

/*****************************************************
 *       Poll for input reads                        *
 *****************************************************/
        if (*events & POLLNORM) { 8
                if (scanning) { 9
                        nread = xxnread(dev); 10
                        if (nread > 0)
                                *revents |= POLLNORM; [11]
                        else
                                select_enqueue(xxdevice); [12]
                } else
                        select_dequeue(xxdevice); [13]
        }
/***************************************************
 *      Poll for output write                      *
 ***************************************************/
        if (*events & POLLOUT) { [14]
                if (scanning) { [15]
                        if (xxnwrite(dev)) [16]
                                *revents |= POLLOUT; [17]
                        else
                                select_enqueue(xxdevice); [18]
                } else
                        select_dequeue(xxdevice); [19]
        }
        return (0); [20]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific `xx` device. The minor device number is used to determine the logical unit number for the `xx` device on which the select call is to be performed.

[2] Declares a pointer to an argument that specifies the events to be polled on. This argument is an input to the device driver. The caller of select is the user level process that issued the `select` system call. The `select` system call then calls the driver's *xx*`select` interface. The kernel can set this argument to the bitwise inclusive OR of one or more of the polling bit masks defined in the file `/usr/sys/include/sys/poll.h`: `POLLNORM`, `POLLOUT`, and `POLLPRI`.

[3] Declares a pointer to an argument that specifies the events that are ready. The driver writer sets this value in the driver's *xx*`select` interface. The driver writer can set this argument to the bitwise inclusive OR of one or more of the polling bit masks defined in `/usr/sys/include/sys/poll.h`: `POLLNVAL`, `POLLHUP`, `POLLNORM`, and `POLLOUT`.

[4] Declares an argument that specifies the initiation and termination of a select call. The kernel sets this argument to the value 1 to indicate initiation of a select call. The caller of select is the user level process that issued the `select` system call. The `select` system call then calls the driver's *xx*`select` interface.

[5] Declares a variable to contain the number of characters available for input.

6 Declares a pointer to a data structure that is specific to the xx driver.

7 For the purposes of this example, assume that the xx driver has declared a static array of xx device-specific data structures, one structure for each instance of the device. Assume that the array is called xxdevs.

In this line, a pointer is established to point to the specific xxdevstruct data structure associated with this xx device. Note the use of the minor interface to obtain the device minor number associated with this xx device.

8 Determines if the kernel set the read input select bit, which indicates that the caller of select wants to know if input data is available on this device. The caller of select is the user level process that issued the select system call. The select system call then calls the driver's *xx*select interface.

9 If the kernel sets this argument to the value 1 (true), then a select call is being initiated.

10 For the purpose of this example, assume that the xx driver has a separate interface called xxnread, which returns the count of the number of characters available for input.

11 If the count is greater than zero (0), there are characters available for input. Set the read input select bit in the pointer to the *revents* argument. The select call can complete without waiting for input to be available.

12 If the count is not greater than zero (0), there are no characters available for input. Call the select_enqueue interface to allow the select call to remember that a caller of select wants to be notified when input is available to be read. This interface takes one argument: a pointer to a sel_queue_t structure. One of the members of this structure is a pointer to an event. Thus, this line passes the pointer to the xxdevice structure, which identifies this instance of the xx device. The select_enqueue interface adds the current thread to the list of threads waiting for a select event on this xx device. When input is available, the select can complete.

At a different interface in the xxdriver (typically, either the interrupt section or an interface called by the interrupt section) when new input has been received on the device, the driver calls the select_wakeup interface. The driver passes to it the same parameter passed to select_enqueue to notify the upper levels of the select system call that the caller of select can now be notified that the driver has new input available to be read.

13 Executes when the kernel sets the *scanning* parameter to the value zero (0). This indicates that the upper level select system call code is no longer interested in being notified when input is available on this device.

The `xxdriver` calls the `select_dequeue` interface to remove any instances of this `xx` device registered as waiting for notification of input.

14 Determines if the kernel set the write output select bit, which indicates that the caller of select wants to know if the device is ready to accept data to be output. Typically, this involves verifying that the device's output buffers have sufficient space to accept additional characters to be transmitted.

15 If the kernel sets the *`scanning`* argument to the value 1 (true), then a select call is being initiated.

16 For the purpose of this example, assume that the `xx` driver has a separate interface called `xxnwrite`, which returns a nonzero value if the device is in a state where it is ready to output data.

17 To indicate that this instance of the `xx` device is ready to accept additional output, sets the *`revents`* argument to the polling bit `POLLOUT`.

18 The device is not ready to accept additional output. Therefore, the `xx` driver calls the `select_enqueue` interface to cause the select call to later be notified when the device is ready to accept output. At a different interface in the `xxdriver` (typically, either the interrupt section or an interface called by the interrupt section) after previous output transmission has completed, the driver calls the `select_wakeup` interface. The driver passes to it the same parameter passed to `select_enqueue` to notify the upper levels of the `select` system call that the caller of select can now be notified that the driver has new output available to be written.

19 If the kernel sets the *`scanning`* argument to the value zero (false), then a select call is being terminated. This indicates that the upper level `select` system call code is no longer interested in being notified when the device is ready to accept output characters. The `xxdriver` calls the `select_dequeue` interface to remove any instances of this `xx` device registered as waiting for notification of output ready status.

20 The value zero (0) is returned to indicate that the driver's *`xx`*`select` interface completed successfully.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *`xx`*`select` interface.

3.13 Dump Section

The dump section applies only to block device drivers and it contains a dump interface. A device driver's dump interface is called to copy system memory to the dump device. You specify the entry point for a driver's dump interface in the bdevsw table. Section 8.2.2 describes the bdevsw table. The following code fragment shows you how to set up a dump interface:

```
xxdump(dumpdev)
dev_t dumpdev;  /* Device to dump system memory to */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to copy system memory to the dump device */
   •
   •
   •
}
```

The code fragment declares the argument associated with a dump interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the dump interface.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an *xx*dump interface.

3.14 The psize Section

The psize section applies only to block device drivers and it contains a psize interface. A device driver's psize interface is called to return the size of a disk partition. You specify the entry point for a driver's psize interface in the bdevsw table. Section 8.2.2 describes the bdevsw table. The following code fragment shows you how to set up a psize interface:

```
xxpsize(dev)
dev_t dev;   /* Device and partition for which size */
             /* is requested */
{
     /* Variable and structure declarations */
   •
   •
   •
     /* Code to return the size of a disk partition */
   •
   •
   •
}
```

The code fragment declares the argument associated with a psize interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the psize interface.

Writing Device Drivers, Volume 2: Reference provides a reference (man) page that gives additional information on the arguments and tasks associated with an `xxpsize` interface.

3.15 Memory Map Section

Some Alpha AXP CPUs do not support an application's use of the `mmap` system call. Therefore, if you are writing a device driver that operates on such CPUs, you need to use a mechanism other than the memory map interface. The memory map section applies only to character device drivers, and it contains a memory map interface. The memory map interface is invoked by the kernel as the result of an application calling the `mmap` system call. You specify the entry for the driver's `xxmmap` interface in the `cdevsw` table. Section 8.2.1 describes the `cdevsw` table. The following code fragment shows you how to set up a memory map interface:

```
xxmmap(dev, offset, prot)
dev_t dev;     /* Major/minor device number */
off_t offset; /* Offset into device memory */
int prot;      /* Protection flag */
{
     /* Variable and structure declarations */
  •
  •
  •
     /* Code to map kernel space to user space */
  •
  •
  •
}
```

The code fragment declares the three arguments associated with a memory map interface that operates on any bus. It also sets up the sections where you declare local variables and structures and where you write the code to implement the memory map interface. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page that gives additional information on the arguments and tasks associated with an `xxmmap` interface.

Coding, Configuring, and Testing a Device Driver 4

The previous chapter showed you how to set up the interfaces associated with a device driver. This chapter shows you how to implement some of those interfaces for the `/dev/none` device driver. Specifically, the chapter shows you how to:

- Code a device driver
- Configure a device drivier
- Test a device driver

4.1 Coding a Device Driver

The `/dev/none` device driver is a simple device driver that performs the following tasks:

- Determines if the `none` device exists on the system
- Checks to ensure that the open request is unique and marks the device as open
- Copies data from the specified address space to the device
- Obtains and clears the count of bytes previously written to the device
- Closes the device
- Implements the tasks associated with the loadable version of the driver

Note that the `/dev/none` driver is implemented as one driver that can be configured either as a loadable or static driver.

The `/dev/none` device driver references some structures (for example, the `driver` structure) you may not be familiar with yet. However, to understand this simple device driver you do not need an intimate understanding of the structures.

For convenience in learning how this driver is implemented, the source code is divided into parts. Table 4-1 lists the parts of the `/dev/none` device driver and the sections of the chapter where each is described. For those who prefer to read the `/dev/none` source code with inline comments, see Section B.1.

Table 4-1: Parts of the /dev/none Device Driver

Part	Section
The nonereg.h Header File	Section 4.1.1
Include Files Section	Section 4.1.2
Autoconfiguration Support Declarations and Definitions Section	Section 4.1.3
Loadable Driver Configuration Support Declarations and Definitions Section	Section 4.1.4
Loadable Driver Local Structure and Variable Definitions Section	Section 4.1.5
The Autoconfiguration Support Section	Section 4.1.6
Loadable Device Driver Section	Section 4.1.7
Open and Close Device Section	Section 4.1.8
Read and Write Device Section	Section 4.1.9
Interrupt Section	Section 4.1.10
The ioctl Section	Section 4.1.11

4.1.1 The nonereg.h Header File

The `nonereg.h` file is the device register header file for the `/dev/none` device driver. It contains public declarations and the device register structure for the `none` device. The following declarations are applicable to the loadable or static version of the driver:

```
#define DN_GETCOUNT    _IOR(0,1,int) 1
#define DN_CLRCOUNT    _IO(0,2)      2

#define NONE_CSR 0 3
```

1 Uses the `_IOR` macro to construct an `ioctl` macro called `DN_GETCOUNT`. The `_IOR` macro defines `ioctl` types for situations where data is transferred from the kernel into the user's buffer. Typically, this data consists of device control or status information returned to the application program. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page style descriptions of the `_IO`, `_IOR`, `_IOW`, and `_IOWR` `ioctl` macros.

   Section 4.1.11 shows how the `noneioctl` interface uses `DN_GETCOUNT`.

2 Uses the `_IO` macro to construct an `ioctl` macro called `DN_CLRCOUNT`. The `_IO` macro defines `ioctl` types for situations where no data is actually transferred between the application program and the kernel. For example, this could occur in a device control operation. Section 4.1.11 shows how the `noneioctl` interface uses `DN_CLRCOUNT`.

3 Defines the device register offset for the `none` device. All real devices have registers and the offsets defining the layout of these registers is usually defined in the device register header file. Although the `none` device is not a real device, the example shows how a device register offset would be defined.

   The `NONE_CSR` offset is a 64-bit read/write CSR/LED register.

4.1.2 Include Files Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. It identifies the following header files needed by the `/dev/none` device driver:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/tty.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/map.h>
#include <sys/buf.h>
#include <sys/vm.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/devio.h>
#include <hal/cpuconf.h>
#include <sys/exec.h>
#include <io/common/devdriver.h>
#include <sys/sysconfig.h>
#include <io/dec/tc/tc.h>
#include <machine/cpu.h>

#include <kits/ESA100/nonereg.h> 1

#define NNONE 4 2
```

1 Includes the device register header file, which is discussed in Section 4.1.1. The directory specification adheres to the third-party device driver configuration model discussed in Section 11.1.2. If the traditional device driver configuration model is followed, the directory specification is `<io/EasyInc/nonereg.h>`. The directory specification you make here depends on where you put the device register file.

The above lines include the common header files.

Writing Device Drivers, Volume 2: Reference provides reference (man) page-style descriptions of the header files most frequently used by DEC OSF/1 device drivers.

2 Defines a constant called `NNONE` that is used to allocate data structures needed by the `/dev/none` driver. There can be at most four instances of the none controller on the system. This value represents a small number of instances of the driver on the system and the data structures themselves are not large, so it is acceptable to allocate for the maximum configuration. This example uses the static allocation model 2 technique

described in Section 2.3.2. Note that you could also define this constant in a `name_data.c` file.

4.1.3 Autoconfiguration Support Declarations and Definitions Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. It contains the following declarations needed by the `/dev/none` device driver. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer. Thus, the driver writer can implement one device driver that is loadable or static.

```
#define DN_RESET 0001 [1]
#define DN_ERROR 0002 [2]

#define DN_OPEN  1 [3]
#define DN_CLOSE 0 [4]

int noneprobe(), nonecattach(), noneintr();
int noneopen(),  noneclose(),   noneread(), nonewrite();
int noneioctl(), none_ctlr_unattach(); [5]

struct controller *noneinfo[NNONE]; [6]

struct driver nonedriver = {
        noneprobe,
        0,
        nonecattach,
        0,
        0,
        0,
        0,
        0,
        "none",
        noneinfo,
        0,
        0,
        0,
        0,
        0,
        none_ctlr_unattach,
        0
}; [7]

struct none_softc {
     int sc_openf;
     int sc_count;
     int sc_state;
} none_softc[NNONE]; [8]
```

[1] Declares a constant called `DN_RESET` to indicate that the specified `none` device is ready for data transfer. Section 4.1.6.1 shows that the `noneprobe` interface uses this constant to set the `NONE_CSR` device

register offset associated with a specific none device. If this driver operated on actual hardware, setting the DN_RESET bit could force the device to reset.

2 Declares a constant called DN_ERROR to indicate when an error occurs. Section 4.1.6.1 shows that the noneprobe interface uses this constant in a bitwise AND operation with the NONE_CSR device register offset associated with a specific none device. An actual hardware device could set this bit in the NONE_CSR device register offset to indicate that an error condition occurred.

3 Declares a constant called DN_OPEN to represent the device open bit. Section 4.1.8.1 shows that the noneopen interface uses this constant to set the open bit for a specific none device. This bit represents the driver's software state.

4 Declares a constant called DN_CLOSE to represent the device close bit. Section 4.1.8.2 shows that the noneclose interface uses this constant to clear the open bit for a specific none device. This bit represents the driver's software state.

5 Declares the driver interfaces for the /dev/none driver. Note that the nonecattach and noneintr interfaces are merely stubs. These interfaces have no associated code but are declared here to handle any future development.

6 Declares an array of pointers to controller structures and calls it noneinfo. The controller structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation. Section 7.4 describes the controller structure.

Note that the NNONE constant is used to represent the maximum number of none controllers. This number is used to size the array of pointers to controller structures.

7 Declares and initializes the driver structure called nonedriver. This structure is used to connect the driver entry points and other information to the DEC OSF/1 code. This structure is used primarily during autoconfiguration. Some members of this structure are not used by the /dev/none device driver. Section 7.6 describes the driver structure.

The value zero (0) indicates that the /dev/none driver does not make use of a specific member of the driver structure. The following list describes those members initialized to a nonzero value by the example driver:

- The driver's probe interface, noneprobe. Section 4.1.6.1 shows how to implement noneprobe.

- The driver's cattach interface, nonecattach. The none device does not need an attach interface. However, Section 4.1.6.2 provides a nonecattach interface stub for future expansion.
- The string none, which is the name of the device.
- The value noneinfo, which references the array of pointers to the previously declared controller structures. You index this array with the controller number as specified in the ctlr_num member of the controller structure.
- The driver's controller unattach interface, none_ctlr_unattach. The none_ctlr_unattach interface removes the controller structure associated with specific none devices from the list of controller structures that it handles. Section 4.1.6.3 shows how to implement none_ctlr_unattach.

8 Declares an array of softc structures and calls it none_softc. Like noneinfo, the none_softc structure's size is the value represented by the NNONE constant. The softc structure is found in many device drivers to allow driver interfaces to share data. The none_softc structure contains the following members:

- sc_openf

  Stores a constant value that indicates the none device is open. This member is set by the noneopen and noneclose interfaces in Section 4.1.8.1 and Section 4.1.8.2.
- sc_count

  Stores the count of characters. This member is set by the nonewrite and noneioctl interfaces in Section 4.1.9.2 and Section 4.1.11.
- sc_state

  Stores a constant value that indicates the state the driver is in. This member is not currently used, but may be used in a future implementation of the example.

4.1.4 Loadable Driver Configuration Support Declarations and Definitions Section

This section is applicable only to the loadable version of the /dev/none device driver. It contains the following declarations used by the loadable version of the /dev/none device driver:

```
extern int nodev(), nulldev();  [1]
extern ihandler_id_t handler_add(), handler_del();
extern ihandler_id_t handler_enable(), handler_disable(); [2]
ihandler_id_t none_id_t[NNONE]; [3]
#define DN_BUSNAME     "tc" [4]

int none_is_dynamic = 0; [5]

struct tc_option none_option_snippet [] =
/***************************************************/
{
/*  module          driver  intr_b4 itr_aft        adpt    */
/*  name            name    probe   attach  type   config  */
/*  ------          ------  ------- ------- ----   ------  */
{   "NONE     ",    "none",   0,       1,    'C',   0},
{   "",             ""      } /* Null terminator in the */
                              /* table */
}; [6]
int num_none = 0;   [7]
```

[1] Declares external references for the nodev and nulldev interfaces, which are used to initialize members of the cdevsw table under specific circumstances. The cdevsw_add kernel interface, called by the driver's none_configure interface, initializes the cdevsw table. Section 8.2.1 provides a description of the cdevsw table and examples of the nodev and nulldev interfaces.

[2] Declares an external reference for the handler_add interface, which registers the interrupt service interface for the loadable driver. Note also the declarations of the external references for the handler_del, handler_enable, and handler_disable interfaces. Section 4.1.6.1 shows how the noneprobe interface calls handler_add.

[3] Declares an array of IDs used to deregister the interrupt handlers. Note that the NNONE constant is used to represent the maximum number of none controllers. This number is used to size the array of IDs. Thus, there is one ID per none device. Section 4.1.6.1 shows how noneprobe uses the *none_id_t* array.

[4] Defines a constant that represents a 2-character string that indicates this is a driver that operates on the TURBOchannel bus. This constant is passed as an argument to the ldbl_stanza_resolver, ldbl_ctlr_configure, and ldbl_ctlr_unconfigure

interfaces. This bus name is used in calls to the configuration code. Other bus types can use a different name. Section 4.1.7.2 shows how to call `ldbl_stanza_resolver` and `ldbl_ctlr_configure`. Section 4.1.7.3 shows how to call `ldbl_ctlr_unconfigure`.

5 Declares a variable called *`none_is_dynamic`* and initializes it to the value zero (0). This variable is used to control any differences in the tasks performed by the static and loadable versions of the `/dev/none` device driver at run time. Thus, the `/dev/none` driver can be compiled once for the loadable and static versions. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer.

Section 4.1.6.1 shows how `noneprobe` uses *`none_is_dynamic`*. Section 4.1.6.3 shows how `none_ctlr_unattach` uses *`none_is_dynamic`*. Section 4.1.7.2 shows how `none_configure` uses *`none_is_dynamic`*.

6 These lines are specific to drivers written for the TURBOchannel bus. Other bus types may use a different mechanism.

Declares a `tc` option table snippet that includes an entry for the loadable version of this driver. For the static version, a similar entry is made in the `tc_option` table located in the `tc_option_data.c` file. The entry in the `tc_option` table is used only when the driver is configured statically; the `none_option_snippet` entry is used only when the driver is configured dynamically.

The `tc_option` option table contains the bus-specific ROM module name for the driver. This information forms the bus-specific parameter that is passed to the `ldbl_stanza_resolver` interface to search for matches in the `tc_option` table. The `tc_option` table is used by the bus configuration interfaces associated with the TURBOchannel bus.

The items in the `tc` option table snippet are identical to those in the `tc_option` table. These items have the following meanings:

- **module name**

  In this column, you specify the device name in the ROM on the hardware device. The module name can be up to 8 characters in length. You must blank-pad the name to 8 bytes for those names that are less than 8 characters in length. Thus, the entry for the `/dev/none` driver consists of the letters ''NONE'' followed by four spaces.

- **driver name**

  In this column, you specify the driver name as it appears in the `Module_Config_Name` field of the `stanza.loadable` file fragment. In this example, the driver name is `none`. Because you specify the same name in the `Module_Config_Name` field and the driver name field of the `tc option` snippet table, the bus configuration code initializes the correct `controller` and `device` structures during device autoconfiguration for loadable drivers.

- **intr_b4 probe**

  In this column, you specify whether the device needs interrupts enabled during execution of the driver's `probe` interface. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled. In the example, the value zero (0) is specified to indicate that the `none` device does not need interrupts enabled.

- **itr_aft attach**

  In this column, you specify whether the device needs interrupts enabled after the driver's `probe` and `attach` interfaces complete. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled. In the example, the value 1 is specified to indicate that the `none` device needs interrupts enabled after its `noneprobe` and `nonecattach` interfaces complete.

- **type**

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

- **adpt config**

  If the device in the type column is `A` (adapter), you specify the name of the interface to configure the adapter. Otherwise, you specify the value zero (0). In the example, the value zero (0) is specified because the device in the previous column is a controller. Section 4.1.7.2 shows how the `none_configure` interface uses `none_option_snippet`.

7 Declares a variable called *`num_none`* to store the count on the number of controllers probed during autoconfiguration. This variable is initialized to the value zero (0) to indicate that no instances of the controller have been initialized yet. Section 4.1.6.3 shows how `none_ctlr_unattach` uses this variable. Section 4.1.7.2 shows how `none_configure` uses this variable.

4.1.5 Loadable Driver Local Structure and Variable Definitions Section

This section is applicable only to the loadable version of the /dev/none device driver. It contains the following declaration of the /dev/none driver's cdevsw entry that will be dynamically added to the cdevsw table:

```
int none_config = FALSE;  1
dev_t none_devno = NODEV; 2

struct cdevsw none_cdevsw_entry = {
        noneopen,
        noneclose,
        noneread,
        nonewrite,
        noneioctl,
        nodev,
        nodev,
        0,
        nodev,
        0,
        DEV_FUNNEL_NULL
}; 3
```

1 Declares a variable called *none_config* to store state flags indicating whether the /dev/none driver is configured as a loadable driver. The *none_config* variable is initialized to the value FALSE. Section 4.1.7.2 shows that none_configure sets *none_config* to the value TRUE to indicate that the /dev/none driver has successfully configured as a loadable driver. Section 4.1.7.3 shows that none_configure sets *none_config* to the value FALSE to indicate that the /dev/none driver has successfully unconfigured.

2 Declares a variable called *none_devno* to store the cdevsw table entry slot to use. The *none_devno* variable is initialized to the value NODEV to indicate that no major number for the device has been assigned. Section 4.1.7.2 shows that none_configure sets *none_devno* to the table entry slot.

3 Declares and initializes the cdevsw structure called none_cdevsw_entry. Section 8.2.1 describes the cdevsw table. The following list describes those members initialized to a nonzero value by the /dev/none device driver:

- The driver's open interface, noneopen. Section 4.1.8.1 shows how to implement noneopen.
- The driver's close interface, noneclose. Section 4.1.8.2 shows how to implement noneclose.

- The driver's `read` interface, `noneread`. Section 4.1.9.1 shows how to implement `noneread`.
- The driver's `write` interface, `nonewrite`. Section 4.1.9.2 shows how to implement `nonewrite`.
- The driver's `ioctl` interface, `noneioctl`. Section 4.1.11 shows how to implement `noneioctl`.

4.1.6 The Autoconfiguration Support Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer. Thus, the driver writer can implement one device driver that is loadable or static. Table 4-2 lists the interfaces implemented as part of The Autoconfiguration Support Section along with the sections in the book where each is described.

Table 4-2: Interfaces Implemented as Part of the Autoconfiguration Support Section

Part	Section
Implementing the noneprobe Interface	Section 4.1.6.1
Implementing the nonecattach Interface	Section 4.1.6.2
Implementing the none_ctlr_unattach Interface	Section 4.1.6.3

4.1.6.1 Implementing the noneprobe Interface

The `noneprobe` interface is applicable to the loadable or static version of the `/dev/none` device driver. It is called from the operating system configuration code during boot time. However, there are specific tasks associated with the loadable or static version of the driver. These tasks are identified by a conditional `if` statement that tests the *none_is_dynamic* variable. The interface's main task is to determine whether any `none` devices exist on the system. For the loadable version of the driver, `noneprobe` also calls the appropriate interfaces to register the interrupt handlers for the loadable driver.

For the static version, `noneprobe` calls the BADADDR interface to determine if the device is present. If the device is present, `noneprobe` returns the size of the device register structure. If the device is not present, `noneprobe` returns the value zero (0).

The following code implements the `noneprobe` interface:

```
noneprobe(addr1, ctlr)
io_handle_t addr1; 1
struct controller *ctlr; 2
{
     ihandler_t handler;         3
     struct tc_intr_info info;   4
     int unit = ctlr->ctlr_num;  5
     register io_handle_t reg = addr1; 6

     if (none_is_dynamic) { 7

             handler.ih_bus = ctlr->bus_hd; 8

             info.configuration_st = (caddr_t)ctlr; 9
             info.config_type = TC_CTLR; 10

             info.intr = noneintr; 11

             info.param = (caddr_t)unit; 12
             handler.ih_bus_info = (char *)&info; 13

             none_id_t[unit] = handler_add(&handler); 14
             if (none_id_t[unit] == NULL) { 15
                     return(0);
             }
             if (handler_enable(none_id_t[unit]) != 0) { 16
                     handler_del(none_id_t[unit]);
                     return(0);
             }
     }
     else {
          if (BADADDR( (caddr_t) reg + NONE_CSR, sizeof(long)) !=0)
          {
                     return (0);
           } 17
```

```
            }
      write_io_port(reg + NONE_CSR, 8, 0, DN_RESET); 18
      wbflush();                          19
      if(read_io_port(reg + NONE_CSR, 8, 0) & DN_ERROR)
      {
          return (0);
      } 20
      write_io_port(reg + NONE_CSR, 8, 0, 0); 21
      wbflush();      22
      return (1); 23
}
```

1 Declares an *addr1* argument that specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references the device's I/O address space for the bus where the read operation originates from (in calls to the `read_io_port` interface) and where the write operation occurs (in calls to the `write_io_port` interface).

In previous versions of this book, this first argument was of type `caddr_t`. The argument specified the system virtual address (SVA) that corresponds to the base slot address. The interpretation of this argument depends on the bus that your driver operates on. Although there is no real bus connected to the `/dev/none` driver, the example uses arguments associated with a TURBOchannel bus. Other buses may require a different argument in this position.

2 Declares a pointer to a `controller` structure. Again, the argument specified in this position of the `noneprobe` interface depends on the bus the driver operates on.

3 Declares an `ihandler_t` data structure called `handler` to contain information associated with the `/dev/none` device driver interrupt handling. Section 7.8 describes the `ihandler_t` structure. The `noneprobe` interface initializes two members of this data structure. This line is applicable only to the loadable version of the `/dev/none` device driver.

4 Declares a `tc_intr_info` data structure called `info`. This interrupt handler structure is for use by loadable drivers. Section 7.9 describes the `tc_intr_info` structure.

5 Declares a *unit* variable and initializes it to the controller number. This controller number identifies the specific `none` controller that is being probed.

The controller number is contained in the `ctlr_num` member of the `controller` structure associated with this `none` device. Section 7.4.3 describes the `ctlr_num` member. This line is applicable only to the loadable version of the `/dev/none` device driver.

6 Declares a variable called *reg* and initializes it to the I/O handle passed to the driver's probe interface by the bus configuration code. In this case, the I/O handle describes the system virtual address (SVA) for the `none` device.

7 Registers the interrupt handlers if *none_is_dynamic* evaluates to a nonzero value, indicating that the `/dev/none` device driver is loadable. Because this driver is also static, the driver writer must specify the interrupt handler, `noneintr`, in the system configuration file or the `stanza.static` file fragment. Section 12.2.1.2 describes how to specify the interrupt handlers in the system configuration file and the `stanza.static` file fragment.

The *none_is_dynamic* variable contains a value to control any differences in tasks performed by the static or loadable version of the `/dev/none` device driver. This approach means that any differences are made at run time and not at compile time. The *none_is_dynamic* variable is initialized and set by the `none_configure` interface, discussed in Section 4.1.7.2.

The items from 8 – 16 are applicable only to the loadable version of the driver.

8 Specifies the bus that this controller is attached to. The `bus_hd` member of the `controller` structure contains a pointer to the `bus` structure that this controller is connected to. After the initialization, the `ih_bus` member of the `ihandler_t` structure contains the pointer to the `bus` structure associated with the `/dev/none` device driver.

9 Sets the `configuration_st` member of the `info` data structure to the pointer to the `controller` structure associated with this `none` device. This `controller` structure is the one for which an associated interrupt will be written.

This line also performs a type-casting operation that converts `ctlr` (which is of type pointer to a `controller` structure) to be of type `caddr_t`, the type of the `configuration_st` member.

10 Sets the `config_type` member of the `info` data structure to the constant `TC_CTLR`, which identifies the `/dev/none` driver type as a controller.

11 Sets the `intr` member of the `info` data structure to `noneintr`, the `/dev/none` device driver's interrupt service interface.

12 Sets the `param` member of the `info` data structure to the controller number for the `controller` structure associated with this `none` device.

This line also performs a type-casting operation that converts *unit* (which is of type `int`) to be of type `caddr_t`, the type of the `param` member.

13 Sets the `ih_bus_info` member of the `handler` data structure to the address of the bus-specific information structure, `info`.

This line also performs a type-casting operation that converts `info` (which is of type `ihandler_t`) to be of type `char *`, the type of the `ih_bus_info` member.

14 Calls the `handler_add` interface and saves its return value for use later by the `handler_del` interface. The `handler_add` interface takes one argument: a pointer to an `ihandler_t` data structure, which in the example is the initialized `handler` structure.

This interface returns an opaque `ihandler_id_t` key, which is a unique number used to identify the interrupt service interfaces to be acted on by subsequent calls to `handler_del`, `handler_disable`, and `handler_enable`. Note that this key is stored in the *`none_id_t`* array, which was declared in Section 4.1.4. Section 4.1.6.3 shows how to call `handler_del` and `handler_disable`.

15 If the return value from `handler_add` equals NULL, returns a failure status to indicate that there are no interrupt service interfaces for the `/dev/none` driver.

16 If the `handler_enable` interface returns a nonzero value, returns the value zero (0) to indicate that it could not enable a previously registered interrupt service interface. The `handler_enable` interface takes one argument: a pointer to the interrupt service interface's entry in the interrupt table. In this example, this ID is contained in the *`none_id_t`* array.

If the call to `handler_enable` failed, removes the previously registered interrupt handler by calling `handler_del` prior to returning an error status.

Items 17 – 23 are applicable only to the static version of the `/dev/none` driver.

17 The next sequence of code calls the `BADADDR` interface to determine if the device is present. The `BADADDR` interface takes three arguments. However, only two are needed in this call. The first argument specifies the address of the device whose existence you want to check and the second argument specifies the length of the data to be checked. In this call to `BADADDR`, the I/O handle plus the `NONE_CSR` device register offset maps to the 64-bit control/status register for this `none` 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 `long`.

Because the first argument to `BADADDR` is of type `caddr_t`, this line also performs a type-casting operation that converts the type of the *`reg`* variable (which is of type `io_handle_t`) to type `caddr_t`.

If a device is present, BADADDR returns the value zero (0).

18 Calls the `write_io_port` interface to write the bit represented by the constant DN_RESET to the `none` device's control/status register. This bit instructs the device to reset itself in preparation for data transfer operations. The `write_io_port` interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. In this call, the `/dev/none` driver specifies the device's I/O address space by adding the I/O handle (stored in the *reg* variable) to the device register offset (represented by the NONE_CSR bit).
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the `/dev/none` driver passes the value 8.
- The third argument specifies flags to indicate special processing requests. Because this argument is not currently used, the `/dev/none` driver passes the value 0 (zero).
- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the `/dev/none` driver passes the bit represented by the constant DN_RESET.

19 Calls the kernel interface `wbflush` to ensure that a write to I/O space has completed.

20 If the result of the bitwise AND operation produces a nonzero value (that is, the error bit is set), then `noneprobe` returns the value zero (0) to the configuration code to indicate that the device is broken. To determine if the error bit is set, the `/dev/none` driver reads the device register by calling the `read_io_port` interface. The `read_io_port` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. In this call, the `/dev/none` driver specifies the device's I/O address space by adding the I/O handle (stored in the *reg* variable) to the device register offset (represented by the NONE_CSR bit).
- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the `/dev/none` driver

passes the value 8.

– The third argument specifies flags to indicate special processing requests. Currently, no flags are used. Because this argument is not currently used, the `/dev/none` driver passes the value 0 (zero).

21 If the result of the bitwise AND operation produces a zero value (that is, the error bit is not set), then `noneprobe` initializes the device's CSR/LED register to the value zero (0) by calling the `write_io_port` interface. The `/dev/none` driver passes the same arguments to `write_io_port` as in the previous call except for the fourth argument. In this call, `write_io_port` passes the value 0 to the fourth argument.

22 The `wbflush` interface is called a second time to ensure that a write to I/O space has completed.

23 The `noneprobe` interface returns to the autoconfiguration code a nonzero value, which indicates that the device is present.

4.1.6.2 Implementing the nonecattach Interface

The `nonecattach` interface has no code associated with it and is included as a stub for future expansion. It would be applicable to the loadable or static version of the `/dev/none` device driver.

```
nonecattach(ctlr)
struct controller *ctlr; 1
{
        /* Attach interface goes here. */
    •
    •
    •
        return;
}
```

1 The `none` device does not need an attach interface. However, this line shows that your `cattach` interface would declare a pointer to a `controller` structure. Your device driver could then send any information contained in this structure to the controller. Section 7.4 describes the `controller` structure.

4.1.6.3 Implementing the none_ctlr_unattach Interface

The none_ctlr_unattach interface is a loadable driver-specific interface called indirectly from the bus code when a system manager specifies that the loadable driver is to be unloaded. In other words, this interface would never be called if the /dev/none device driver were configured as a static driver because static drivers cannot be unconfigured. This interface's main tasks are to deregister the interrupt handlers associated with the /dev/none device driver and to remove the specified controller structure from the list of controllers the /dev/none driver handles.

The following code implements the none_ctlr_unattach interface:

```
int none_ctlr_unattach(bus, ctlr)
    struct bus *bus;         [1]
    struct controller *ctlr; [2]
{
        register int unit = ctlr->ctlr_num; [3]

        if ((unit > num_none) || (unit < 0)) { [4]
                return(1);
        }

        if (none_is_dynamic == 0) { [5]
                return(1);
        }

        if (handler_disable(none_id_t[unit]) != 0) { [6]
                return(1);
        }
        if (handler_del(none_id_t[unit]) != 0) { [7]
                return(1);
        }
        return(0); [8]
}
```

[1] Declares a pointer to a bus structure and calls it bus. The bus structure represents an instance of a bus entity. A bus is a real or imagined entity to which other buses or controllers are logically attached. All systems have at least one bus, the system bus, even though the bus may not actually exist physically. The term controller here refers both to devices that control slave devices (for example, disk and tape controllers) and to devices that stand alone (for example, terminal or network controllers). Section 7.3 describes the bus structure.

[2] Declares a pointer to a controller structure and calls it ctlr. This controller structure is the one you want to remove from the list of controllers handled by the /dev/none device driver. Section 7.4 describes the controller structure.

[3] Declares a *unit* variable and initializes it to the controller number. This controller number identifies the specific none controller whose associated controller structure is to be removed from the list of controllers handled by the /dev/none driver. Section 7.4.3 describes the ctlr_num member.

The controller number is contained in the ctlr_num member of the controller structure associated with this none device.

[4] If the controller number is greater than the number of controllers found by the noneprobe interface or the number of controllers is less than zero, returns the value 1 to the bus code to indicate an error.

This sequence of code validates the controller number. The *num_none* variable contains the number of instances of the none controller found by the noneprobe interface. Section 4.1.6.1 describes the implementation of noneprobe.

[5] If *none_is_dynamic* is equal to the value zero (0), returns the value 1 to the bus code to indicate an error.

This sequence of code validates whether the /dev/none driver is a loadable driver. The *none_is_dynamic* variable contains a value to control any differences in tasks performed by the static or loadable version of the /dev/none device driver. This approach means that any differences are made at run time and not at compile time. The *none_is_dynamic* variable is initialized and set by the none_configure interface, discussed in Section 4.1.7.2.

[6] If the return value from the call to the handler_disable interface is not equal to the value zero (0), returns the value 1 to the bus code to indicate an error. Otherwise, the handler_disable interface makes the /dev/none device driver's previously registered interrupt service interfaces unavailable to the system. Section 9.6.4 provides additional information on handler_disable.

This sequence of code is executed if *none_is_dynamic* is not equal to the value zero (0), indicating that the /dev/none device driver is a loadable driver. The handler_disable interface takes one argument: a pointer to the interrupt service's entry in the interrupt table. In this call, the ID is accessed through the *none_id_t* array. Section 4.1.6.1 shows that handler_add fills in this array. Note that the *unit* variable is used as an index to identify the interrupt service interface associated with a specific controller structure. Section 4.1.6.1 shows that noneprobe initializes this variable to the controller number.

[7] If the return value from the call to the handler_del interface is not equal to the value zero (0), returns the value 1 to the bus code to indicate an error. Otherwise, the handler_del interface deregisters the /dev/none device driver's interrupt service interface from the bus-

specific interrupt dispatching algorithm. Section 9.6.4 provides additional information on `handler_del`.

This sequence of code is executed if *`none_is_dynamic`* is not equal to the value zero (0), indicating that the `/dev/none` device driver is a loadable driver. The `handler_del` interface takes the same argument as the `handler_disable` interface: a pointer to the interrupt service's entry in the interrupt table.

8 Returns the value zero (0) to the bus code upon successful completion of the tasks performed by the `none_ctlr_unattach` interface.

4.1.7 Loadable Device Driver Section

This section is applicable only to the loadable version of the `/dev/none` device driver. It implements the `none_configure` interface. Table 4-3 lists the tasks associated with implementing the Loadable Device Driver Section along with the sections in the book where each task is described.

Table 4-3: Loadable Device Driver Section

Part	Section
Setting Up the none_configure Interface	Section 4.1.7.1
Configuring (Loading) the /dev/none Device Driver	Section 4.1.7.2
Unconfiguring (Unloading) the /dev/none Device Driver	Section 4.1.7.3
Querying the /dev/none Device Driver	Section 4.1.7.4

4.1.7.1 Setting Up the none_configure Interface

The following code shows how to set up the `none_configure` interface:

```
none_configure(op,indata,indatalen,outdata,outdatalen)
   sysconfig_op_t op;         1
   device_config_t *indata;   2
   size_t indatalen;          3
   device_config_t *outdata;  4
   size_t outdatalen;         5
{
        dev_t   cdevno;  6
        int     retval;  7
        int     i;       8
```

1 Declares an argument called *op* to contain a constant that describes the configuration operation to be performed on the loadable driver. This argument is used in a `switch` statement and evaluates to one of the following valid constants: `SYSCONFIG_CONFIGURE`, `SYSCONFIG_UNCONFIGURE`, or `SYSCONFIG_QUERY`.

2 Declares a pointer to a `device_config_t` data structure called `indata` that consists of inputs to the `none_configure` interface. This data structure is filled in by the device driver method of `cfgmgr`. The `device_config_t` data structure is used to represent a variety of information, including the `/dev/none` driver's major number requirements. Section 7.11 describes the `device_config_t` structure.

3 Declares an argument called *`indatalen`* to store the size of this input data structure (in bytes).

4 Declares a pointer to a `device_config_t` data structure called `outdata` that is filled in by the `/dev/none` device driver. This data structure contains a variety of information, including the return values from the `/dev/none` driver to `cfgmgr`. This returned information contains the major number assigned to the `none` device.

5 Declares an argument called *`outdatalen`* to store the size of this output data structure (in bytes).

6 Declares a variable called *cdevno* to store the major device number for the `none` device.

7 Declares a variable called *`retval`* to store the return value from the `cdevsw_del` interface.

8 Declares a variable called *`i`* used in the `for` loop when `none_configure` unloads the loadable driver.

4.1.7.2 Configuring (Loading) the /dev/none Device Driver

The following code shows how to implement the loadable driver's configure or loading operation. This section of code executes when the system manager requests that the `/dev/none` device driver be dynamically configured.

```
switch (op) {

        case SYSCONFIG_CONFIGURE: 1
          if (indata->dc_dsflags & IH_DRV_DYNAMIC) {
                  none_is_dynamic = 1;
          } 2
          if (none_is_dynamic) { 3

           if (strlen(indata->config_name) <= 0) {
               printf("none_configure, null config name.\n");
               return(EINVAL);
                   } 4

           if (ldbl_stanza_resolver(indata->config_name,
               DN_BUSNAME, &nonedriver,
               (caddr_t *)none_option_snippet) != 0) {
               return(EINVAL);
                   } 5

                   if (ldbl_ctlr_configure(DN_BUSNAME,
                           LDBL_WILDNUM, indata->config_name,
                           &nonedriver, 0)) {
                           return(EINVAL);
                   } 6

                   if (num_none == 0) {
                       return(EINVAL);
                   }
          } 7

          cdevno = makedev(indata->dc_cmajnum,
                  (indata->dc_cmajnum == -1)?-1:0); 8
          cdevno = cdevsw_add(cdevno,&none_cdevsw_entry); 9
          if (cdevno == NODEV) {

                  return(ENODEV);
          } 10

          none_devno = cdevno; 11

          outdata->dc_cmajnum = major(none_devno); 12

          outdata->dc_begunit = 0;
          outdata->dc_numunit = num_none;

          outdata->dc_version = DRIVER_BUILD_LEVEL;
          outdata->dc_dsflags = indata->dc_dsflags;
          outdata->dc_bmajnum = NODEV;

          outdata->dc_errcode = 0;
          outdata->dc_ihflags = 0;
          outdata->dc_ihlevel = 0;
```

```
                    none_config = TRUE; [13]
                    break;
```

[1] Specifies the SYSCONFIG_CONFIGURE constant to indicate that this section of code implements the configure loadable driver operation. The file /usr/sys/include/sys/sysconfig.h contains the definition of this constant.

[2] If the device switch configuration flag is set to the IH_DRV_DYNAMIC bit, then this is the loadable version of the driver. To indicate this condition, sets the *none_is_dynamic* variable to 1. The file /usr/sys/include/sys/sysconfig.h contains the definition of this constant.

The device switch configuration flag is set by the device driver method of cfgmgr in the dc_dsflags member of the input data structure, indata. This if statement is included because it is possible for a static driver to call the configure interface.

[3] If this is the loadable version of the driver, calls the strlen kernel interface.

[4] If the number of characters returned by strlen is less than or equal to zero:

- Calls printf to print a message indicating that no device driver name was specified in the stanza.loadable file fragment. The absence of a device driver name in stanza.loadable indicates that the driver name cannot be determined. The driver name is required for the configuration of the loadable driver. In this case, the interface must return an error status.

  Section 12.6 describes the stanza file format and syntax.

- Returns the constant EINVAL to indicate an invalid argument. This constant is defined in the file /usr/sys/include/sys/errno.h. This error return value gets indirectly passed back to cfgmgr.

The strlen interface takes one argument: a pointer to an array of characters terminated by a null character. In this call, the array of characters is the config_name member of the pointer to the indata input data structure. This member is set by the driver method of cfgmgr, which obtains the driver's configuration name from the Module_Config_Name field in the stanza.loadable file fragment. Because the driver's configuration name is a required field in the stanza.loadable file fragment, the driver must return an error if the name does not exist. Section 9.1.5 provides additional information on strlen. Section 12.6.2.19 describes the Module_Config_Name field.

5 If the ldbl_stanza_resolver kernel interface returns a value not equal to zero, it did not find matches in the tc_slot table. This condition indicates that no instances of the controller exist on the bus. It returns the constant EINVAL to indicate an invalid argument. Otherwise, ldbl_stanza_resolver allows the device driver to merge the system configuration data specified in the stanza.loadable file fragment into the hardware topology tree created at static configuration time. The ldbl_stanza_resolver interface takes four arguments:

- The name of the driver specified by the driver writer in the stanza.loadable file fragment

  In this call, the driver name is obtained from the config_name member of the pointer to the indata input data structure.

- The name of the parent bus structure associated with this controller

  In this call, the constant DN_BUSNAME represents the characters ``tc'', indicating that the parent bus structure is a TURBOchannel bus. The bus structure name is obtained from the config program.

- A pointer to the driver structure for the controlling device driver

  In this call, the address of the nonedriver structure is passed.

- A bus-specific parameter

  The bus-specific parameter for a TURBOchannel bus is usually a tc_option snippet table. In this call, the address of the none_option_snippet table is passed. This table contains the appropriate entry for the loadable version of the /dev/none device driver. Section 4.1.4 shows the declaration of this table.

  Note that a type-casting operation converts none_option_snippet (which is of type struct tc_option) to be of type caddr_t *, the type of the bus-specific argument. However, ldbl_stanza_resolver does not do anything with this argument but pass it to the bus configuration code that performs the correct type-casting operation to handle none_option_snippet. Section 9.6.5 provides additional information on ldbl_stanza_resolver.

6 Calls the ldbl_ctlr_configure interface to cause the driver's noneprobe interface to be called once for each instance of the controller found on the system. If ldbl_ctlr_configure fails, it returns the constant EINVAL. The ldbl_ctlr_configure interface takes five arguments:

- The bus name

  In this call, the bus name ``tc'' is represented by the DN_BUSNAME constant.

- The bus number

  In this call, the bus number is represented by the wildcard constant LDBL_WILDNUM. This usage allows for the configuration of all instances of the none device present on the system. This constant is defined in the file /usr/sys/include/io/common/devdriver.h.

- The name of the controlling device driver

  In this call, this name is obtained from the config_name member of the indata input data structure.

- A pointer to the driver structure for the controlling device driver, which in this call is nonedriver.

- Miscellaneous flags from /usr/sys/include/io/common/devdriver_loadable.h

  In this call, the value zero (0) is passed to indicate that no flags are specified.

7 If the noneprobe interface does not find any controllers, sets the variable that keeps count of the number of controllers found to the value zero (0) and returns the constant EINVAL to indicate no controllers were found.

8 Calls the makedev interface, which makes a device number of type dev_t based on the specified major and minor numbers. Upon successful completion, makedev returns the major number for this none device in the *cdevno* variable. Note that the driver configuration is performed before obtaining the device major number to prevent user level programs from gaining access to the /dev/none driver's entry points in the cdevsw.

The makedev interface takes two arguments. The first argument is the major number for the device, which in this call is obtained from the dc_cmajnum member of the pointer to the indata input data structure associated with this device.

The second argument is the minor number for the device, which in this call is also obtained from the dc_cmajnum member, indicating that the major and minor numbers are identical. This interface does not make use of the minor number. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of makedev.

9 Calls the cdevsw_add interface to add the driver entry points for the /dev/none driver to the cdevsw table. This interface takes two arguments. The first argument specifies the device switch table entry (slot) to use. This entry represents the requested major number. In this call, the slot to use was obtained in a previous call to makedev. The second argument is the character device switch structure that contains the

character device driver's entry points. In this call, this structure is called none_cdevsw_entry. Upon successful completion, cdevsw_add returns the device number associated with the device switch table. Section 9.6.2 provides additional information on cdevsw_add.

10 If the device number associated with the device switch table is equal to the constant NODEV, returns the error constant ENODEV. The NODEV constant indicates that the requested major number is currently in use or that the cdevsw table is currently full. The NODEV constant is defined in /usr/sys/include/sys/param.h, and /usr/sys/include/sys/errno.h contains the ENODEV constant.

11 Stores the cdevsw table entry slot for this none device in the *none_devno* variable. Section 4.1.7.3 shows that cdevsw_del uses this slot value when the device is unconfigured.

12 This line and the following lines set up the pointer to the outdata output data structure to contain the returned information from the driver configuration. The cfgmgr program uses this information to determine whether the driver was successfully configured and what device special files need to be created. The cfgmgr program initializes the output data structure as follows:

- Sets dc_cmajnum to the major number for this device by calling the major interface. In this call, the number of the device is contained in the *none_devno* variable. Section 9.9.2 provides additional information on major.
- Sets dc_begunit to the first minor device number in the range. In this case, the first minor number is zero (0).
- Sets dc_numunit to the number of instances of the controller found by the noneprobe interface. In this case, this number is contained in the *num_none* variable, which was incremented by noneprobe upon locating each controller on the system.
- Sets dc_version to the version of the kernel interfaces that the driver was compiled to. This member is examined by cfgmgr upon driver loading to ensure compatibility. In this call, the version is specified with the constant DRIVER_BUILD_LEVEL, which is defined in /usr/sys/include/sys/sysconfig.h.
- Sets dc_dsflags to the device switch configuration flags that were passed to the input data structure by cfgmgr.
- Sets dc_bmajnum to the constant NODEV. The none device is a character device and, therefore, has no block device major number.
- Because the dc_errcode, dc_ihflags, and dc_ihlevel members are not used, sets them to the value zero (0).

$\boxed{13}$ Sets the state flag to indicate that the /dev/none device driver is now configured as a loadable device driver.

4.1.7.3 Unconfiguring (Unloading) the /dev/none Device Driver

The following code shows how to implement the loadable driver's unconfigure or unloading operation. This section of code executes when the system manager requests that the currently loadable `/dev/none` device driver be unconfigured.

```
case SYSCONFIG_UNCONFIGURE: 1

  if (none_config != TRUE) {
          return(EINVAL);
  } 2

  for (i = 0; i < num_none; i++) {
          if (none_softc[i].sc_openf != 0) {
                  return(EBUSY);
          } 3
  }

  retval = cdevsw_del(none_devno);
  if (retval) {
          return(ESRCH);
  } 4

  if (none_is_dynamic) { 5

          if (ldbl_ctlr_unconfigure(DN_BUSNAME,
                  LDBL_WILDNUM, &nonedriver,
                  LDBL_WILDNAME, LDBL_WILDNUM) != 0) { 6

                  return(ESRCH);
          }
  }
  none_config = FALSE; 7
  break;
```

1 Specifies the `SYSCONFIG_UNCONFIGURE` constant to indicate that this section of code implements the unconfigure operation of the loadable driver. The file `/usr/sys/include/sys/sysconfig.h` contains the definition of this constant.

2 If the `/dev/none` device driver is not currently configured as a loadable driver, fails the unconfiguration by returning the constant `EINVAL`. This error code is defined in `/usr/sys/include/sys/errno.h`.

3 Prevents the system manager from unloading the device driver if it is currently active. To determine if the driver is active, checks the `sc_openf` member of this `none` device's `none_softc` structure to determine if the device is opened. If so, returns the constant `EBUSY`. This error code is defined in `/usr/sys/include/sys/errno.h`.

4 Calls the `cdevsw_del` interface to delete the `/dev/none` driver's entry points from the `cdevsw` table.

The `cdevsw_del` interface takes one argument: the device switch table entry (slot) to use. In this call, the slot is contained in the *none_devno* variable, which was set when the driver was configured. If `cdevsw_del` fails, returns `ESRCH` to indicate that the driver is not currently present in the `cdevsw` table.

5 If *none_is_dynamic* evaluates to TRUE, calls the `ldbl_ctlr_unconfigure` interface to unconfigure the specified controller. Section 4.1.7.2 shows that the *none_is_dynamic* variable was previously set to TRUE (the value 1) when the driver was configured.

6 If the `ldbl_ctlr_unconfigure` interface returns a nonzero value, returns the error constant `ESRCH` to indicate that it did not successfully unconfigure the specified controller. Otherwise, unconfigures the controller. A call to this interface results in a call to the driver's `none_ctlr_unattach` interface for each instance of the controller. Section 4.1.6.3 describes `none_ctlr_unattach`.

The `ldbl_ctlr_unconfigure` interface takes five arguments:

– The bus name

  In this call, the bus name is represented by the constant `DN_BUSNAME`.

– The bus number

  In this call, the wildcard constant indicates that the interface `ldbl_ctlr_unconfigure` deregisters all instances of the controllers connected to the TURBOchannel bus.

– A pointer to the `driver` structure for the controlling device driver, which in this call is `nonedriver`

– The controller name and controller number

  In this call, the wildcard constants indicate that `ldbl_ctlr_unconfigure` scans all `controller` structures. Section 9.6.6 provides additional information on `ldbl_ctlr_unconfigure`.

7 Sets the *none_config* variable to the value FALSE to indicate that the `/dev/none` device driver is now unconfigured.

4.1.7.4 Querying the /dev/none Device Driver

The following code shows how to implement the loadable driver's query operation. This section of code executes when the system manager requests a query of information associated with the loadable version of the driver.

```
                case SYSCONFIG_QUERY: [1]

                  if (none_config != TRUE) {
                          return(EINVAL);
                  } [2]
                  outdata->dc_cmajnum = major(none_devno);
                  outdata->dc_bmajnum = NODEV;
                  outdata->dc_begunit = 0;
                  outdata->dc_numunit = num_none;
                  outdata->dc_version = DRIVER_BUILD_LEVEL;
                  break;
                default:
                  return(EINVAL); [3]
        }

        return(0); [4]
}
```

[1] Specifies the SYSCONFIG_QUERY constant to indicate that this section of code implements the query operation of the loadable driver. The file /usr/sys/include/sys/sysconfig.h contains the definition of this constant.

[2] Fails the query if the driver is not currently configured as a loadable driver. Otherwise, sets the following members of the outdata output data structure:

- Sets dc_cmajnum to the major number for this device by calling the major interface.
- Sets dc_bmajnum to the constant NODEV. The none device is a character device and, therefore, has no block device major number.
- Sets dc_begunit to the first minor device number in the range. In this case, the first minor number is zero (0).
- Sets dc_numunit to the number of instances of the controller found by the noneprobe interface. In this case, this number is contained in the *num_none* variable, which was incremented by noneprobe upon locating each controller on the system.
- Sets dc_version to the version of the kernel interfaces that the driver was compiled to. This member is examined by cfgmgr upon driver loading to ensure compatibility. In this call, the version is specified with the constant DRIVER_BUILD_LEVEL, which is defined in /usr/sys/include/sys/sysconfig.h.

[3] Defines an unknown operation type and returns the error constant `EINVAL` to indicate this condition. This section of code is called if the *op* argument is set to anything other than `SYSCONFIG_CONFIGURE`, `SYSCONFIG_UNCONFIGURE`, or `SYSCONFIG_QUERY`.

[4] To indicate that the `/dev/none` driver's `none_configure` interface completed successfully, the value zero (0) is returned.

4.1.8 Open and Close Device Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer. Thus, the driver writer can implement one device driver that is loadable or static. Table 4-4 lists the two interfaces implemented as part of the Open and Close Device Section along with the sections in the book where each is described.

Table 4-4: Interfaces Implemented as Part of the Open and Close Device Section

Part	Section
Implementing the noneopen Interface	Section 4.1.8.1
Implementing the noneclose Interface	Section 4.1.8.2

4.1.8.1 Implementing the noneopen Interface

The noneopen interface is called as the result of an open system call. The following code implements a noneopen interface, which performs the following tasks: checks to ensure that the open is unique, marks the device as open, and returns the value zero (0) to the open system call to indicate success:

```
noneopen(dev, flag, format)
dev_t dev;  [1]
int flag;   [2]
int format; [3]
{

     register int unit = minor(dev);                [4]

     struct controller *ctlr = noneinfo[unit];  [5]

     struct none_softc *sc = &none_softc[unit]; [6]

     if(unit >= NNONE)
          return ENODEV; [7]

     if (sc->sc_openf == DN_OPEN)
          return (EBUSY); [8]

     if ((ctlr !=0) && (ctlr->alive & ALV_ALIVE))
     {
          sc->sc_openf = DN_OPEN;
          return(0); [9]
     }
     else return(ENXIO); [10]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific none device. The minor device number is used to determine the logical unit number for the none device that is to be opened.

[2] Declares an argument to contain flag bits from the file /usr/sys/include/sys/file.h. These flags indicate whether the device is being opened for reading, writing, or both.

[3] Declares an argument to contain a constant that identifies whether the device is a character or a block device. These constants are defined in /usr/sys/include/sys/mode.h.

[4] Declares a *unit* variable and initializes it to the device minor number. Note the use of the minor interface to obtain the device minor number.

The minor interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument. Section 9.9.3 provides additional information on minor.

5 Declares a pointer to a controller structure and calls it ctlr. Initializes ctlr to the controller structure associated with this none device. The minor device number, *unit*, is used as an index into the array of controller structures to determine which controller structure is associated with this none device.

6 Declares a pointer to a none_softc structure and calls it sc. Initializes sc to the address of the none_softc structure associated with this none device. The minor device number, *unit*, is used as an index into the array of none_softc structures to determine which none_softc structure is associated with this none device.

7 The none device requires no real work to open; therefore, the code could simply ignore the call and return the value zero (0). To demonstrate some of the checking that a real driver might perform, the example provides code that checks to be sure that the device exists.

If the device minor number, *unit*, is greater than or equal to the number of devices configured by the system, returns the error code ENODEV, which indicates no such device on the system. This error code is defined in /usr/sys/include/sys/errno.h.

8 If the sc_openf member of the sc pointer is equal to DN_OPEN, returns the error code EBUSY, which indicates that the none device has already been opened. This error code is defined in /usr/sys/include/sys/errno.h. This example test is used to ensure that this unit of the driver can be opened only once at a time. This type of open is referred to as an exclusive access open.

9 If the ctlr pointer is not equal to 0 and the alive member of ctlr has the ALV_ALIVE bit set, then the device exists. If this is the case, the noneopen interface sets the sc_openf member of the sc pointer to the open bit DN_OPEN and returns 0 to indicate a successful open.

10 If the device does not exist, noneopen returns the error code ENXIO, which indicates that the device does not exist. This error code is defined in /usr/sys/include/sys/errno.h.

4.1.8.2 Implementing the noneclose Interface

The `noneclose` interface uses the same arguments as `noneopen`, gets the device minor number in the same way, and initializes the `controller` and `none_softc` structures identically. The purpose of `noneclose` is to turn off the open flag for the specified `none` device. The following code implements the `noneclose` interface:

```
noneclose(dev, flag, format)
dev_t dev;  [1]
int flag;   [2]
int format; [3]
{

     register int unit = minor(dev);            [4]
     struct controller *ctlr = noneinfo[unit]; [5]
     struct none_softc *sc = &none_softc[unit]; [6]
          register io_handle_t reg =
          (io_handle_t) ctlr->addr; [7]

     sc->sc_openf = DN_CLOSE; [8]

        write_io_port(reg + NONE_CSR, 8, 0, 0); [9]

     wbflush(); [10]

     return(0); [11]
}
```

[1] Like the `noneopen` interface, the `noneclose` interface declares an argument that specifies the major and minor numbers for a specific `none` device. The minor device number is used to determine the logical unit number for the `none` device to be closed.

[2] Like the `noneopen` interface, the `noneclose` interface also declares an argument to contain flag bits from the file `/usr/sys/include/sys/file.h`. Typically, a driver's `close` interface does not make use of this argument.

[3] Although the *`format`* argument is shown here, a driver's `close` interface does not typically make use of this argument.

[4] Declares a *`unit`* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *`dev`* argument. Section 9.9.3 provides additional information on `minor`.

[5] Declares a pointer to a `controller` structure and calls it `ctlr`. Initializes `ctlr` to the `controller` structure associated with this

none device. The minor device number, *unit*, is used as an index into the array of controller structures to determine which controller structure is associated with this none device.

6 Declares a pointer to a none_softc structure and calls it sc. Initializes sc to the address of the none_softc structure associated with this none device. The minor device number, *unit*, is used as an index into the array of none_softc structures to determine which none_softc structure is associated with this none device.

7 Section 4.1.6.1 shows that the /dev/none device driver stored the I/O handle passed to its probe interface in the *reg* variable. The /dev/none device driver now initializes *reg* to the system virtual address (SVA) for the none device. This address is obtained from the addr member of the controller structure associated with this none device. Because the data types are different, this line performs a type-casting operation that converts the addr member (which is of type caddr_t) to be of type io_handle_t.

8 Turns off the open flag by setting the sc_openf member of the sc pointer to the close bit DN_CLOSE. This action frees up the unit so that subsequent calls to noneopen will succeed.

9 The none device is not a real device and therefore would not initiate interrupts. However, this line shows how to turn off interrupts by writing the value zero (0) to the device's control/status register. To accomplish the write operation, the /dev/none device driver calls the write_io_port interface. The write_io_port interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. In this call, the /dev/none driver specifies the device's I/O address space by adding the system virtual address (SVA) (stored in the *reg* variable) to the device register offset (represented by the NONE_CSR bit).
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the /dev/none driver passes the value 8.
- The third argument specifies flags to indicate special processing requests. Because this argument is not currently used, the /dev/none driver passes the value 0 (zero).
- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the /dev/none

driver passes the value zero (0).

10 Calls the kernel interface `wbflush` to ensure that a write to I/O space has completed.

11 The `noneclose` interface returns the value 0 to the `close` system call to indicate a successful close of the `none` device.

4.1.9 Read and Write Device Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer. Thus, the driver writer can implement one device driver that is loadable or static. Table 4-5 lists the two interfaces implemented as part of the Read and Write Device Section along with the sections in the book where each is described.

Table 4-5: Interfaces Implemented As Part of the Read and Write Device Section

Part	Section
Implementing the noneread Interface	Section 4.1.9.1
Implementing the nonewrite Interface	Section 4.1.9.2

4.1.9.1 Implementing the noneread Interface

The `noneread` interface simply returns success to the `read` system call because the `/dev/none` driver always returns EOF (End Of File) on read operations. The following code implements the `noneread` interface:

```
noneread(dev, uio, flag)
dev_t dev;        [1]
struct uio *uio; [2]
int flag;
{
     return (0); [3]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific `none` device. The minor device number is used to determine the logical unit number for the `none` device on which the read operation is performed.

[2] Declares 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 pointer unmodified to the `uiomove` or `physio` kernel interface.

[3] Returns success to the `read` system call. Because the `/dev/none` driver always returns EOF on read operations, the `noneread` interface simply returns 0. More complicated drivers would need to copy data from the device into the address space pointed to by the `uio` structure.

4.1.9.2 Implementing the nonewrite Interface

The `nonewrite` interface copies data from the address space pointed to by the `uio` structure to the device. Upon a successful write, `nonewrite` returns the value zero (0) to the write system call. The following code implements the `nonewrite` interface:

```
nonewrite(dev, uio, flag)
dev_t dev;           1
struct uio *uio;     2
int flag;
{
     int unit = minor(dev);                          3
     struct controller *ctlr = noneinfo[unit];       4
     struct none_softc *sc = &none_softc[unit];      5
     unsigned int count;                             6
     struct iovec *iov;                              7

     while(uio->uio_resid > 0) {      8
          iov = uio->uio_iov;      9
          if(iov->iov_len == 0) {  10
               uio->uio_iov++;
               uio->uio_iovcnt--;
               if(uio->uio_iovcnt < 0)  11
                    panic("none write");
               continue;
          }

     count = iov->iov_len;  12

     iov->iov_base += count;  13
     iov->iov_len -= count;  14
     uio->uio_offset += count;  15
     uio->uio_resid -= count;  16

     sc->sc_count +=count;  17
     }
     return (0);
}
```

1 Declares an argument that specifies the major and minor device numbers for a specific `none` device. The minor device number is used to determine the logical unit number for the device on which the write operation is performed.

2 Declares 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 pointer unmodified to the `uiomove` or `physio` kernel interface.

3 Declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument.

4 Declares a pointer to a `controller` structure and calls it `ctlr`. Initializes `ctlr` to the `controller` structure associated with this `none` device. The minor device number, *unit*, is used as an index into the array of `controller` structures to determine which `controller` structure is associated with this `none` device.

5 Declares a pointer to a `none_softc` structure and calls it `sc`. Initializes `sc` to the address of the `none_softc` structure associated with this `none` device. The minor device number, *unit*, is used as an index into the array of `none_softc` structures to determine which `none_softc` structure is associated with this `none` device.

6 Declares a variable that stores the size of the write request.

7 Declares a pointer to an `iovec` structure and calls it `iov`.

8 Checks the size of the remaining logical buffer (represented by the `uio_resid` member) to determine if `nonewrite` must copy data from the address space pointed to by the `uio` structure to the device. The loop continues until all the bytes of data are copied to the device.

9 Sets the `iov` pointer to the address of the current logical buffer segment (represented by the `uio_iov` member).

10 If the remaining size of the current segment (represented by the `iov_len` member) is equal to 0, increments the address of the current logical buffer segment (represented by the `uio_iov` member) and decrements the number of remaining logical buffer segments (represented by the `uio_iovcnt` member).

11 If the number of remaining logical buffer segments is less than 0, there is no data to write; therefore, calls the `panic` interface to cause a system crash and displays the message ''none write'' on the console terminal. This code represents an error condition that should never occur.

12 Sets the *count* variable to the number of bytes contained in the current segment (represented by the `iov_len` member). This value is the size of the write request.

13 Adds the number of bytes in the write request to the address of the current byte within the logical buffer segment (represented by the `iov_base` member).

14 Subtracts the number of bytes in the write request from the current segment (represented by the `iov_len` member).

15 Adds the number of bytes in the write request to the current offset into the full logical buffer (represented by the `uio_offset` member).

16 Subtracts the number of bytes in the write request from the size of the remaining logical buffer (represented by the `uio_resid` member).

17 Adds the number of bytes in the write request to the `sc_count` member of the `sc` pointer. When there are no more bytes, `nonewrite` returns the value zero (0) to indicate a successful write. Otherwise, it returns an appropriate error code that identifies the problem. You obtain the error codes from the file `/usr/sys/include/sys/errno.h`. Because there is no physical device associated with `/dev/none`, `nonewrite` does not actually copy data anywhere.

4.1.10 Interrupt Section

The interrupt entry point for the /dev/none device driver is a stub because there is no physical device to generate an interrupt. However, most devices can generate interrupts. For example, a terminal might generate an interrupt when a character is keyed into it. There is an interrupt entry point for those drivers that are written for devices that generate interrupts. There are a number of important issues relating to interrupts, which are discussed in Section 2.1.3.4. Some devices generate more than one type of interrupt. Thus, drivers controlling these devices can contain more than one interrupt section.

For the /dev/none driver, these issues are not relevant because there are no interrupts, but they will be important for most drivers.

The following shows the interrupt section for the /dev/none device driver:

```
noneintr(unit)
int unit; [1]
{
     struct controller *ctlr = noneinfo[unit]; [2]
     struct none_softc *sc = &none_softc[unit];
/* Code to perform the interrupt */
   •
   •
   •
}
```

[1] Declares a *unit* argument that specifies the logical unit number for this none device. This logical unit number would have been previously specified in the system configuration file.

[2] This line and the following line show some of the typical data structures you would define for an interrupt interface.

4.1.11 The ioctl Section

This section is applicable to the loadable or static version of the `/dev/none` device driver. The decision as to whether it is loadable or static is made at run time by the system manager and not at compile time by the driver writer. Thus, the driver writer can implement one device driver that is loadable or static. The ioctl section implements the `noneioctl` interface, which obtains and clears the count of bytes that was previously written by `nonewrite`. When a user program issues the command to obtain the count, the `/dev/none` driver returns the count through the data pointer passed to the `noneioctl` interface. When a user program asks to clear the count, the `/dev/none` driver does so. The following code implements the `noneioctl` interface:

```
noneioctl(dev, cmd, data, flag)
dev_t dev;               [1]
unsigned int cmd;        [2]
caddr_t data;            [3]
int flag;                [4]
{
     int unit = minor(dev); [5]
     int *res; [6]
     struct none_softc *sc = &none_softc[unit]; [7]

     res = (int *) data; [8]

     if(cmd == DN_GETCOUNT)
          *res = sc->sc_count; [9]

     if(cmd == DN_CLRCOUNT)
          sc->sc_count = 0;  [10]

     return (0); [11]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific `none` device. The minor device number is used to determine the logical unit number for the `none` device on which the ioctl operation is to be performed.

[2] Declares an argument to contain the ioctl command as specified in `/usr/sys/include/sys/ioctl.h` or in another include file that you define.

[3] Declares a pointer to the `ioctl` command-specified data that is to be passed to the device driver or filled in by the device driver. This argument is a kernel address. The size of this data cannot exceed the size of a page. At least 128 bytes is guaranteed. Any size between 128 bytes and the page size may fail if memory cannot be allocated. The particular

`ioctl` command implicitly determines the action to be taken. The `ioctl` system call performs all the necessary copy operations to move data to and from user space.

4 Declares an argument that holds the access mode of the device. The access modes are represented by flag constants defined in `/usr/sys/include/sys/file.h`.

5 Declares a *`unit`* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *`dev`* argument. Section 9.9.3 provides additional information on `minor`.

6 Declares a pointer to a variable that will store the character count. The `nonewrite` interface stores this character count in the `sc_count` member of the softc structure associated with this `none` device.

7 Declares a pointer to a `none_softc` structure and calls it `sc`. Initializes `sc` to the address of the `none_softc` structure associated with this `none` device. The minor device number, *`unit`*, is used as an index into the array of `none_softc` structures to determine which `none_softc` structure is associated with this `none` device.

8 Sets the *`res`* variable to point to the kernel memory allocated by the `ioctl` system call. The `ioctl` system call copies the data to and from user address space.

Because the data types are different, this line performs a type-casting operation that converts the *`data`* argument (which is of type `caddr_t`) to be of type pointer to an `int`.

9 If the `ioctl` command is equal to the `DN_GETCOUNT` macro, sets *`res`* to the number of bytes in the write request. This count was previously set by the `nonewrite` interface in the `sc_count` member of the `sc` pointer.

10 If the `ioctl` command is equal to the `DN_CLRCOUNT` macro, sets the count of characters written to the device to the value 0. This line has the effect of clearing the `sc_count` member of the softc structure associated with this `none` device.

11 Returns success to the `ioctl` system call.

4.2 Configuring the Device Driver

The DEC OSF/1 operating system provides two models for configuring device drivers: the third-party device driver configuration model and the traditional device driver configuration model. Chapter 11 describes each of these models. Chapter 12 reviews the files associated with configuring device drivers and describes the syntaxes and mechanisms used to populate these files. Chapter 13 provides instructions for configuring the /dev/none device driver, using the third-party and the traditional models.

4.3 Testing a Device Driver

One way to test a device driver is to write a test program. Implementing the test program involves the following tasks:

- Writing the test program
- Deciding on more rigorous testing
- Tracking down problems in testing

Each of these tasks is discussed in the following sections.

4.3.1 Writing the Test Program

The test program for the /dev/none device driver performs the following tasks:

- Opens the none device for both reading and writing.
- Gets the count of bytes written to the device.
- Writes 100 bytes.
- Gets the byte count a second time. The byte count should be 100 more than the first time.
- Resets the count to the value zero (0).
- Gets the count again, to verify that the reset worked.
- Tries to read 100 bytes. The test program should not be able to read any because /dev/none returns EOF on a read.

The test program follows:

```
/***************************************************
 * testdevnone.c - Test program for /dev/none      *
 *                 driver                          *
 ***************************************************/
/***************************************************
 *                                                 *
 * Author: Digital Equipment Corporation           *
 *                                                 *
 ***************************************************/
/***************************************************
 *         INCLUDE FILES                           *
 *                                                 *
 ***************************************************/
/***************************************************
```

```
 *                                                   *
 *                                                   *
 * Header files required by testdevnone.c            *
 *                                                   *
 *                                                   *
 *****************************************************/

#include <stdio.h>
#include <sys/ioctl.h> [1]
#include <sys/file.h>
#include <sys/limits.h>
#include <kits/ESA100/nonereg.h> [2]


char buf[100]; [3]

main()
{
     int d; [4]
     int count; [5]

     if((d = open("/dev/none",O_RDWR)) == -1) [6]
     {
          perror("/dev/none");
                exit(1);
     }
     ioctl(d,DN_GETCOUNT,&count); [7]
     printf("saw %d bytes\n",count); [8]

     write(d,&buf[0],100); [9]
     printf("wrote 100 bytes\n"); [10]

     ioctl(d,DN_GETCOUNT,&count); [11]
     printf("saw %d bytes\n",count); [12]

     ioctl(d,DN_CLRCOUNT,&count); [13]
     printf("set count\n"); [14]

     ioctl(d,DN_GETCOUNT,&count); [15]
        printf("saw %d bytes\n",count);

     count = read(d,&buf[0],100); [16]
        printf("was able to read %d bytes\n",count); [17]

     exit(0); [18]
}
```

[1] Includes the `ioctl.h` header file because the test program uses the `ioctl` macros defined in the device register header file `nonereg.h`.

[2] Includes the device register header file, which is discussed in Section 4.1.1. The directory specification adheres to the third-party device driver configuration model discussed in Section 11.1.2. If the traditional device driver configuration model is followed, the directory specification is `<io/EasyInc/nonereg.h>`. The directory specification you make here depends on where you put the device register file.

[3] Declares an array containing 100 character elements. This array is used to store the 100 bytes written by the nonewrite interface.

[4] Declares a variable to contain the return from the open system call.

[5] Declares a variable to store the character count.

[6] Calls the open system call, which in this example opens the device for read/write operations. The example passes two arguments. The first argument is the pathname that identifies the file to be opened. In the example, /dev/none is passed. This path causes the kernel to:

- Look up /dev/none in the file system
- Notice that /dev/none is a character-special device
- Look up the driver entry for /dev/none in the cdevsw table
- Locate the pointer to the noneopen interface that was put there when the cdevsw table was edited for the static version of the driver or inserted by the cdevsw_add kernel interface for the loadable version of the driver.
- Call the noneopen interface, passing to it a data structure that describes the device being opened

The second argument is a flag to tell the kernel how to open the file. In the example, the constant O_RDWR is passed. This constant tells the kernel to open /dev/none for reading and writing.

If the open system call executes successfully, it opens the device and returns a nonnegative value to the test program. However, if the system call fails, the kernel (in cooperation with the driver's noneopen interface) will have set the global variable *errno* to a constant value that describes the error. If an error occurs, the test program prints the error, using the perror interface, and quits.

[7] After the device is opened, gets the initial character count that was written to it. Although the count should be zero, it may not be if other users have already written data to the device.

Remember that the noneioctl interface in the driver is used to read or clear the count of characters written to /dev/none. Here, the ioctl system call is used to read the count of characters written to the device. The kernel notices that the file associated with the descriptor *d* is actually a special file, looks up its ioctl interface in the cdevsw table, and invokes the driver's noneioctl interface. The kernel copies the return value from noneioctl back to the *count* variable defined in the test program because the bitmask for DN_GETCOUNT instructs it to do so.

[8] Calls the printf interface to display on the console terminal a message indicating the number of bytes.

9 Writes 100 bytes of data to `/dev/none`. The kernel invokes `nonewrite` because the file descriptor *d* is associated with a special file. The `nonewrite` interface counts these characters.

10 Calls the `printf` interface to display on the console terminal a message indicating that `nonewrite` wrote 100 bytes.

11 Reads the count of characters written to the device now. The count should be 100 higher than it was in the previous section of code. The `noneioctl` interface is used exactly the same as it was to get the initial count.

12 Calls the `printf` interface to display on the console terminal the number of bytes.

13 The next test is to clear the count of characters written to the `none` device. The `DN_CLRCOUNT` macro is used to accomplish this task. This time, no data will be returned by the device driver, so the kernel will not copy any data back to the test program. All three arguments are passed to the `ioctl` system call for the sake of correctness. When this `ioctl` call returns, the test program expects the current count to be the value zero (0).

14 Calls the `printf` interface to display the message ‘‘set count’’ on the console terminal.

15 To determine if the previous code cleared the count to 0, the test program reads the characters from the `none` device again. The `printf` interface is used to display the count.

16 One design goal for the `/dev/none` device driver is that it should always return EOF on read operations. The test program checks this action by trying to read 100 bytes from the device. If the design goal was met, the `read` system call should return zero (0) bytes.

The test program passes three arguments to the `read` system call. The first argument is the file descriptor. The second argument points to where the data is stored. The final argument tells how many bytes to read, in this example 100 bytes. As discussed previously, the kernel notices that the file descriptor *d* is associated with a device and finds the `cdevsw` entry for the device's read interface, `noneread`. The kernel calls `noneread` to service the read request. The number of bytes successfully read is returned by the `read` system call.

17 The `printf` interface prints the number of bytes that were read.

18 The test program exits and the kernel automatically calls the `noneclose` interface.

The following shows the output from a sample run of the test program, which verifies that the driver is working correctly:

```
saw 0 bytes
wrote 100 bytes
saw 100 bytes
set count
saw 0 bytes
was able to read 0 bytes
```

4.3.2 Deciding on More Rigorous Testing

The test program for the `/dev/none` device driver can be expanded to do even more complete testing. For example, the test program could check the character count between each of several writes.

Because `/dev/none` is a simple driver, the test program is relatively straightforward. More complicated drivers require more complicated test programs, but the strategy is the same: for each interface you provide in the driver, test it independently from the other interfaces.

4.3.3 Hints for Tracking Problems in Testing

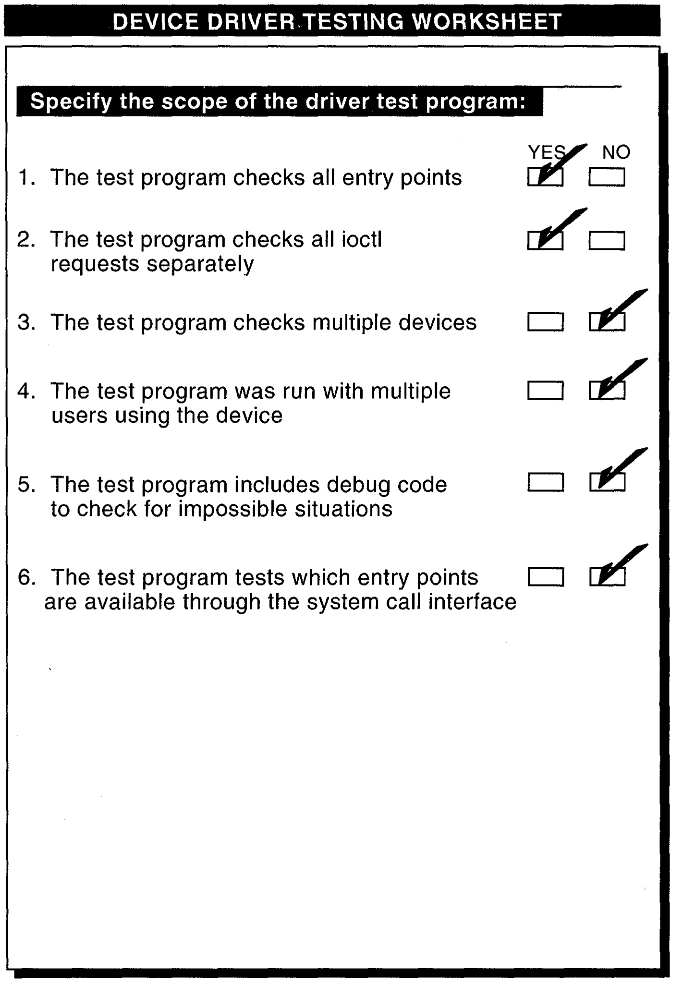
If the driver fails to behave as expected, you need to track down the problem and correct it. Here are several aids and hints:

- Refer to the *Kernel Debugging* for guidelines in debugging the driver.
- Use the `printf` kernel interface to display descriptive messages on the console terminal when the driver calls different interfaces. The messages that appear on the console should give you some insight into the problem. The `syslog` utility also logs these messages into a message file.

  You cannot use the `printf` interface to debug all driver types. For example, `printf` frequently is not useful in debugging terminal drivers because use of the terminal driver may be required to display the contents of `printf` to the terminal.
- Recompile and reinstall the kernel.
- Rerun the test program with the reinstalled kernel.
- Use synchronous traces of flow and variables where practical.
- Do not use synchronous traces in the interrupt handler. This means you should not use the `printf` interface in the interrupt handler.
- Use asynchronous traces where synchronous traces cannot be used.

Figure 4-1 shows the device driver testing worksheet for the `/dev/none` device driver. This worksheet is provided in Appendix C for use in identifying the scope of your driver test programs.

Figure 4-1: Testing Worksheet for /dev/none

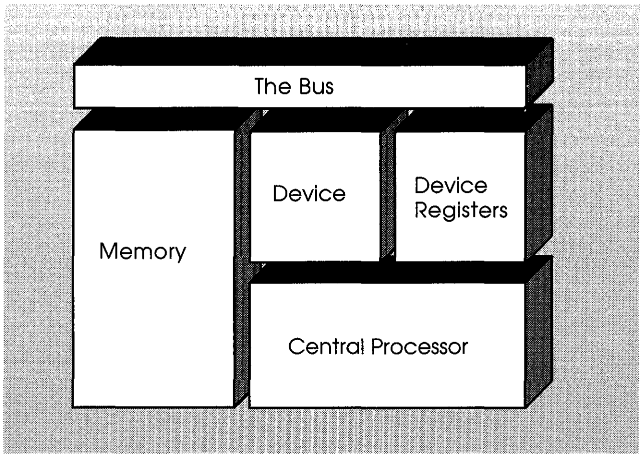


DEVICE DRIVER TESTING WORKSHEET

Specify the scope of the driver test program:

	YES	NO
1. The test program checks all entry points	✓	
2. The test program checks all ioctl requests separately	✓	
3. The test program checks multiple devices		✓
4. The test program was run with multiple users using the device		✓
5. The test program includes debug code to check for impossible situations		✓
6. The test program tests which entry points are available through the system call interface		✓

Part 3 Hardware Environment



Hardware-Independent Model and Device Drivers 5

One of the goals of an open systems environment is to provide hardware and software platforms that promote the use of standards. By adhering to a set of standard interfaces, these platforms make it easier for third-party application programmers to write applications that can run on a variety of operating systems and hardware. This open systems environment can also make it easier for systems engineers to write device drivers for numerous peripheral devices that operate on this same variety of operating systems and hardware.

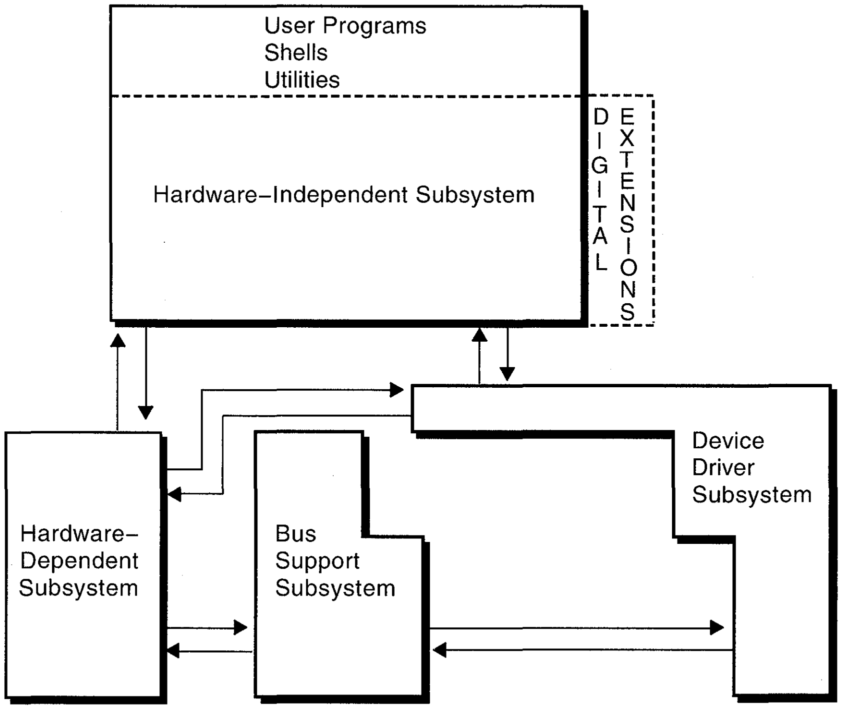
The hardware-independent model describes the hardware and software components that make up an open systems environment. This chapter provides an overview of the hardware-independent model and shows how it relates to device drivers. Although you do not need to know all of the details of this model, a high-level discussion can help to clarify how device drivers fit into the independent model.

Figure 5-1 shows that the hardware-independent model consists of a:

- Hardware-independent subsystem
- Hardware-dependent subsystem
- Bus support subsystem
- Device driver subsystem

The sections following Figure 5-1 briefly describe these subsystems as they relate to device drivers.

Figure 5-1: Hardware-Independent Model



5.1 Hardware-Independent Subsystem

The hardware-independent subsystem contains all of the hardware-independent pieces of an operating system, including the hardware-independent kernel interfaces, user programs, shells, and utilities. The Open Software Foundation (OSF) provides the hardware-independent subsystem. As the figure shows, however, this subsystem also contains extensions and enhancements made by Digital Equipment Corporation. Examples of these extensions are DECnet and support for CDROM file systems.

Writing device drivers becomes easier in such an open systems environment because the system design standardizes interfaces and data structures whenever possible. For example, the hardware-independent subsystem contains those kernel interfaces (such as `kalloc` and `timeout`) and data structures (such as `buf` and `uio`) that are not affected by the hardware. On the other hand, those interfaces (such as `fubyte` and `suword`) and data structures (such as the timer and memory management structures) affected by the hardware are contained in the hardware-dependent subsystem.

Figure 5-1 shows that the hardware-independent subsystem communicates with:

- The hardware-dependent subsystem

  The hardware-independent subsystem communicates with the hardware-dependent subsystem by calling interfaces and referencing data structures and global variables provided by the hardware-dependent subsystem.

- The device driver subsystem

  The hardware-independent subsystem communicates with the device driver subsystem by using specific data structures to make calls into the device driver subsystem. These data structures are the device switch tables `bdevsw` and `cdevsw`.

5.2 Hardware-Dependent Subsystem

The hardware-dependent subsystem contains all of the hardware-dependent pieces of an operating system with the exception of device drivers. This subsystem provides the code that supports a specific CPU platform and, therefore, is implemented by specific vendors. In the case of Digital Equipment Corporation, this code supports the Alpha AXP architecture. However, code could be written to support other CPU architectures. Writing device drivers becomes easier in such an open systems environment because the CPU-specific changes are hidden in the hardware-dependent layer, so minimal or no changes would need to be made to device drivers.

Figure 5-1 shows that the hardware-dependent subsystem communicates with:

- The hardware-independent subsystem

  The hardware-independent subsystem initiates communication with the hardware-dependent subsystem by calling interfaces and referencing data structures and global variables provided by the hardware-dependent subsystem.

- The device driver subsystem

  The device driver subsystem initiates communication with the hardware-dependent subsystem by calling some of the same interfaces and referencing some of the same data structures as the hardware-independent subsystem.

- The bus support subsystem

  The hardware-dependent subsystem initiates communication with the bus support subsystem by calling the bus configuration interfaces. The bus support subsystem also initiates communication with the device driver by calling the driver's autoconfiguration entry points. These entry points are the driver's `probe`, `attach`, and `slave` interfaces.

5.3 Bus Support Subsystem

The bus support subsystem contains all of the bus adapter-specific code. Many buses provide interfaces that are publicly documented and, therefore, allow vendors to more easily plug in hardware and software components. However, the way one vendor implements the bus-specific code can be different from another vendor. For example, the implementation of the VMEbus on Digital RISC systems is different from Sun Microsystem's implementation of the VMEbus. Isolating the bus-specific code and data structures into a bus support subsystem makes it easier for independent software vendors to implement different bus adapters.

Figure 5-1 shows that the bus support subsystem communicates with:

- The hardware-dependent subsystem

  The bus support subsystem communicates with the hardware-dependent subsystem by calling interfaces and referencing data structures and global variables provided by the hardware-dependent subsystem.

- The device driver subsystem

  The bus support subsystem communicates with the device driver subsystem during device autoconfiguration through the driver's `probe`, `slave`, and `attach` interfaces. In addition, the bus support subsystem provides interfaces that allow drivers to perform bus-specific tasks.

5.4 Device Driver Subsystem

The device driver subsystem contains all of the driver-specific code. This subsystem is also supplied by Digital Equipment Corporation, and it includes device drivers for hardware supported by Digital. Third-party device driver writers would place their drivers in the device driver subsystem.

Figure 5-1 shows that the device driver subsystem communicates with:

- The hardware-dependent subsystem

  The device driver subsystem communicates with the hardware-dependent subsystem by calling interfaces and referencing data structures and global variables provided by the hardware-dependent subsystem.

- The bus support subsystem

  The device driver subsystem communicates with the bus support subsystem during device autoconfiguration. This is accomplished through the driver's `probe`, `slave`, and `attach` interfaces. In addition, device drivers can call bus-specific interfaces to perform a variety of tasks. For example, a TURBOchannel device driver calls the `tc_enable_option` interface to enable a device's interrupt line to the CPU. Other buses have their own specific interfaces that drivers use to communicate with this subsystem.

- The hardware-independent subsystem

  The device driver subsystem communicates with the hardware-independent subsystem through calls to interfaces and data structures provided by the hardware-independent subsystem.

Hardware Components and Hardware Activities 6

This chapter discusses the hardware environment from the point of view of the device driver writer. Specifically, the chapter describes the following hardware components:

- The central processing unit (CPU)
- The bus
- The device

The chapter also discusses a variety of hardware activities that are of interest to you when writing device drivers.

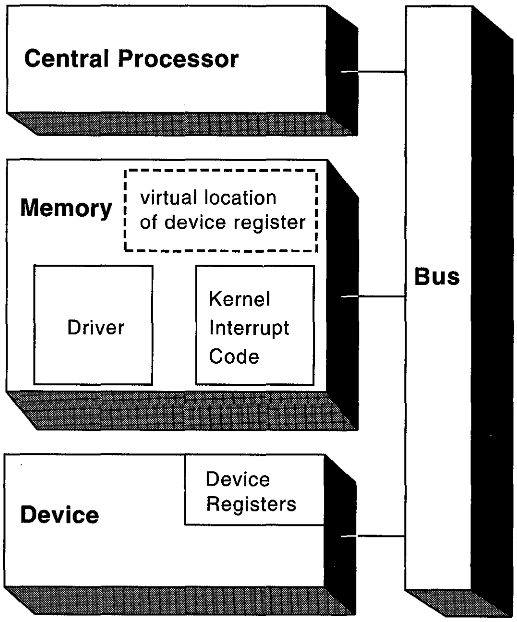
6.1 Hardware Components

Figure 6-1 shows the following hardware components:

- The central processing unit (CPU)
- Memory
- The bus
- The device

Each of these components is discussed briefly following Figure 6-1.

Figure 6-1: Hardware Components of Interest to a Device Driver Writer



6.1.1 Central Processing Unit

The central processing unit (CPU) is the main computational unit in a computer and the one that executes instructions. The CPU is of interest to device driver writers because its associated architecture influences the design of the driver. Writing device drivers for OPENbus architectures in an open systems environment means that you may need to become familiar with a variety of CPU architectures, such as Alpha AXP and MIPS. Section 2.4 describes the CPU issues that influence the design of device drivers.

6.1.2 Memory

Figure 6-1 shows that both the kernel and device drivers reside in memory, as does the kernel interrupt code that determines what driver will handle each interrupt from a device. The device driver accesses device registers (often referred to as control status register or CSR addresses) as though they were in memory; however, these registers are not really in memory but are located in the device. Figure 6-1 shows this arrangment by identifying a virtual location for the device registers. The hardware manual for the device you are writing the driver for should describe these device register addresses or offsets.

You need to include the addresses assigned to these device registers in one of the following places:

- The device driver

  For the TURBOchannel bus, the address is based on the slot and is provided to the device driver through the `probe` interface. A device driver that operates on the TURBOchannel accesses the device registers by referencing the location that results from the base address (determined by the slot the device is plugged into) plus the register offset address.

- The system configuration file

  You (or the system manager) specify the device register addresses in the system configuration file when you configure a static device driver. Section 12.2.1.2 discusses the syntax used to describe a device's register addresses in the system configuration file. For loadable drivers, you specify the device register addresses in the driver's `probe` interface and pass them through a call to the `handler_add` interface.

  For some buses (for example, the VMEbus), you may need to change the device register address because another device is using the address. If so, some change will also need to be made to the device. The hardware manual for the device should state how this change can be accomplished.

6.1.3 Bus

The bus is the path for all communication among the CPU, memory, and devices. Thus, when a device driver reads from or writes to a device's registers, the information transfer is by way of the bus. If you write device drivers for OPEN systems, you will probably need to know about a variety of buses, including the following:

- TURBOchannel
- VMEbus
- SCSI

There are three situations when the bus is of concern to you:

- When specifying which bus a device is on
- When writing `probe` and `slave` interfaces
- When direct memory access (DMA) is done by a device

Each of these situations is briefly considered in the following sections.

6.1.3.1 Specifying Which Bus a Device Is On

For the most part, you can think of the bus as a single path, but at kernel configuration time this view is not adequate. The bus, as used in this book, includes:

- The system bus
- Other buses attached to the system bus, for example, a VMEbus

When adding a device driver, you very well may be adding a device to the system. If you are, you will need to know what bus to attach its device controller to.

You specify which bus a device controller is connected to in the following files:

- The system configuration file, for static drivers that follow the traditional device driver configuration model. Section 11.2 describes the traditional device driver configuration model and Section 12.2.1.2 discusses the syntax for specifying which bus is connected to a device controller.
- The `config.file` file fragment, for static device drivers that follow the third-party device driver configuration model. Section 11.1 describes the third-party device driver configuration model. Section 11.1.2.3 discusses the `config.file` fragment. Section 12.2.1.2 discusses the syntax for specifying which bus is connected to a device controller.
- The `stanza.loadable` file fragment, for loadable drivers that follow the third-party device driver configuration model. Section 11.1.2.5 describes this file and Section 12.6.2.21 explains the syntax for specifying which bus is connected to a device controller.

6.1.3.2 When Writing probe and slave Interfaces

When writing `probe` and `slave` interfaces, you need to consider the bus on which the driver will operate. The bus affects the formal parameters you specify for these interfaces. Section 3.3.1 and Section 3.3.2 show you how to set up `probe` and `slave` interfaces for the TURBOchannel bus. See the bus-specific driver manual to learn how to set up these interfaces for the bus your driver operates on.

6.1.3.3 When Direct Memory Access (DMA) Is Done by a Device

Some devices are capable of directly accessing memory, generally to transfer large blocks of data. Section 6.1.4.5 discusses DMA as it relates to devices.

6.1.4 Device

A device (often referred to as a peripheral device) can be a printer, an acquisition device, terminal, and so forth. Whatever the device is, it has:

- One or more device registers for communicating with other hardware
- The ability (in most devices) to generate interrupts

Major distinctions between devices are the type of device (block, character, or network) and whether the device is capable of directly accessing memory. A direct memory access (DMA) device is one that can directly access (read from and write to) CPU memory, without CPU intervention. Non-DMA devices cannot directly access CPU memory.

The following sections briefly discuss the device registers, block and character devices, network devices, and DMA and non-DMA devices.

6.1.4.1 Device Registers

A device register is commonly referred to as a control status register, or CSR. The device register can be used to:

- Control what a device does
- Report the status of a device
- Transfer data to or from the device

The device register can be in the device or in a separate controller. In most cases, the location of the device register is of no concern to the device driver writer.

The types of device registers can vary widely, depending on the device used. Most devices have multiple registers. A device register can be read-only, such as for device status or the results of an I/O operation, or it can be a write-only command/control register, or it can be both readable and writeable.

It is often the case that after writing to a read/write register, the subsequent read from it will return a completely different value. The read value will be defined to be a status or result, while the write value will be command or control information to the device. In many cases, reading a control status register once will clear that register's values. A second read may, in fact, return unexpected or unwanted results. See the documentation for the device to determine if this situation exists.

6.1.4.2 Block and Character Devices

A block device is one that is designed to operate in terms of the block I/O supported by DEC OSF/1. It is accessed through the buffer cache. A block device has an associated block device driver that performs I/O by using file system block-sized buffers from a buffer cache supplied by the kernel. Block device drivers are particularly well-suited for disk drives, the most common block devices.

A character device is any device that can have streams of characters read from or written to it. A character device has a character device driver associated with it that can be used for a device such as a line printer that handles one character at a time. However, character drivers are not limited to performing I/O a single character at a time (despite the name ''character'' driver). For example, tape drivers frequently perform I/O in 10K chunks. 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 in handling I/O, many drivers are character drivers. Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

6.1.4.3 Terminal Devices

A terminal device is a special type of character device that can have streams of characters read from or written to it. Terminal devices have terminal (character) device drivers associated with them. 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. It cannot be mounted as a file system and, therefore, does not use data caching.

6.1.4.4 Network Devices

A network device is any device associated with network activities and is responsible for both transmitting and receiving frames to and from the network medium. Network devices have network device drivers associated with them. 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.

6.1.4.5 DMA and non-DMA Devices

A direct memory access (DMA) device is one that can directly access (read from and write to) CPU memory, without CPU intervention. Non-DMA devices cannot directly access CPU memory. The object of DMA is faster data transfer. When character drivers perform DMA, they do it to or from the user's address space. When block drivers perform DMA, they do it to or from the buffer cache.

6.2 Hardware Activities

When writing device drivers, you need to consider the following hardware-related activities:

- How a device driver accesses device registers
- How the device uses the registers
- How the device driver interrupts the CPU

The following sections discuss each of these activities as they relate to device drivers.

6.2.1 How a Device Driver Accesses Device Registers

Alpha AXP CPU platforms vary in the mechanisms available to access device registers. Some platforms, such as the DEC 4000 and DEC 7000 series, use the mailbox mechanism. Other platforms, such as the DEC 3000 series, allow ''direct'' access to device registers through special address spaces. The following discussion applies to those platforms that use special address spaces.

Alpha AXP CPUs can access longwords (32 bits) and quadwords (64 bits) atomically. However, many devices have CSRs that are only 16- or 8-bits wide. Program fragments accessing those CSRs will actually be sequences of Alpha AXP instructions that fetch longwords and mask out bytes. For example, for adjacent 16-bit CSRs this longword access is probably not what is intended. The fetch of the longword may result in the reading of multiple CSRs, with unwanted side effects.

6.2.2 How the Device Uses the Registers

Devices use their registers to report status and to return data. DMA devices can move data to or from memory directly. DMA device drivers typically request this type of data transfer by:

- Writing a base address and a count of characters in the device register to specify what data to move
- Setting a bit that requests that the transfer begin

The device requests access to memory and then manages a sequence of data transfers to memory. Upon completion of the transfer, the device:

- Sets a bit indicating that the transfer is done
- Issues an interrupt

You do not need to know the details of how the device and the bus interact to handle these transfers of data. However, you do need to know the exact register fields to use to make the request. This information should be in the hardware manual for the device.

6.2.3 How the Device Driver Interrupts the CPU

The specifics of how interrupts are generated varies from bus to bus and even from CPU architecture to CPU architecture. The following is a general description of how interrupts are processed by the kernel on some systems. The important point to remember is that the end result of an interrupt is the calling of the device driver's interrupt handling interface.

When a device generates an interrupt, it also provides an interrupt index. This interrupt index is passed by the bus to the kernel assembly language interface that does initial handling of interrupts. That interface uses the interrupt index to determine which entry in the interrupt vector table contains the address of the interrupt interface for the device driver that handles this device. Among other things, the assembly language interface uses that address to transfer control to the appropriate driver. The `config` program adds a static device driver's interrupt service interface to the system interrupt vector table. A loadable driver's interrupt service interface is added to this table through a call to the `handler_add` interface.

6.3 Parts of the System Accessing Hardware

There are only two parts of the system that access hardware. These are:

- The bus support subsystem

  There are specific bus support interfaces that access the device registers of the bus adapter.

- The device driver subsystem

  There are specific device driver interfaces that access the device registers of specific devices attached to a controller.

Part 4 Kernel Environment

Device Autoconfiguration 7

Autoconfiguration is a process that determines what hardware actually exists during the current instance of the running kernel. This chapter discusses the events that occur during the autoconfiguration of devices, with an emphasis on how autoconfiguration relates to static and loadable drivers. In addition, the chapter provides detailed information on the data structures related to autoconfiguration. The chapter frequently uses the generic term autoconfiguration software. The autoconfiguration software consists of the programs that accomplish the tasks associated with the events that occur during the autoconfiguration of devices. In most cases, it is not necessary for device driver writers to know the specific programs that execute during autoconfiguration.

Specifically, this chapter discusses:

- Files used by the autoconfiguration software
- The autoconfiguration process
- The `bus` structure
- The `controller` structure
- The `device` structure
- The `driver` structure
- The `port` structure
- The `ihandler_t` structure (for loadable drivers)
- The `tc_intr_info` structure (for loadable drivers)
- The `handler_key` structure (for loadable drivers)
- The `device_config_t` structure (for loadable drivers)

7.1 Files Used by the Autoconfiguration Software

The autoconfiguration software reads the following files pertinent to device drivers during the autoconfiguration process:

- The `config.file` file fragment and/or the system configuration file (for static device drivers)

  The `config.file` file fragment and the system configuration file (sometimes referred to as the *NAME* file) identify, among other things, device connectivity information. Section 12.2 describes the parts of these files pertinent to device drivers and provides details on the syntaxes that specify each current or planned device on the system.

- The /etc/sysconfigtab database (for loadable device drivers)

  The sysconfigtab database contains the information provided in the stanza.loadable file fragments. This information is appended to the sysconfigtab database during installation of the device driver kit.

  The stanza.loadable file fragment contains an entry for each device driver, providing such information as the driver's name, location of the loadable object, device connectivity information, and device special file information. Parts of the stanza.loadable file fragment are functionally similar to the system configuration file in that it uses a subset of the syntaxes used in the system configuration file to specify each current or planned device on the system. Section 12.6 describes the valid device syntaxes for the stanza.loadable file fragment.

The following example shows a sample system configuration file and a sample /etc/sysconfigtab database:

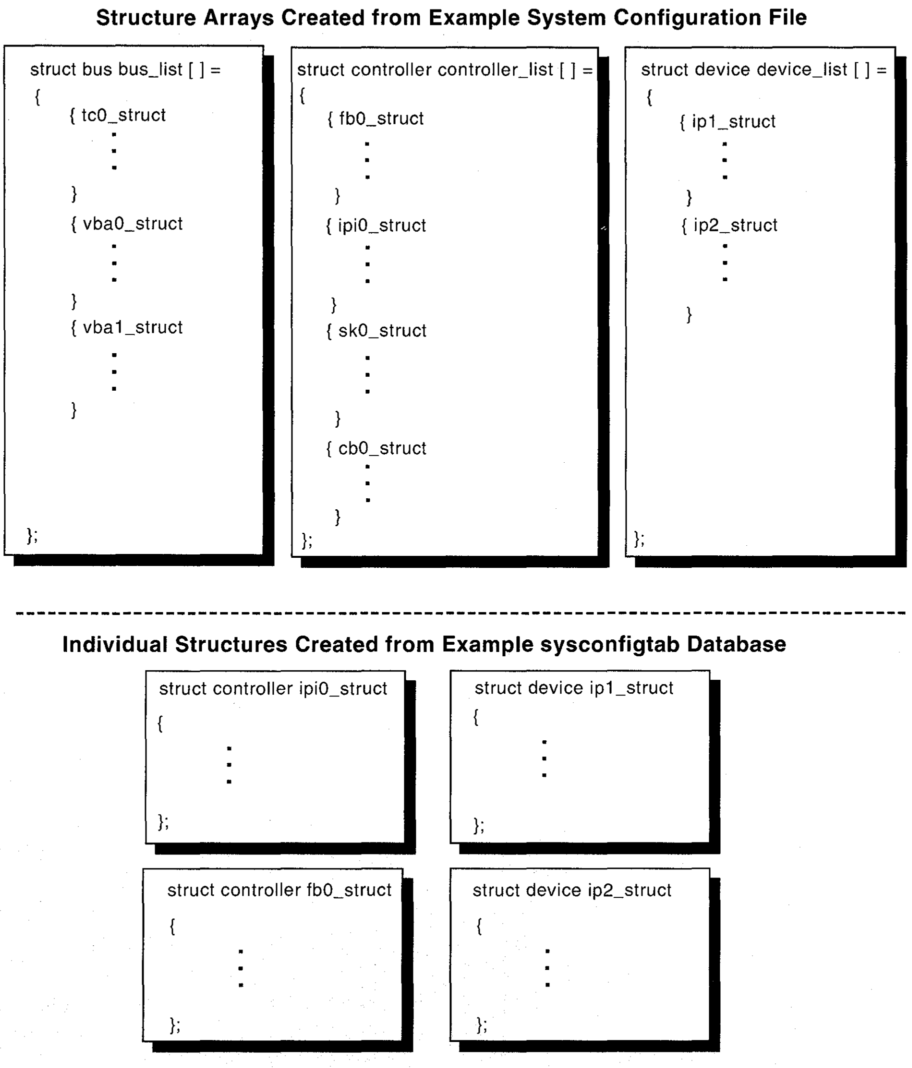
```
/* Example system configuration file */
bus          tc0    at     nexus
controller   fb0    at     tc0  vector fbintr
controller   ipi0   at     tc0  vector ipiintr
device disk  ip1    at     ipi0 unit 1
device disk  ip2    at     ipi0 unit 2
bus          vba0   at     tc0  slot 2 vector vbaerrors
controller   sk0    at     vba0 csr 0x8000 vector skintr 0xc8
bus          vba1   at     tc0  slot 1
controller   cb0    at     vba1 csr 0x80001000 vector cbintr 0x45

/* Example sysconfigtab database */
Module_Config2 = controller  fb0    at      tc0
Module_Config3 = controller  ipi0   at      tc0
Module_Config4 = device disk ip1    at      ipi0    unit 1
Module_Config5 = device disk ip2    at      ipi0    unit 2
```

The two samples are shown together so that you can clearly see the similarities and differences. Note that these samples show only a subset of all the possible syntaxes.

The autoconfiguration software uses the information in this system configuration file and database to create the associated bus, controller, and device structures. These structures are designed so that they are generic enough to handle not only autoconfiguration for static and loadable drivers but also to simplify configuration management. It is important to note that the kernel doubly links these structures top-down and bottom-up to make the traversal of the configuration tree more efficient. Section 7.2 discusses the links the kernel makes between these structures during the autoconfiguration process.

Figure 7-1 shows the structures created by the autoconfiguration software after it reads the sample system configuration file and `sysconfigtab` database. Of course, the autoconfiguration software does not store these structures in memory with the structure names as identified in the figure. These names and the figure are used to make it easier to understand how these structures are created and manipulated.

Figure 7-1: Structures Created from the Example


Structure Arrays Created from Example System Configuration File
Individual Structures Created from Example sysconfigtab Database

```
struct bus bus_list [ ] =
{
     { tc0_struct
          .
          .
          .
     }
     { vba0_struct
          .
          .
          .
     }
     { vba1_struct
          .
          .
          .
     }
};
```

```
struct controller controller_list [ ] =
{
     { fb0_struct
          .
          .
          .
     }
     { ipi0_struct
          .
          .
          .
     }
     { sk0_struct
          .
          .
          .
     }
     { cb0_struct
          .
          .
          .
     }
};
```

```
struct device device_list [ ] =
{
     { ip1_struct
          .
          .
          .
     }
     { ip2_struct
          .
          .
          .
     }
};
```

```
struct controller ipi0_struct
{
          .
          .
          .
};
```

```
struct device ip1_struct
{
          .
          .
          .
};
```

```
struct controller fb0_struct
{
          .
          .
          .
};
```

```
struct device ip2_struct
{
          .
          .
          .
};
```

The following sections discuss the similarities and differences between the structures created by the autoconfiguration software for static and loadable drivers.

7.1.1 For bus Structures

For each bus (specified with the bus keyword in the system configuration file), the autoconfiguration software creates an array of bus structures called bus_list, which it stores in ioconf.c. As the figure shows, the autoconfiguration software fills this array with the bus structures it finds in the system configuration file: tc0_struct, vba0_struct, and vba1_struct.

Similarly, for each bus (specified with the bus keyword in the stanza.loadable file fragment), the autoconfiguration software dynamically creates individual bus structures, instead of an array, and stores these structures in memory. As the figure shows, tc0_struct is not created during the autoconfiguration of loadable drivers. The system bus would have already been created during the autoconfiguration of static drivers. In this example, tc0_struct already exists in the bus_list array. The system bus must be in the system configuration file because a loadable system bus is not supported. Here, the controller structures are dynamically created and linked through the pointer to the system bus structure that was statically allocated in bus_list. Because tc0_struct represents the system bus, there is no need to specify it in the stanza.loadable file fragment. However, if the system bus was not specified in the system configuration file (for static drivers), no bus structures are created. The autoconfiguration software always detects a system bus during autoconfiguration of loadable drivers.

7.1.2 For controller Structures

For each controller (specified with the controller keyword in the system configuration file), the autoconfiguration software creates an array of controller structures called controller_list, which it stores in ioconf.c. As the figure shows, the autoconfiguration software fills this array with the controller structures it finds in the system configuration file: fb0_struct, ipi0_struct, sk0_struct, and cb0_struct.

Similarly, for each controller (also specified with the controller keyword in the stanza.loadable file fragment), the autoconfiguration software dynamically creates individual controller structures, instead of an array, and stores these structures in memory. As the figure shows, the autoconfiguration software creates the controller structures it finds in the stanza.loadable file fragment: fb0_struct and ipi0_struct.

7.1.3 For device Structures

For each device (specified with the device keyword in the system configuration file), the autoconfiguration software creates an array of device structures called device_list, which it stores in ioconf.c.

As the figure shows, the autoconfiguration software fills this array with the device structures it finds in the system configuration file: ip1_struct and ip2_struct.

Similarly, for each device (also specified with the device keyword in the stanza.loadable file fragment), the autoconfiguration software dynamically creates individual device structures, instead of an array, and stores these structures in memory. As the figure shows, the autoconfiguration software creates the device structures it finds in the stanza.loadable file fragment: ip1_struct and ip2_struct.

7.2 Autoconfiguration Process

When the kernel boots, the autoconfiguration software determines what hardware actually exists during the current instance of the running kernel. System managers often create (compile and link) kernels that define the greatest possible amount and variety of hardware available on the system. For example, a large and complex system with many bus adapters might alternately run a certain device on one bus adapter at a certain time and on another bus adapter later on for testing purposes.

The following sections describe autoconfiguration from the static and loadable driver points of view.

7.2.1 Autoconfiguration for Static Device Drivers

This section describes autoconfiguration from the point of view of static device drivers. When the kernel boots, the autoconfiguration software configures all the buses on the system by performing the following tasks:

- Locating the bus structure for the system bus
- Calling the level 1 bus configuration interfaces
- Configuring all devices
- Calling the level 1 configuration interfaces for any other buses
- Calling the level 2 configuration interfaces for any other buses
- Creating a system configuration tree

Each of these tasks is discussed in the following sections.

7.2.1.1 Locating the bus Structure for the System Bus

The autoconfiguration software searches bus_list, the array of bus structures created from the system configuration file, to locate the structure for the system bus. The system bus structure is identified with a backpointer of –1 indicating that it is connected to the keyword nexus in the system configuration file. The nexus keyword indicates the top of the system configuration tree.

All systems have a system bus, even if there is no physical bus. For example, workstations that do not have a physical bus have a logical `ibus` to which the onboard devices are connected.

7.2.1.2 Calling the Level 1 Bus Configuration Interfaces

After the autoconfiguration software locates the `bus` structure for the system bus, it calls the level 1 bus configuration interface specified in the `bus` structure. If a bus supports autoconfiguration (that is, the devices support a device type identifier and the bus supports a well-defined search algorithm), the autoconfiguration software searches `controller_list`, the array of `controller` structures created from the system configuration file, to locate a match for each controller on the system bus. If a match is found, the `controller` structure for that controller is linked to the `bus` structure and a back link to the `bus` structure is placed in the `controller` structure. This action continues until all controllers have been configured.

If the bus does not support autoconfiguration, the `controller_list` array is searched for controllers that were attached to the bus in the system configuration file. In both cases, the driver's `probe` interface (as specified in the `driver` structure) for each device is called to verify the existence of the controller. If the `probe` interface returns success, the controller's `attach` interface (if one exists) is called.

7.2.1.3 Configuring All Devices

After a successful probe, the autoconfiguration software searches `device_list`, the array of `device` structures created from the system configuration file, to locate a match for each device connected to a controller. If a match is found, the `device` structure is connected to its respective `controller` structure. For each device found, the autoconfiguration software calls the driver's `slave` and `attach` interfaces.

7.2.1.4 Calling the Level 1 Configuration Interfaces for Any Other Buses

Any other buses located during the configuration of the system bus are handled in the same manner. The level 1 configuration interface for each bus is called at this time. All configured buses are also connected by means of a linked list.

7.2.1.5 Calling the Level 2 Configuration Interfaces for Any Other Buses

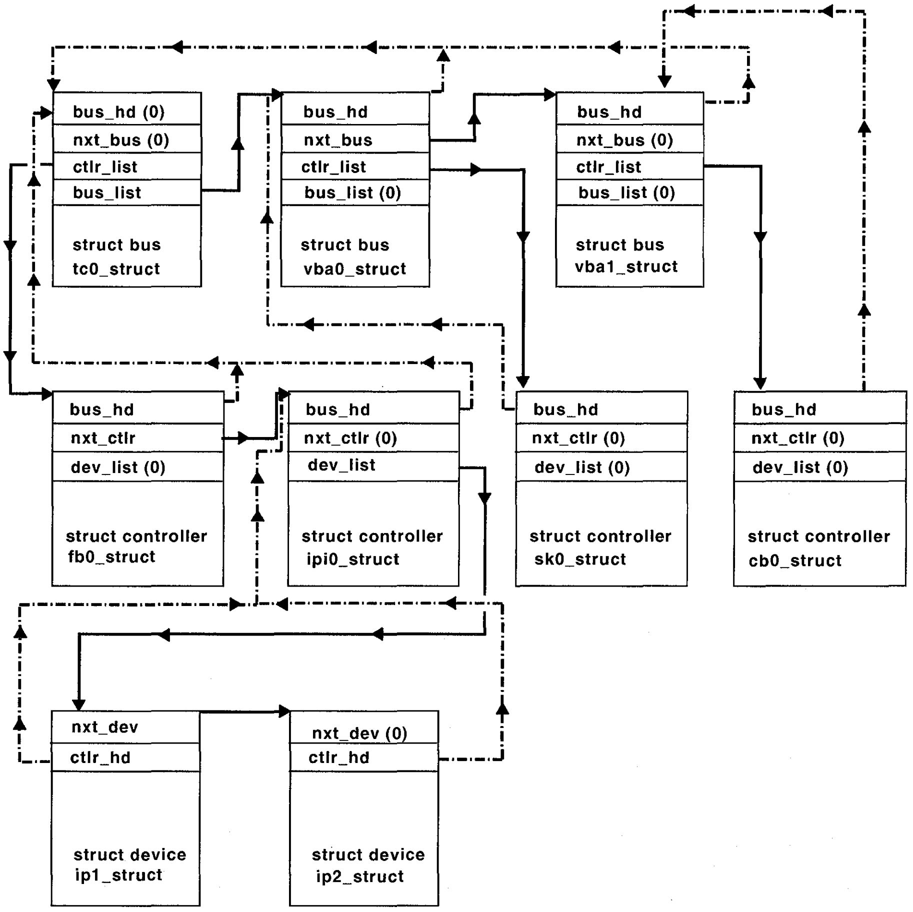
After all the buses have been configured, the autoconfiguration software calls the system bus level 2 configuration interface specified in the `bus` structure. This interface then calls the level 2 configuration interface for each directly connected bus. Each of those buses performs any second pass configuration work and calls the level 2 configuration interface of any connected bus.

7.2.1.6 Creating a System Configuration Tree

The end result of this process is to have a completely connected tree that represents the current system configuration. Note that there may be buses, controllers, and devices that are not connected. This indicates the entity was not physically connected on this system.

Figure 7-2 shows the configuration tree the autoconfiguration software would create, based on the entries in the example system configuration file. The figure shows the members of the `bus`, `controller`, and `device` structures used to establish the correct links.

Figure 7-2: Configuration Tree Based on Example System Configuration File



The following text provides information on how to traverse this configuration tree.

Pointing to the bus Structure to Which This Bus Is Connected

The bus_hd member specifies a pointer to the bus structure that this bus is connected to. The vba0 and vba1 buses are both connected to the system bus. The autoconfiguration software establishes this connection by setting each bus_hd member to the address of the system bus structure, tc0_struct. Figure 7-2 uses a broken arrow to show the setting of the bus_hd members.

The system bus is not connected to any other bus and the figure indicates that, as a result, the autoconfiguration software sets this bus_hd member to the value zero (0).

Pointing to the Next Bus at This Level

The nxt_bus member specifies a pointer to the next bus at this level. The vba1 bus follows the vba0 bus, thus making it the next bus at this level. The autoconfiguration software establishes this relationship by setting the nxt_bus member of vba0_struct to the address of vba1_struct. Figure 7-2 uses a solid arrow to show the setting of the nxt_bus member.

No buses follow the system bus and the vba1 bus and the figure indicates that, as a result, the autoconfiguration software sets their nxt_bus members to the value zero (0).

Pointing to a Linked List of Controllers Connected to This Bus

The ctlr_list member specifies a linked list of controllers connected to this bus. The fb0 and ipi0 controllers are connected to the system bus. The autoconfiguration software establishes the linked list by setting the ctlr_list member of tc0_struct to the address of fb0_struct. Figure 7-2 uses a solid arrow to show the setting of the ctlr_list member.

The sk0 controller is connected to the vba0 bus and the cb0 controller is connected to the vba1 bus. As it did with the system bus, the autoconfiguration software establishes the linked lists by setting the ctlr_list member for each bus to the address of its associated controller structure. Figure 7-2 uses a solid arrow to show the setting of the ctlr_list members.

Pointing to the Linked List of Buses Connected to This Bus

The bus_list member specifies a linked list of buses connected to this bus. The vba0 and vba1 buses are connected to the system bus. The autoconfiguration software establishes the linked list by setting the bus_list member of tc0_struct to the address of vba0_struct. Figure 7-2 uses a solid arrow to show the setting of the bus_list member.

The vba0 and vba1 buses do not have a linked list of buses and the figure indicates that, as a result, the autoconfiguration software sets their bus_list members to the value zero (0).

Pointing to the bus Structure to Which This Controller Is Connected

The bus_hd member appears not only in the bus structure, but also in the controller structure. In this case, bus_hd specifies a pointer to the bus structure that this controller is connected to. The fb0 and ipi0 controllers are connected to the system bus. The autoconfiguration software establishes the backpointer to the system bus by setting the bus_hd members of fb0_struct and ipi0_struct to the address of tc0_struct. Figure 7-2 uses a broken arrow to show the setting of the backpointer.

The sk0 and cb0 controllers are connected to the vba0 and vba1 buses. The autoconfiguration software establishes these backpointers by setting the bus_hd members of sk0_struct and cb0_struct to the address of their associated bus structures. Figure 7-2 uses a broken arrow to show the setting of these backpointers.

Pointing to the Next Controller at This Level

The nxt_ctlr member specifies a pointer to the next controller at this level. The ipi0 controller follows the fb0 controller, thus making it the next controller at this level. The autoconfiguration software establishes this relationship by setting the nxt_ctlr member of fb0_struct to the address of ipi0_struct. Figure 7-2 uses a solid arrow to show this. No controllers follow the ipi0 controller and the figure indicates that, as a result, the autoconfiguration software sets the nxt_ctlr member of ipi0_struct to the value zero (0).

The sk0 and cb0 controllers are the only controllers at their respective levels and the figure indicates that, as a result, the autoconfiguration software sets their nxt_ctlr members to the value zero (0).

Pointing to the Linked List of Devices Connected to This Controller

The dev_list member specifies a linked list of devices connected to this controller. The ipi0 controller has two devices connected to it: device ip1 and device ip2. The autoconfiguration software establishes the linked list by setting the dev_list member of ipi0_struct to the address of ip1_struct. Figure 7-2 uses a solid arrow to show the setting of the linked list.

Because the fb0, sk0, and cb0 controllers have no connected devices, Figure 7-2 shows that the autoconfiguration software sets the dev_list member in their respective structures to the value zero (0).

Pointing to the Next Device at This Level

The nxt_dev member specifies a pointer to the next device at this level. The ip2 device follows the ip1 device, thus making it the next device at this level. The autoconfiguration software establishes this relationship by setting the nxt_dev member of ip1_struct to the address of ip2_struct. Figure 7-2 uses a solid arrow to show the setting of the nxt_dev member.

Because there are no devices that follow device ip2, Figure 7-2 shows that the autoconfiguration software sets the nxt_dev member of ip2_struct to the value zero (0).

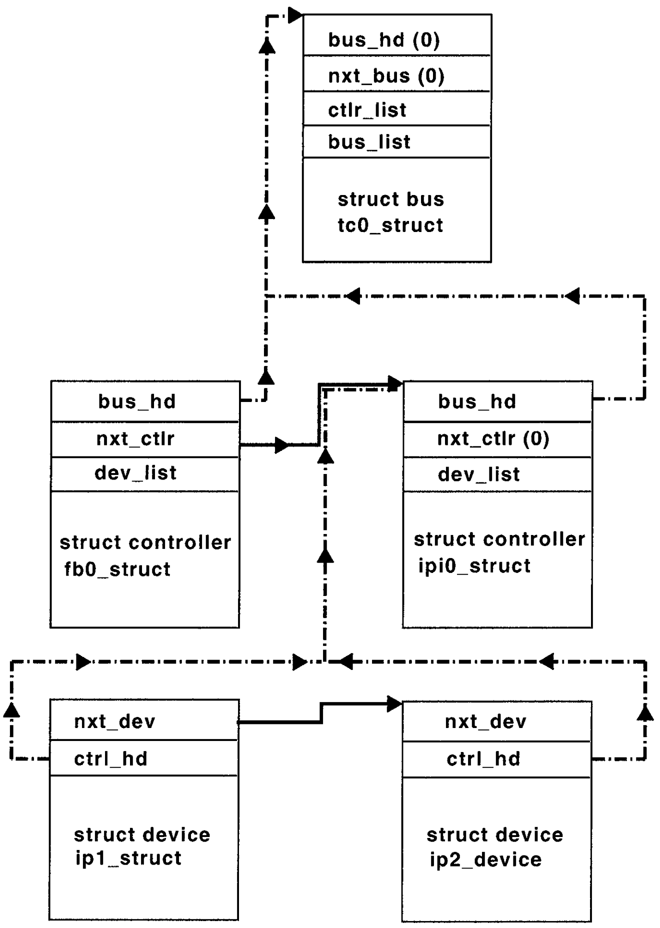
Pointing to the controller Structure to Which This Device Is Connected

The ctlr_hd member specifies a pointer to the controller structure that this device is connected to. Both the ip1 and ip2 devices are connected to the same controller, ipi0. The autoconfiguration software establishes these backpointers by setting the ctlr_hd member of ip1_struct and ip2_struct to the address of ipi0_struct. Figure 7-2 uses a broken arrow to show the setting of the backpointers.

7.2.2 Autoconfiguration for Loadable Device Drivers

Figure 7-3 shows the configuration tree the autoconfiguration software would create, based on the entries in the sysconfigtab database (which is built from entries specified in stanza.loadable file fragments) presented in Section 7.1. The figure shows the members of the bus, controller, and device structures used to establish the correct links. Note that the system bus, tc0_struct, was previously created during the autoconfiguration of static device drivers.

Figure 7-3: Configuration Tree for Loadable Drivers Based on Example sysconfigtab



7.3 The bus Structure

The bus structure represents an instance of a bus entity. A bus is a real or imagined entity to which other buses or controllers are logically attached. All systems have at least one bus, the system bus, even though the bus may not actually exist physically. The term controller here refers both to devices that control slave devices (for example, disk and tape controllers) and to devices that stand alone (for example, terminal or network controllers). You (or the system manager) specify a bus entity as follows:

- For static drivers

  Specify a valid syntax for the bus in the `config.file` file fragment or the system configuration file.

- For loadable drivers

  Specify a valid syntax for the bus in the `stanza.loadable` file fragment.

Chapter 11 discusses the device driver configuration models. Chapter 12 describes the valid syntaxes for a bus specification in these files. Chapter 13 provides examples of how to configure device drivers.

Table 7-1 lists the members of the `bus` structure along with their associated data types.

Table 7-1: Members of the bus Structure

Member Name	Data Type
`bus_mbox`	`u_long`
`bus_hd`	`struct bus *`
`nxt_bus`	`struct bus *`
`ctlr_list`	`struct controller *`
`bus_list`	`struct bus *`
`bus_type`	`int`
`bus_name`	`char *`
`bus_num`	`int`
`slot`	`int`
`connect_bus`	`char *`
`connect_num`	`int`
`confl1`	`int (*confl1)()`
`confl2`	`int (*confl2)()`
`pname`	`char *`
`port`	`struct port *`
`intr`	`int (**intr)()`
`alive`	`int`
`framework`	`struct bus_framework *`
`driver_name`	`char *`
`private`	`void * [8]`

Table 7-1: (continued)

Member Name	Data Type
conn_priv	void * [8]
rsvd	void * [7]

The following sections discuss all of these members except for `bus_hd`, `nxt_bus`, `ctlr_list`, and `bus_list`, which are presented in Section 7.2.1.6. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of this data structure.

7.3.1 The bus_mbox Member

The `bus_mbox` member specifies a pointer to the mailbox data structure for hardware platforms that access I/O space through hardware mailboxes. This member is set by the adapter code. As the adapter code probes the buses, the first bus that has a mailbox allocates the initial software mailbox data structure and sets its `bus_mbox` pointer to that data structure. As the adapter code continues to probe for buses and controllers, the `MBOX_GET` macro allocates the `ctlr_mbox` members for devices accessed by mailboxes. Typically, this macro is called in the controller configuration interface after the `controller` data structure has been found, but before probing the controller. The following code fragment gives an idea of the tasks that occur prior to calling the `MBOX_GET` macro:

```
config_con(name, node, bus)
{
   register struct controller *ctlr;

   if((ctlr = get_ctlr(name, node, bus->bus_name, bus->bus_num)) ||
          /* other wildcards here */
      (ctlr = get_ctlr(name, -1, "*", -99))) {
   •
   •
   •
       }

/**************************************************
 * Found the controller                           *
 **************************************************/

          int savebusnum;
          char *savebusname;
          int saveslot;

          if(ctlr->alive & ALV_ALIVE) {
                  printf("config_con: %s%d alive0,
                         ctlr->ctlr_name, ctlr->ctlr_num);
                  return(stat);
          }
          savebusnum = ctlr->bus_num;
          savebusname = ctlr->bus_name;
          saveslot = ctlr->slot;
          ctlr->bus_name = bus->bus_name;
          ctlr->bus_num = bus->bus_num;
          ctlr->slot = node;

/***************************************************
 *            Allocate and initialize a software    *
 *            mailbox data structure for the        *
 *            controller if it is attached to a     *
 *            bus that is accessed by mailboxes     *
 ***************************************************/
          MBOX_GET(bus, ctlr);

/***************************************************
 * Now get the controller's driver structure       *
 * and probe                                       *
 ***************************************************/
   •
   •
   •
}
```

7.3.2 The bus_type Member

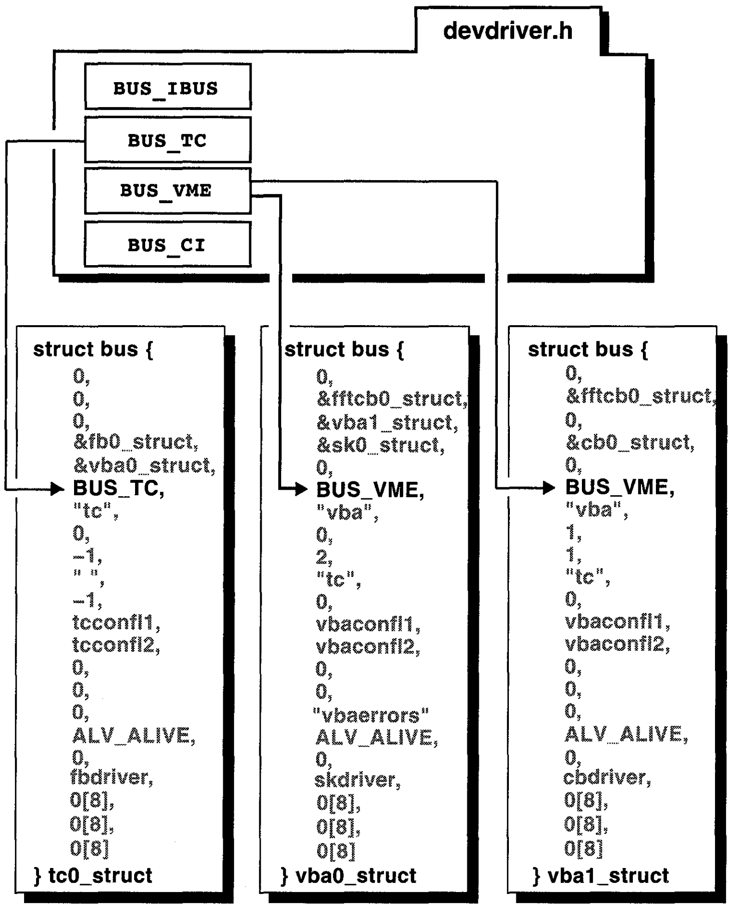
The bus_type member specifies the type of bus. The devdriver.h file defines constants that represent a variety of bus types.

For the example system configuration file, Figure 7-4 shows that the autoconfiguration software sets the bus_type member of tc0_struct to BUS_TC (a TURBOchannel bus). Likewise, the autoconfiguration software sets the bus_type members of vba_struct and vba1_struct to BUS_VME (a VMEbus).

Similarly for loadable drivers, the autoconfiguration software sets the bus_type members for any buses in the sysconfigtab database.

Device driver writers seldom need to reference this member. However, bus adapter driver writers might reference this member in their bus adapter code.

Figure 7-4: The bus_type Member Initialized



7.3.3 The bus_name and bus_num Members

The `bus_name` member specifies the bus name. The `bus_num` member specifies the bus number of this bus.

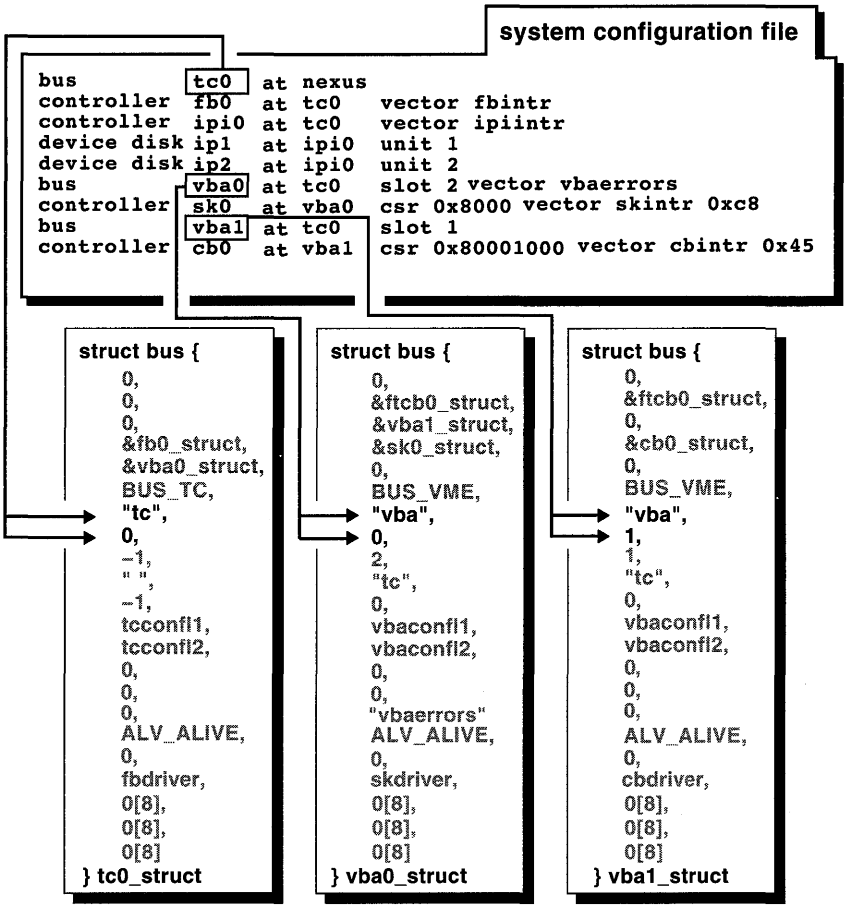
You (or the system manager) specify the bus name and the bus number by using the keyword `bus` followed by a character string and number that represent the bus name and bus number. You enter these values in the `config.file` file fragment or the system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers. For example, `tc0` specifies a TURBOchannel bus with a bus number of zero (0).

Figure 7-5 shows the values in the example system configuration file that the autoconfiguration software parses to obtain the bus name and bus number. It also shows that the autoconfiguration software sets the `bus_name` and `bus_num` members to these values for `tc0_struct`, `vba0_struct`, and `vba1_struct`.

The autoconfiguration software performs a similar parsing operation in the `sysconfigtab` database for loadable drivers to obtain the bus name and bus number and to initialize the associated `bus_name` and `bus_num` members.

Device driver writers seldom need to reference the `bus_name` and `bus_num` members in their device drivers. Instead, driver writers often pass a pointer to a `bus` structure to the `handler` interfaces, and the interrupt handlers use the information contained in these members.

Figure 7-5: The bus_name and bus_num Members Initialized



```
bus         tc0   at nexus
controller  fb0   at tc0    vector fbintr
controller  ipi0  at tc0    vector ipiintr
device disk ip1   at ipi0   unit 1
device disk ip2   at ipi0   unit 2
bus         vba0  at tc0    slot 2 vector vbaerrors
controller  sk0   at vba0   csr 0x8000 vector skintr 0xc8
bus         vba1  at tc0    slot 1
controller  cb0   at vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct bus {            struct bus {            struct bus {
    0,                      0,                      0,
    0,                      &ftcb0_struct,          &ftcb0_struct,
    0,                      &vba1_struct,           0,
    &fb0_struct,            &sk0_struct,            &cb0_struct,
    &vba0_struct,           0,                      0,
    BUS_TC,                 BUS_VME,                BUS_VME,
    "tc",                   "vba",                  "vba",
    0,                      0,                      1,
    -1,                     2,                      1,
    " ",                    "tc",                   "tc",
    -1,                     0,                      0,
    tcconfl1,               vbaconfl1,              vbaconfl1,
    tcconfl2,               vbaconfl2,              vbaconfl2,
    0,                      0,                      0,
    0,                      0,                      0,
    0,                      "vbaerrors"             0,
    ALV_ALIVE,              ALV_ALIVE,              ALV_ALIVE,
    0,                      0,                      0,
    fbdriver,               skdriver,               cbdriver,
    0[8],                   0[8],                   0[8],
    0[8],                   0[8],                   0[8],
    0[8]                    0[8]                    0[8]
} tc0_struct            } vba0_struct           } vba1_struct
```

7.3.4 The slot, connect_bus, and connect_num Members

The `slot` member specifies the bus slot or node number. The `connect_bus` member specifies the name of the bus that this bus is connected to. The `connect_num` member specifies the number of the bus that this bus is connected to.

You (or the system manager) specify the bus slot or node number by using the `slot` keyword followed by a valid number. You specify the bus name and bus number that this bus is connected to by using the keyword `at` followed by a character string or the wildcard character (a question mark (?)), and number that represent the bus name and bus number. For static drivers, if the bus is the system bus, you use the keyword `nexus`. For loadable drivers, you never identify the system bus with this keyword because the autoconfiguration software always assumes there is a system bus. Because it is not valid for the system bus to be loadable, the `nexus` keyword is invalid for loadable drivers. You enter these values in the `config.file` file fragment or the system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers.

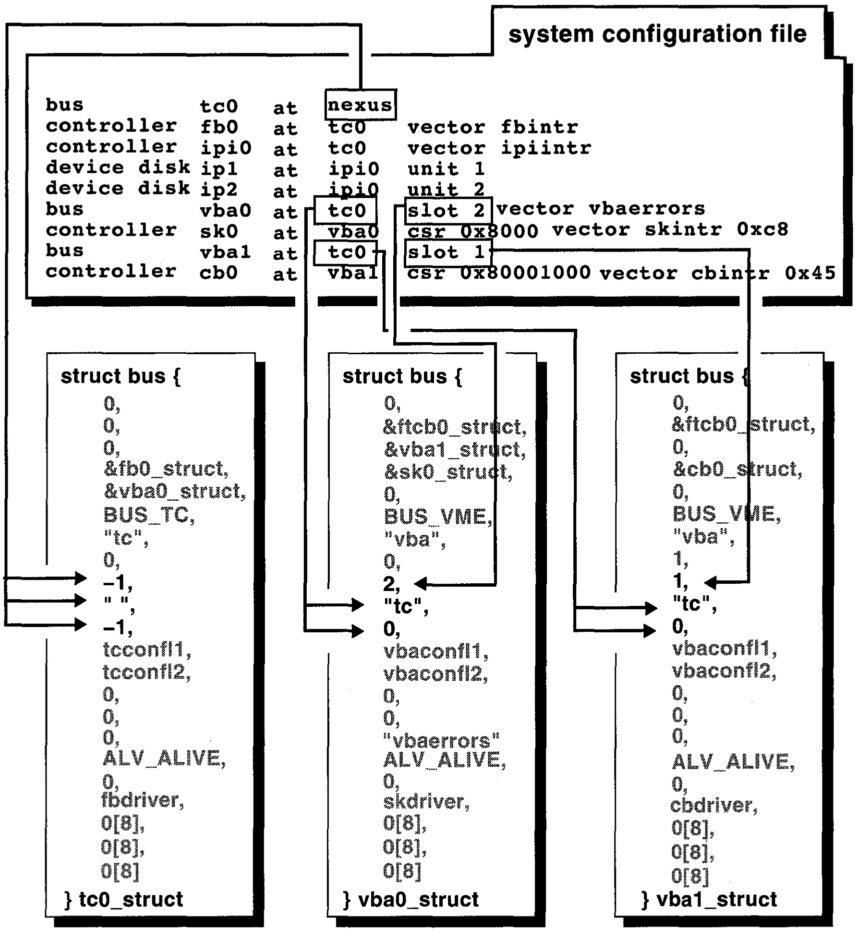
Figure 7-6 shows the values in the example system configuration file that the autoconfiguration software parses to obtain the bus slot, bus name, and bus number for the system bus. The autoconfiguration software uses the keyword `nexus` to identify the system bus. Because the slot number is not specified in the system configuration file, the `slot` member of the system bus structure, `tc0_struct`, defaults to the value –1. Because the system bus is not connected to any other bus, the autoconfiguration software sets its `connect_bus` member to the null string and its `connect_num` member to the value –1.

Figure 7-6 also shows the values in the example system configuration file that the autoconfiguration software parses to obtain the bus slot, bus name, and bus number for the other specified buses. It uses the keyword `slot` to set the `slot` member for `vba_struct` and `vba1_struct` to the values 2 and 1. It also sets the `connect_bus` and `connect_num` members of `vba_struct` and `vba1_struct` to the values tc and zero (0).

The autoconfiguration software performs a similar parsing operation in the `sysconfigtab` database to obtain the slot, bus name, and bus number and to initialize the associated `slot`, `bus_name`, and `bus_num` members. The only difference is that the autoconfiguration software locates the system bus by name (for example, the string `tc` represents a TURBOchannel bus), without using the `nexus` keyword.

Device driver writers seldom need to reference these members in their device drivers. However, bus adapter driver writers might reference these members in their bus adapter code.

Figure 7-6: The slot, connect_bus, and connect_num Members Initialized



```
bus            tc0   at  nexus
controller     fb0   at  tc0    vector fbintr
controller     ipi0  at  tc0    vector ipiintr
device disk    ip1   at  ipi0   unit 1
device disk    ip2   at  ipi0   unit 2
bus            vba0  at  tc0    slot 2 vector vbaerrors
controller     sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus            vba1  at  tc0    slot 1
controller     cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct bus {
      0,
      0,
      0,
      &fb0_struct,
      &vba0_struct,
      BUS_TC,
      "tc",
      0,
      -1,
      " ",
      -1,
      tcconfl1,
      tcconfl2,
      0,
      0,
      0,
      ALV_ALIVE,
      0,
      fbdriver,
      0[8],
      0[8],
      0[8]
} tc0_struct
```

```
struct bus {
      0,
      &ftcb0_struct,
      &vba1_struct,
      &sk0_struct,
      0,
      BUS_VME,
      "vba",
      0,
      2,
      "tc",
      0,
      vbaconfl1,
      vbaconfl2,
      0,
      0,
      "vbaerrors"
      ALV_ALIVE,
      0,
      skdriver,
      0[8],
      0[8],
      0[8]
} vba0_struct
```

```
struct bus {
      0,
      &ftcb0_struct,
      0,
      &cb0_struct,
      0,
      BUS_VME,
      "vba",
      1,
      1,
      "tc",
      0,
      vbaconfl1,
      vbaconfl2,
      0,
      0,
      0,
      ALV_ALIVE,
      0,
      cbdriver,
      0[8],
      0[8],
      0[8]
} vba1_struct
```

7.3.5 The confl1 and confl2 Members

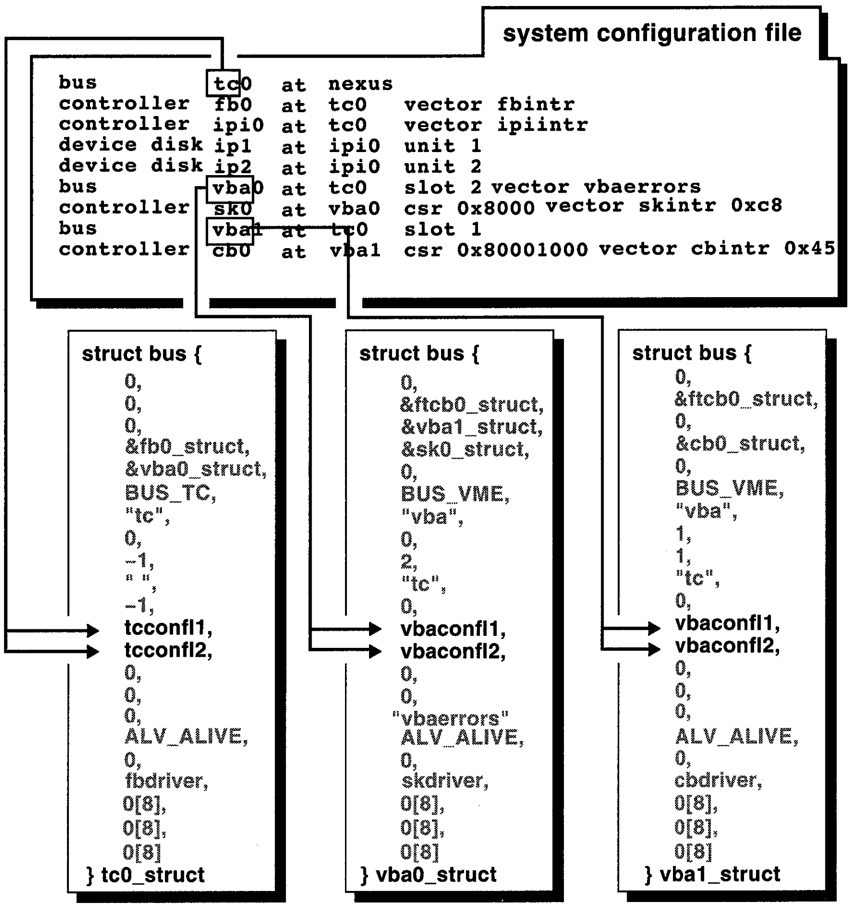
The `confl1` member specifies a pointer to an entry point of the level 1 bus configuration interface. The `confl2` member specifies a pointer to an entry point of the level 2 bus configuration interface. These interfaces are not typically used by device driver writers, but by systems engineers who want to implement a configuration procedure for a specific bus.

Figure 7-7 shows that the autoconfiguration software initializes the confl1 and confl2 members of the tc0_struct, vba0_struct, and vba1_struct to their respective level 1 and level 2 bus configuration interfaces.

The autoconfiguration software obtains the names of these interfaces by using the bus name and appending the string confl1 or confl2. Thus, tcconfl1 and tcconfl2 are the level 1 and level 2 bus configuration interfaces for the system bus, tc0_struct.

The autoconfiguration software performs a similar operation by using entries in the sysconfigtab database to initialize the confl1 and confl2 members of the associated bus structures.

Figure 7-7: The confl1 and confl2 Members Initialized



```
bus           tc0   at  nexus
controller    fb0   at  tc0    vector fbintr
controller    ipi0  at  tc0    vector ipiintr
device disk   ip1   at  ipi0   unit 1
device disk   ip2   at  ipi0   unit 2
bus           vba0  at  tc0    slot 2 vector vbaerrors
controller    sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus           vba1  at  tc0    slot 1
controller    cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct bus {
    0,
    0,
    0,
    &fb0_struct,
    &vba0_struct,
    BUS_TC,
    "tc",
    0,
    -1,
    " ",
    -1,
    tcconfl1,
    tcconfl2,
    0,
    0,
    0,
    ALV_ALIVE,
    0,
    fbdriver,
    0[8],
    0[8],
    0[8]
} tc0_struct
```

```
struct bus {
    0,
    &ftcb0_struct,
    &vba1_struct,
    &sk0_struct,
    0,
    BUS_VME,
    "vba",
    0,
    2,
    "tc",
    0,
    vbaconfl1,
    vbaconfl2,
    0,
    0,
   "vbaerrors"
    ALV_ALIVE,
    0,
    skdriver,
    0[8],
    0[8],
    0[8]
} vba0_struct
```

```
struct bus {
    0,
    &ftcb0_struct,
    0,
    &cb0_struct,
    0,
    BUS_VME,
    "vba",
    1,
    1,
    "tc",
    0,
    vbaconfl1,
    vbaconfl2,
    0,
    0,
    0,
    ALV_ALIVE,
    0,
    cbdriver,
    0[8],
    0[8],
    0[8]
} vba1_struct
```

7.3.6 The pname and port Members

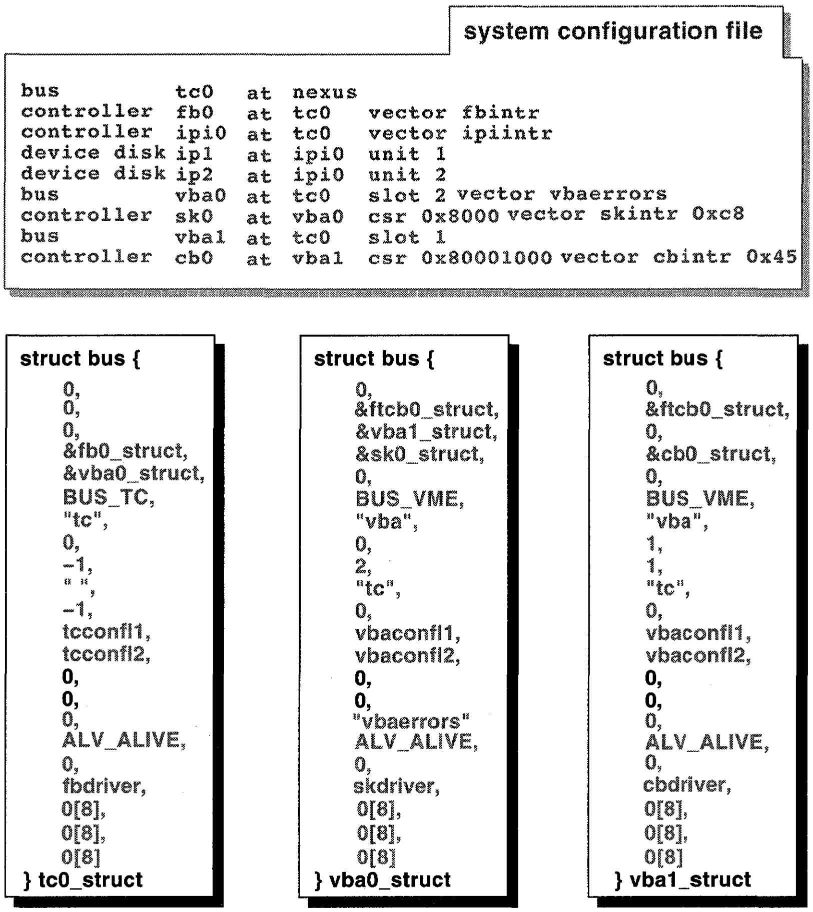
The `pname` member specifies a pointer to the port name for this bus, if applicable. The `port` member specifies a pointer to the `port` structure for this bus, if applicable.

You (or the system manager) specify the port name by using the `port` keyword followed by a string that represents the name of the port. You enter this keyword and string in the `config.file` file fragment or system configuration file for static drivers. This keyword is not currently supported for loadable drivers; therefore, you cannot specify it in the `stanza.loadable` file fragment.

Figure 7-8 shows that the autoconfiguration software sets these members for all of the structures to the value zero (0) to indicate that no port was specified in the system configuration file. If port names were specified, the autoconfiguration software would initialize the `pname` and `port` members for all of the example `bus` structures to the specified values.

Device driver writers do not reference these members in their device drivers. However, bus adapter writers use the `port` structure to implement the initialization for a class of devices or controllers that have common characteristics.

Figure 7-8: The pname and port Members Initialized



```
bus          tc0   at  nexus
controller   fb0   at  tc0    vector fbintr
controller   ipi0  at  tc0    vector ipiintr
device disk  ip1   at  ipi0   unit 1
device disk  ip2   at  ipi0   unit 2
bus          vba0  at  tc0    slot 2 vector vbaerrors
controller   sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus          vba1  at  tc0    slot 1
controller   cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct bus {
      0,
      0,
      0,
      &fb0_struct,
      &vba0_struct,
      BUS_TC,
      "tc",
      0,
      -1,
      " ",
      -1,
      tcconfl1,
      tcconfl2,
      0,
      0,
      0,
      ALV_ALIVE,
      0,
      fbdriver,
      0[8],
      0[8],
      0[8]
} tc0_struct
```

```
struct bus {
      0,
      &ftcb0_struct,
      &vba1_struct,
      &sk0_struct,
      0,
      BUS_VME,
      "vba",
      0,
      2,
      "tc",
      0,
      vbaconfl1,
      vbaconfl2,
      0,
      0,
      "vbaerrors"
      ALV_ALIVE,
      0,
      skdriver,
      0[8],
      0[8],
      0[8]
} vba0_struct
```

```
struct bus {
      0,
      &ftcb0_struct,
      0,
      &cb0_struct,
      0,
      BUS_VME,
      "vba",
      1,
      1,
      "tc",
      0,
      vbaconfl1,
      vbaconfl2,
      0,
      0,
      0,
      ALV_ALIVE,
      0,
      cbdriver,
      0[8],
      0[8],
      0[8]
} vba1_struct
```

7.3.7 The intr Member

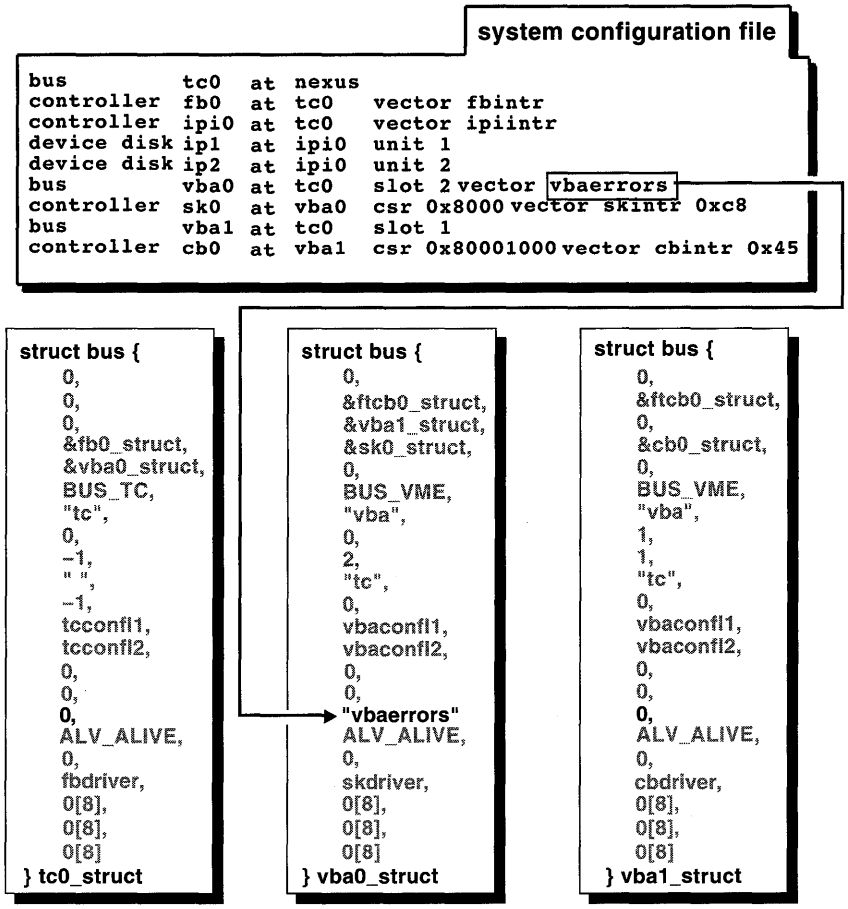
The `intr` member specifies an array that contains an entry point or points for the bus interrupt interfaces. You specify the bus interrupt interface or interfaces by using the `vector` keyword followed by a string that represents the name of the bus interrupt interface. You enter this keyword and string in the `config.file` file fragment or system configuration file for static drivers.

A loadable driver knows the name of its bus interrupt interface. The loadable driver uses the `handler_add` and `handler_enable` interfaces to register its interrupt handlers.

Figure 7-9 shows that the autoconfiguration software initializes this member to the value `vbaerrors` for `vba0_struct`. The figure also shows that the autoconfiguration software initializes this member for all of the other structures in the example to the value zero (0) because there are no bus interrupt interfaces associated with these buses.

Device driver writers seldom need to reference this member in their device drivers. However, bus adapter driver writers might use this member in their bus adapter code to reference the interrupt handlers that handle error interrupts related to the bus.

Figure 7-9: The intr Member Initialized



```
bus            tc0   at  nexus
controller     fb0   at  tc0    vector fbintr
controller     ipi0  at  tc0    vector ipiintr
device disk    ip1   at  ipi0   unit 1
device disk    ip2   at  ipi0   unit 2
bus            vba0  at  tc0    slot 2 vector vbaerrors
controller     sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus            vba1  at  tc0    slot 1
controller     cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct bus {
        0,
        0,
        0,
        &fb0_struct,
        &vba0_struct,
        BUS_TC,
        "tc",
        0,
        -1,
        " ",
        -1,
        tcconfl1,
        tcconfl2,
        0,
        0,
        0,
        ALV_ALIVE,
        0,
        fbdriver,
        0[8],
        0[8],
        0[8]
} tc0_struct
```

```
struct bus {
        0,
        &ftcb0_struct,
        &vba1_struct,
        &sk0_struct,
        0,
        BUS_VME,
        "vba",
        0,
        2,
        "tc",
        0,
        vbaconfl1,
        vbaconfl2,
        0,
        0,
        "vbaerrors"
        ALV_ALIVE,
        0,
        skdriver,
        0[8],
        0[8],
        0[8]
} vba0_struct
```

```
struct bus {
        0,
        &ftcb0_struct,
        0,
        &cb0_struct,
        0,
        BUS_VME,
        "vba",
        1,
        1,
        "tc",
        0,
        vbaconfl1,
        vbaconfl2,
        0,
        0,
        0,
        ALV_ALIVE,
        0,
        cbdriver,
        0[8],
        0[8],
        0[8]
} vba1_struct
```

7.3.8 The alive Member

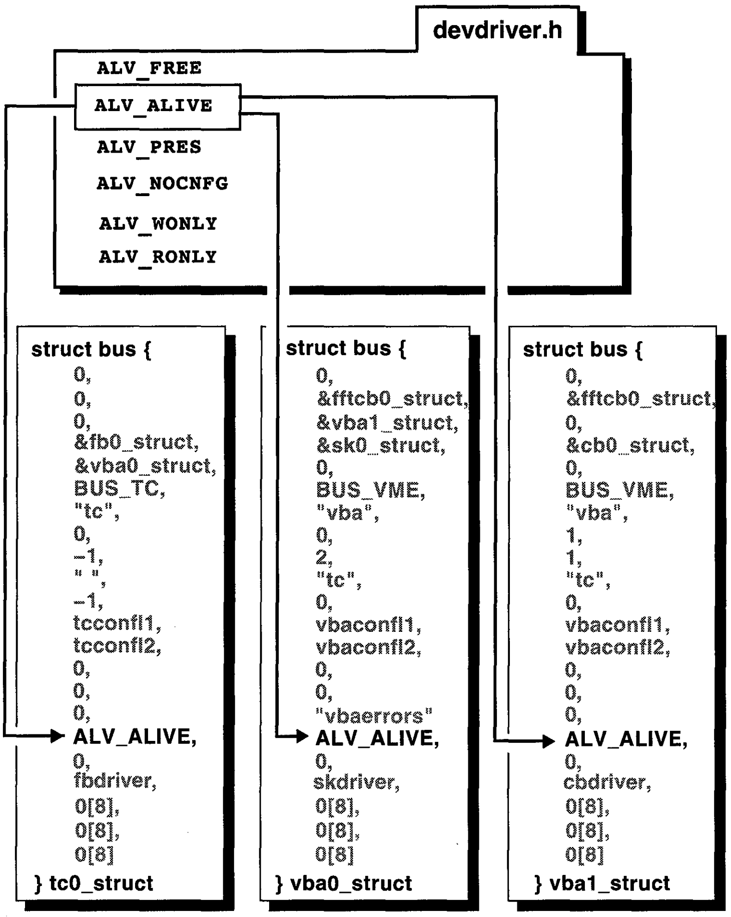
The `alive` member specifies a flag word to indicate the current status of the bus. This member is set by the system to the bitwise inclusive OR of the valid alive bits defined in `/usr/sys/include/io/common/devdriver.h`.

For the example system configuration file, Figure 7-10 shows that the autoconfiguration software initializes the `alive` member for all of the example `bus` structures to the bit `ALV_ALIVE`. This bit indicates that the device is present and configured on the system.

Similarly for loadable drivers, the autoconfiguration software (specifically, the `ldbl_stanza_resolver` interface) sets the `alive` member for any buses in the `sysconfigtab` database to the bitwise inclusive OR of the valid alive bits defined in `/usr/sys/include/io/common/devdriver.h`. The following list shows the valid alive bits that loadable and static drivers can use:

`ALV_FREE`	The bus is not yet processed.
`ALV_ALIVE`	The bus is alive and configured.
`ALV_PRES`	The bus is present but not yet configured.
`ALV_NOCNFG`	The bus is not to be configured.
`ALV_LOADABLE`	The bus is present and resolved as loadable.
`ALV_NOSIZER`	The `sizer` program should ignore these `bus` structures. This bit is set for loadable drivers. It indicates that this `bus` structure is not part of the static configuration.
`ALV_RONLY`	The device is read-only. This bit is set for static drivers if the corresponding entry is specified in the `stanza.static` file fragment or system configuration file. This bit is not supported for loadable drivers.
`ALV_WONLY`	The device is write-only. This bit is set for static drivers if the corresponding entry is specified in the `stanza.static` file fragment or system configuration file. This bit is not supported for loadable drivers.

Figure 7-10: The alive Member Initialized



7.3.9 The framework and driver_name Members

The `framework` member specifies a pointer to the `bus_framework` structure. This structure contains pointers to bus interfaces for loadable device drivers. These interfaces provide dynamic extensions to bus functionality. They are used in the autoconfiguration of loadable drivers to perform bus-specific tasks, such as the registration of interrupt handlers. The `framework` member is initialized by the bus adapter driver. Device driver writers seldom need to reference this member in their device drivers. However, bus adapter driver writers might reference this member in their bus adapter code.

The `driver_name` member specifies the name of the controlling device driver. For the example system configuration file, Figure 7-11 shows that the autoconfiguration software initializes the `driver_name` members for the example `bus` structures to the addresses of their respective controlling device drivers: `fbdriver`, `skdriver`, and `cbdriver`. The autoconfiguration software obtains the driver names by using the controller name and appending the string `driver` to it.

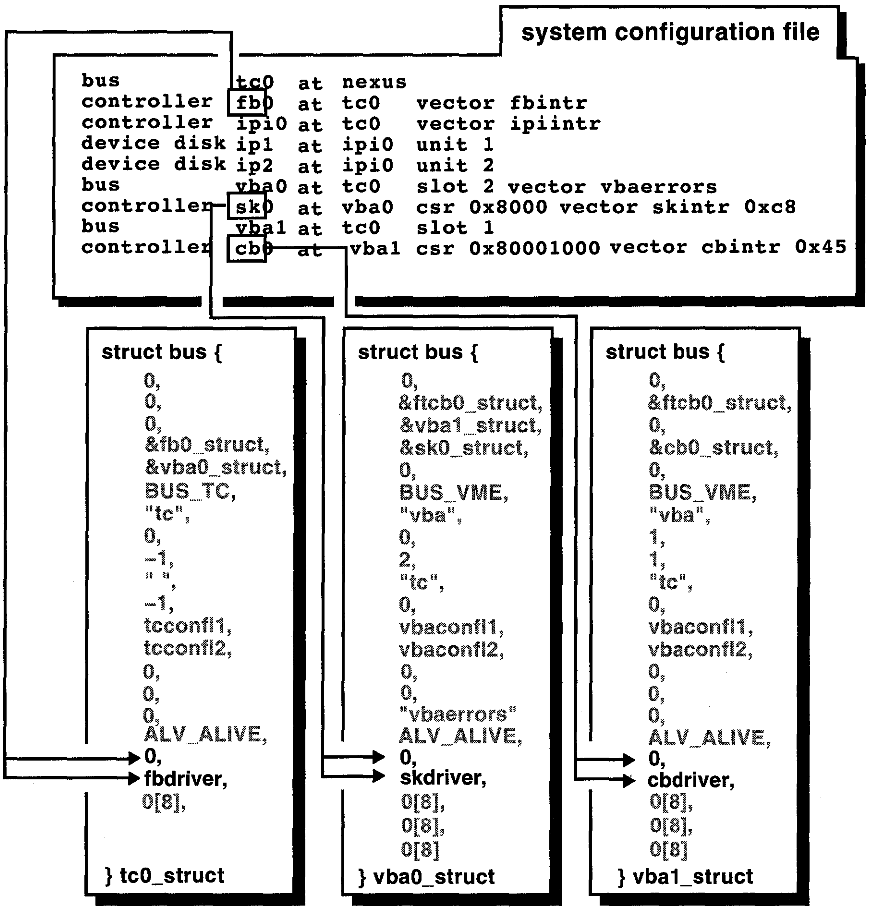
Similarly for loadable drivers, the autoconfiguration software sets the `driver_name` members to *xx*`driver`, where *xx* is the driver name as specified by the `Module_Config_Name` field in the `stanza.loadable` file fragment. The following example shows one such entry:

```
   •
   •
   •
Module_Config_Name = cb
   •
   •
   •
```

In this case, the autoconfiguration software sets the `driver_name` member to `cbdriver`.

Device driver writers seldom need to reference this member in their device drivers. However, bus adapter driver writers might reference this member in their bus adapter code.

Figure 7-11: The framework and driver_name Members Initialized

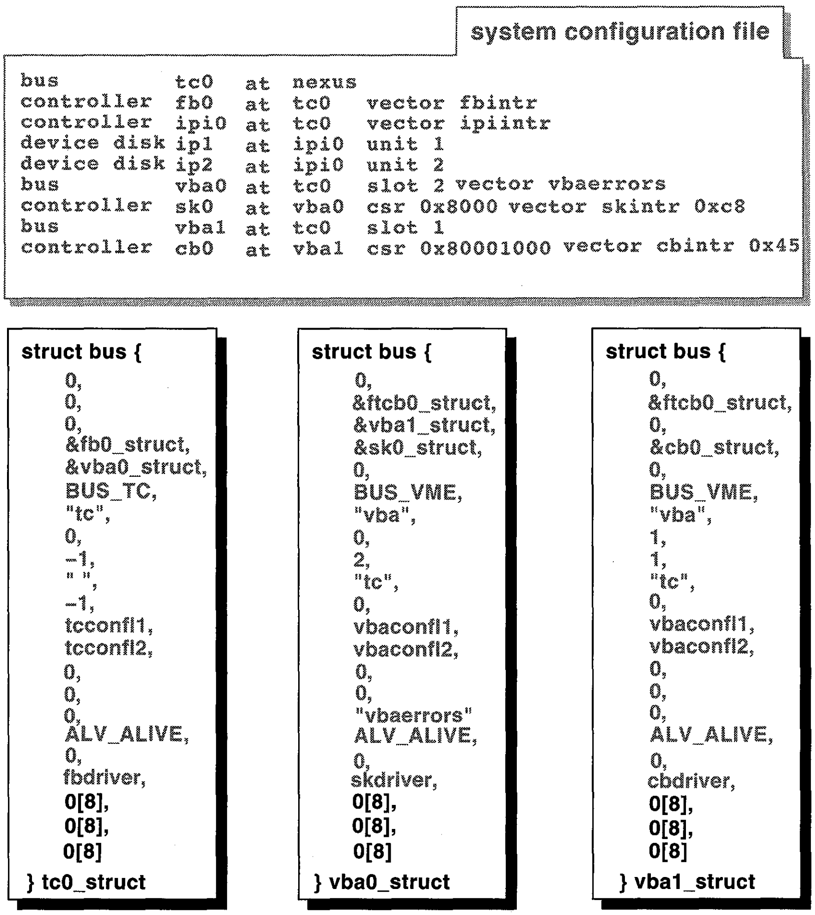

system configuration file
bus tc0 at nexus
controller fb0 at tc0 vector fbintr
controller ipi0 at tc0 vector ipiintr
device disk ip1 at ipi0 unit 1
device disk ip2 at ipi0 unit 2
bus vba0 at tc0 slot 2 vector vbaerrors
controller sk0 at vba0 csr 0x8000 vector skintr 0xc8
bus vba1 at tc0 slot 1
controller cb0 at vba1 csr 0x80001000 vector cbintr 0x45
struct bus { 0, 0, 0, &fb0_struct, &vba0_struct, BUS_TC, "tc", 0, -1, " ", -1, tcconfl1, tcconfl2, 0, 0, 0, ALV_ALIVE, 0, fbdriver, 0[8], } tc0_struct
struct bus { 0, &ftcb0_struct, &vba1_struct, &sk0_struct, 0, BUS_VME, "vba", 0, 2, "tc", 0, vbaconfl1, vbaconfl2, 0, 0, "vbaerrors" ALV_ALIVE, 0, skdriver, 0[8], 0[8], 0[8] } vba0_struct
struct bus { 0, &ftcb0_struct, 0, &cb0_struct, 0, BUS_VME, "vba", 1, 1, "tc", 0, vbaconfl1, vbaconfl2, 0, 0, 0, ALV_ALIVE, 0, cbdriver, 0[8], 0[8], 0[8] } vba1_struct

7.3.10 The private, conn_priv, and rsvd Members

The `private` member specifies private storage for use by this bus or bus class. The `conn_priv` member specifies private storage for use by the bus that this bus is connected to. The `rsvd` member is reserved for future expansion of the data structure.

For the example system configuration file, Figure 7-12 shows that the system initializes these members for all of the structures to the value zero (0). The code controlling the bus can use these members for any storage purposes. The `conn_priv` member is often used by TUROBchannel bus writers as an index to the `tc_option` table.

Figure 7-12: The private, conn_priv, and rsvd Members Initialized



```
bus          tc0  at  nexus
controller   fb0  at  tc0   vector fbintr
controller   ipi0 at  tc0   vector ipiintr
device disk  ip1  at  ipi0  unit 1
device disk  ip2  at  ipi0  unit 2
bus          vba0 at  tc0   slot 2 vector vbaerrors
controller   sk0  at  vba0  csr 0x8000 vector skintr 0xc8
bus          vba1 at  tc0   slot 1
controller   cb0  at  vba1  csr 0x80001000 vector cbintr 0x45
```

```
struct bus {
    0,
    0,
    0,
    &fb0_struct,
    &vba0_struct,
    BUS_TC,
    "tc",
    0,
    -1,
    " ",
    -1,
    tcconfl1,
    tcconfl2,
    0,
    0,
    0,
    ALV_ALIVE,
    0,
    fbdriver,
    0[8],
    0[8],
    0[8]
} tc0_struct
```

```
struct bus {
    0,
    &ftcb0_struct,
    &vba1_struct,
    &sk0_struct,
    0,
    BUS_VME,
    "vba",
    0,
    2,
    "tc",
    0,
    vbaconfl1,
    vbaconfl2,
    0,
    0,
    "vbaerrors"
    ALV_ALIVE,
    0,
    skdriver,
    0[8],
    0[8],
    0[8]
} vba0_struct
```

```
struct bus {
    0,
    &ftcb0_struct,
    0,
    &cb0_struct,
    0,
    BUS_VME,
    "vba",
    1,
    1,
    "tc",
    0,
    vbaconfl1,
    vbaconfl2,
    0,
    0,
    0,
    ALV_ALIVE,
    0,
    cbdriver,
    0[8],
    0[8],
    0[8]
} vba1_struct
```

7.4 The controller Structure

The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation. You (or the system manager) specify a controller entity as follows:

- For static drivers

  Specify a valid syntax for the controller in the `config.file` file fragment or the system configuration file.

- For loadable drivers

  Specify a valid syntax for the controller in the `stanza.loadable` file fragment.

Chapter 11 discusses the device driver configuration models. Chapter 12 describes the valid syntaxes for a controller specification. Chapter 13 provides examples of how to configure device drivers.

Table 7-2 lists the members of the `controller` structure along with their associated data types.

Table 7-2: Members of the controller Structure

Member Name	Data Type
`ctlr_mbox`	`u_long`
`bus_hd`	`struct bus *`
`nxt_ctlr`	`struct controller *`
`dev_list`	`struct device *`
`driver`	`struct driver *`
`ctlr_type`	`int`
`ctlr_name`	`char *`
`ctlr_num`	`int`
`bus_name`	`char *`
`bus_num`	`int`
`rctlr`	`int`
`slot`	`int`
`alive`	`int`
`pname`	`char *`
`port`	`struct port *`
`intr`	`int (**intr)()`
`addr`	`caddr_t`
`addr2`	`caddr_t`

Table 7-2: (continued)

Member Name	Data Type
`flags`	`int`
`bus_priority`	`int`
`ivnum`	`int`
`priority`	`int`
`cmd`	`int`
`physaddr`	`caddr_t`
`physaddr2`	`caddr_t`
`private`	`void * [8]`
`conn_priv`	`void * [8]`
`rsvd`	`void * [8]`

The following sections discuss all of these members except for `bus_hd`, `nxt_ctlr`, and `dev_list`, which are presented in Section 7.2.1.6. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of this data structure.

7.4.1 The ctlr_mbox Member

The `ctlr_mbox` member specifies a pointer to the mailbox data structure for hardware platforms that access I/O space through hardware mailboxes. This member is set by the adapter code. As the adapter code probes the buses, the first bus that has a mailbox allocates the initial software mailbox data structure and sets its `bus_mbox` pointer to that data structure. As the adapter code continues to probe for buses and controllers, the `MBOX_GET` macro allocates the `ctlr_mbox` members for devices accessed by mailboxes. Typically, this macro is called in the controller configuration interface after the `controller` data structure has been found, but before probing the controller. The following code fragment gives an idea of the tasks that occur prior to calling the `MBOX_GET` macro:

```
config_con(name, node, bus)
{
   register struct controller *ctlr;

   if((ctlr = get_ctlr(name, node, bus->bus_name, bus->bus_num)) ||
          /* other wildcards here */
      (ctlr = get_ctlr(name, -1, "*", -99))) {
   •
   •
   •
       }
/***************************************************
 * Found the controller                            *
 ***************************************************/

          int savebusnum;
          char *savebusname;
          int saveslot;

          if(ctlr->alive & ALV_ALIVE) {
                  printf("config_con: %s%d alive0,
                         ctlr->ctlr_name, ctlr->ctlr_num);
                  return(stat);
          }
          savebusnum = ctlr->bus_num;
          savebusname = ctlr->bus_name;
          saveslot = ctlr->slot;
          ctlr->bus_name = bus->bus_name;
          ctlr->bus_num = bus->bus_num;
          ctlr->slot = node;

/***************************************************
 *            Allocate and initialize a software   *
 *            mailbox data structure for the       *
 *            controller if it is attached to a    *
 *            bus that is accessed by mailboxes    *
 ***************************************************/
          MBOX_GET(bus, ctlr);

/***************************************************
 * Now get the controller's driver structure       *
 * and probe                                       *
 ***************************************************/
   •
   •
   •
}
```

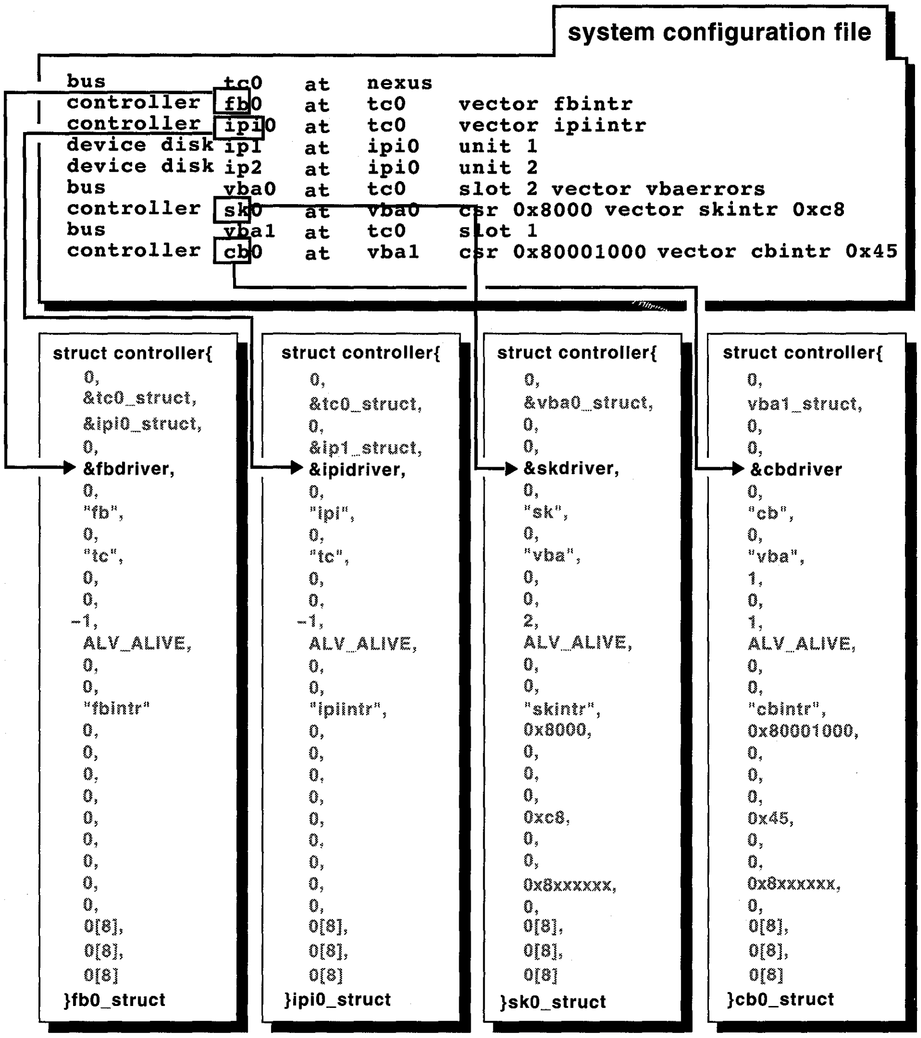
7.4.2 The driver Member

The `driver` member specifies a pointer to the `driver` structure for this controller. The device driver writer initializes a `driver` structure, usually in the Declarations Section of the driver.

For the example system configuration file, Figure 7-13 shows that the autoconfiguration software initializes the `driver` member for the example `controller` structures to their respective controlling device drivers: `fbdriver`, `ipidriver`, `skdriver`, and `cbdriver`. The autoconfiguration software obtains the driver names by using the keyword for the controller and appending the string `driver` to it.

Similarly for loadable drivers, the driver writer provides the device driver name in the Configure Section of the device driver by calling the `ldbl_stanza_resolver` and `ldbl_ctlr_configure` interfaces. Thus, the autoconfiguration software sets the `driver` members for any controllers specified in the `sysconfigtab` database to the name specified by the driver writer.

Figure 7-13: The driver Member Initialized



```
bus             tc0    at   nexus
controller      fb0    at   tc0     vector fbintr
controller      ipi0   at   tc0     vector ipiintr
device disk     ip1    at   ipi0    unit 1
device disk     ip2    at   ipi0    unit 2
bus             vba0   at   tc0     slot 2 vector vbaerrors
controller      sk0    at   vba0    csr 0x8000 vector skintr 0xc8
bus             vba1   at   tc0     slot 1
controller      cb0    at   vba1    csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
   0,
   &tc0_struct,
   &ipi0_struct,
   0,
   &fbdriver,
   0,
   "fb",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "fbintr"
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}fb0_struct
```

```
struct controller{
   0,
   &tc0_struct,
   0,
   &ip1_struct,
   &ipidriver,
   0,
   "ipi",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "ipiintr",
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}ipi0_struct
```

```
struct controller{
   0,
   &vba0_struct,
   0,
   0,
   &skdriver,
   0,
   "sk",
   0,
   "vba",
   0,
   0,
   2,
   ALV_ALIVE,
   0,
   0,
   "skintr",
   0x8000,
   0,
   0,
   0,
   0xc8,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}sk0_struct
```

```
struct controller{
   0,
   vba1_struct,
   0,
   0,
   &cbdriver
   0,
   "cb",
   0,
   "vba",
   1,
   0,
   1,
   ALV_ALIVE,
   0,
   0,
   "cbintr",
   0x80001000,
   0,
   0,
   0,
   0x45,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}cb0_struct
```

7.4.3 The ctlr_type, ctlr_name, and ctlr_num Members

The `ctlr_type` member specifies the controller type. The `ctlr_name` member specifies the controller name. The `ctlr_num` member specifies the controller number.

The `ctlr_type` member is not currently used by the autoconfiguration software. Thus, Figure 7-14 shows that in the example system configuration file, the autoconfiguration software initializes the `ctlr_type` members for the example `controller` structures to the value zero (0).

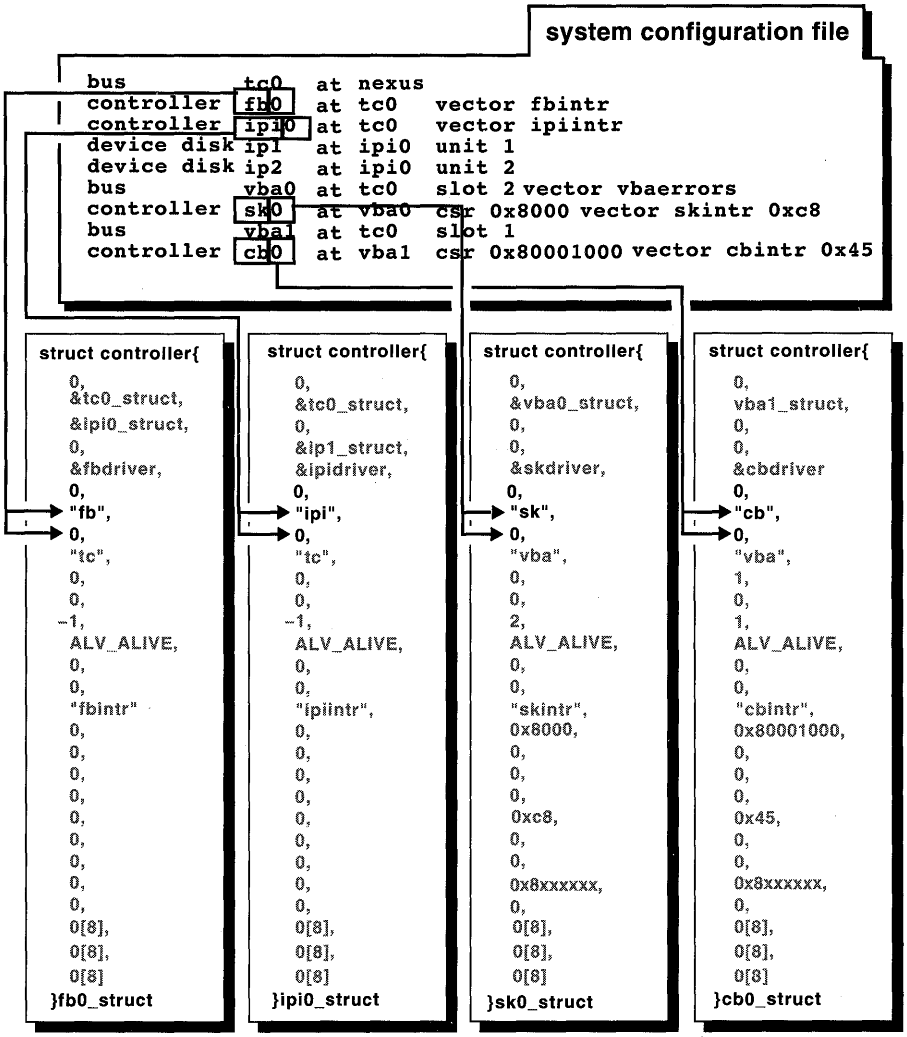
You (or the system manager) specify the controller name and the controller number by using the keyword `controller` followed by a character string and number that represent the controller name and controller number. You enter these values in the `config.file` file fragment or the system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers. For example, `fb0` specifies a graphics frame buffer controller with a controller number of zero (0).

Figure 7-14 shows the values in the example system configuration file that the autoconfiguration software parses to obtain the controller name and controller number. It also shows that the autoconfiguration software sets the `ctlr_name` and `ctlr_num` members to these values for `fb0_struct`, `ipi0_struct`, `sk0_struct`, and `cb0_struct`.

The autoconfiguration software performs a similar parsing operation in the `sysconfigtab` database for loadable drivers to obtain the controller name and controller number and to initialize the associated `ctlr_name` and `ctlr_num` members.

Driver writers are unlikely to use the `ctlr_name` member. Driver writers often use the `ctlr_num` member as an index to identify which instance of the controller the request is for. The `/dev/none` and `/dev/cb` device drivers illustrate the use of the `ctlr_num` member.

Figure 7-14: The ctlr_type, ctlr_name, and ctlr_num Members Initialized



```
bus          tc0   at nexus
controller   fb0   at tc0   vector fbintr
controller   ipi0  at tc0   vector ipiintr
device disk  ip1   at ipi0  unit 1
device disk  ip2   at ipi0  unit 2
bus          vba0  at tc0   slot 2 vector vbaerrors
controller   sk0   at vba0  csr 0x8000 vector skintr 0xc8
bus          vba1  at tc0   slot 1
controller   cb0   at vba1  csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
   -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr"
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
 }fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
   -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
 }ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
 }sk0_struct
```

```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
 }cb0_struct
```

7.4.4 The bus_name and bus_num Members

The `bus_name` member specifies the name of the bus to which this controller is connected. The `bus_num` member specifies the number of the bus to which this controller is connected.

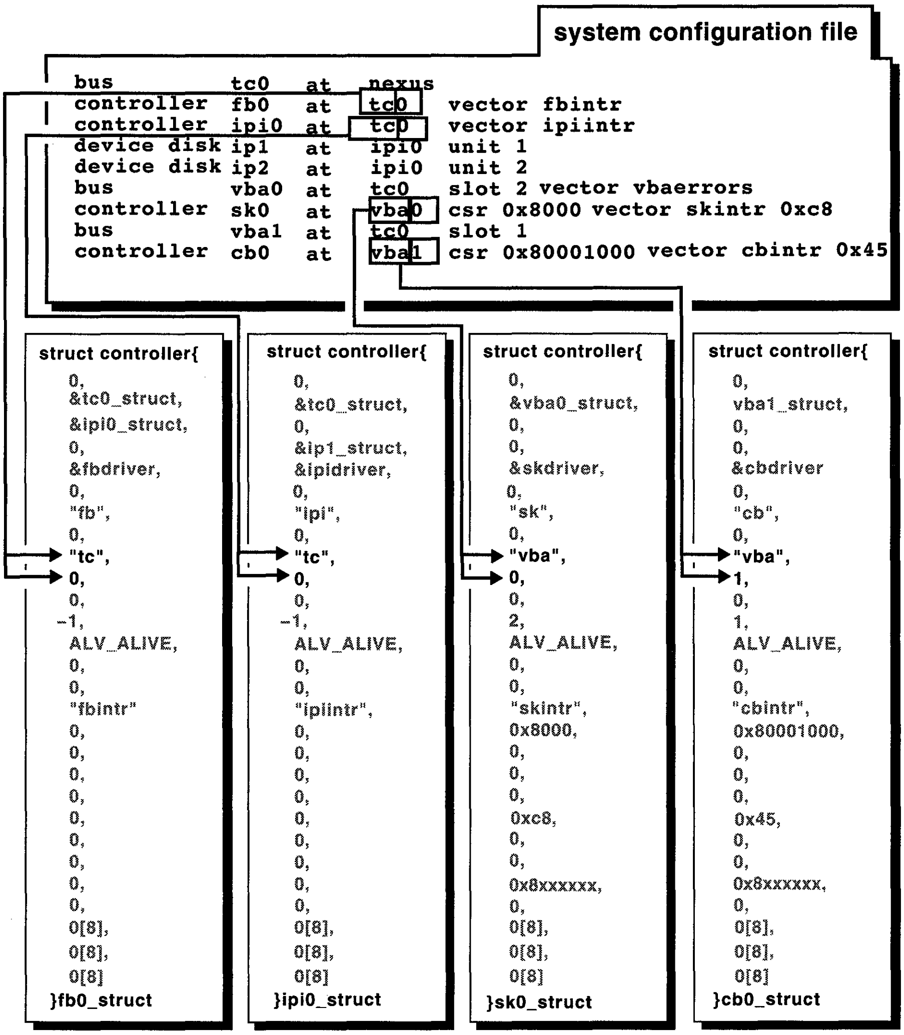
You (or the system manager) specify the bus name and the bus number by using the keyword `at` followed by a character string and number that represent the bus name and bus number. You enter these values in the `config.file` file fragment or the system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers. For example, `tc0` specifies a TURBOchannel bus with a bus number of zero (0).

Figure 7-15 shows the values in the example system configuration file that the autoconfiguration software parses to obtain the bus name and bus number. It also shows that the autoconfiguration software sets the `bus_name` and `bus_num` members to these values for `fb0_struct`, `ipi0_struct`, `sk0_struct`, and `cb0_struct`.

The autoconfiguration software performs a similar parsing operation in the `sysconfigtab` database for loadable drivers to obtain the bus name and bus number and to initialize the associated `bus_name` and `bus_num` members.

Device driver writers seldom need to reference these members in their device drivers. However, bus adapter driver writers might reference these members in their bus adapter code.

Figure 7-15: The bus_name and bus_num Members Initialized



```
bus            tc0   at  nexus
controller     fb0   at  tc0    vector fbintr
controller     ipi0  at  tc0    vector ipiintr
device disk    ip1   at  ipi0   unit 1
device disk    ip2   at  ipi0   unit 2
bus            vba0  at  tc0    slot 2 vector vbaerrors
controller     sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus            vba1  at  tc0    slot 1
controller     cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
  0,
  &tc0_struct,
  &ipi0_struct,
  0,
  &fbdriver,
  0,
  "fb",
  0,
  "tc",
  0,
  0,
 -1,
  ALV_ALIVE,
  0,
  0,
  "fbintr"
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}fb0_struct
```

```
struct controller{
  0,
  &tc0_struct,
  0,
  &ip1_struct,
  &ipidriver,
  0,
  "ipi",
  0,
  "tc",
  0,
  0,
 -1,
  ALV_ALIVE,
  0,
  0,
  "ipiintr",
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}ipi0_struct
```

```
struct controller{
  0,
  &vba0_struct,
  0,
  0,
  &skdriver,
  0,
  "sk",
  0,
  "vba",
  0,
  0,
  2,
  ALV_ALIVE,
  0,
  0,
  "skintr",
  0x8000,
  0,
  0,
  0,
  0xc8,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}sk0_struct
```

```
struct controller{
  0,
  vba1_struct,
  0,
  0,
  &cbdriver
  0,
  "cb",
  0,
  "vba",
  1,
  0,
  1,
  ALV_ALIVE,
  0,
  0,
  "cbintr",
  0x80001000,
  0,
  0,
  0,
  0x45,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}cb0_struct
```

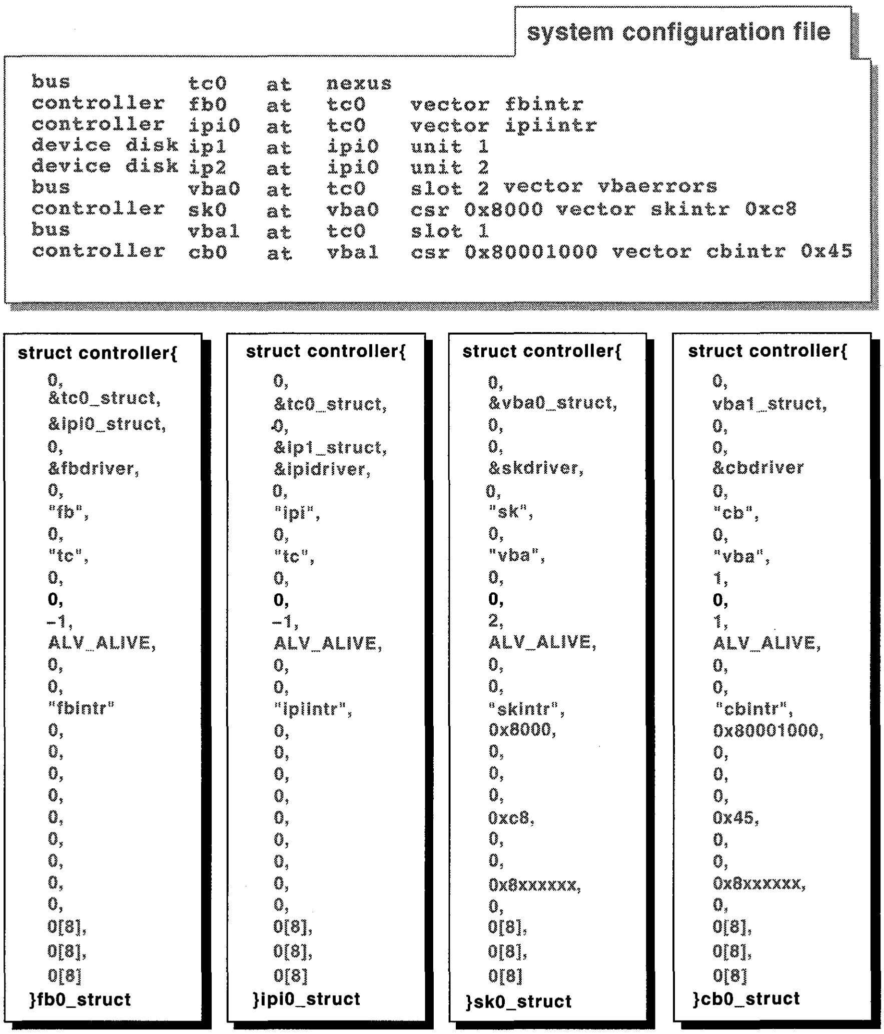
7.4.5 The rctlr Member

The `rctlr` member specifies the remote controller number (for example, the SCSI ID).

You (or the system manager) specify the remote controller number by using the `rctlr` keyword followed by a number. You enter this keyword and number in the `config.file` file fragment or system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers.

Figure 7-16 shows that the example system configuration file does not contain an entry for the remote controller. Therefore, the autoconfiguration software sets this member to the value zero (0) for each of the controller structures.

Figure 7-16: The rctlr Member Initialized



```
bus          tc0   at   nexus
controller   fb0   at   tc0    vector fbintr
controller   ipi0  at   tc0    vector ipiintr
device disk  ip1   at   ipi0   unit 1
device disk  ip2   at   ipi0   unit 2
bus          vba0  at   tc0    slot 2 vector vbaerrors
controller   sk0   at   vba0   csr 0x8000 vector skintr 0xc8
bus          vba1  at   tc0    slot 1
controller   cb0   at   vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
  0,
  &tc0_struct,
  &ipi0_struct,
  0,
  &fbdriver,
  0,
  "fb",
  0,
  "tc",
  0,
  0,
  -1,
  ALV_ALIVE,
  0,
  0,
  "fbintr"
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}fb0_struct
```

```
struct controller{
  0,
  &tc0_struct,
  0,
  &ip1_struct,
  &ipidriver,
  0,
  "ipi",
  0,
  "tc",
  0,
  0,
  -1,
  ALV_ALIVE,
  0,
  0,
  "ipiintr",
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}ipi0_struct
```

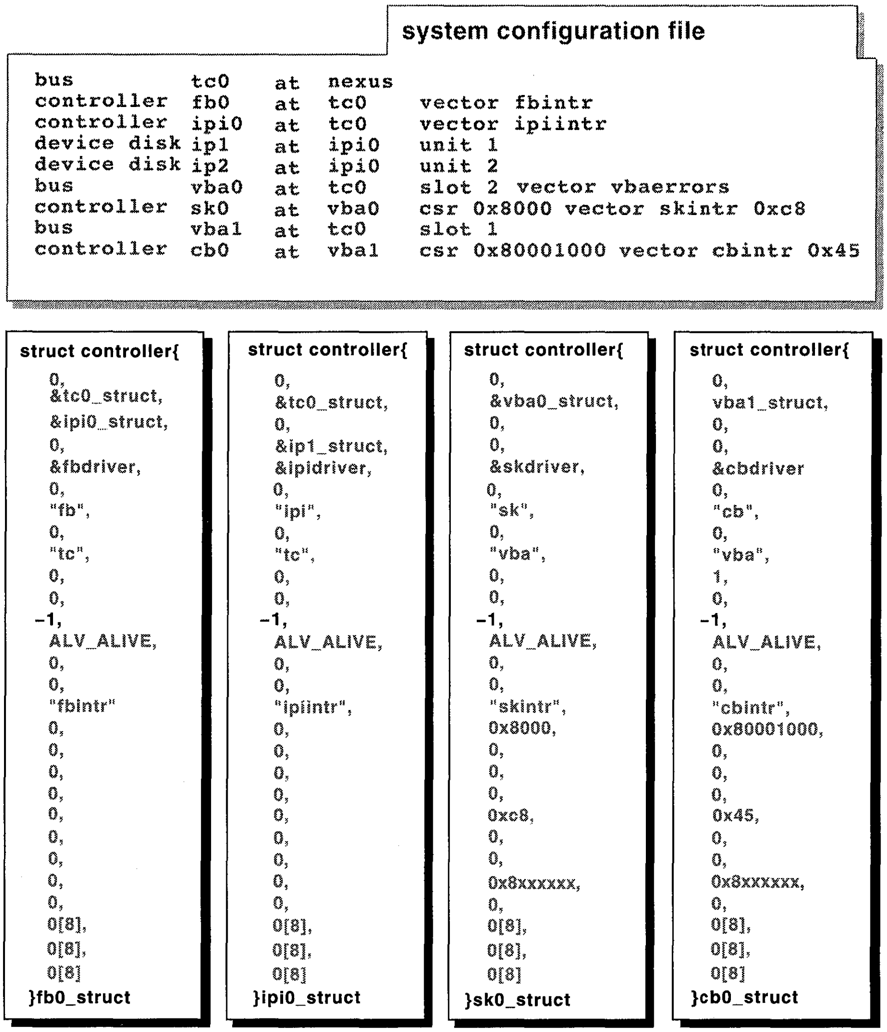
```
struct controller{
  0,
  &vba0_struct,
  0,
  0,
  &skdriver,
  0,
  "sk",
  0,
  "vba",
  0,
  0,
  2,
  ALV_ALIVE,
  0,
  0,
  "skintr",
  0x8000,
  0,
  0,
  0,
  0xc8,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}sk0_struct
```

```
struct controller{
  0,
  vba1_struct,
  0,
  0,
  &cbdriver
  0,
  "cb",
  0,
  "vba",
  1,
  0,
  1,
  ALV_ALIVE,
  0,
  0,
  "cbintr",
  0x80001000,
  0,
  0,
  0,
  0x45,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}cb0_struct
```

7.4.6 The slot Member

The `slot` member specifies the bus slot or node number. Figure 7-17 shows that the example system configuration file does not contain an entry for the slot. Because no slots were specified for the controllers connected to these buses, the kernel sets the `slot` member to the value –1 for each of the `controller` structures.

Figure 7-17: The slot Member Initialized



system configuration file

```
bus          tc0    at  nexus
controller   fb0    at  tc0     vector fbintr
controller   ipi0   at  tc0     vector ipiintr
device disk  ip1    at  ipi0    unit 1
device disk  ip2    at  ipi0    unit 2
bus          vba0   at  tc0     slot 2 vector vbaerrors
controller   sk0    at  vba0    csr 0x8000 vector skintr 0xc8
bus          vba1   at  tc0     slot 1
controller   cb0    at  vba1    csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
  -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr"
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
  -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
  -1,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}sk0_struct
```

```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
  -1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}cb0_struct
```

7.4.7 The alive Member

The `alive` member specifies a flag word to indicate the current status of the controller. Figure 7-18 shows that the autoconfiguration software sets the `alive` member for all of the example `controller` structures to the bit `ALV_ALIVE`. The autoconfiguration software obtains this and the other alive bits in the file `devdriver.h`.

Similarly for loadable drivers, the autoconfiguration software (specifically, the `ldbl_stanza_resolver` interface) sets the `alive` member for any controllers in the sysconfigtab database to the bitwise inclusive OR of the valid alive bits defined in /usr/sys/include/io/common/devdriver.h. The following list shows the valid alive bits that loadable and static drivers can use:

ALV_FREE	The controller is not yet processed.
ALV_ALIVE	The controller is alive and configured.
ALV_PRES	The controller is present but not yet configured.
ALV_NOCNFG	The controller is not to be configured.
ALV_LOADABLE	The controller is present and resolved as loadable.
ALV_NOSIZER	The sizer program should ignore these controller structures. This bit is set for loadable drivers. It indicates that this controller structure is not part of the static configuration.
ALV_RONLY	The device is read-only. This bit is set for static drivers if the corresponding entry is specified in the stanza.static file fragment or system configuration file. This bit is not supported for loadable drivers.
ALV_WONLY	The device is write-only. This bit is set for static drivers if the corresponding entry is specified in the stanza.static file fragment or system configuration file. This bit is not supported for loadable drivers.

Digital does not make use of the functionality associated with the ALV_RONLY and ALV_WONLY bits. However, the config program supports the functionality for third-party driver writers who test for the bits and perform the corresponding logic.

The following examples show situations where third-party driver writers could use the functionality associated with the ALV_RONLY and ALV_WONLY bits:

- You might be implementing a device driver to handle a disk that can be write protected (logically) much the same way as a write-protect button works. Because SCSI disks do not have write-protect buttons, the entry in the system configuration file or stanza.static file fragment could look like this:

  ```
  readonly device disk rz7 at asc0 drive 56
  ```

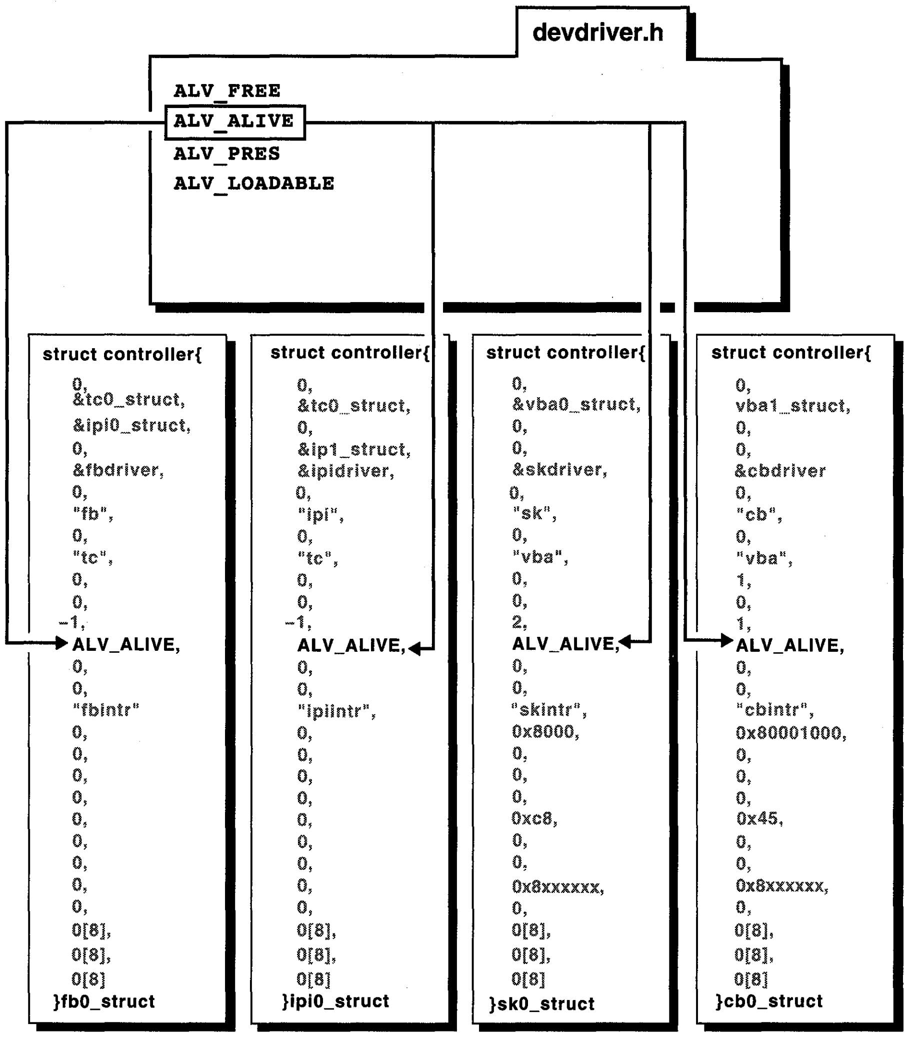
- You might now have a device that crashes the system every time the system touches it while booting multiuser. You suspect a hardware problem with a disk and you are waiting for field service, but you need the system immediately. However, you do not know the bad disk is in the critical path. One solution is to mark the disk as not to be configured in the system configuration file, build a kernel, and boot multiuser until field service fixes the problem. Following is the entry in the system configuration file:

```
not device disk rz6 at asc0 drive 55
```

- The following entry in the system configuration file passes a flag to the driver indicating that a terminal device is for display-only, will not accept input, and is a write-only device:

```
writeonly device dmb0 at vaxbi? node? flags 0xff
vector dmbsint dmbaint dmblint
```

Figure 7-18: The alive Member Initialized


devdriver.h
ALV_FREE
ALV_ALIVE
ALV_PRES
ALV_LOADABLE
struct controller{ 0, &tc0_struct, &ipi0_struct, 0, &fbdriver, 0, "fb", 0, "tc", 0, 0, -1, ALV_ALIVE, 0, 0, "fbintr" 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[8], 0[8], 0[8] }fb0_struct
struct controller{ 0, &tc0_struct, 0, &ip1_struct, &ipidriver, 0, "ipi", 0, "tc", 0, 0, -1, ALV_ALIVE, 0, 0, "ipiintr", 0, 0, 0, 0, 0, 0, 0, 0, 0, 0[8], 0[8], 0[8] }ipi0_struct
struct controller{ 0, &vba0_struct, 0, 0, &skdriver, 0, "sk", 0, "vba", 0, 0, 2, ALV_ALIVE, 0, 0, "skintr", 0x8000, 0, 0, 0, 0xc8, 0, 0, 0x8xxxxxx, 0, 0[8], 0[8], 0[8] }sk0_struct
struct controller{ 0, vba1_struct, 0, 0, &cbdriver 0, "cb", 0, "vba", 1, 0, 1, ALV_ALIVE, 0, 0, "cbintr", 0x80001000, 0, 0, 0, 0x45, 0, 0, 0x8xxxxxx, 0, 0[8], 0[8], 0[8] }cb0_struct

7.4.8 The pname and port Members

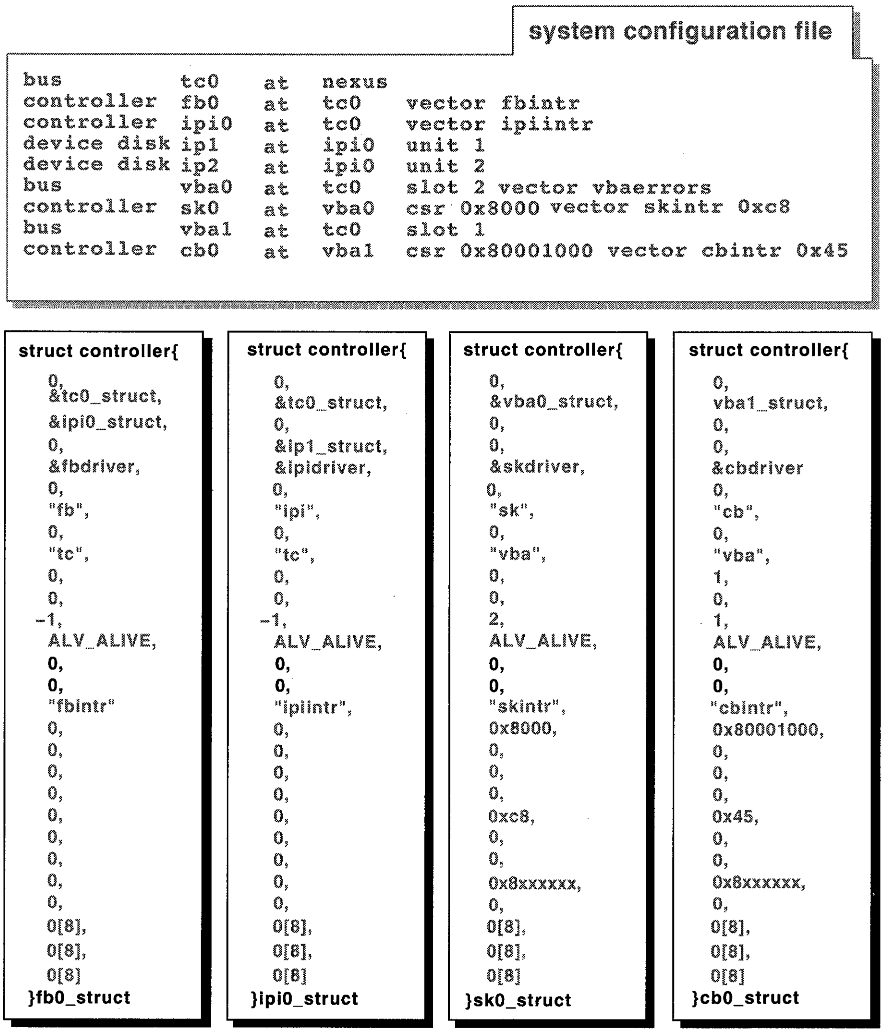
The `pname` member specifies a pointer to the port name for this controller, if applicable. The `port` member specifies a pointer to the `port` structure for this controller, if applicable.

You (or the system manager) specify the port name by using the `port` keyword followed by a string that represents the name of the port. You enter this keyword and string in the `config.file` file fragment or system configuration file for static drivers. This keyword is not currently supported for loadable drivers; therefore, you cannot specify it in the `stanza.loadable` file fragment.

Figure 7-19 shows that the autoconfiguration software sets these members for all of the structures to the value zero (0) to indicate that no port was specified in the system configuration file. If port names were specified, the autoconfiguration software would initialize the `pname` and `port` members for all of the example `controller` structures to the specified values.

Device driver writers do not reference these members in their device drivers. However, bus adapter writers use the `port` structure to implement the initialization for a class of devices or controllers that have common characteristics.

Figure 7-19: The pname and port Members Initialized



```
bus            tc0    at   nexus
controller     fb0    at   tc0    vector fbintr
controller     ipi0   at   tc0    vector ipiintr
device disk    ip1    at   ipi0   unit 1
device disk    ip2    at   ipi0   unit 2
bus            vba0   at   tc0    slot 2 vector vbaerrors
controller     sk0    at   vba0   csr 0x8000 vector skintr 0xc8
bus            vba1   at   tc0    slot 1
controller     cb0    at   vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
   -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr"
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
 }fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
   -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
 }ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
 }sk0_struct
```

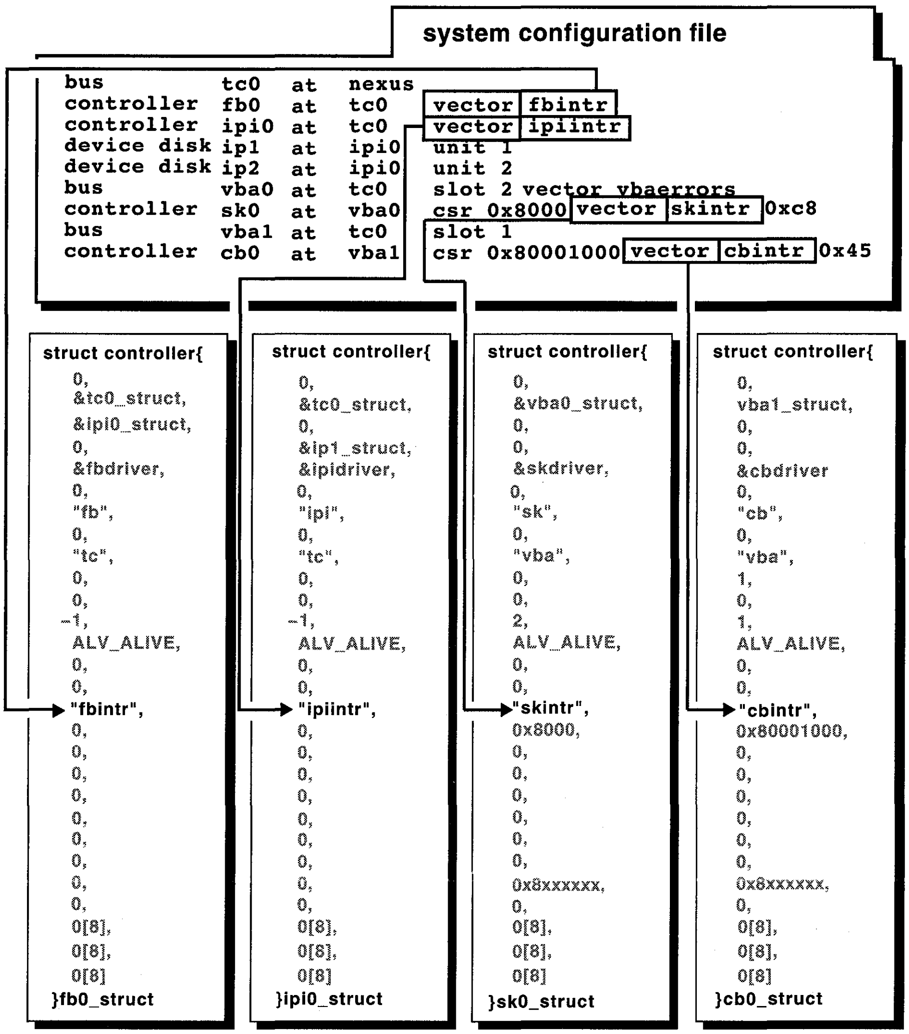
```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
 }cb0_struct
```

7.4.9 The intr Member

The `intr` member specifies an array that contains one or more entry points for the controller interrupt interfaces. Figure 7-20 shows that the autoconfiguration software initializes the `intr` members to `fbintr` for `fb0_struct`, `ipiintr` for `ipi0_struct`, `skintr` for `sk0_struct`, and `cbintr` for `cb0_struct`. The autoconfiguration software obtains these names following the keyword `vector` in the example system configuration file.

For loadable drivers, the `intr` members of these data structures would be set up indirectly through calls to the `handler_add` interface.

Figure 7-20: The intr Member Initialized



```
bus         tc0  at nexus
controller  fb0  at tc0   vector fbintr
controller  ipi0 at tc0   vector ipiintr
device disk ip1  at ipi0  unit 1
device disk ip2  at ipi0  unit 2
bus         vba0 at tc0   slot 2 vector vbaerrors
controller  sk0  at vba0  csr 0x8000 vector skintr 0xc8
bus         vba1 at tc0   slot 1
controller  cb0  at vba1  csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}sk0_struct
```

```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}cb0_struct
```

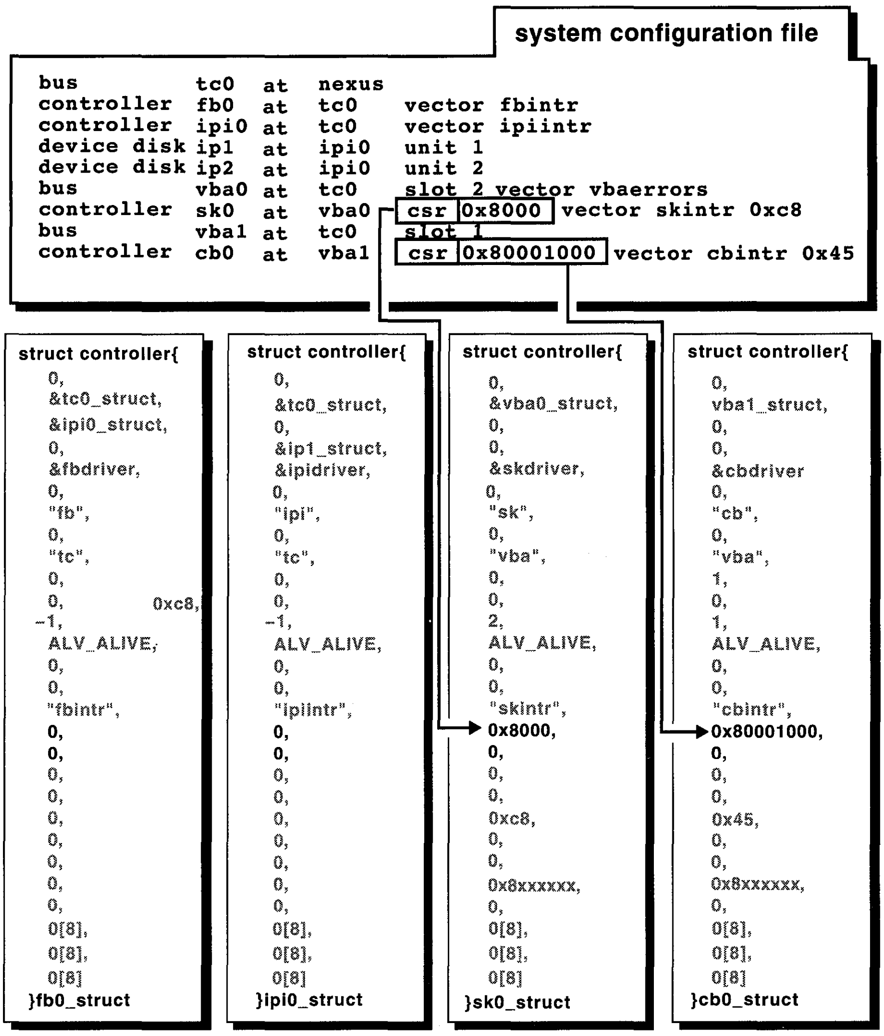
7.4.10 The addr and addr2 Members

The `addr` member specifies the address of the device registers or memory. Figure 7-21 shows that the autoconfiguration software uses the value following the `csr` keyword in the system configuration file to obtain the address. This address is the first control status register (CSR) for the CPU. Thus, the autoconfiguration software initializes the `addr` member to the value zero (0) for `fb0_struct` and `ipi0_struct` because no CSR addresses were specified. It initializes the `addr` members for `sk0_struct` and `cb0_struct` to the values 0x8000 and 0x80001000.

The `addr2` member specifies an optional second virtual address for this controller. This member is set if there are two CSR spaces. Figure 7-21 shows that because no second CSR address was specified, the autoconfiguration software initializes all of the `addr2` members to the value zero (0). The autoconfiguration software would obtain the second CSR address from the `csr2` keyword.

Note that the addresses that appear in these members may not be the values specified in the system configuration file because the kernel could perform an address mapping operation to produce different values.

Figure 7-21: The addr and addr2 Members Initialized



```
bus         tc0  at  nexus
controller  fb0  at  tc0    vector fbintr
controller  ipi0 at  tc0    vector ipiintr
device disk ip1  at  ipi0   unit 1
device disk ip2  at  ipi0   unit 2
bus         vba0 at  tc0    slot 2 vector vbaerrors
controller  sk0  at  vba0   csr 0x8000 vector skintr 0xc8
bus         vba1 at  tc0    slot 1
controller  cb0  at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
  0,
  &tc0_struct,
  &ipi0_struct,
  0,
  &fbdriver,
  0,
  "fb",
  0,
  "tc",
  0,
  0,          0xc8,
 -1,
  ALV_ALIVE,
  0,
  0,
  "fbintr",
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}fb0_struct
```

```
struct controller{
  0,
  &tc0_struct,
  0,
  &ip1_struct,
  &ipidriver,
  0,
  "ipi",
  0,
  "tc",
  0,
  0,
 -1,
  ALV_ALIVE,
  0,
  0,
  "ipiintr",
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}ipi0_struct
```

```
struct controller{
  0,
  &vba0_struct,
  0,
  0,
  &skdriver,
  0,
  "sk",
  0,
  "vba",
  0,
  0,
  2,
  ALV_ALIVE,
  0,
  0,
  "skintr",
  0x8000,
  0,
  0,
  0,
  0xc8,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}sk0_struct
```

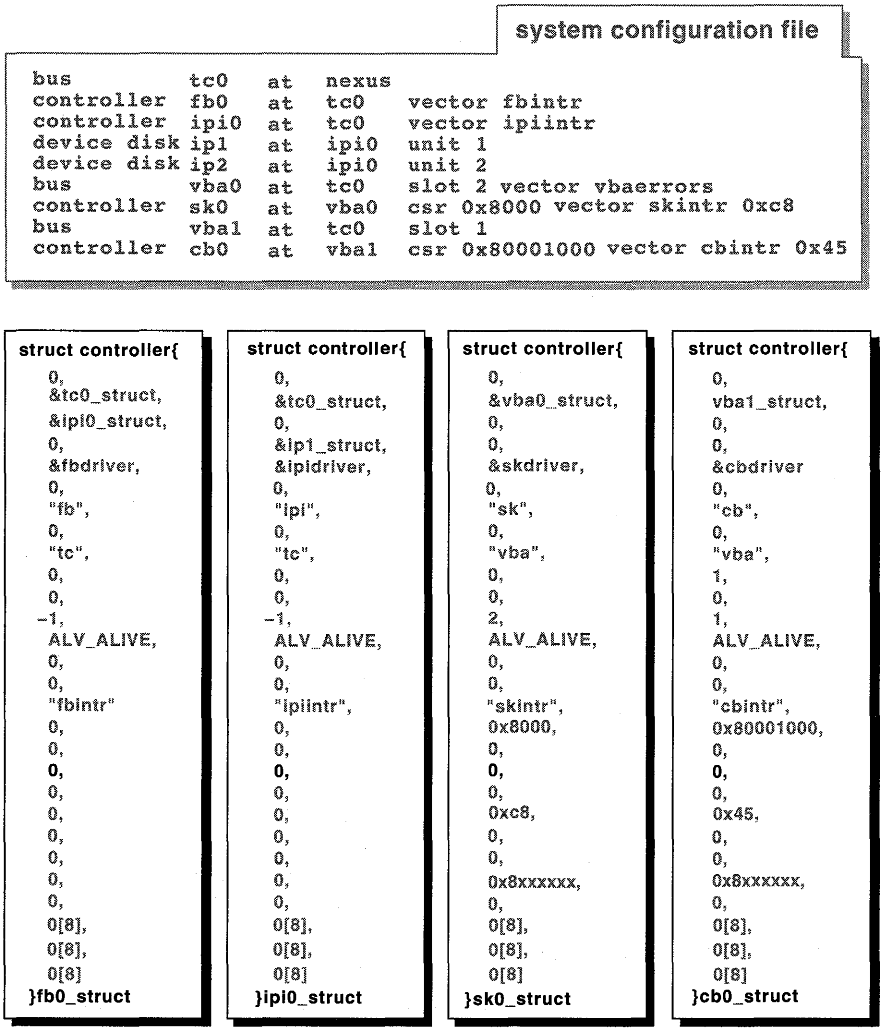
```
struct controller{
  0,
  vba1_struct,
  0,
  0,
  &cbdriver
  0,
  "cb",
  0,
  "vba",
  1,
  0,
  1,
  ALV_ALIVE,
  0,
  0,
  "cbintr",
  0x80001000,
  0,
  0,
  0,
  0x45,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}cb0_struct
```

7.4.11 The flags Member

The `flags` member specifies controller-specific flags. The driver writer (or system manager) can specify controller-specific flags by using the `flags` keyword. Thus, the autoconfiguration software initializes the `flags` member with the value that follows the `flags` keyword in the system configuration file.

Figure 7-22 shows that, because no controller-specific flags were specified in the example system configuration file, the autoconfiguration software initializes the `flags` members for all of the `controller` structures to the value zero (0).

Figure 7-22: The flags Member Initialized



system configuration file

```
bus         tc0   at   nexus
controller  fb0   at   tc0    vector fbintr
controller  ipi0  at   tc0    vector ipiintr
device disk ip1   at   ipi0   unit 1
device disk ip2   at   ipi0   unit 2
bus         vba0  at   tc0    slot 2 vector vbaerrors
controller  sk0   at   vba0   csr 0x8000 vector skintr 0xc8
bus         vba1  at   tc0    slot 1
controller  cb0   at   vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
  0,
  &tc0_struct,
  &ipi0_struct,
  0,
  &fbdriver,
  0,
  "fb",
  0,
  "tc",
  0,
  0,
  -1,
  ALV_ALIVE,
  0,
  0,
  "fbintr"
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}fb0_struct
```

```
struct controller{
  0,
  &tc0_struct,
  0,
  &ip1_struct,
  &ipidriver,
  0,
  "ipi",
  0,
  "tc",
  0,
  0,
  -1,
  ALV_ALIVE,
  0,
  0,
  "ipiintr",
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0,
  0[8],
  0[8],
  0[8]
}ipi0_struct
```

```
struct controller{
  0,
  &vba0_struct,
  0,
  0,
  &skdriver,
  0,
  "sk",
  0,
  "vba",
  0,
  0,
  2,
  ALV_ALIVE,
  0,
  0,
  "skintr",
  0x8000,
  0,
  0,
  0,
  0xc8,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}sk0_struct
```

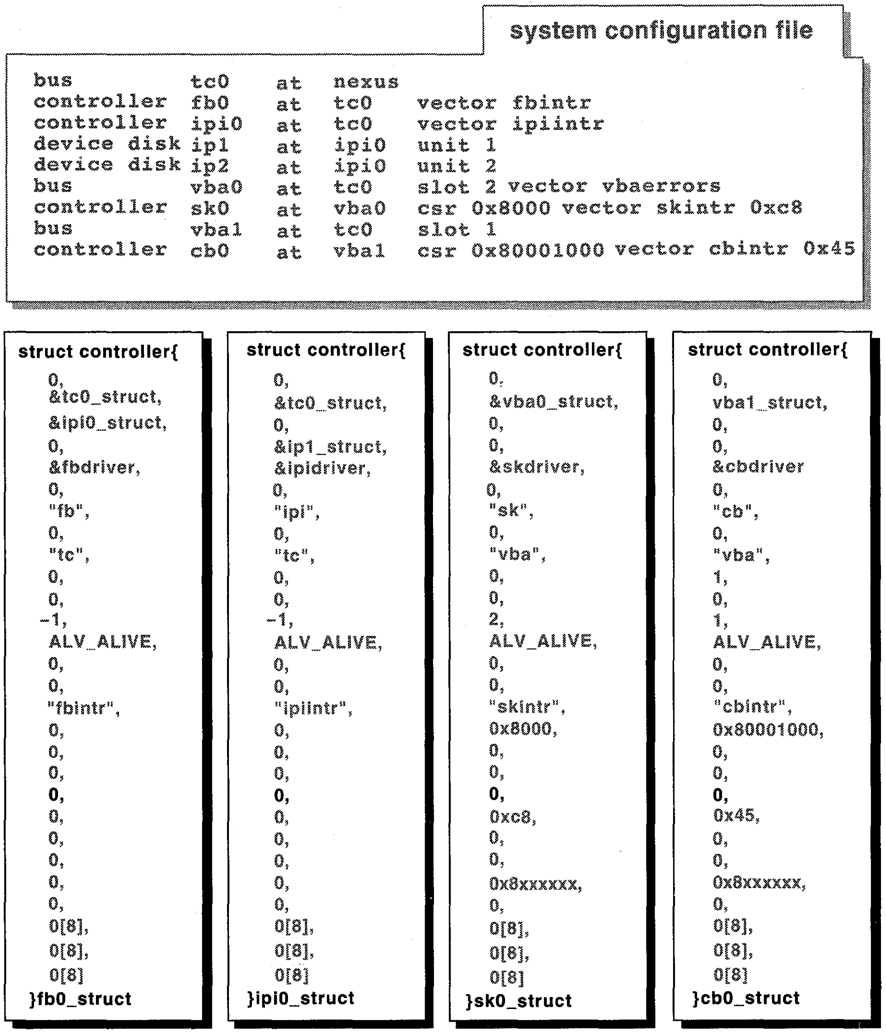
```
struct controller{
  0,
  vba1_struct,
  0,
  0,
  &cbdriver
  0,
  "cb",
  0,
  "vba",
  1,
  0,
  1,
  ALV_ALIVE,
  0,
  0,
  "cbintr",
  0x80001000,
  0,
  0,
  0,
  0x45,
  0,
  0,
  0x8xxxxxx,
  0,
  0[8],
  0[8],
  0[8]
}cb0_struct
```

7.4.12 The bus_priority Member

The `bus_priority` member specifies the configured VMEbus priority level of the device. Only drivers operating on the VMEbus use this member. You (or the system manager) can specify the bus priority by using the `priority` keyword. Thus, the autoconfiguration software initializes the `bus_priority` member with the value that follows the `priority` keyword in the system configuration file.

Figure 7-23 shows that because no bus priorities were specified in the example system configuration file, the autoconfiguration software initializes the `bus_priority` members for all of the `controller` structures to the value zero (0).

Figure 7-23: The bus_priority Member Initialized



```
bus            tc0    at   nexus
controller     fb0    at   tc0     vector fbintr
controller     ipi0   at   tc0     vector ipiintr
device disk    ip1    at   ipi0    unit 1
device disk    ip2    at   ipi0    unit 2
bus            vba0   at   tc0     slot 2 vector vbaerrors
controller     sk0    at   vba0    csr 0x8000 vector skintr 0xc8
bus            vba1   at   tc0     slot 1
controller     cb0    at   vba1    csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}sk0_struct
```

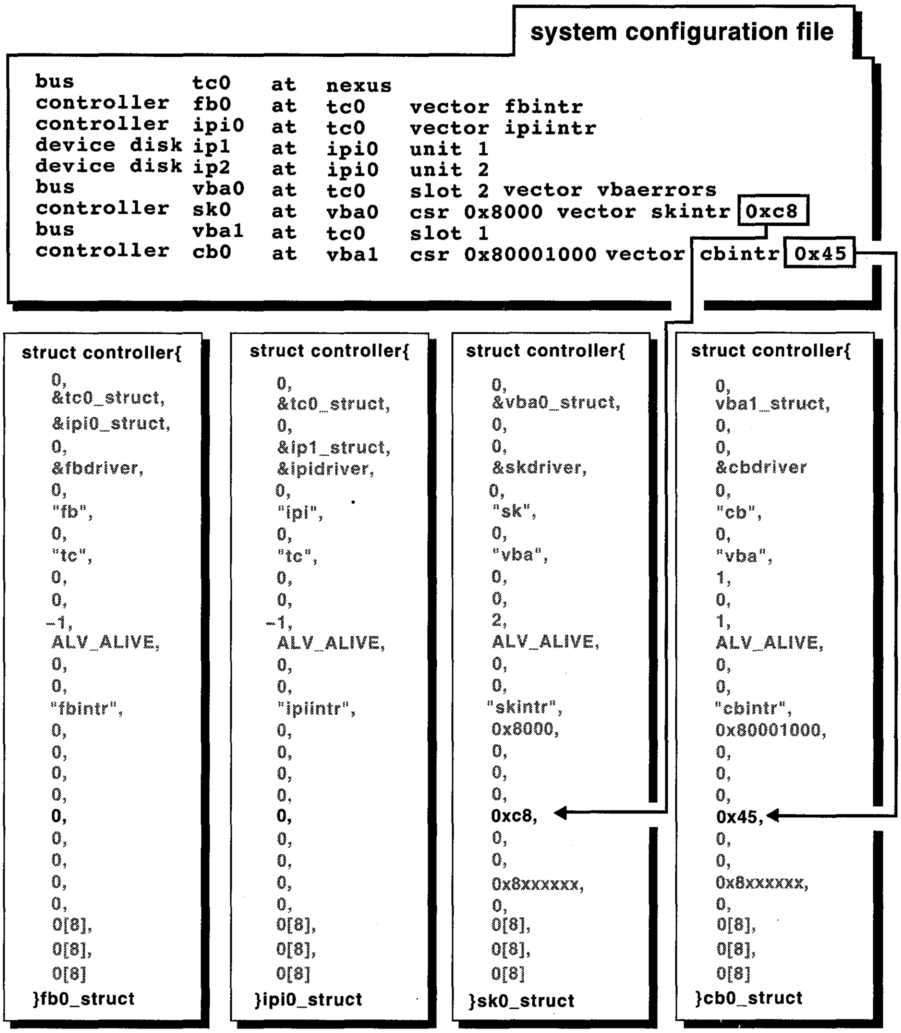
```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}cb0_struct
```

7.4.13 The ivnum Member

The `ivnum` member specifies an interrupt vector number. Only drivers operating on the VMEbus use this member. The device driver writer can specify an interrupt vector number after the interrupt interface in the system configuration file. Thus, the autoconfiguration software initializes the `ivnum` member with the value that follows the interrupt interface in the system configuration file.

Figure 7-24 shows the values in the example system configuration file that the autoconfiguration software parses to obtain the interrupt vector numbers. It also shows that the autoconfiguration software sets the `ivnum` members to these values for `sk0_struct` and `cb0_struct`. Because no interrupt vector numbers were specified in the example system configuration file for `fb0_struct` and `ipi0_struct`, the autoconfiguration software initializes their `ivnum` members to the value zero (0).

Figure 7-24: The ivnum Member Initialized



```
bus          tc0    at   nexus
controller   fb0    at   tc0     vector fbintr
controller   ipi0   at   tc0     vector ipiintr
device disk  ip1    at   ipi0    unit 1
device disk  ip2    at   ipi0    unit 2
bus          vba0   at   tc0     slot 2 vector vbaerrors
controller   sk0    at   vba0    csr 0x8000 vector skintr 0xc8
bus          vba1   at   tc0     slot 1
controller   cb0    at   vba1    csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
   0,
   &tc0_struct,
   &ipi0_struct,
   0,
   &fbdriver,
   0,
   "fb",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "fbintr",
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}fb0_struct
```

```
struct controller{
   0,
   &tc0_struct,
   0,
   &ip1_struct,
   &ipidriver,
   0,
   "ipi",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "ipiintr",
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}ipi0_struct
```

```
struct controller{
   0,
   &vba0_struct,
   0,
   0,
   &skdriver,
   0,
   "sk",
   0,
   "vba",
   0,
   0,
   2,
   ALV_ALIVE,
   0,
   0,
   "skintr",
   0x8000,
   0,
   0,
   0,
   0xc8,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}sk0_struct
```

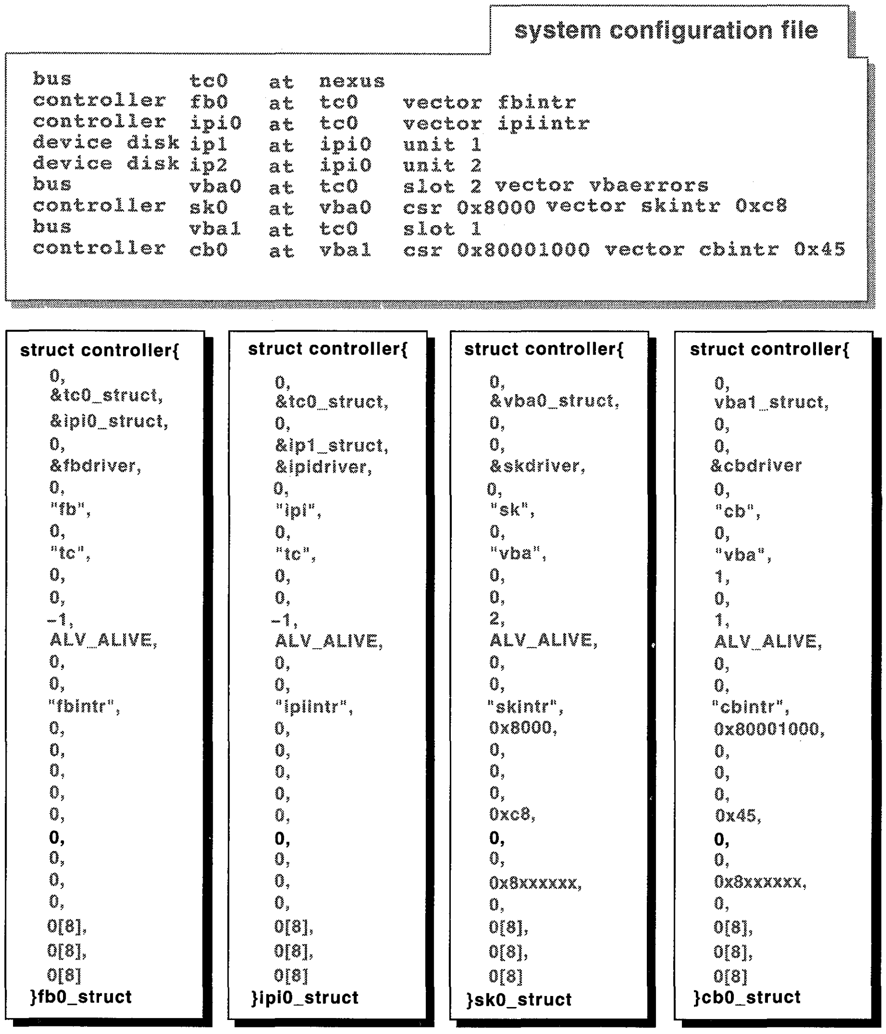
```
struct controller{
   0,
   vba1_struct,
   0,
   0,
   &cbdriver
   0,
   "cb",
   0,
   "vba",
   1,
   0,
   1,
   ALV_ALIVE,
   0,
   0,
   "cbintr",
   0x80001000,
   0,
   0,
   0,
   0x45,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}cb0_struct
```

7.4.14 The priority Member

The `priority` member specifies the system priority level (spl) to block interrupts from this device. Only drivers operating on the VMEbus use this member. Thus, the autoconfiguration software initializes the `priority` member with an appropriate value based on the value stored in the `bus_priority` member (if specified) and the system implementation.

You use this value as an argument to the `splx` interface to block interrupts for the device. Thus, the autoconfiguration software initializes the `priority` member with an appropriate value based on the value stored in the `bus_priority` member (if specified) and the system implementation. The driver writer uses this value as an argument to the `splx` interface to block interrupts for the device. Figure 7-25 shows that the autoconfiguration software sets the members for all of the `controller` structures to the value zero (0).

Figure 7-25: The priority Member Initialized



```
bus          tc0   at  nexus
controller   fb0   at  tc0    vector fbintr
controller   ipi0  at  tc0    vector ipiintr
device disk  ip1   at  ipi0   unit 1
device disk  ip2   at  ipi0   unit 2
bus          vba0  at  tc0    slot 2 vector vbaerrors
controller   sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus          vba1  at  tc0    slot 1
controller   cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
   0,
   &tc0_struct,
   &ipi0_struct,
   0,
   &fbdriver,
   0,
   "fb",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "fbintr",
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}fb0_struct
```

```
struct controller{
   0,
   &tc0_struct,
   0,
   &ip1_struct,
   &ipidriver,
   0,
   "ipi",
   0,
   "tc",
   0,
   0,
   -1,
   ALV_ALIVE,
   0,
   0,
   "ipiintr",
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0,
   0[8],
   0[8],
   0[8]
}ipi0_struct
```

```
struct controller{
   0,
   &vba0_struct,
   0,
   0,
   &skdriver,
   0,
   "sk",
   0,
   "vba",
   0,
   0,
   2,
   ALV_ALIVE,
   0,
   0,
   "skintr",
   0x8000,
   0,
   0,
   0,
   0xc8,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}sk0_struct
```

```
struct controller{
   0,
   vba1_struct,
   0,
   0,
   &cbdriver
   0,
   "cb",
   0,
   "vba",
   1,
   0,
   1,
   ALV_ALIVE,
   0,
   0,
   "cbintr",
   0x80001000,
   0,
   0,
   0,
   0x45,
   0,
   0,
   0x8xxxxxx,
   0,
   0[8],
   0[8],
   0[8]
}cb0_struct
```

7.4.15 The cmd Member

The `cmd` member specifies a field that is not currently used.

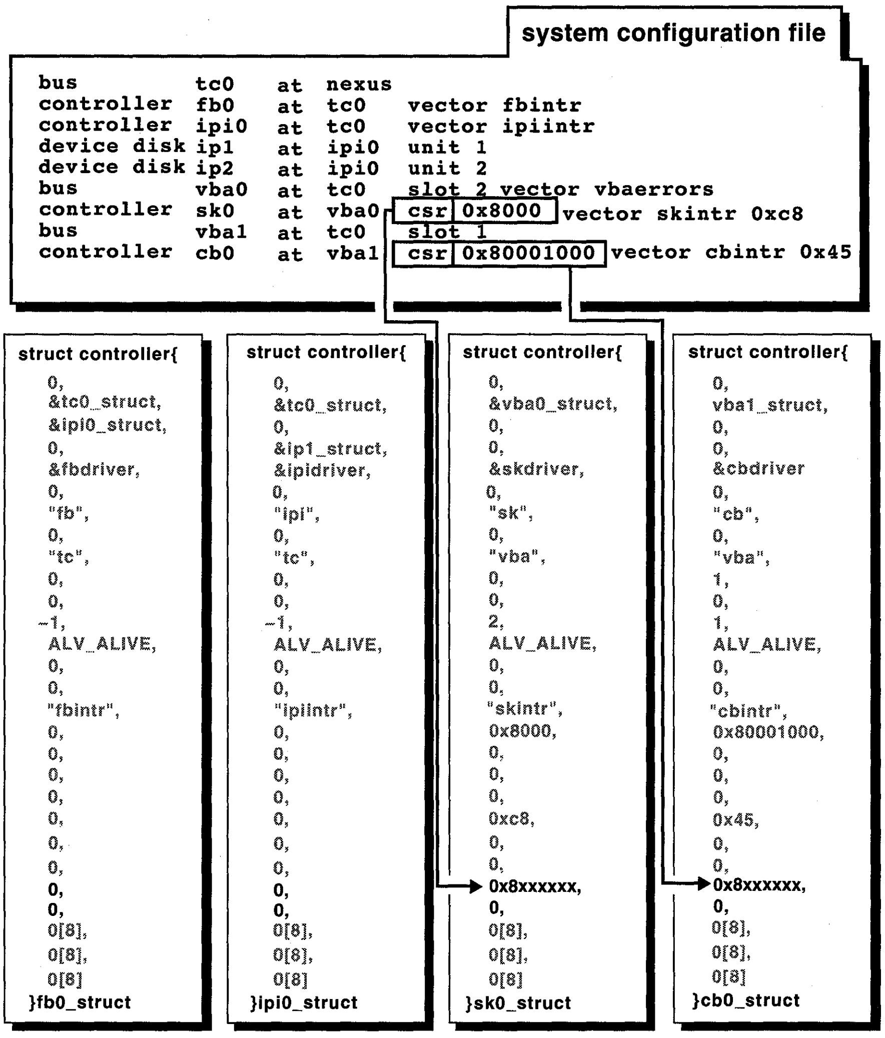
7.4.16 The physaddr and physaddr2 Members

The `physaddr` member specifies the physical address that corresponds to the virtual address set in the `addr` member. Because no CSR addresses were specified for `fb0_struct` and `ipi0_struct`, Figure 7-26 shows that the autoconfiguration software initializes their `physaddr` members to the value zero (0). The figure also shows that the autoconfiguration software uses the CSR addresses specified after the `csr` keywords to calculate corresponding physical addresses and to use them to initialize the `physaddr` members for `sk0_struct` and `cb0_struct`.

The `physaddr2` member specifies the physical address that corresponds to the virtual address set in the `addr2` member. The autoconfiguration software would use the CSR addresses specified after the `addr2` keyword to calculate a corresponding second physical address. Because no second CSR addresses were specified in the example system configuration file, Figure 7-26 shows that the autoconfiguration software initializes the `physaddr2` members for all of the `controller` structures to the value zero (0).

Note that the addresses that appear in these members may not be the values specified in the system configuration file because the kernel could perform an address mapping operation to produce a different value.

Figure 7-26: The physaddr and physaddr2 Members Initialized



```
bus          tc0   at  nexus
controller   fb0   at  tc0    vector fbintr
controller   ipi0  at  tc0    vector ipiintr
device disk  ip1   at  ipi0   unit 1
device disk  ip2   at  ipi0   unit 2
bus          vba0  at  tc0    slot 2 vector vbaerrors
controller   sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus          vba1  at  tc0    slot 1
controller   cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}sk0_struct
```

```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}cb0_struct
```

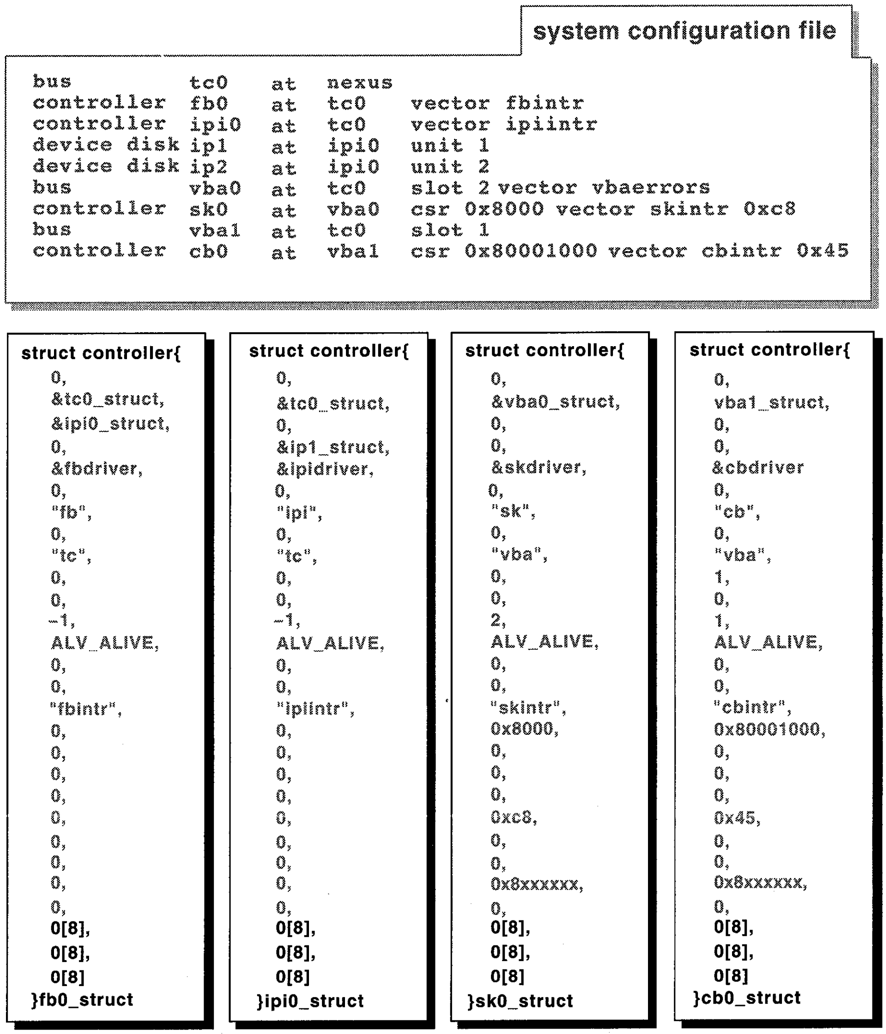
7.4.17 The private, conn_priv, and rsvd Members

The `private` member specifies private storage for use by this controller or controller type. For the example system configuration file, Figure 7-27 shows that the autoconfiguration software initializes these members for all of the structures to the value zero (0). The code controlling the controller can use these members for any storage purpose.

The conn_priv member specifies private storage for use by the bus that this controller is connected to. Figure 7-27 shows that the autoconfiguration software initializes these members for all of the structures to the value zero (0).

The rsvd member is reserved for future expansion of the data structure.

Figure 7-27: The private, conn_priv, and rsvd Members Initialized



```
bus         tc0   at  nexus
controller  fb0   at  tc0   vector fbintr
controller  ipi0  at  tc0   vector ipiintr
device disk ip1   at  ipi0  unit 1
device disk ip2   at  ipi0  unit 2
bus         vba0  at  tc0   slot 2 vector vbaerrors
controller  sk0   at  vba0  csr 0x8000 vector skintr 0xc8
bus         vba1  at  tc0   slot 1
controller  cb0   at  vba1  csr 0x80001000 vector cbintr 0x45
```

```
struct controller{
    0,
    &tc0_struct,
    &ipi0_struct,
    0,
    &fbdriver,
    0,
    "fb",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "fbintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}fb0_struct
```

```
struct controller{
    0,
    &tc0_struct,
    0,
    &ip1_struct,
    &ipidriver,
    0,
    "ipi",
    0,
    "tc",
    0,
    0,
    -1,
    ALV_ALIVE,
    0,
    0,
    "ipiintr",
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0[8],
    0[8],
    0[8]
}ipi0_struct
```

```
struct controller{
    0,
    &vba0_struct,
    0,
    0,
    &skdriver,
    0,
    "sk",
    0,
    "vba",
    0,
    0,
    2,
    ALV_ALIVE,
    0,
    0,
    "skintr",
    0x8000,
    0,
    0,
    0,
    0xc8,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}sk0_struct
```

```
struct controller{
    0,
    vba1_struct,
    0,
    0,
    &cbdriver
    0,
    "cb",
    0,
    "vba",
    1,
    0,
    1,
    ALV_ALIVE,
    0,
    0,
    "cbintr",
    0x80001000,
    0,
    0,
    0,
    0x45,
    0,
    0,
    0x8xxxxxx,
    0,
    0[8],
    0[8],
    0[8]
}cb0_struct
```

7.5 The device Structure

The `device` structure represents an instance of a device entity. A device is an entity that connects to and is controlled by a controller. A device does not connect directly to a bus. You (or the system manager) specify a device entity in the system configuration file as follows:

- For static drivers

  Specify a valid syntax for the device in the `config.file` file fragment or the system configuration file
- For loadable drivers

  Specify a valid syntax for the device in the `stanza.loadable` file fragment.

Chapter 11 discusses the device driver configuration models. Chapter 12 describes the valid syntaxes for a device specification. Chapter 13 provides examples of how to configure device drivers.

Table 7-3 lists the members of the `device` structure along with their associated data types.

Table 7-3: Members of the device Structure

Member Name	Data Type
`nxt_dev`	`struct device *`
`ctlr_hd`	`struct controller *`
`dev_type`	`char *`
`dev_name`	`char *`
`logunit`	`int`
`unit`	`int`
`ctlr_name`	`char *`
`ctlr_num`	`int`
`alive`	`int`
`private`	`void * [8]`
`conn_priv`	`void * [8]`
`rsvd`	`void * [8]`

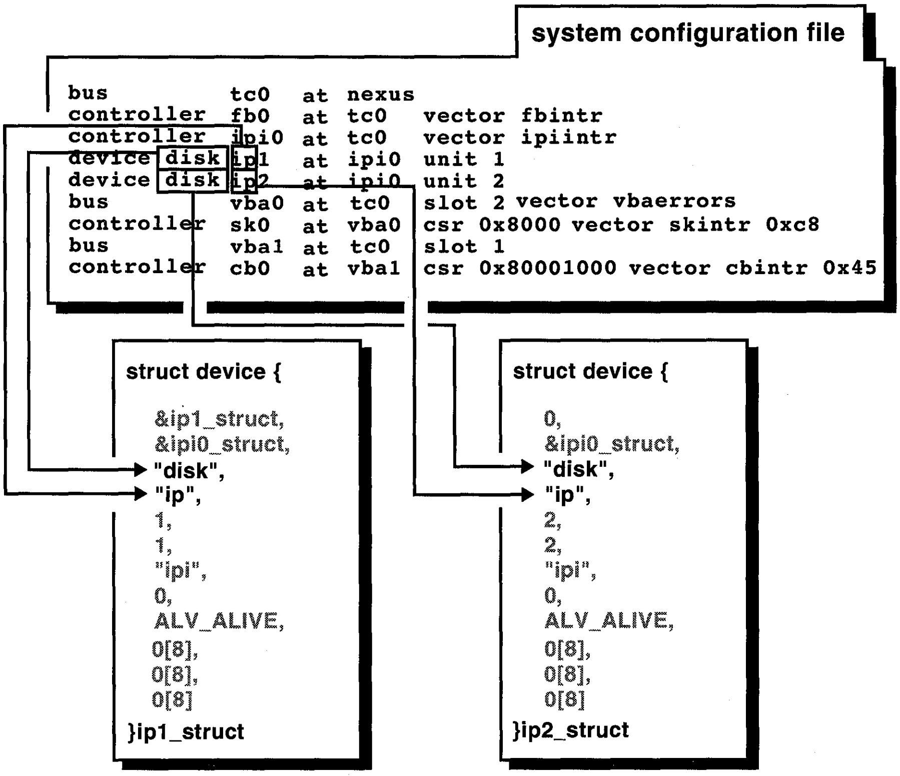
The following sections discuss all of these members except for `nxt_dev`, and `ctlr_hd`, which are presented in Section 7.2.1.6. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of this data structure.

7.5.1 The dev_type and dev_name Members

The dev_type member specifies the device type (for example, disk and tape). The dev_name member specifies the device name type. You (or the system manager) specify the device type by using any string and the device name by using any string up to a maximum of eight characters. For example, the string disk can indicate that the device is a disk and the string ip can indicate that this is an ip disk. For loadable drivers, the only supported strings are disk and tape.

Figure 7-28 shows the lines in the system configuration file that the autoconfiguration software parses to obtain the device type and device name for the specified devices. It sets the dev_type members of ip1_struct and ip2_struct to the value disk. The autoconfiguration software also sets the dev_name members to the string ip.

Figure 7-28: The dev_type and dev_name Members Initialized



```
bus          tc0   at  nexus
controller   fb0   at  tc0   vector fbintr
controller   ipi0  at  tc0   vector ipiintr
device disk  ip1   at  ipi0  unit 1
device disk  ip2   at  ipi0  unit 2
bus          vba0  at  tc0   slot 2 vector vbaerrors
controller   sk0   at  vba0  csr 0x8000 vector skintr 0xc8
bus          vba1  at  tc0   slot 1
controller   cb0   at  vba1  csr 0x80001000 vector cbintr 0x45
```

```
struct device {
    &ip1_struct,
    &ipi0_struct,
    "disk",
    "ip",
    1,
    1,
    "ipi",
    0,
    ALV_ALIVE,
    0[8],
    0[8],
    0[8]
}ip1_struct
```

```
struct device {
    0,
    &ipi0_struct,
    "disk",
    "ip",
    2,
    2,
    "ipi",
    0,
    ALV_ALIVE,
    0[8],
    0[8],
    0[8]
}ip2_struct
```

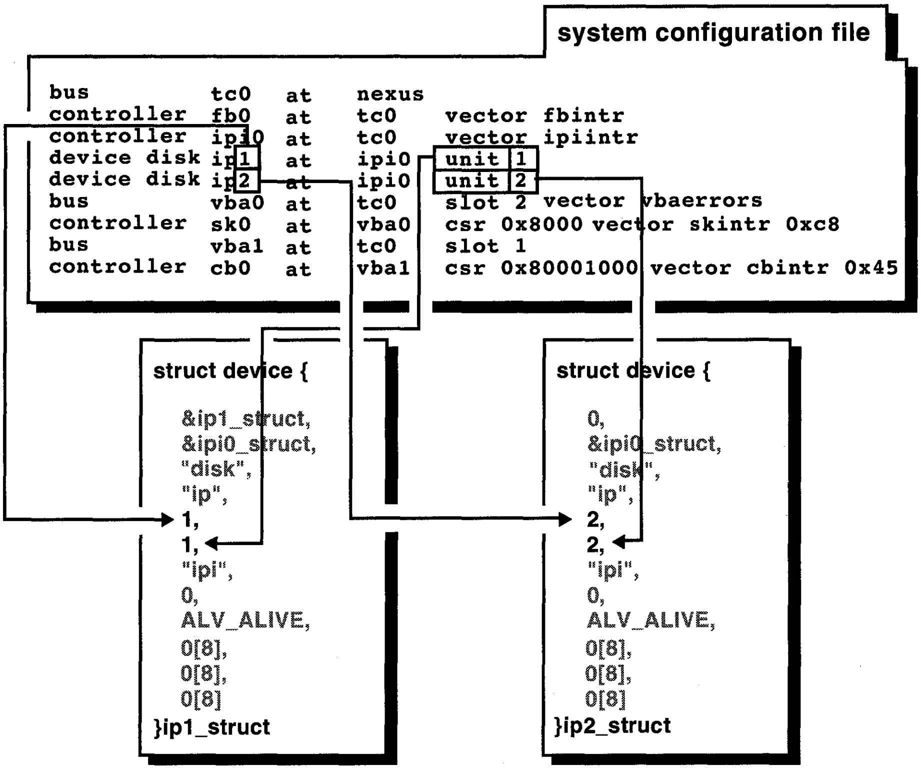
7.5.2 The logunit and unit Members

The `logunit` member specifies the device logical unit number. The `unit` member specifies the device physical unit number. You (or the system manager) specify the logical unit number by using a valid number after the device name string and the physical unit number by using a valid number after the keyword `unit`.

Figure 7-29 shows the lines in the system configuration file that the autoconfiguration software parses to obtain the logical unit number and the physical unit number for the specified devices. It sets the `logunit` members of `ip1_struct` and `ip2_struct` to the values 1 and 2.

The figure also shows that the autoconfiguration software sets the `unit` members for these structures to 1 and 2.

Figure 7-29: The logunit and unit Members Initialized


system configuration file
bus tc0 at nexus
controller fb0 at tc0 vector fbintr
controller ipi0 at tc0 vector ipiintr
device disk ip1 at ipi0 unit 1
device disk ip2 at ipi0 unit 2
bus vba0 at tc0 slot 2 vector vbaerrors
controller sk0 at vba0 csr 0x8000 vector skintr 0xc8
bus vba1 at tc0 slot 1
controller cb0 at vba1 csr 0x80001000 vector cbintr 0x45
struct device {
&ip1_struct,
&ipi0_struct,
"disk",
"ip",
1,
1,
"ipi",
0,
ALV_ALIVE,
0[8],
0[8],
0[8]
}ip1_struct
struct device {
0,
&ipi0_struct,
"disk",
"ip",
2,
2,
"ipi",
0,
ALV_ALIVE,
0[8],
0[8],
0[8]
}ip2_struct

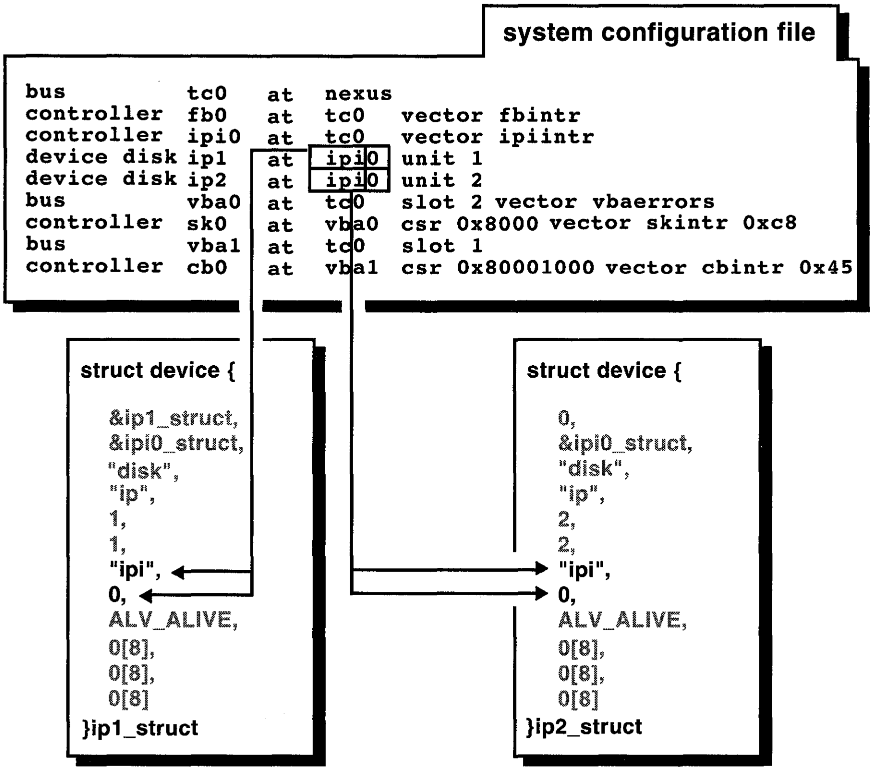
7.5.3 The ctlr_name and ctlr_num Members

The `ctlr_name` member specifies the name of the controller that this device is connected to. The `ctlr_num` member specifies the number of the controller that this device is connected to. You (or the system manager) specify the controller name and the controller number by using the keyword `at` followed by a character string and number that represent the controller name and controller number. For example, the character string `ipi` represents a controller supported by Digital. You enter these values in the `config.file` file fragment or the system configuration file for static drivers and the `stanza.loadable` file fragment for loadable drivers.

Figure 7-30 shows the values in the system configuration file that the autoconfiguration software parses to obtain the controller name and the controller number for the specified devices. It sets the `ctlr_name` members of `ip1_struct` and `ip2_struct` to the string `ipi` because both devices are connected to the same controller.

The figure also shows that the autoconfiguration software sets the `ctlr_num` members for these structures to the value zero (0).

Figure 7-30: The ctlr_name and ctlr_num Members Initialized

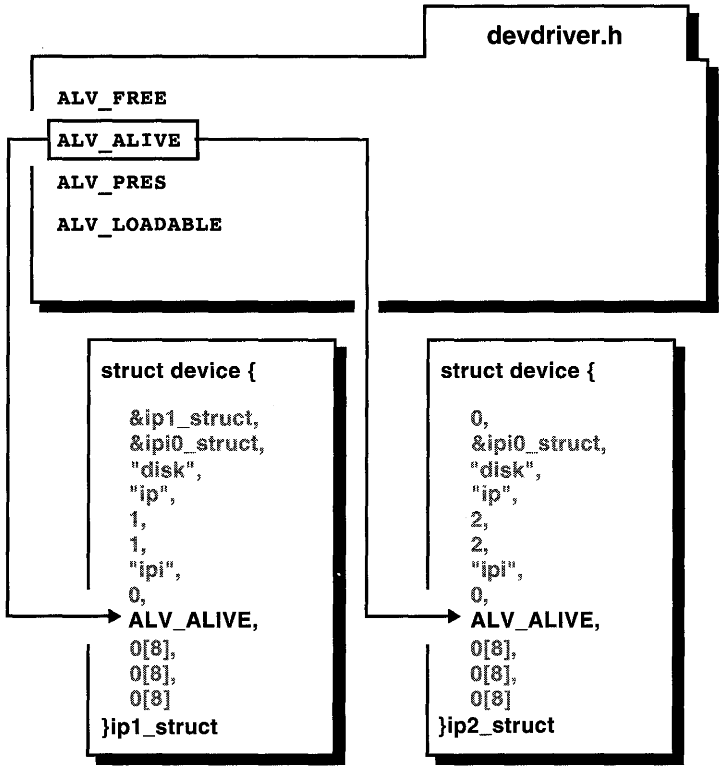


```
bus          tc0   at  nexus
controller   fb0   at  tc0   vector fbintr
controller   ipi0  at  tc0   vector ipiintr
device disk  ip1   at  ipi0  unit 1
device disk  ip2   at  ipi0  unit 2
bus          vba0  at  tc0   slot 2 vector vbaerrors
controller   sk0   at  vba0  csr 0x8000 vector skintr 0xc8
bus          vba1  at  tc0   slot 1
controller   cb0   at  vba1  csr 0x80001000 vector cbintr 0x45
```

7.5.4 The alive Member

The `alive` member specifies a flag word to indicate the current status of the device. Figure 7-31 shows that the autoconfiguration software sets the `alive` member for both of the example `device` structures to the bit `ALV_ALIVE`. The autoconfiguration software obtains this and the other alive bits in the file `/usr/sys/include/io/common/devdriver.h`.

Figure 7-31: The alive Member Initialized

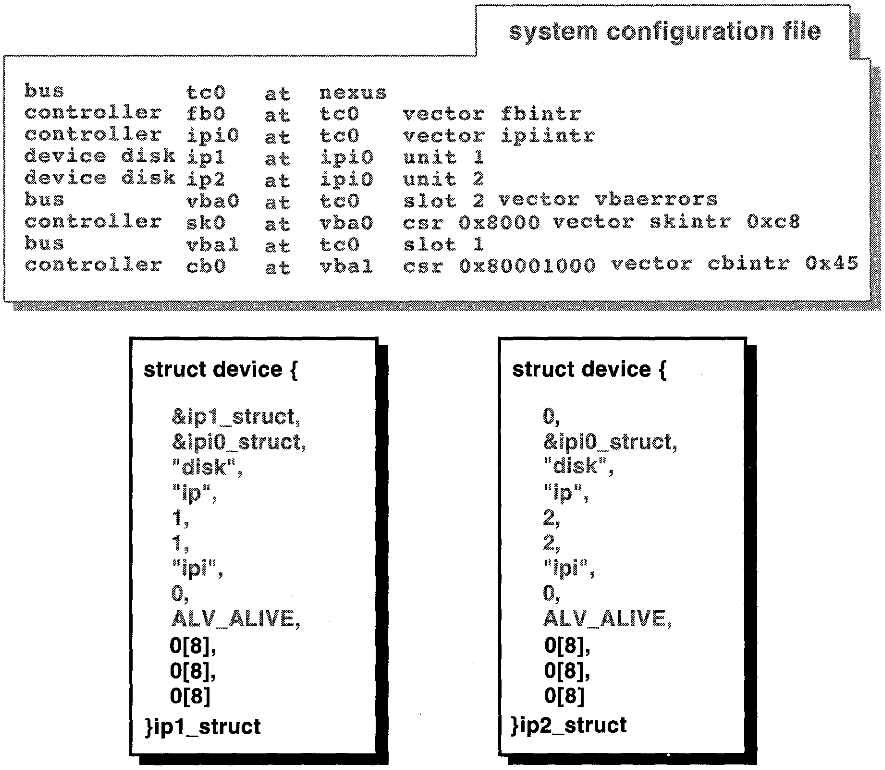


7.5.5 The private, conn_priv, and rsvd Members

The `private` member specifies private storage for use by this device or device class. The `conn_priv` member specifies private storage for use by the controller that this device is connected to.

The `rsvd` member is reserved for future expansion of the data structure. For the example system configuration file, Figure 7-32 shows that the autoconfiguration software initializes these members for all of the structures to the value zero (0). The code controlling the device can use these members for any storage purposes.

Figure 7-32: The private, conn_priv, rsvd Members Initialized



```
bus         tc0   at  nexus
controller  fb0   at  tc0    vector fbintr
controller  ipi0  at  tc0    vector ipiintr
device disk ip1   at  ipi0   unit 1
device disk ip2   at  ipi0   unit 2
bus         vba0  at  tc0    slot 2 vector vbaerrors
controller  sk0   at  vba0   csr 0x8000 vector skintr 0xc8
bus         vba1  at  tc0    slot 1
controller  cb0   at  vba1   csr 0x80001000 vector cbintr 0x45
```

```
struct device {

   &ip1_struct,
   &ipi0_struct,
   "disk",
   "ip",
   1,
   1,
   "ipi",
   0,
   ALV_ALIVE,
   0[8],
   0[8],
   0[8]
}ip1_struct
```

```
struct device {

   0,
   &ipi0_struct,
   "disk",
   "ip",
   2,
   2,
   "ipi",
   0,
   ALV_ALIVE,
   0[8],
   0[8],
   0[8]
}ip2_struct
```

7.6 The driver Structure

The `driver` structure defines driver entry points and other driver-specific information. You declare and initialize an instance of this structure in the device driver. The bus configuration code uses the entry points defined in this structure during system configuration. The bus configuration code fills in the `dev_list` and `ctlr_list` arrays. These arrays (members of the `device` and `controller` structures) are used by the driver interfaces to get the structures for specific devices or controllers.

Table 7-4 lists the members of the `driver` structure along with their associated data types.

Table 7-4: Members of the driver Structure

Member Name	Data Type
`probe`	`int (*probe)()`
`slave`	`int (*slave)()`
`cattach`	`int (*cattach)()`
`dattach`	`int (*dattach)()`
`go`	`int (*go)()`
`addr_list`	`caddr_t *`
`dev_name`	`char *`
`dev_list`	`struct device **`
`ctlr_name`	`char *`
`ctlr_list`	`struct controller **`
`xclu`	`short`
`addr1_size`	`int`
`addr1_atype`	`int`
`addr2_size`	`int`
`addr2_atype`	`int`
`ctlr_unattach`	`int (*ctlr_unattach)()`
`dev_unattach`	`int (*dev_unattach)()`

The following sections discuss all of these members. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of this data structure.

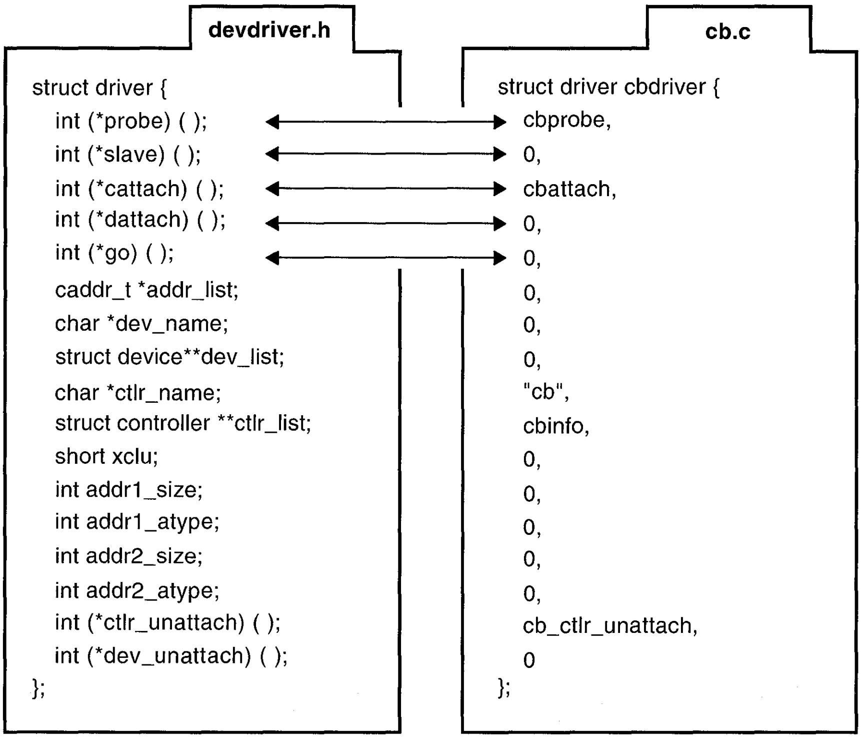
7.6.1 The probe, slave, cattach, dattach, and go Members

The `probe` member specifies a pointer to the driver's `probe` interface, which is called to verify that the controller exists. The `slave` member specifies a pointer to the driver's `slave` interface, which is called once for each device connected to the controller.

The `cattach` member specifies a pointer to the driver's `cattach` interface, which is called to allow controller-specific initialization. This pointer can be set to NULL. The `dattach` member specifies a pointer to the driver's `dattach` interface, which is called once for each `slave` call that returns success. The `dattach` interface is used for device-specific initialization. This pointer can be set to NULL.

The go member specifies a pointer to the driver's go interface, which is not currently used. Figure 7-33 shows that the driver writer initializes these members to the values contained in the cbdriver structure. This data structure appears in cb.c, the source file for the /dev/cb device driver.

Figure 7-33: The probe, slave, cattach, dattach, and go Members Initialized



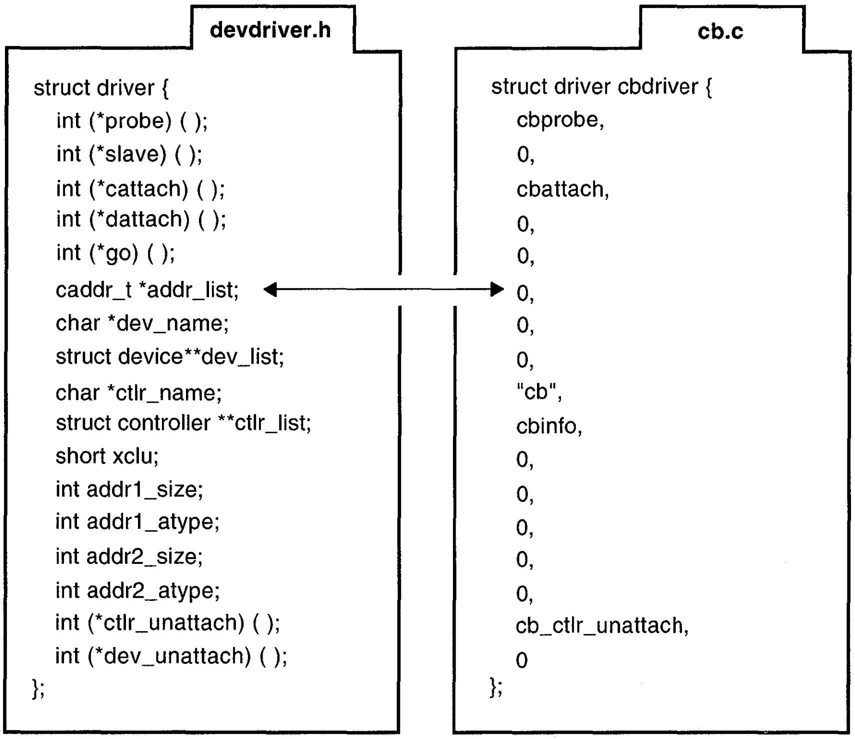
```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

7.6.2 The addr_list Member

The addr_list member specifies a list of optional CSR addresses. Figure 7-34 shows that the driver writer initializes this member to the value zero (0) because this entry is not used for TURBOchannel device drivers.

Figure 7-34: The addr_list Member Initialized



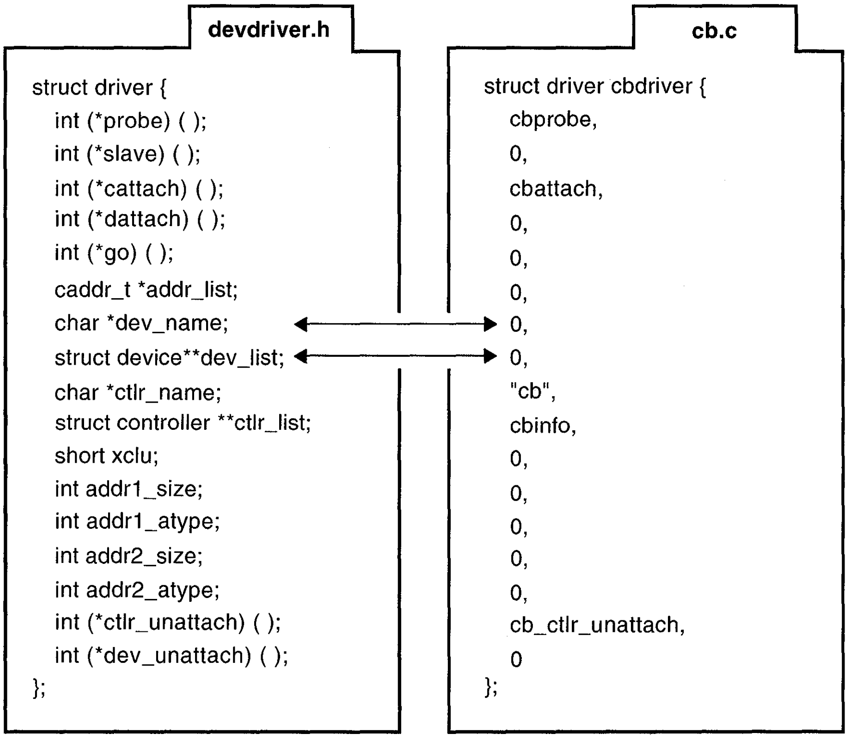
```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

7.6.3 The dev_name and dev_list Members

The `dev_name` member specifies the name of the device connected to this controller. The `dev_list` member specifies an array of pointers to device structures currently connected to this controller. This member is indexed through the `logunit` member of the `device` structure associated with this device. Figure 7-35 shows that the driver writer initializes these members to the value zero (0), which indicates that there is no device connected to the controller and that there is no array of pointers to device structures.

Figure 7-35: The dev_name and dev_list Members Initialized



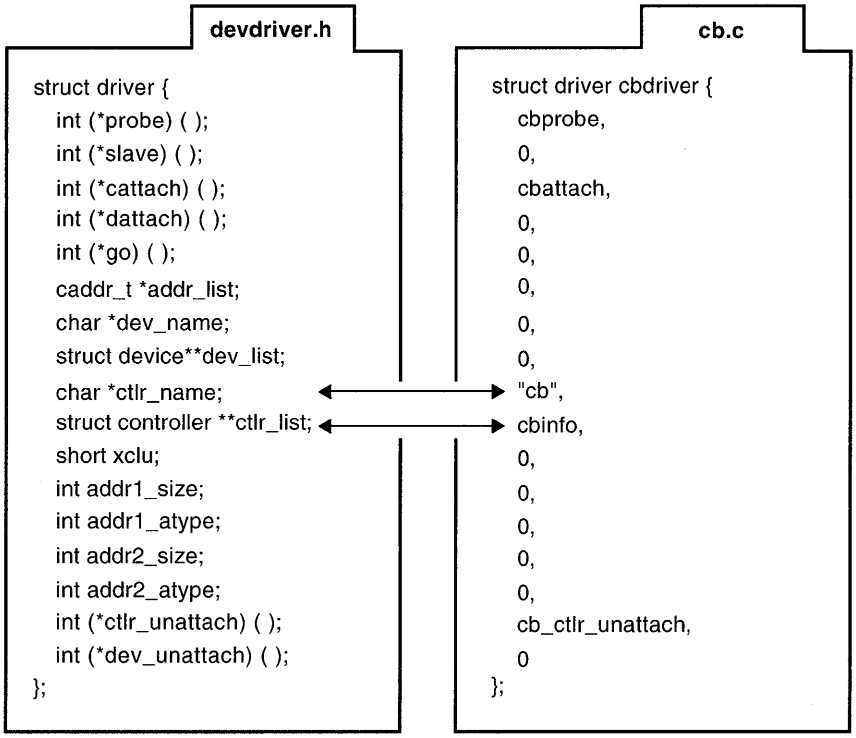
```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

7.6.4 The ctlr_name and ctlr_list Members

The `ctlr_name` member specifies the controller name. The `ctlr_list` member specifies an array of pointers to `controller` structures. This member is used when multiple controllers are controlled by a single device driver. This member is indexed through the `ctlr_num` member of the `controller` structure associated with this device. Figure 7-36 shows that the driver writer initializes these members to the values `cb` and `cbinfo`.

Figure 7-36: The ctlr_name and ctlr_list Members Initialized



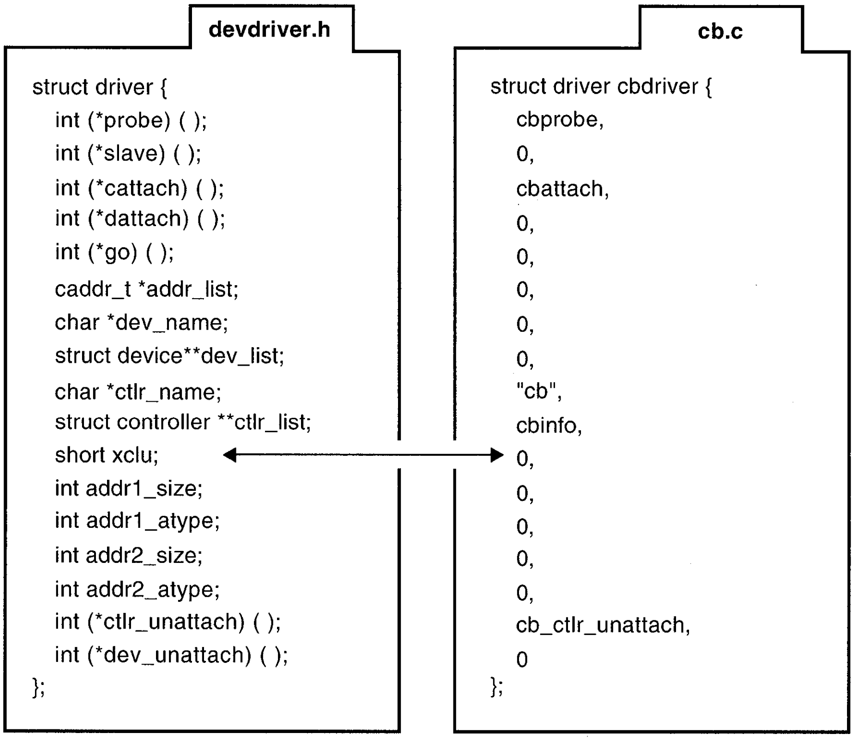
```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

7.6.5 The xclu Member

The `xclu` member specifies a field that is not currently used. Figure 7-37 shows that the driver writer initializes this member to the value zero (0).

Figure 7-37: The xclu Member Initialized



```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

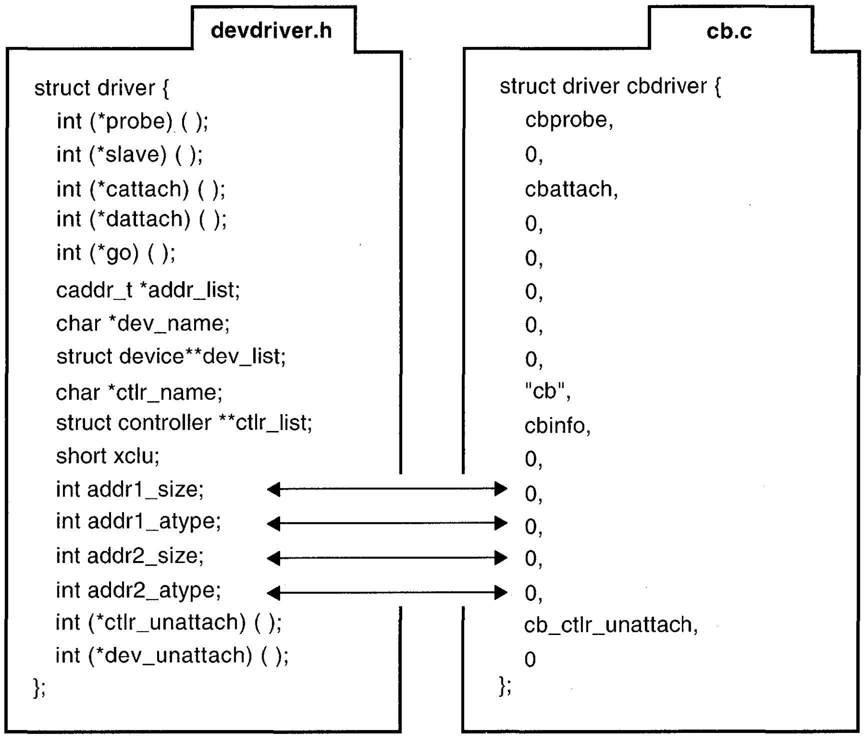
7.6.6 The addr1_size, addr1_atype, addr2_size, and addr2_atype Members

The `addr1_size` member specifies the size (in bytes) of the first control status register (CSR) area. This area is usually the control status register of the device. Only drivers operating on the VMEbus use this member. The `addr1_atype` member specifies the address space and data size of the first CSR area. Only drivers operating on the VMEbus use this member.

The `addr2_size` member specifies the size (in bytes) of the second CSR area. This area is usually the data area that the system uses with devices that have two separate CSR areas. Only drivers operating on the VMEbus use this member. The `addr2_atype` member specifies the address space and data size of the second CSR area. Only drivers operating on the VMEbus use this member. Figure 7-38 shows that the driver writer initializes these members to the value zero (0) to indicate that they are not used by the

/dev/cb driver.

Figure 7-38: The addr1_size, addr1_atype, addr2_size, and addr2_atype Members Initialized



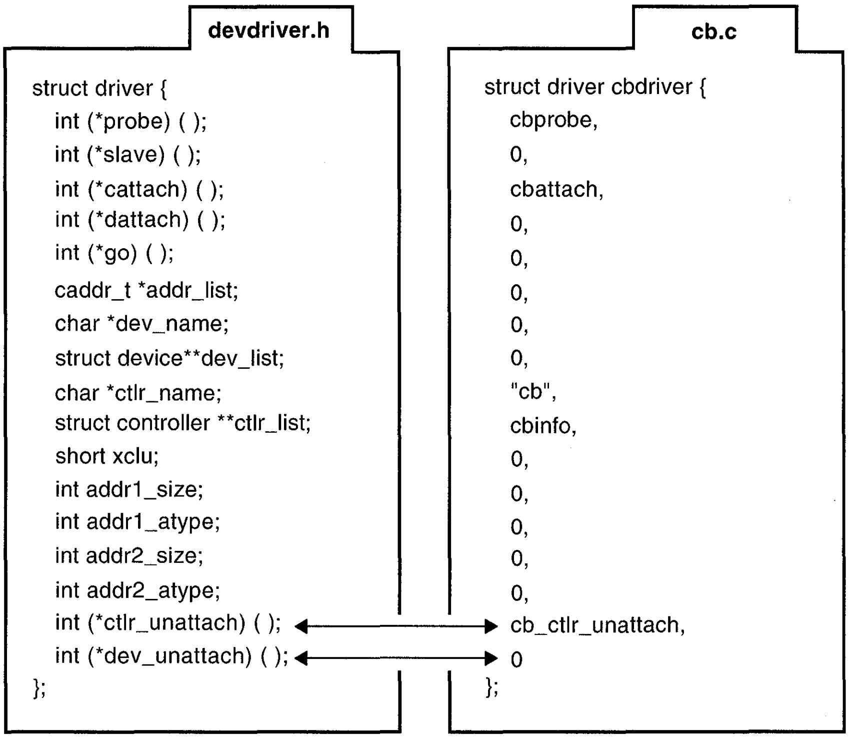
```
struct driver {
   int (*probe) ( );
   int (*slave) ( );
   int (*cattach) ( );
   int (*dattach) ( );
   int (*go) ( );
   caddr_t *addr_list;
   char *dev_name;
   struct device**dev_list;
   char *ctlr_name;
   struct controller **ctlr_list;
   short xclu;
   int addr1_size;
   int addr1_atype;
   int addr2_size;
   int addr2_atype;
   int (*ctlr_unattach) ( );
   int (*dev_unattach) ( );
};
```

```
struct driver cbdriver {
   cbprobe,
   0,
   cbattach,
   0,
   0,
   0,
   0,
   0,
   "cb",
   cbinfo,
   0,
   0,
   0,
   0,
   0,
   cb_ctlr_unattach,
   0
};
```

7.6.7 The ctlr_unattach and dev_unattach Members

The ctlr_unattach member specifies a pointer to the controller's unattach interface. Loadable driver use the controller unattach interface. The dev_unattach member specifies a pointer to the device's unattach interface. Loadable driver use the device unattach interface. Figure 7-39 shows that the driver writer initializes the ctlr_unattach member to the interface cb_ctlr_unattach. The driver writer initializes the dev_unattach member to the value zero (0) to indicate that there is no device unattach interface.

Figure 7-39: The ctlr_unattach and dev_unattach Members Initialized



7.7 The port Structure

The `port` structure contains information about a port.

Table 7-5 lists the member of the `port` structure along with its associated data type.

Table 7-5: The Member of the port Structure

Member Name	Data Type
`conf`	`int (*conf) ()`

The conf member specifies a pointer to the configuration interface for this port.

7.8 The ihandler_t Structure

The ihandler_t structure contains information associated with device driver interrupt handling. Loadable drivers use this data structure. This model of interrupt dispatching uses the bus as the means of interrupt dispatching for all drivers. For this reason, all of the information needed to register an interrupt is considered to be bus-specific. As a result, no attempt is made to represent all the possible permutations within the ihandler_t data structure.

Table 7-6 lists the members of the ihandler_t structure along with their associated data types.

Table 7-6: Members of the ihandler_t Structure

Member Name	Data Type
ih_id	ihandler_id_t
ih_bus	struct bus *
ih_bus_info	char *

The ih_id member specifies a unique ID.

The ih_bus member specifies a pointer to the bus structure associated with this device driver. This member is needed because the interrupt dispatching methodology requires that the bus be responsible for dispatching interrupts in a bus-specific manner.

The ih_bus_info member specifies bus registration information. Each bus type could have different mechanisms for registering interrupt handlers on that bus. Thus, the ih_bus_info member contains the bus-specific information needed to register the interrupt handlers.

For example, on a TURBOchannel bus, the bus-specific information might consist of:

- An interrupt service interface
- A parameter passed to the interrupt service interface
- A slot number

Device driver writers pass the ihandler_t structure to the handler_add interface to specify how interrupt handlers are to be registered with the bus-specific interrupt dispatcher. This task is usually done within the driver's `probe` interface. The following code fragment shows how the `/dev/cb` driver uses the `ihandler_t` data structure:

```
   •
   •
   •
ihandler_id_t cb_id_t[NCB]; 1
   •
   •
   •
ihandler_t handler; 2
   •
   •
   •
struct tc_intr_info info; 3
   •
   •
   •
handler.ih_bus = ctlr->bus_hd; 4
   •
   •
   •
handler.ih_bus_info = (char *)&info; 5
   •
   •
   •
none_id_t[unit] = handler_add(&handler); 6
```

1 Declares an array of IDs used to deregister the interrupt handlers. The `NCB` constant represents the maximum number of `CB` controllers. This number sizes the array of IDs. Thus, there is one ID per `CB` device.

Section 10.5 contains the declaration of this array of IDs.

2 Declares an `ihandler_t` data structure called `handler` to contain information associated with the `/dev/cb` device driver interrupt handling.

3 Declares a `tc_intr_info` data structure called `info`. Note that the `ih_bus_info` member is set to the address of this structure.

4 Specifies the bus that this controller is attached to. The `bus_hd` member of the `controller` structure contains a pointer to the `bus` structure that this controller is connected to. After the initialization, the `ih_bus` member of the `ihandler_t` structure contains the pointer to the `bus` structure associated with the `/dev/cb` device driver.

5 Sets the `ih_bus_info` member of the `handler` data structure to the address of the bus-specific information structure, `info`. This setting is necessary because registration of the interrupt handlers will indirectly call bus-specific interrupt registration interfaces.

6 Calls the .L "handler_add" interface and saves its return value for use later by the `handler_del` interface. The `handler_add` interface

takes one argument: a pointer to an ihandler_t data structure, which in the example is the initialized handler structure. Section 10.8.1 provides additional information on the use of the ihandler_t structure with the handler interfaces.

7.9 The tc_intr_info Structure

The tc_intr_info structure contains interrupt handler information for device controllers connected to the TURBOchannel bus. Loadable drivers initialize the members of the tc_intr_info structure, usually in the driver's probe interface.

Table 7-7 lists the members of the tc_intr_info structure with their associated data types.

Table 7-7: Members of the tc_intr_info Structure

Member Name	Data Type
configuration_st	caddr_t
intr	int (*intr)()
param	caddr_t
config_type	unsigned int

The configuration_st member specifies a pointer to the bus or controller structure for which an associated interrupt handler is written.

The intr member specifies a pointer to the interrupt handler for the specified bus or controller.

The param member specifies a member whose contents are passed to the interrupt service interface.

The config_type member specifies the driver type. You can set this member to one of the following constants defined in /usr/sys/include/io/dec/tc/tc.h: TC_CTLR (controller) TC_ADPT (bus), TC_DEV (device).

The following code fragment shows how the /dev/cb driver uses the tc_intr_info data structure:

```
   •
   •
   •
struct tc_intr_info info; 1
   •
   •
   •
info.configuration_st = (caddr_t)ctlr; 2
info.config_type = TC_CTLR; 3
info.intr = cbintr; 4
info.param = (caddr_t)unit; 5
   •
   •
   •
```

1 Declares a tc_intr_info data structure called info. Note that the ih_bus_info member was set to the address of this structure in the code fragment presented in Section 7.8.

2 Sets the configuration_st member of the info data structure to the pointer to the controller structure associated with this CB device. This controller structure is the one for which an associated interrupt will be written.

3 Sets the config_type member of the info data structure to the constant TC_CTLR, which identifies the /dev/cb driver type as a TURBOchannel controller.

4 Sets the intr member of the info data structure to cbintr, the /dev/cb device driver's interrupt service interface (ISI).

5 Sets the param member of the info data structure to the controller number for the controller structure associated with this CB device. Once the driver is operational and interrupts are generated, the cbintr interface is called with the controller number, which specifies which instance of the controller the interrupt is associated with. Section 10.8.1 provides additional information on the use of the tc_intr_info structure members.

7.10 The handler_key Structure

The handler_key structure contains handler-specific information. This structure is allocated by the handler_add interface, which device drivers call to register the driver's interrupt service interfaces. This structure contains all of the information needed for the bus-specific implementations of the handler_del, handler_disable, and handler_enable interfaces. Table 7-8 lists the members of the handler_key structure along with their associated data types.

Table 7-8: Members of the handler_key Structure

Member Name	Data Type
`next`	`struct handler_key *`
`prev`	`struct handler_key *`
`bus`	`struct bus *`
`bus_id_t`	`ihandler_id_t`
`state`	`unsigned int`
`key`	`ihandler_id_t`
`lock`	`lock_data_t`

The `next` member specifies a pointer to the next handler entry.

The `prev` member specifies a pointer to the previous handler entry.

The `bus` member specifies a pointer to a `bus` structure. The `bus` member is used to find pointers to the bus-specific implementations of `handler_del`, `handler_disable`, and `handler_enable`.

The `bus_id_t` member specifies a bus-specific unique key. The `bus_id_t` member allows the bus-specific implementation to identify which particular interrupt to act upon.

The `state` member specifies state information. You can set this member to the constant `IH_STATE_ENABLED`.

The `key` member specifies a unique key for this entry.

The `lock` member specifies the symmetric multiprocessor (SMP) lock.

The device driver writer does not initialize any of the members of `handler_key`.

7.11 The device_config_t Structure

The `device_config_t` structure contains device configuration information and is the primary mechanism of communication between `cfgmgr` (the configuration manager daemon) and the device driver's `configure` interface.

Table 7-9 lists the members of the `device_config_t` structure with their associated data types.

Table 7-9: Members of the device_config_t Structure

Member Name	Data Type
dc_version	uint
dc_errcode	uint
dc_bmajnum	long
dc_cmajnum	long
dc_begunit	long
dc_numunit	long
dc_dsflags	long
dc_ihflags	long
dc_ihlevel	long
config_name	char

The dc_version member specifies the version of the kernel interfaces that you compiled the driver with. This member is an output field. This member is examined by cfgmgr upon driver loading to ensure compatibility. You can set this member to the constant DRIVER_BUILD_LEVEL, which is defined in /usr/sys/include/sys/sysconfig.h.

The dc_errcode member specifies additional error information. This member is an output field. The driver's configure interface can set this member to a specific error code indicating exactly why the configuration operation failed. Currently, this member is not used.

The dc_bmajnum member specifies the preferred major number for the block device. This member is both an input and an output field. Upon entry to the driver's configure interface, this member is set to specify the desired major number in the bdevsw table. The driver's configure interface passes the major number to bdevsw_add, which registers the device driver's entry points. As an output field, dc_bmajnum specifies the major number that was actually assigned to the driver.

The dc_cmajnum member specifies the preferred major number for the character device. This member is both an input and an output field. Upon entry to the driver's configure interface, this member is set to specify the desired major number in the cdevsw table. The driver's configure interface passes the major number to cdevsw_add, which registers the device driver's entry points.

As an output field, dc_cmajnum specifies the major number that was actually assigned to the driver. The dc_begunit member specifies the first minor device number in the range. This member is an output field. The device driver's configure interface sets this member to be the first minor number used to create the device special files for the driver. Currently, the driver method portion of cfgmgr does not examine this member because the driver's minor number requirements are obtained from the entry in the stanza.loadable file fragment.

The dc_numunit member specifies the number of minor device numbers. This member is an output field. The device driver's configure interface sets this member to the number of minor numbers used to represent this driver. Currently, the driver method portion of cfgmgr does not examine this member because the driver's minor number requirements are obtained from the entry in the stanza.loadable file fragment.

The dc_dsflags member specifies device switch configuration flags. This member is an input field that consists of a set of bit masks used to specify driver attributes. The device driver configuration method portion of cfgmgr can set this member to IH_DRV_DYNAMIC or IH_DRV_SAMEMAJOR. The IH_DRV_DYNAMIC bit mask specifies that the driver has been dynamically loaded. The IH_DRV_SAMEMAJOR bit mask specifies that the driver needs to use the same major number in both the bdevsw and cdevsw tables. In this case, to obtain its major numbers the driver's configure interface should call the dualdevsw_add interface.

The dc_ihflags member specifies a field for future use.

The dc_ihlevel member specifies a field for future use.

The config_name member specifies the driver's configuration name. This member is an input field. The device driver configuration method portion of cfgmgr sets this member to the string name specified by the Module_Config_Name field in the driver's stanza entry in the stanza.loadable file fragment. The driver's configure interface passes this name to the ldbl_stanza_resolver and ldbl_ctlr_configure interfaces to locate the driver's configuration information. The maximum name length is 80 bytes.

The following code fragment shows how the /dev/cb driver uses the device_config_t data structure:

```
   •
   •
   •
device_config_t *indata;
   •
   •
   •
device_config_t *outdata;
   •
   •
   •
outdata->dc_cmajnum = major(cb_devno);
outdata->dc_begunit = 0;
outdata->dc_numunit = num_cb;
```

```
outdata->dc_version = DRIVER_BUILD_LEVEL;
outdata->dc_dsflags = indata->dc_dsflags;
outdata->dc_bmajnum = NODEV;
outdata->dc_errcode = 0;
outdata->dc_ihflags = 0;
outdata->dc_ihlevel = 0;
```

Section 10.9.2 provides additional information on these members as they are initialized by the `/dev/cb` device driver.

Data Structures Used in I/O Operations 8

Data structures are the mechanism used to pass information between the DEC OSF/1 kernel and device driver interfaces. This chapter describes the data structures used in input/output (I/O) operations. Specifically, the chapter discusses:

- The `buf` structure
- The device switch tables
- The `uio` structure
- Buffer cache management
- The interrupt code
- Bus resource management

8.1 The buf Structure

The `buf` structure describes arbitrary I/O, but is 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 for use with the `physio` kernel interface. The `buf` structure does not contain data. Instead, it contains information about where the data resides and information about the types of I/O operations. You need to be familiar with the following topics associated with the `buf` structure:

- Using the systemwide pool of `buf` structures
- Declaring locally defined `buf` structures
- Understanding `buf` structure members used by device drivers

8.1.1 Using the Systemwide Pool of buf Structures

The following code fragment shows how the `/dev/cb` driver discussed in Chapter 10 uses the systemwide pool of `buf` structures with the cbminphys interface:

```
cbminphys(bp)
register struct buf *bp; [1]
{
   •
   •
   •
```

[1] Declares a pointer to a buf structure called bp. The cbminphys interface references the systemwide pool of buf structures to perform a variety of tasks, including checking the size of the requested transfer.

8.1.2 Declaring Locally Defined buf Structures

The following code fragment shows how the /dev/cb driver discussed in Chapter 10 declares an array of locally defined buf structures and references it with the cbattach interface:

```
#define NCB TC_OPTION_SLOTS [1]
   •
   •
   •
struct buf cbbuf[NCB]; [2]
   •
   •
   •
cbattach(ctlr)
struct controller *ctlr; [3]
{
struct cb_unit *cb; [4]
   •
   •
   •
cb->cbbuf = &cbbuf[ctlr->ctlr_num]; [5]
```

[1] The NCB constant is used to allocate the buf structures associated with the CB devices that currently exist on the system. Section 10.3 shows that this constant is defined in the Include Files Section of the /dev/cb device driver.

[2] Declares an array of buf structures called cbbuf. The NCB constant is used to allocate the buf structures for the maximum number of CB devices that currently exist on the system. Thus, there is one buf structure per CB device. Section 10.6 shows that this array is declared in the Local Structure and Variable Definitions Section of the /dev/cb driver.

[3] Declares a pointer to a controller structure associated with a specific CB device. The ctlr_num member of this pointer is used as an index to obtain a specific CB device's associated buf structure.

4 Declares a pointer to the cb_unit data structure associated with this CB device. Section 10.6 shows the declaration of this data structure. It contains members that store such information as whether the CB device is opened and the CB device's TC slot number. It also declares a pointer to the cbbuf structure.

5 Sets the buffer structure address (the cbbuf member of this CB device's cb_unit structure) to the address of this CB device's buf structure. The ctlr_num member is used as an index into the array of buf structures associated with this CB device.

8.1.3 Understanding buf Structure Members Used by Device Drivers

Table 8-1 lists those members of the buf structure with their associated data types that device drivers might reference.

Table 8-1: Members of the buf Structure

Member Name	Data Type
`b_flags`	`int`
`b_forw`	`struct buf *`
`b_back`	`struct buf *`
`av_forw`	`struct buf *`
`av_back`	`struct buf *`
`b_bcount`	`int`
`b_error`	`short`
`b_dev`	`dev_t`
`b_un.b_addr`	`caddr_t`
`b_lblkno`	`daddr_t`
`b_blkno`	`daddr_t`
`b_resid`	`int`
`b_iodone`	`void (*b_iodone) ()`
`b_proc`	`struct proc *`

Writing Device Drivers, Volume 2: Reference provides a reference (man) page-style description of this data structure. The following sections discuss all of these members.

8.1.3.1 The b_flags Member

The `b_flags` member specifies binary status flags. These flags indicate how a request is to be handled and the current status of the request. These status flags are defined in `buf.h` and get set by various parts of the kernel. The flags supply the device driver with information about the I/O operation.

The device driver can also send information back to the kernel by setting `b_flags`. Table 8-2 lists the binary status flags applicable to device drivers.

Table 8-2: Binary Status Flags 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` interface. The device driver writer must call `iodone` to mark a buffer as completed.
`B_ERROR`	This flag specifies that an error occurred on this data transfer. Device drivers set this flag if an error occurs.
`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.
`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` interface when the buffer is freed by the current process. The driver passes the address of the buffer as an argument to `wakeup`.

8.1.3.2 The b_forw and b_back Members

The `b_forw` and `b_back` members specify a file system buffer hash chain. When the kernel performs an I/O operation on a buffer, the `buf` structures are not on any list. Device driver writers sometimes use these members to link `buf` structures to lists.

8.1.3.3 The av_forw and av_back Members

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. The kernel initializes these members. However, when the driver gets use of the buf structure, these members are available for local use by the device driver.

8.1.3.4 The b_bcount and b_error Members

The b_bcount member specifies the size of the requested transfer (in bytes). This member is initialized by the kernel as the result of an I/O request. The driver writer references this member to determine the size of the I/O request. This member is often used in the driver's strategy interface.

The b_error member specifies that an error occurred on this data transfer. This member is set to an error code if the b_flags member bit was set. The driver writer sets this member with the errors defined in the file errno.h.

8.1.3.5 The b_dev Member

The b_dev member specifies the special device to which the transfer is directed. The data type for this member is dev_t, which maps to major and minor construction macros. The device driver writer should not access the dev_t bits directly. Instead, the driver writer should use the major and minor interfaces to obtain the major and minor numbers for a special device. These numbers are specified by the driver writer in the stanza.static file fragment for static drivers and the stanza.loadable file fragment for loadable drivers. Chapter 11 describes these file fragments.

8.1.3.6 The b_un.b_addr Member

The b_un.b_addr member specifies the address at which to pull or push the data. This member is set by the kernel and is the main memory address where the I/O occurs. Driver writers use this member when their drivers need to perform DMA operations. It tells the driver where the data comes from and goes to in memory.

8.1.3.7 The b_lblkno and b_blkno Members

The b_lblkno member specifies the logical block number. The b_blkno member specifies the block number on the partition of a disk or on the file system. The b_blkno member is set by the kernel and it indicates the starting block number on the device where the I/O operation is to begin. This member is used only with block devices. For disk devices, this member is the block number relative to the start of the partition.

8.1.3.8 The b_resid and b_iodone Members

The b_resid member specifies (in bytes) the data not transferred because of some error. The b_iodone member specifies the interface called by iodone. The device driver calls iodone at the completion of an I/O operation. The driver calls the iodone interface, which calls the interface pointed to by the b_iodone member. The driver writer does not need to know anything about the interface pointed to by this argument.

8.1.3.9 The b_proc Member

The b_proc member specifies a pointer to the proc structure that represents the process performing the I/O. A device driver operating on the TURBOchannel bus might reference b_proc in the tc_loadmap interface in order to get the proper physical addresses into the map registers.

8.2 Device Switch Tables

Associated with each device is a unique device number consisting of a major number and a minor number. The major number is used as an index into one of two device switch tables: the cdevsw table for character devices or the bdevsw table for block devices. The device switch tables are located in the file /usr/sys/io/common/conf.c.

The device switch tables have the following characteristics:

- They are arrays of structures that contain device driver entry points. These entry points are actually the addresses of the specific interfaces within the drivers.
- They may contain stubs for device driver entry points for devices that do not exist on a specific machine.
- The location in the table corresponds to the device major number.

The /usr/sys/io/common/conf.c text file is built into the kernel to initialize the bdevsw and cdevsw tables. These tables contain entries for all statically configured drivers. The kernel sizes these tables to include a number of unused entries that will be used at kernel run time to represent loadable drivers. Therefore, loadable device drivers are entered into the memory-resident versions of the bdevsw and cdevsw tables and do not appear in the text file /usr/sys/io/common/conf.c.

The following sections describe the cdevsw and bdevsw tables.

8.2.1 Character Device Switch Table

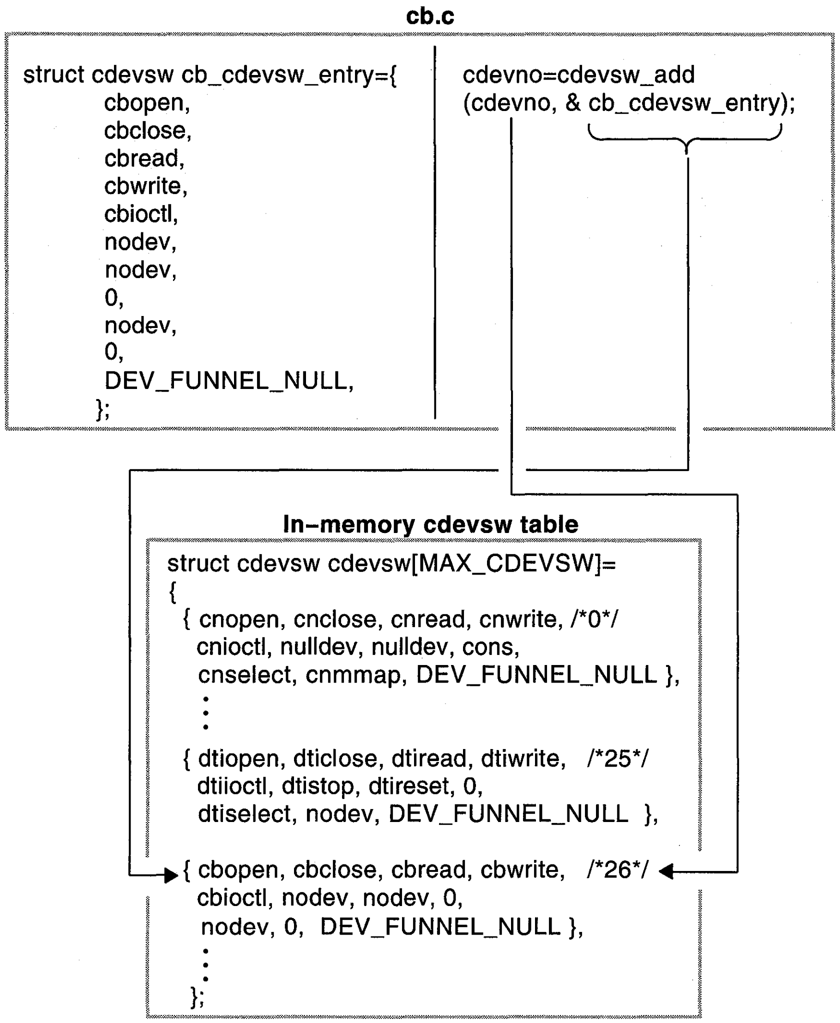
The character device switch, or cdevsw, table is an array of data structures that contains pointers to device driver entry points for each character device supported by the system. In addition, the table can contain stubs for device driver entry points for character mode devices that do not exist or entry points not used by a device driver.

The following shows the cdevsw structure defined in /usr/sys/include/sys/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_funnel; /* serial code compatibility */
};
```

There are two methods for adding device driver interfaces to the cdevsw table, depending on whether the system manager chooses to install the device driver dynamically or statically. Figure 8-1 shows how the device driver interfaces associated with the loadable version of the /dev/cb driver are added to the cdevsw table. As the figure shows, the driver writer declares and initializes a structure called cb_cdevsw_entry that is of type cdevsw. This data structure is usually declared in a section of the driver that contains declarations and definitions used by the loadable version of the driver. Section 10.7 shows that the /dev/cb driver declares and initializes cb_cdevsw_entry in the Loadable Driver Local Structure and Variable Definitions Section.

Figure 8-1: Adding Entries to the cdevsw Table for Loadable Drivers



```
struct cdevsw cb_cdevsw_entry={
        cbopen,
        cbclose,
        cbread,
        cbwrite,
        cbioctl,
        nodev,
        nodev,
        0,
        nodev,
        0,
        DEV_FUNNEL_NULL,
       };
```

```
cdevno=cdevsw_add
(cdevno, & cb_cdevsw_entry);
```

```
struct cdevsw cdevsw[MAX_CDEVSW]=
{
  { cnopen, cnclose, cnread, cnwrite, /*0*/
    cnioctl, nulldev, nulldev, cons,
    cnselect, cnmmap, DEV_FUNNEL_NULL },
    .
    .
    .
  { dtiopen, dticlose, dtiread, dtiwrite,   /*25*/
    dtiioctl, dtistop, dtireset, 0,
    dtiselect, nodev, DEV_FUNNEL_NULL  },

  { cbopen, cbclose, cbread, cbwrite,   /*26*/
    cbioctl, nodev, nodev, 0,
    nodev, 0,  DEV_FUNNEL_NULL },
    .
    .
    .
  };
```

The figure also shows that the driver writer uses `cdevsw_add` as the mechanism for adding the driver interfaces to the `cdevsw` table. The first argument to `cdevsw_add` is the major number associated with the CB device. The second argument is the address of the previously initialized `cb_cdevsw_entry` structure.

Furthermore, the figure shows that when the loadable version of the driver is configured, `cdevsw_add` locates the position in the table corresponding to the major number, which in this example is 26, and adds the entries from cb_cdevsw_entry. Note that the cdevsw_add interface adds the driver's entry points into the in-memory resident cdevsw table. It does not change the /usr/sys/io/common/conf.c file, which is used to build the kernel. Thus, the driver's entry points are dynamically added to the cdevsw table for loadable drivers.

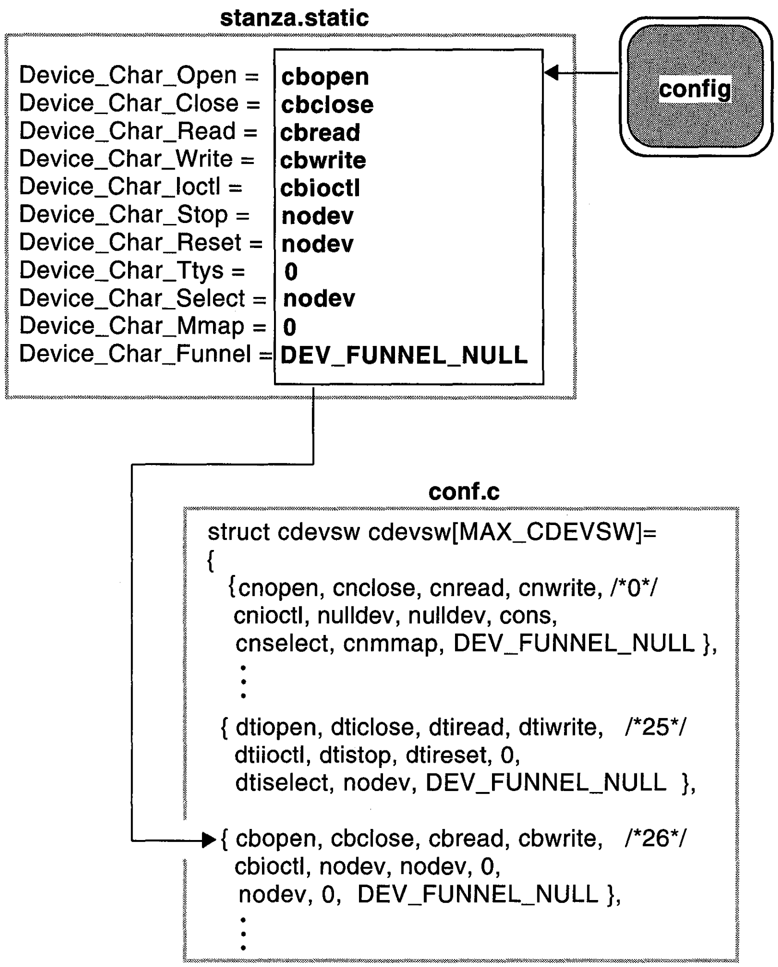
The loadable version of the driver usually calls the cdevsw_add interface in a Loadable Device Driver Section. Section 10.9.2 describes in detail the section of the /dev/cb driver that calls cdevsw_add.

For static drivers that follow the traditional device driver configuration model, the device driver interfaces are added by the device driver writer by manually editing the cdevsw table. For static drivers that follow the third-party device driver configuration model, the device driver interfaces are added by the config program as shown in Figure 8-2.

As the figure shows, config parses the appropriate driver interface entries in the stanza.static file fragment, locates the appropriate major device number (in this case 26), and adds these entries to the table.

The following sections discuss the members of the cdevsw structure.

Figure 8-2: Adding Entries to the cdevsw Table for Static Drivers


stanza.static
Device_Char_Open = cbopen
Device_Char_Close = cbclose
Device_Char_Read = cbread
Device_Char_Write = cbwrite
Device_Char_Ioctl = cbioctl
Device_Char_Stop = nodev
Device_Char_Reset = nodev
Device_Char_Ttys = 0
Device_Char_Select = nodev
Device_Char_Mmap = 0
Device_Char_Funnel = DEV_FUNNEL_NULL
config
conf.c
struct cdevsw cdevsw[MAX_CDEVSW]=
{
{cnopen, cnclose, cnread, cnwrite, /*0*/
cnioctl, nulldev, nulldev, cons,
cnselect, cnmmap, DEV_FUNNEL_NULL },
{ dtiopen, dticlose, dtiread, dtiwrite, /*25*/
dtiioctl, dtistop, dtireset, 0,
dtiselect, nodev, DEV_FUNNEL_NULL },
{ cbopen, cbclose, cbread, cbwrite, /*26*/
cbioctl, nodev, nodev, 0,
nodev, 0, DEV_FUNNEL_NULL },

8.2.1.1 The d_open and d_close Members

The `d_open` member specifies a pointer to an entry point for the driver's `open` interface, which opens a device. As shown in Figure 8-1 and Figure 8-2, the `d_open` member is initialized to `cbopen` for both the dynamically configured and statically configured `/dev/cb` driver.

The `cdevsw_add` interface is used to dynamically assign device major numbers to loadable character device drivers. The `cdevsw_add` interface checks the `d_open` member to determine if the entry is available to be assigned a dynamic major device number. If your device driver does not have an open interface, you must set this member to nulldev to tell cdevsw_add that the entry is not available for dynamic assignment of the device major number.

The d_close member specifies a pointer to an entry point for the driver's close interface, which closes a device. As shown in Figure 8-1 and Figure 8-2, the d_close member is initialized to cbclose for the dynamically configured and statically configured /dev/cb driver.

8.2.1.2 The d_read and d_write Members

The d_read member specifies a pointer to an entry point for the driver's read interface, which reads characters or raw data. As shown in Figure 8-1 and Figure 8-2, the d_read member is initialized to cbread for both the dynamically configured and statically configured /dev/cb driver.

The d_write member specifies a pointer to an entry point for the driver's write interface, which writes characters or raw data. As shown in Figure 8-1 and Figure 8-2, the d_write member is initialized to cbwrite for both the dynamically configured and statically configured /dev/cb driver.

8.2.1.3 The d_ioctl and d_stop Members

The d_ioctl member specifies a pointer to an entry point for the driver's ioctl interface, which performs special functions or I/O control. As shown in Figure 8-1 and Figure 8-2, the d_ioctl member is initialized to cbioctl for both the dynamically configured and statically configured /dev/cb driver.

The d_stop member specifies a pointer to an entry point for the driver's stop interface, which suspends other processing on behalf of the current process. The d_stop member is typically used only for terminal drivers. As shown in Figure 8-1 and Figure 8-2, the d_stop member is initialized to nodev for both the dynamically configured and statically configured /dev/cb driver. The nodev entry calls the nodev interface, which returns an ENODEV (error, no such device). You should specify nodev when it is not appropriate to call that interface for a particular driver. For example, a device driver written for a write-only printer has no need for a read interface. Therefore, the read entry point would contain a nodev entry. In this example, it is not appropriate to call a stop interface for the /dev/cb driver; therefore, the nodev entry is specified.

You could also specify nulldev. The nulldev entry calls the nulldev interface, which returns the value zero (0). You should specify nulldev when it is appropriate for the interface to be called, but the driver does not need to perform any actions to support the interface. If the stop interface had no functionality for the /dev/cb device, the nulldev entry would have been specified instead of nodev.

8.2.1.4 The d_reset and d_ttys Members

The d_reset member specifies a pointer to an entry point for the driver's reset interface, which stops all current work and places the device connected to the controller in a known, quiescent state. As shown in Figure 8-1 and Figure 8-2, the d_reset member is initialized to nodev for both the dynamically configured and statically configured /dev/cb driver. The nodev entry calls the nodev interface, which returns an ENODEV (error, no such device). You should specify nodev when it is not appropriate to call that interface for a particular driver. For example, a device driver written for a write-only printer has no need for a read interface. Therefore, the read entry point would contain a nodev entry. In this example, it is not appropriate to call a reset interface for the /dev/cb driver; therefore, the nodev entry is specified.

The d_ttys member specifies a pointer to driver private data. As shown in Figure 8-1 and Figure 8-2, the d_ttys member is initialized to the value zero (0) for both the dynamically configured and statically configured /dev/cb driver. The value zero (0) indicates that the /dev/cb device does not support the d_ttys member. A possible value for this member is an array of tty data structures. A tty data structure is associated with terminal device drivers, which are not discussed in this book.

8.2.1.5 The d_select and d_mmap Members

The d_select member specifies a pointer to an entry point for the driver's select interface, which determines if a call to a read or write interface will block. As shown in Figure 8-1 and Figure 8-2, the d_select member is initialized to nodev for both the dynamically configured and statically configured /dev/cb driver. The nodev entry calls the nodev interface, which returns an ENODEV (error, no such device). You should specify nodev when it is not appropriate to call that interface for a particular driver. For example, a device driver written for a write-only printer has no need for a read interface. Therefore, the read entry point would contain a nodev entry. In this example, it is not appropriate to call a select interface for the /dev/cb driver; therefore, the nodev entry is specified.

The d_mmap member specifies a pointer to an entry point for the driver's mmap interface, which maps kernel memory to user address space. As shown in Figure 8-1 and Figure 8-2, the d_mmap member is initialized to the value zero (0) for both the dynamically configured and statically configured /dev/cb driver.

8.2.1.6 The d_funnel Member

The d_funnel member schedules a device driver onto a CPU in a multiprocessor configuration. Because multiprocessor configurations are not supported on DEC OSF/1, set this member to the constant DEV_FUNNEL_NULL. As shown in Figure 8-1 and Figure 8-2, the d_funnel member is initialized to DEV_FUNNEL_NULL for both the dynamically configured and statically configured /dev/cb device driver.

8.2.2 Block Device Switch Table

The block device switch, or bdevsw, table is an array of data structures that contains pointers to 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 entry points not used by a device driver.

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

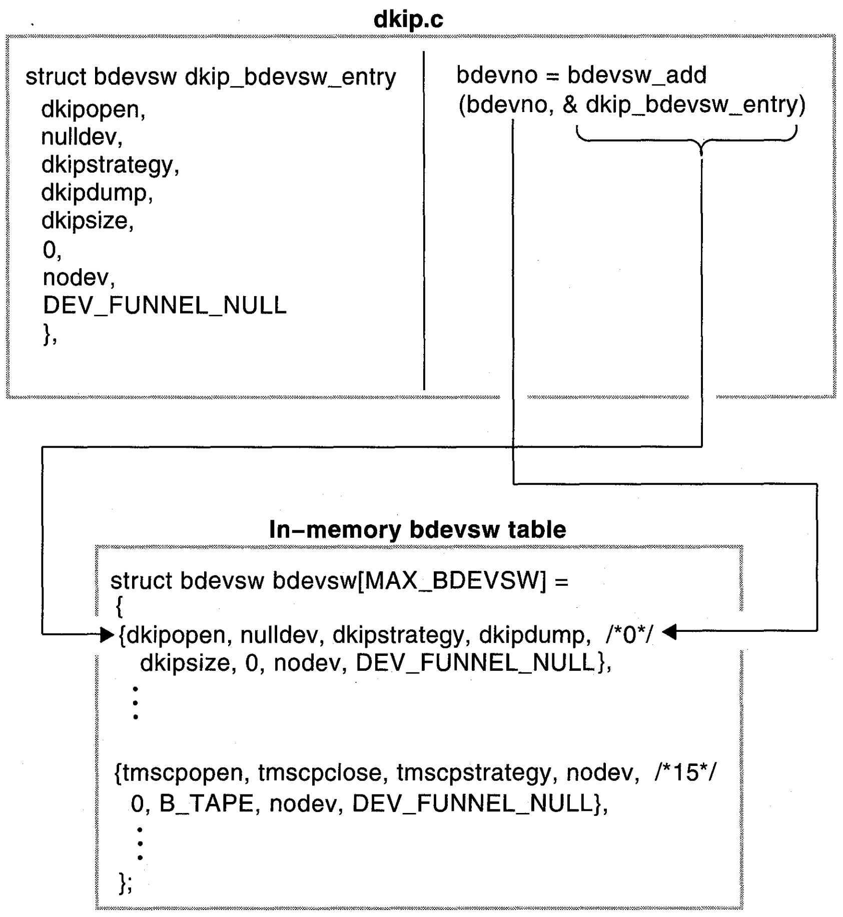
```
struct bdevsw
{
   int      (*d_open)();
   int      (*d_close)();
   int      (*d_strategy)();
   int      (*d_dump)();
   int      (*d_psize)();
   int      d_flags;
   int      (*d_ioctl)();
   int      d_funnel; /* serial code compatibility */
};
```

The way the bdevsw table gets filled in differs, depending on whether the system manager configures the loadable or static version of the driver. Before discussing the members of the bdevsw data structure, it is useful to describe how entries are added to the bdevsw table for loadable and static drivers.

The method for adding device driver interfaces to the bdevsw table differs, depending on whether the system manager chooses to install the device driver dynamically or statically. Figure 8-3 shows how the device driver interfaces are added to the bdevsw table when the system manager dynamically configures the dkip driver. As the figure shows, the driver writer declares and initializes a structure called dkip_bdevsw_entry that is of type bdevsw. This data structure is usually declared in a section of the driver that contains declarations and definitions used by the loadable version of the driver.

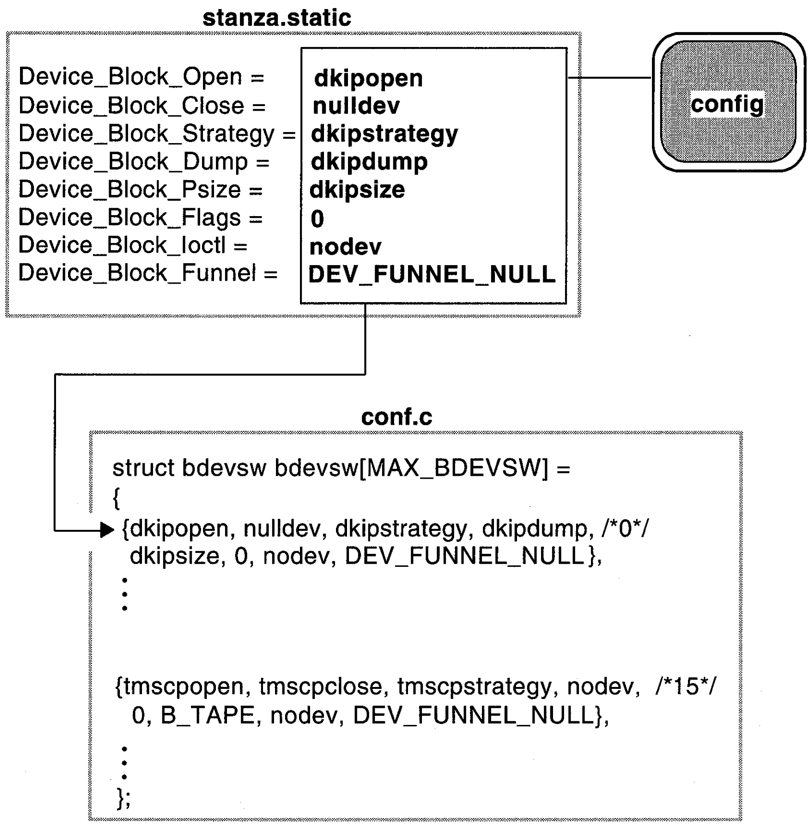
The figure also shows that the driver writer uses bdevsw_add as the mechanism for adding the driver interfaces to the bdevsw table. The first argument to bdevsw_add is the major number associated with the dkip device. The second argument is the address of the previously initialized dkip_bdevsw_entry structure. Furthermore, the figure shows that when the loadable version of the driver is configured, bdevsw_add locates the major device number in the table, which in this example is zero (0), and adds the entries from dkip_bdevsw_entry. Note that the bdevsw_add interface adds the driver's entry points into the in-memory resident bdevsw table. It does not change the /usr/sys/io/common/conf.c file, which is used to build the kernel. Thus, the driver's entry points are dynamically added to the bdevsw table for loadable drivers.

Figure 8-3: Adding Entries to the bdevsw Table for Loadable Drivers


dkip.c
struct bdevsw dkip_bdevsw_entry
dkipopen,
nulldev,
dkipstrategy,
dkipdump,
dkipsize,
0,
nodev,
DEV_FUNNEL_NULL
},
bdevno = bdevsw_add
(bdevno, & dkip_bdevsw_entry)
In-memory bdevsw table
struct bdevsw bdevsw[MAX_BDEVSW] =
{
{dkipopen, nulldev, dkipstrategy, dkipdump, /*0*/
dkipsize, 0, nodev, DEV_FUNNEL_NULL},
{tmscpopen, tmscpclose, tmscpstrategy, nodev, /*15*/
0, B_TAPE, nodev, DEV_FUNNEL_NULL},
};

For static drivers that follow the traditional device driver configuration model, the device driver interfaces are added by the device driver writer by manually editing the `bdevsw` table. For static drivers that follow the third-party device driver configuration model, the device driver interfaces are added by the `config` program as shown in Figure 8-4.

Figure 8-4: Adding Entries to the bdevsw Table for Static Drivers



```
Device_Block_Open =      dkipopen
Device_Block_Close =     nulldev
Device_Block_Strategy =  dkipstrategy
Device_Block_Dump =      dkipdump
Device_Block_Psize =     dkipsize
Device_Block_Flags =     0
Device_Block_Ioctl =     nodev
Device_Block_Funnel =    DEV_FUNNEL_NULL
```

```
struct bdevsw bdevsw[MAX_BDEVSW] =
{
{dkipopen, nulldev, dkipstrategy, dkipdump, /*0*/
 dkipsize, 0, nodev, DEV_FUNNEL_NULL},
 .
 .
 .

{tmscpopen, tmscpclose, tmscpstrategy, nodev,  /*15*/
 0, B_TAPE, nodev, DEV_FUNNEL_NULL},
 .
 .
 .
};
```

As the figure shows, `config` parses the appropriate driver interface entries in the `stanza.static` file fragment, locates the appropriate major device number (in this case 0), and adds these entries to the table.

The following sections discuss the members of the `bdevsw` structure.

8.2.2.1 The d_open and d_close Members

The d_open member specifies a pointer to an entry point for the driver's open interface, which opens a device. As shown in Figure 8-3 and Figure 8-4, the d_open member is initialized to dkipopen and tmscpopen for both the dynamically configured and statically configured dkip and tmscp drivers.

The bdevsw_add interface is used to dynamically assign device major numbers to loadable block device drivers. The bdevsw_add interface checks the d_open member to determine if the entry is available to be assigned a dynamic major device number. If your device driver does not have an open interface, you must set this member to nulldev to tell bdevsw_add that the entry is not available for dynamic assignment of the device major number.

The d_close member specifies a pointer to an entry point for the driver's close interface, which closes a device. As shown in Figure 8-3 and Figure 8-4, the d_close member is initialized to nulldev for both the dynamically configured and statically configured dkip driver. The nulldev entry calls the nulldev interface, which returns the value zero (0). You should specify nulldev when it is appropriate for the interface to be called, but the driver does not need to perform any actions to support the interface. The close interface has no functionality for the dkip device; therefore, the nulldev entry is specified.

The figure also shows that d_close is initialized to tmscpclose for both the dynamically configured and statically configured tmscp driver.

8.2.2.2 The d_strategy and d_dump Members

The d_strategy member specifies the device driver's strategy interface for the device. As shown in Figure 8-3 and Figure 8-4, the d_strategy member is initialized to dkipstrategy and tmscpstrategy for both the dynamically configured and statically configured dkip and tmscp drivers.

The d_dump member specifies a pointer to an entry point for the driver's dump interface, which is used for panic dumps of the system image. As shown in Figure 8-3 and Figure 8-4, the d_dump member is initialized to dkipdump for both the dynamically configured and statically configured dkip driver and to nodev for both the dynamically configured and statically configured tmscp driver. The nodev entry calls the nodev interface, which returns an ENODEV (error, no such device). You should specify nodev when it is not appropriate to call that interface for a particular driver. In this example, it is not appropriate to call a dump interface for the tmscp driver; therefore, the nodev entry is specified.

8.2.2.3 The d_psize and d_flags Members

The d_psize member specifies a pointer to an entry point for the driver's psize interface, which returns the size in physical blocks of a device (disk partition). As shown in Figure 8-3 and Figure 8-4, the d_psize member is initialized to dkipsize for both the dynamically configured and statically configured dkip driver. The figure also shows that d_psize is initialized to the value zero (0) for both the dynamically configured and statically configured tmscp driver. The value zero (0) indicates that the tmscp device does not support disk partitions (because it is a tape device).

The d_flags member specifies device-related flags. For tape drivers, this member gets set to the flag B_TAPE. This flag is set in the b_flags member of the buf structure. The B_TAPE flag determines whether to use delayed writes, which are not allowed for tape devices. For all other drivers, this member is set to the value zero (0). As shown in Figure 8-3 and Figure 8-4, the d_flags member is initialized to the value zero (0) for both the dynamically configured and statically configured dkip driver. The figure also shows that for the tmscp driver the d_flags member is initialized to the constant B_TAPE.

8.2.2.4 The d_ioctl and d_funnel Members

The d_ioctl member specifies a pointer to an entry point for the driver's ioctl interface, which performs special functions or I/O control. As shown in Figure 8-3 and Figure 8-4, the d_ioctl member is initialized to nodev for both the dynamically configured and statically configured dkip and tmscp drivers. The nodev entry calls the nodev interface, which returns an ENODEV (error, no such device). You should specify nodev when it is not appropriate to call that interface for a particular driver. In this example, it is not appropriate to call an ioctl interface for the dkip and tmscp block drivers; therefore, the nodev entry is specified.

The d_funnel member schedules a device driver onto a CPU in a multiprocessor configuration. Because multiprocessor configurations are not supported on DEC OSF/1, set this member to the constant DEV_FUNNEL_NULL. As shown in Figure 8-3 and Figure 8-4, the d_funnel member is initialized to the constant DEV_FUNNEL_NULL for both the dynamically configured and statically configured dkip and tmscp drivers.

8.3 The uio Structure

The uio structure describes I/O, either single vector or multiple vectors. Typically, device drivers do not manipulate the members of this structure. However, the structure is presented here for the purpose of understanding the uiomove kernel interface, which operates on the members of the uio structure. Table 8-3 lists the members of the `uio` structure along with their associated data types that you might need to understand.

Table 8-3: Members of the uio Structure

Member Name	Data Type
`uio_iov`	`struct iovec *`
`uio_iovcnt`	`int`
`uio_offset`	`off_t`
`uio_segflg`	`enum uio_seg`
`uio_resid`	`int`
`uio_rw`	`enum uio_rw`

8.3.1 The uio_iov and uio_iovcnt Members

The `uio_iov` member specifies a pointer to the first `iovec` structure. The `iovec` structure has two members: one that specifies the address of the segment and another that specifies the size of the segment. The system allocates contiguously the `iovec` structures for a given transfer.

The `uio_iovcnt` member specifies the number of `iovec` structures for this transfer.

8.3.2 The uio_offset and uio_segflg Members

The `uio_offset` member specifies the offset within the file.

The `uio_segflg` member specifies the segment type. This member can be set to one of the following 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).

8.3.3 The uio_resid and uio_rw Members

The `uio_resid` member specifies the number of bytes that still need to be transferred.

The `uio_rw` member specifies whether the transfer is a read or a write. This member is set by `read` and `write` system calls according to the corresponding field in the file descriptor. This member can be set to one of the following values: `UIO_READ` (read transfer) or `UIO_WRITE` (write transfer).

8.4 Buffer Cache Management

When the file system deals with regular files, directories, and block devices, the I/O requests are serviced through the buffer cache system. Because the buffer cache deals with fixed-size buffers, it is often necessary to translate the user's request for I/O into buffer-size pieces called blocks. Only in the case where the size of the user's I/O matches a block and aligns to a block boundary will the underlying request match with the user's size. A large I/O request will be broken down into many block requests, with each block request going to the buffer cache system separately. Both read and write requests smaller than a block force the file system to request a read of the entire block and deal with the small read or write in the buffer.

Regular files and directories go through an extra translation process to map their logical block number into the physical blocks of the disk device. This mapping process itself can generate block requests to the buffer cache system to deal with file extension or to obtain or modify indirect file system blocks.

Buffer reads and buffer writes do not necessarily cause I/O to occur. In the case of buffer reads, the request can be satisfied by data already in the cache. On the other hand, buffer writes can modify or replace data in the cache, but the physical write might be delayed. Using a buffer cache enhances performance because data that changes often in the cache does not require a physical write for each change.

The nature of the buffer cache system's delayed physical I/O requires that each buffer request, or each block read or write, be a self contained I/O request to the device driver's `strategy` interface. The buffer cache system and the block device driver `strategy` interface cannot assume any particular process context; therefore, the context of the process must be severed from the I/O request. The buffer passed to the buffer interface has all of the context necessary to perform the I/O.

8.4.1 Buffer Header

I/O requests come from a buffer cache interface as follows:

```
(*bdevsw[major](dev)].d_strategy)(bp);
```

The `bdevsw` table is referenced and the appropriate driver interface is called through the block device's major number. The driver's `strategy` interface is passed a pointer to a `buf` structure.

8.5 Interrupt Code

When the kernel interrupt interface that does the initial handling of interrupts receives an interrupt, it:

- Saves the state of the CPU (for example, the registers)
- Sets up the argument list for the call to the driver (that is, the unit number)
- Transfers control to the appropriate driver, based on the interrupt vector index provided by the bus

Upon return from the driver's interrupt service interface, the kernel restores the state of the CPU to allow previously running processes to run.

Using Kernel Interfaces with Device Drivers 9

This chapter discusses the kernel interfaces most commonly used by device drivers and provides code fragments to illustrate how to call these interfaces in device drivers. These code fragments and associated descriptions supplement the reference (man) page-style descriptions for these and the other kernel interfaces presented in *Writing Device Drivers, Volume 2: Reference.* Specifically, the chapter discusses the following:

- String interfaces
- Virtual memory interfaces
- Data copying interfaces
- Hardware-related interfaces
- Kernel-related interfaces
- Loadable driver interfaces
- Input/output (I/O) handle-related interfaces
- Direct memory access related interfaces
- Miscellaneous interfaces

9.1 String Interfaces

String interfaces allow device drivers to:

- Compare two null-terminated strings
- Compare two strings by using a specified number of characters
- Copy a null-terminated character string
- Copy a null-terminated character string with a specified limit
- Return the number of characters in a null-terminated string

The following sections describe the kernel interfaces that perform these tasks.

9.1.1 Comparing Two Null-Terminated Strings

To compare two null-terminated character strings, call the `strcmp` interface.

The following code fragment shows a call to `strcmp`:

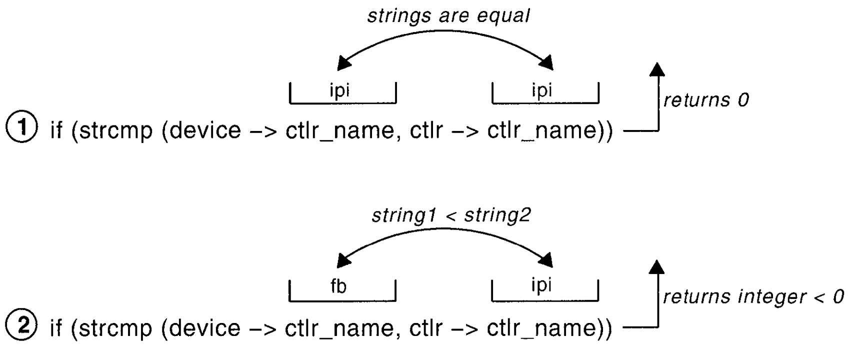
```
   •
   •
   •
register struct device *device;
struct controller *ctlr;
   •
   •
   •
if (strcmp(device->ctlr_name, ctlr->ctlr_name)) { 1
   •
   •
   •
}
```

1 Shows that the `strcmp` interface takes two arguments. The first argument specifies a pointer to a string (an array of characters terminated by a null character). In this example, this is the controller name pointed to by the `ctlr_name` member of the pointer to the `device` structure. The second argument also specifies a pointer to a string (an array of characters terminated by a null character). In the example, this is the controller name pointed to by the `ctlr_name` member of the pointer to the `controller` structure.

The code fragment sets up a condition statement that performs some tasks based on the results of the comparison. Figure 9-1 shows how `strcmp` compares two sample character string values in the code fragment. In item 1, `strcmp` compares the two controller names and returns the value zero (0) because `strcmp` performed a lexicographical comparison between the two strings and they were identical.

In item 2, `strcmp` returns an integer that is less than zero because the lexicographical comparison indicates that the characters in the first controller name, `fb`, come before the letters in the second controller name, `ipi`.

Figure 9-1: Results of the strcmp Interface


strings are equal
ipi
ipi
returns 0
① if (strcmp (device -> ctlr_name, ctlr -> ctlr_name))
string1 < string2
fb
ipi
returns integer < 0
② if (strcmp (device -> ctlr_name, ctlr -> ctlr_name))

9.1.2 Comparing Two Strings by Using a Specified Number of Characters

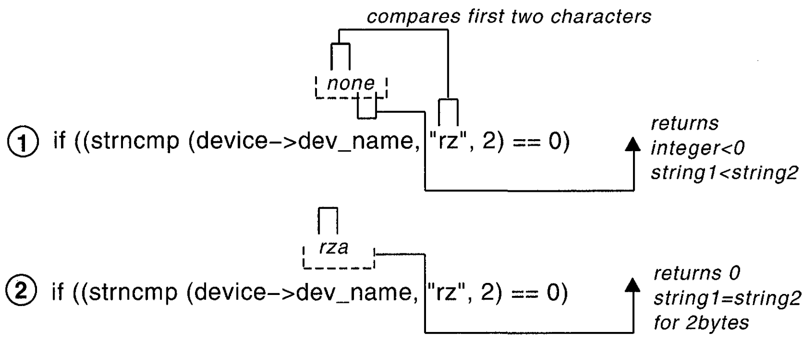
To compare two strings by using a specified number of characters, call the `strncmp` interface. The following code fragment shows a call to `strncmp`:

```
   •
   •
   •
register struct device *device;
   •
   •
   •
if( (strncmp(device->dev_name, "rz", 2) == 0)) 1
   •
   •
   •
```

1 Shows that the `strncmp` interface takes three arguments. The first argument specifies a pointer to a string (an array of characters terminated by a null character). In the example, this is the device name pointed to by the `dev_name` member of the pointer to the `device` structure. The second argument also specifies a pointer to a string (an array of characters terminated by a null character). In the example, this is the character string `rz`. The third argument specifies the number of bytes to be compared. In the example, the number of bytes to compare is 2.

The code fragment sets up a condition statement that performs some tasks based on the results of the comparison. Figure 9-2 shows how `strncmp` compares two sample character string values in the code fragment. In item 1, `strncmp` compares the first two characters of the device name `none` with the string `rz` and returns an integer less than the value zero (0). The reason for this is that `strncmp` makes a lexicographical comparison between the two strings and the string `no` comes before the string `rz`. In item 2, `strncmp` compares the first two characters of the device name `rz3a` with the string `rz` and returns the value zero (0). The reason for this is that `strncmp` makes a lexicographical comparison between the two strings and the string `rz` is equal to the string `rz`.

Figure 9-2: Results of the strncmp Interface


compares first two characters
none
① if ((strncmp (device->dev_name, "rz", 2) == 0)
returns
integer<0
string1<string2
rza
② if ((strncmp (device->dev_name, "rz", 2) == 0)
returns 0
string1=string2
for 2bytes

9.1.3 Copying a Null-Terminated Character String

To copy a null-terminated character string, call the `strcpy` interface. The following code fragment shows a call to `strcpy`:

```
   •
   •
   •
struct tc_slot  tc_slot[TC_IOSLOTS]; [1]
char curr_module_name[TC_ROMNAMLEN + 1]; [2]
   •
   •
   •
strcpy(tc_slot[i].modulename, curr_module_name); [3]
   •
   •
   •
```

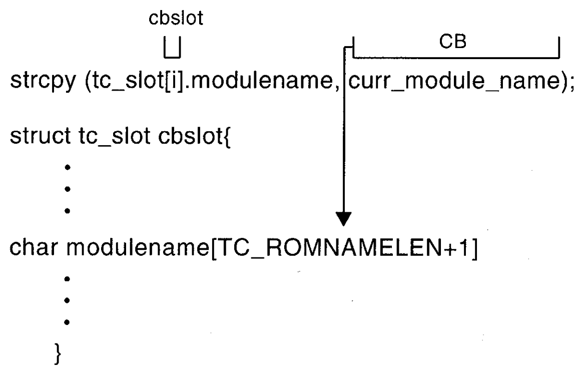
[1] Declares an array of `tc_slot` structures of size `TC_IOSLOTS`.

[2] Declares a variable to store the module name from the ROM of a device on the TURBOchannel bus.

3 Shows that the strcpy interface takes two arguments. The first argument specifies a pointer to a buffer large enough to hold the string to be copied. In the example, this buffer is the modulename member of the tc_slot structure associated with the specified bus. The second argument specifies a pointer to a string (an array of characters terminated by a null character). This is the string to be copied to the buffer specified by the first argument. In the example, this is the module name from the ROM, which is stored in the *curr_module_name* variable.

Figure 9-3 shows how strcpy copies a sample value in the code fragment. The interface copies the string CB (the value contained in *curr_module_name*) to the modulename member of the tc_slot structure associated with the specified bus. This member is presumed large enough to store the character string. The strcpy interface returns the pointer to the location following the end of the destination buffer.

Figure 9-3: Results of the strcpy Interface


cbslot
CB
strcpy (tc_slot[i].modulename, curr_module_name);
struct tc_slot cbslot{
char modulename[TC_ROMNAMELEN+1]
}

9.1.4 Copying a Null-Terminated Character String with a Specified Limit

To copy a null-terminated character string with a specified limit, call the strncpy interface. The following code fragment shows a call to strncpy:

```
   •
   •
   •
register struct device *device;
char * buffer;
   •
   •
   •
strncpy(buffer, device->dev_name, 2); 1
```

```
if (buffer == somevalue)
   •
   •
   •
```

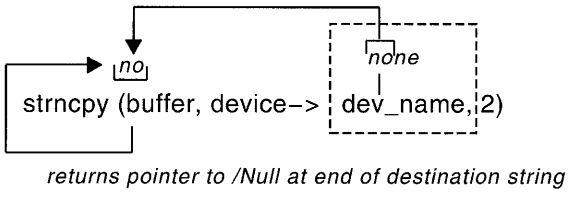
1 Shows that `strncpy` takes three arguments. The first argument specifies a pointer to a buffer of at least the same number of bytes as specified in the third argument. In the example, this is the pointer to the *`buffer`* variable. The second argument specifies a pointer to a string (an array of characters terminated by a null character). This is the character string to be copied and in the example is the value pointed to by the `dev_name` member of the pointer to the `device` structure. The third argument specifies the number of characters to copy, which in the example is two characters.

The code fragment sets up a condition statement that performs some tasks based on the characters stored in the pointer to the *`buffer`* variable.

Figure 9-4 shows how `strncpy` copies a sample value in the code fragment. The interface copies the first two characters of the string `none` (the value pointed to by the `dev_name` member of the pointer to the `device` structure). The `strncpy` interface stops copying after it copies a null character or the number of characters specified in the third argument, whichever comes first.

The figure also shows that `strncpy` returns a pointer to the /NULL character at the end of the first string (or to the location following the last copied character if there is no NULL). The copied string will not be null terminated if its length is greater than or equal to the number of characters specified in the third argument.

Figure 9-4: Results of the strncpy Interface


no
none
strncpy (buffer, device-> dev_name, 2)
returns pointer to /Null at end of destination string

9.1.5 Returning the Number of Characters in a Null-Terminated String

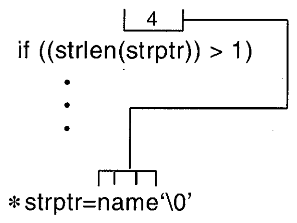
To return the number of characters in a null-terminated character string, call the `strlen` interface. The following code fragment shows a call to

```
strlen:
   •
   •
   •
char *strptr;
   •
   •
   •
if ((strlen(strptr)) > 1) [1]
   •
   •
   •
```

[1] Shows that the `strlen` interface takes one argument, which specifies a pointer to a string (an array of characters terminated by a null character). In the example, this pointer is the variable *`strptr`*.

The code fragment sets up a condition statement that performs some tasks based on the length of the string. Figure 9-5 shows how `strlen` checks the number of characters in a sample string in the code fragment. As the figure shows, `strlen` returns the number of characters pointed to by the *`strptr`* variable, which in the code fragment is four. Note that `strlen` does not count the terminating null character.

Figure 9-5: Results of the strlen Interface



9.2 Virtual Memory Interfaces

The virtual memory interfaces allow device drivers to:

- Allocate a section of kernel virtual memory
- Return previously allocated kernel virtual memory
- Perform nonblocking allocation of kernel virtual memory

The following sections describe the kernel interfaces that perform these tasks.

9.2.1 Allocating a Section of Kernel Virtual Memory

To allocate a variable-sized section of kernel virtual memory, call the `kalloc` interface. This interface is often used to dynamically allocate data structures in the driver's `probe` interface. See Section 2.3.3.3 for an example of how to use `kalloc` to accomplish this task.

The following code fragment shows a call to `kalloc`:

```
   •
   •
   •
struct device *device;
struct device *new_dev;
   •
   •
   •
new_dev = (struct device *) kalloc(sizeof (struct device));
1
   •
   •
   •
```

1 Shows that the `kalloc` interface takes one argument: the number of bytes of memory to allocate. To allocate the number of bytes associated with a `device` structure, the code fragment uses the `sizeof` operator.

The return type of `kalloc` is defined as `caddr_t`. Because the data types of the return values are different, the code fragment performs a type-casting operation that converts the return type to be a pointer to a `device` structure.

If this call to `kalloc` completes successfully, it returns the address of the memory where the `device` structure was allocated. If the memory allocation request cannot be fulfilled, `kalloc` returns the value zero (0).

9.2.2 Returning Previously Allocated Kernel Virtual Memory

To free a previously allocated section of kernel virtual memory, call the `kfree` interface. This interface is used to free the memory space that was previously allocated by `kalloc` or `kget`. The following code fragment shows a call to `kfree`:

```
   •
   •
   •
struct device *device;
struct device *new_dev;
   •
   •
   •
kfree(new_dev, sizeof (struct device)); 1
```

•
•
•

1 Shows that the kfree interface takes two arguments. The first argument is a pointer to the memory to be freed. In the code fragment, this is the pointer to the device structure whose associated memory was allocated in a previous call to kalloc.

The second argument specifies the size (in bytes) of the memory to free. In the example, this size is expressed through the use of the sizeof operator. In this example, kfree frees the size previously allocated for the device structure.

9.2.3 Performing Nonblocking Allocation of Kernel Virtual Memory

To perform nonblocking allocation of a variable-sized section of kernel virtual memory, call the kget interface. The major difference between kget and kalloc is that kget is intended for use by nonblocking code, such as interrupt service interfaces. The following code fragment shows a call to kget:

```
   •
   •
   •
struct cb_unit *cb;
struct cb_unit *return;
   •
   •
   •
return = (struct cb_unit *) kget(sizeof (struct cb_unit));
1
   •
   •
   •
```

1 Shows that the kget interface takes one argument, which is the number of bytes to allocate. To allocate the number of bytes associated with a cb_unit structure, the code fragment uses the sizeof operator.

The return type of kget is defined as caddr_t. Because the data types of the return values are different, the code fragment performs a type-casting operation that converts the return type to be a pointer to a cb_unit structure.

If this call to kget completes successfully, it returns the address of the memory where the cb_unit structure was allocated. If no memory is available, kget returns the value zero (0).

You call kfree to free kernel virtual memory previously allocated with kget. Make sure that the pointer to the memory to be freed was previously set in a call to kget.

9.3 Data Copying Interfaces

The data copying interfaces allow device drivers to:

- Copy a series of bytes with a specified limit
- Zero a block of memory
- Copy data from user address space to kernel address space
- Copy data from kernel address space to user address space
- Move data between user virtual space and system virtual space

The following sections describe the kernel interfaces that perform these tasks.

9.3.1 Copying a Series of Bytes with a Specified Limit

To copy a series of bytes with a specified limit, call the bcopy interface. The following code fragment shows a call to bcopy:

```
   •
   •
   •
struct tc_slot  tc_slot[TC_IOSLOTS]; 1
   •
   •
   •
char *cp; 2
   •
   •
   •
bcopy(tc_slot[index].modulename, cp, TC_ROMNAMLEN + 1); 3
   •
   •
   •
```

1 Declares an array of tc_slot structures of size TC_IOSLOTS.

2 Declares a pointer to a buffer that stores the bytes of data copied from the first argument.

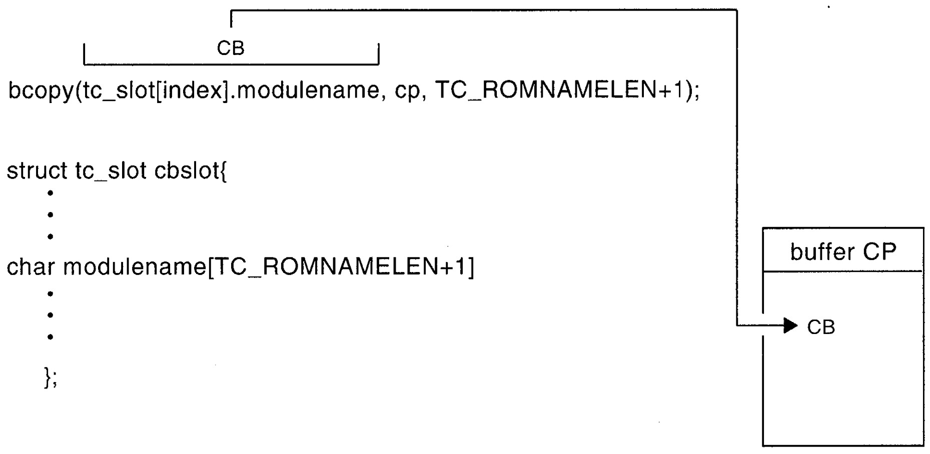
3 Shows that the bcopy interface takes three arguments. The first argument is a pointer to a byte string (array of characters). In the example, this array is the modulename member of the tc_slot structure associated with this bus.

The second argument is a pointer to a buffer that is at least the size specified in the third argument. In the example, this buffer is represented by the pointer to the *cp* variable.

The third argument is the number of bytes to be copied. In the example, the number of bytes is contained in the constant TC_ROMNAMLEN.

Figure 9-6 shows how bcopy copies a series of bytes by using a sample value in the code fragment. As the figure shows, bcopy copies the characters CB to the buffer *cp*. No check is made for null bytes. The copy is nondestructive; that is, the address ranges of the first two arguments can overlap.

Figure 9-6: Results of the bcopy Interface


CB
bcopy(tc_slot[index].modulename, cp, TC_ROMNAMELEN+1);
struct tc_slot cbslot{
char modulename[TC_ROMNAMELEN+1]
};
buffer CP
CB

9.3.2 Zeroing a Block of Memory

To zero a block of memory, call the bzero or blkclr interface. The following code fragment shows a call to bzero. (The blkclr interface has the same arguments.)

```
  •
  •
  •
struct bus *new_bus;
  •
  •
  •
bzero(new_bus, sizeof(struct bus)); 1
  •
  •
  •
```

1 Shows that the bzero interface takes two arguments. The first argument is a pointer to a string whose size is at least the size specified in the second argument. In the example, the first argument is a pointer to a bus structure.

The second argument is the number of bytes to be zeroed. In the example, this size is expressed through the use of the sizeof operator, which returns the size of a bus structure.

In the example, bzero zeroes the number of bytes associated with the size of the bus structure, starting at the address specified by *new_bus*.

The blkclr interface performs the equivalent task.

9.3.3 Copying Data from User Address Space to Kernel Address Space

To copy data from the unprotected user address space to the protected kernel address space, call the copyin interface. The following code fragment shows a call to copyin:

```
   •
   •
   •
register struct buf *bp;
int err;
caddr_t buff_addr;
caddr_t kern_addr;
   •
   •
   •
if (err = copyin(buff_addr,kern_addr,bp->b_resid)) { 1
   •
   •
   •
```

1 Shows that the copyin interface takes three arguments. The first argument specifies the address in user space of the data to be copied. In the example, this address is the user buffer's address.

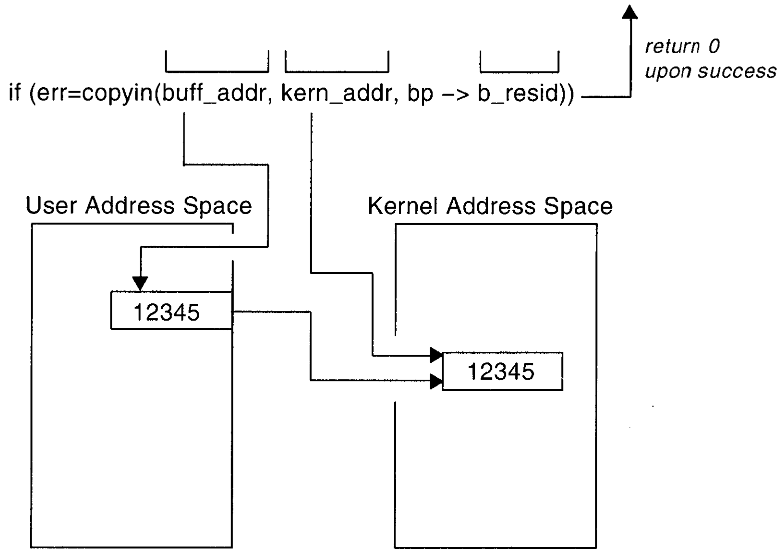
The second argument specifies the address in kernel space to copy the data to. In the example, this address is the address of the kernel buffer.

The third argument specifies the number of bytes to copy. In the example, the number of bytes is contained in the b_resid member of the pointer to the buf structure.

The code fragment sets up a condition statement that performs some tasks based on whether copyin executes successfully. Figure 9-7 shows how copyin copies data from user address space to kernel address space by using sample data.

As the figure shows, copyin copies the data from the unprotected user address space, starting at the address specified by *buff_addr* to the protected kernel address space specified by *kern_addr*. The number of bytes is indicated by the b_resid member. The figure also shows that copyin returns the value zero (0) upon successful completion. If the address in user address space could not be accessed, copyin returns the error EFAULT.

Figure 9-7: Results of the copyin Interface



9.3.4 Copying Data from Kernel Address Space to User Address Space

To copy data from the protected kernel address space to the unprotected user address space, call the copyout interface. The following code fragment shows a call to copyout:

```
   •
   •
   •
register struct buf *bp;
int err;
caddr_t buff_addr;
caddr_t kern_addr;
   •
   •
   •
if (err = copyout(kern_addr,buff_addr,bp->b_resid)) { [1]
   •
   •
   •
```

[1] Shows that the copyout interface takes three arguments. The first argument specifies the address in kernel space of the data to be copied. In the example, this address is the kernel buffer's address, which is stored

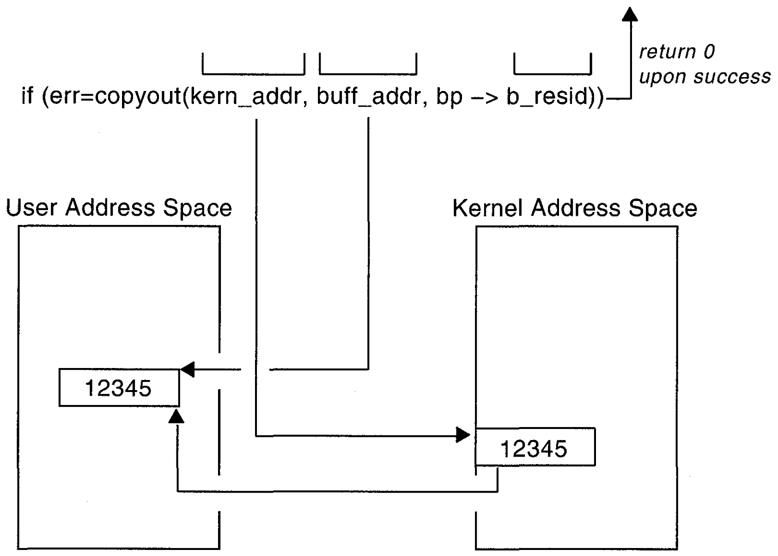
in the *kern_addr* argument.

The second argument specifies the address in user space to copy the data to. In the example, this address is the user buffer's virtual address, which is stored in the *buff_addr* argument.

The third argument specifies the number of bytes to copy. In the example, the number of bytes is contained in the `b_resid` member of the pointer to the `buf` structure.

Figure 9-8 shows the results of `copyout`, based on the code fragment. As the figure shows, `copyout` copies the data from the protected kernel address space, starting at the address specified by *kern_addr* to the unprotected user address space specified by *buff_addr*. The number of bytes is indicated by the `b_resid` member. The figure also shows that `copyout` returns the value zero (0) upon successful completion. If the address in kernel address space could not be accessed or if the number of bytes to copy is invalid, `copyout` returns the error `EFAULT`.

Figure 9-8: Results of the copyout Interface



9.3.5 Moving Data Between User Virtual Space and System Virtual Space

To move data between user virtual and system virtual space, call the `uiomove` interface. The following code fragment shows a call to

```
uiomove:
   •
   •
   •
struct uio *uio;
register struct buf *bp;
int err;
int cnt;
unsigned tmp;
   •
   •
   •
err = uiomove(&tmp,cnt,uio); [1]
   •
   •
   •
```

[1] Shows that the `uiomove` interface takes three arguments. The first argument specifies a pointer to the kernel buffer in system virtual space. The second argument specifies the number of bytes of data to be moved. In this example, the number of bytes to be moved is stored in the *`cnt`* variable. The third argument specifies a pointer to a `uio` structure. This structure describes the current position within a logical user buffer in user virtual space. Section 10.11.1 shows how you use `uiomove` with the `/dev/cb` device driver.

9.4 Hardware-Related Interfaces

The hardware-related interfaces allow device drivers to perform the following tasks related to the hardware:

- Check the read accessibility of addressed data
- Delay the calling interface a specified number of microseconds
- Set the interrupt priority mask

The following sections describe the kernel interfaces that perform these tasks.

9.4.1 Checking the Read Accessibility of Addressed Data

To check the read accessibility of addressed data, call the `BADADDR` interface. The following code fragment shows a call to this interface:

```
   •
   •
   •
caddr_t addr1;
   •
   •
   •
typedef volatile struct {
          int csr;
}none_registers;
```

```
   •
   •
   •
register struct none_registers *reg = (struct none_registers *) addr1;
if (BADADDR( (caddr_t) &reg->csr, sizeof(int)) !=0) [1]
{
   •
   •
   •
}
```

[1] Shows that the BADADDR interface takes two arguments. The first argument specifies the address of the device registers or memory. In the example, this address is that of the csr member of the reg pointer. This member maps to the 32-bit control/status register for this none device.

The second argument specifies the length (in bytes) of the data to be checked. Valid values are 1, 2, and 4 on 32-bit machines and 4 and 8 on 64-bit machines. In the example, the length is the value returned by the sizeof operator — the number of bytes needed to contain a value of type int. The reason for this is the csr member is of type int.

Although it is not shown in this example, BADADDR can take an optional third argument. This third argument specifies a pointer to a bus_ctlr_common structure. You cast this argument as a pointer to either a bus or controller structure.

The BADADDR interface generates a call to a machine-dependent interface 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. You call this interface to probe for memory or I/O devices at a specified address during device autoconfiguration.

The BADADDR interface returns the value zero (0) if the data is accessible and nonzero if the data is not accessible.

This line also sets up a condition statement that performs some tasks based on BADADDR determining if the data stored in csr is accessible.

Loadable device drivers cannot call the BADADDR interface because it is usable only in the early stages of system booting. Loadable device drivers are loaded during the multiboot stage. If your driver is both loadable and static, you can declare a variable and use it to control any differences in the tasks performed by the loadable and static drivers. Thus, the static driver can still call BADADDR.

9.4.2 Delaying the Calling Interface a Specified Number of Microseconds

To delay the calling interface a specified number of microseconds, call the

DELAY interface. The following code fragment shows a call to this interface:

```
    •
    •
    •
DELAY(10000) [1]
    •
    •
    •
```

[1] Shows that the DELAY interface takes one argument: the number of microseconds for the calling process to spin.

The DELAY interface delays the calling interface a specified number of microseconds. DELAY spins, waiting for the specified number of microseconds to pass before continuing execution. In the example, there is a 10000-microsecond (10 millisecond) delay. 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 interface results in an unpredictable delay. For any delay value, the actual delay may vary by plus or minus one clock tick.

Using the DELAY interface is discouraged because the processor will be consumed for the specified time interval and therefore is unavailable to service other processes. In cases where device drivers need timing mechanisms, you should use the sleep and timeout interfaces instead of the DELAY interface. The most common usage of the DELAY interface is in the system boot path. Using DELAY in the boot path is often acceptable because there are no other processes in contention for the processor.

9.4.3 Setting the Interrupt Priority Mask

To set the interrupt priority mask to a specified level, call one of the spl interfaces. Table 9-1 summarizes the uses for the different spl interfaces.

Table 9-1: Uses for spl Interfaces

spl Interface	Meaning
getspl	Gets the spl value
splbio	Masks all disk and tape controller interrupts
splclock	Masks all hardware clock interrupts
spldevhigh	Masks all device and software interrupts
splextreme	Blocks against all but halt interrupts

Table 9-1: (continued)

spl Interface	Meaning
`splhigh`	Mask all interrupts except for realtime devices, machine checks, and halt interrupts
`splimp`	Masks all Ethernet hardware interrupts
`splnet`	Masks all network software interrupts
`splnone`	Unmasks (enables) all interrupts
`splsched`	Masks all scheduling interrupts (usually hardware clock)
`splsoftclock`	Masks all software clock interrupts
`spltty`	Masks all tty (terminal device) interrupts
`splvm`	Masks all virtual memory clock interrupts
`splx`	Resets the CPU priority to the level specified by the argument

The `spl` interfaces set the CPU priority to various interrupt levels. The current CPU priority level determines which types of interrupts are masked (disabled) and which are unmasked (enabled). Historically, seven levels of interrupts were supported, with eight different `spl` interfaces to handle the possible cases. For example, calling `spl0` would unmask all interrupts and calling `spl7` would mask all interrupts. Calling an `spl` interface between 0 and 7 would mask out all interrupts at that level and at all lower levels.

Specific interrupt levels were assigned for different device types. For example, before handling a given interrupt, a device driver would set the CPU priority level to mask all other interrupts of the same level or lower. This setting meant that the device driver could be interrupted only by interrupt requests from devices of a higher priority.

DEC OSF/1 supports the naming of `spl` interfaces to indicate the associated device types. Named `spl` interfaces make it easier to determine which interface should be used to set the priority level for a given device type.

The following code fragment illustrates the use of `spl` interfaces as part of a disk strategy interface:

```
   •
   •
   •
int s;
   •
   •
   •
s = splbio(); [1]
   •
   •
   •
```

```
[Code to deal with data that can be modified by the disk interrupt
code]
splx(s); 2
   •
   •
   •
```

1 Calls the `splbio` interface to mask (disable) all disk interrupts. This interface does not take an argument.

2 Calls the `splx` interface to reset the CPU priority to the level specified by the *s* argument. Note that the one argument associated with `splx` is a CPU priority level, which in the example is the value returned by `splbio`. (The `splx` interface is the only one of the `spl` interfaces that takes an argument.) Upon successful completion, each `spl` interface returns an integer value that represents the CPU priority level that existed before it was changed by a call to the specified `spl` interface.

The binding of any `spl` interface with a specific CPU priority level is highly machine dependent. With the exceptions of the `splhigh` and `splnone` interfaces, knowledge of the explicit bindings is not required to create new device drivers. You always use `splhigh` to mask (disable) all interrupts and `splnone` to unmask (enable) all interrupts.

9.5 Kernel-Related Interfaces

The kernel-related interfaces allow device drivers to:

- Cause a system crash
- Print text to the console and error logger
- Put a calling process to sleep
- Wake up a sleeping process
- Initialize a callout queue element
- Remove the scheduled interface from the callout queues

The following sections describe the kernel interfaces that perform these tasks.

9.5.1 Causing a System Crash

To cause a system crash, call the `panic` interface. The following code fragment shows a call to this interface:

```
   •
   •
   •
panic("vba: no adapter error vector"); 1
   •
   •
   •
```

1 Shows that `panic` takes one argument. This argument specifies the message you want the `panic` interface to display on the console terminal.

The `panic` interface causes a system crash, usually because of fatal errors. It sends to the console terminal and error logger the specified message and, possibly, other system-dependent information (for example, register dumps). It also causes a crash dump to be generated. After displaying the message, `panic` reboots the system if the console envrionment variables are set appropriately. If these console environment variables are not set correctly, the system sits at the halt prompt after a system panic.

9.5.2 Printing Text to the Console and Error Logger

To print text to the console terminal and the error logger, call the `printf` interface. The kernel `printf` interface is a scaled-down version of the C library `printf` interface. The `printf` interface prints diagnostic information directly on the console terminal and writes ASCII text to the error logger. Because `printf` is not interrupt driven, all system activities are suspended when you call it. Only a limited number of characters (currently 128) can be sent to the console display during each call to any section of a driver. The reason is that the characters are buffered until the driver returns to the kernel, at which time they are actually sent to the console display. If more than 128 characters are sent to the console display, the storage pointer may wrap around, discarding all previous characters; or it may discard all characters following the first 128.

If you need to see the results on the console terminal, limit the message size to the maximum of 128 whenever you send a message from within the driver. However, `printf` also stores the messages in an error log file. You can view the text of this error log file by using the `uerf` command. See the reference (man) page for this command. The messages are easier to read if you use `uerf` with the `-o terse` option. The following code fragment shows a call to this interface:

```
   •
   •
   •
#ifdef CB_DEBUG
printf("CBprobe @ %8x, vbaddr = %8x, ctlr = %8x\n",cbprobe,vbaddr,ctlr); 1
#endif /*CB_DEBUG*/
   •
   •
   •
```

1 Shows a typical use for the `printf` interface in the debugging of device drivers. The example shows that `printf` takes two arguments. The

first argument 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, %o, or %x.

The second argument specifies the argument list. In this example, the argument list consists of the arguments *cbprobe*, *vbaddr*, and *ctlr*.

The DEC OSF/1 operating system also supports the `uprintf` interface. The `uprintf` interface prints to the current user's terminal. This interface should never be called by interrupt interfaces. It does not perform any space checking, so you should not use this interface to print verbose messages. The `uprintf` interface does not log messages to the error logger.

9.5.3 Putting a Calling Process to Sleep

To put a calling process to sleep, call the `sleep` interface. The `sleep` and `wakeup` pair of interfaces block and then wake up a process. Generally, device drivers call these interfaces to wait for the transfer to complete an interrupt from the device. That is, the `write` interface of the device driver sleeps on the address of a known location, and the device's interrupt service interface wakes the process when the device interrupts. It is the responsibility of the wakened process to check if the condition for which it was sleeping has been removed.

The following code fragment shows a call to this interface:

```
    •
    •
    •
sleep(&ctlr->bus_name, PCATCH); [1]
    •
    •
    •
```

[1] Shows that the `sleep` interface takes two arguments. The first argument specifies a unique address associated with the calling thread to be put to sleep. In this example, the `sleep` interface puts the calling process to sleep on the address of the bus that this controller is connected to.

The second argument specifies whether the sleep request is interruptible. Setting this argument to the `PCATCH` flag causes the process to sleep in an interruptible state. Not setting the `PCATCH` flag causes the process to sleep in an uninterruptible state. The example sets the `PCATCH` flag to indicate that the sleep request is interruptible.

9.5.4 Waking Up a Sleeping Process

To wake up all processes sleeping on a specified address, call the `wakeup` interface. The following code fragment shows a call to this interface:

```
    •
    •
    •
wakeup(&ctlr->bus_name); 1
    •
    •
    •
```

1 Shows that the `wakeup` interface takes one argument. This argument specifies the address on which the wakeup is to be issued. In the example, this address is that of the bus name associated with the bus this controller is connected to. This address was specified in a previous call to the `sleep` interface. All processes sleeping on this address are wakened.

9.5.5 Initializing a Callout Queue Element

To initialize a callout queue element, call the `timeout` interface. The following code fragment shows a call to this interface:

```
    •
    •
    •
#define CBIncSec  1
    •
    •
    •
cb = &cb_unit[unit];
    •
    •
    •
timeout(cbincled, (caddr_t)cb, CBIncSec*hz); 1
    •
    •
    •
```

1 Shows that the `timeout` interface takes three arguments. The first argument specifies a pointer to the interface to be called. In the example, `timeout` will call the `cbincled` interface on the interrupt stack (not in processor context) as dispatched from the `softclock` interface.

The second argument specifies a single argument to be passed to the called interface. In the example, this argument is the pointer to the `CB` device's `cb_unit` data structure. This argument is passed to the `cbincled` interface. Because the data types of the arguments are different, the code fragment performs a type-casting operation that converts the argument type to be of type `caddr_t`.

The third argument specifies the amount of time to delay before calling the specified interface. Time is expressed as time (in seconds) * hz. In the example, the constant CBIncSec is used with the *hz* global variable to determine the amount of time before timeout calls cbincled. The global variable hz contains the number of clock ticks per second. This variable is a second's worth of clock ticks. The example illustrates a 1-second delay.

9.5.6 Removing the Scheduled Interface from the Callout Queues

To remove the scheduled interfaces from the callout queues, call the untimeout interfaces. The following code fragment shows a call to this interface:

```
   •
   •
   •
untimeout(cbincled, (caddr_t)cb); [1]
   •
   •
   •
```

[1] Shows that the untimeout interface takes two arguments. The first argument specifies a pointer to the interface to be removed from the callout queues. In the example, untimeout removes the cbincled interface from the callout queues. This interface was placed on the callout queue in a previous call to the timeout interface.

The second argument specifies a single argument to be passed to the called interface. In the example, this argument is the pointer to the CB device's cb_unit data structure. It matches the parameter that was passed in a previous call to timeout. Because the data types of the arguments are different, the code fragment performs a type-casting operation that converts the argument type to be of type caddr_t.

The argument is used to uniquely identify which timeout to remove. This is useful if more than one process has called timeout with the same interface argument.

9.6 Loadable Driver Interfaces

The loadable driver interfaces allow loadable drivers to perform a variety of tasks specific to loadable drivers. The most commonly performed loadable driver tasks are to:

- Add or delete entry points in the bdevsw table
- Add or delete entry points in the cdevsw table

- Add or delete entry points to the bdevsw and cdevsw tables
- Register and deregister a driver's interrupt service interface
- Enable and disable a driver's interrupt service interface
- Merge the configuration data
- Configure and unconfigure the specified controller

9.6.1 Adding or Deleting Entry Points in the bdevsw Table

To dynamically add entry points to the bdevsw (block device switch) table for loadable drivers, call the bdevsw_add interface. The following code fragment shows a call to bdevsw_add:

```
   •
   •
   •
struct bdevsw dkip_bdevsw_entry = {
         dkipopen,
         nulldev,
         dkipstrategy,
         dkipdump,
         dkipsize,
         0,
         nodev,
         DEV_FUNNEL_NULL
};
   •
   •
   •
dev_t   dkip_devno;
   •
   •
   •
bdevno = bdevsw_add(bdevno,&dkip_bdevsw_entry); [1]
   •
   •
   •
dkip_devno = bdevno; [2]
   •
   •
   •
```

[1] Shows that bdevsw_add takes two arguments. The first argument specifies the device number to use for the bdevsw device switch table entry (slot). In this example, the device number is stored in the *bdevno* variable. The device number is usually obtained in a previous call to the makedev interface.

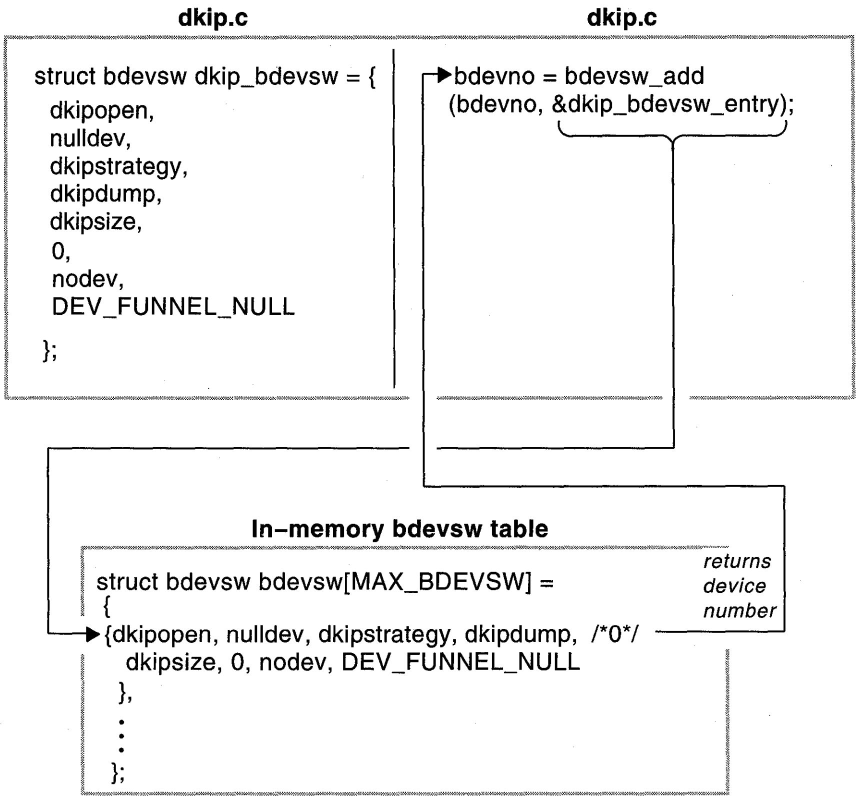
The second argument specifies the block device switch structure that contains the block device driver's entry points. In this example, the dkip_bdevsw_entry structure contains the device drive entry points for a dkip driver.

Figure 9-9 shows the results of bdevsw_add, based on the code fragment. This code would appear in the driver source, dkip.c. As the figure shows, bdevsw_add identifies as zero the device number for the dkip device and adds the driver entry points contained in the dkip_bdevsw_entry structure. The bdevsw_add interface adds the driver's entry points into the in-memory resident bdevsw table. It does not change the /usr/sys/io/common/conf.c file, which is used to build the kernel. Thus, the driver's entry points are dynamically added to the bdevsw table for loadable drivers.

Upon successful completion, bdevsw_add returns the device number associated with the entry in the bdevsw table. In the example, the device number is the value zero (0).

2 Stores the table entry slot for this device in the *dkip_devno* variable for use later by the bdevsw_del interface.

Figure 9-9: Results of the bdevsw_add Interface


dkip.c
dkip.c
struct bdevsw dkip_bdevsw = {
dkipopen,
nulldev,
dkipstrategy,
dkipdump,
dkipsize,
0,
nodev,
DEV_FUNNEL_NULL
};
bdevno = bdevsw_add
(bdevno, &dkip_bdevsw_entry);
In–memory bdevsw table
struct bdevsw bdevsw[MAX_BDEVSW] =
{
{dkipopen, nulldev, dkipstrategy, dkipdump, /*0*/
dkipsize, 0, nodev, DEV_FUNNEL_NULL
},
};
returns
device
number

To dynamically delete entry points from the `bdevsw` (block device switch) table, call the `bdevsw_del` interface. The following code fragment shows a call to `bdevsw_del`:

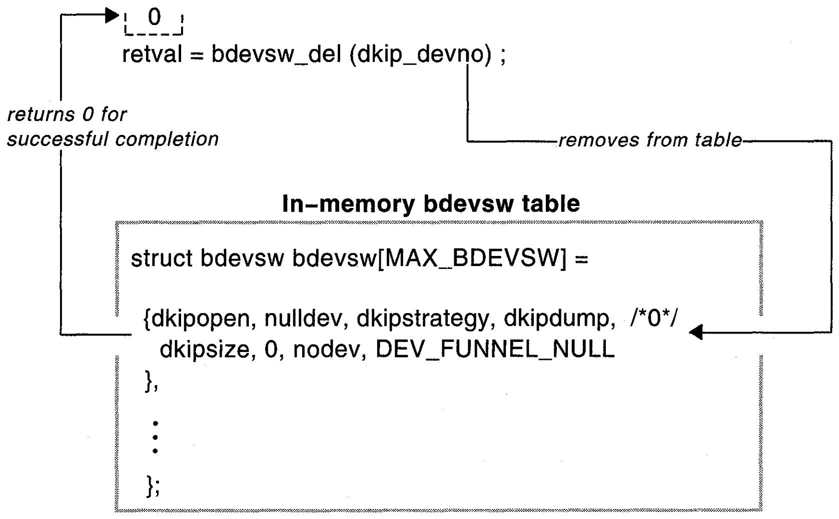
```
   •
   •
   •
int     retval;
   •
   •
   •
retval = bdevsw_del(dkip_devno); [1]
   •
   •
   •
```

[1] Shows that `bdevsw_del` takes one argument: the device number specified in a previous call to `bdevsw_add`. This number was stored in the *`dkip_devno`* variable.

Figure 9-10 shows the results of bdevsw_del, based on the code fragment. As the figure shows, bdevsw_del identifies as zero the device number for the dkip device and deletes the driver entry points from the bdevsw table. The bdevsw_del interface deletes the driver's entry points from the in-memory resident bdevsw table. It does not change the /usr/sys/io/common/conf.c file, which is used to build the kernel. Thus, the driver's entry points are dynamically deleted from the bdevsw table for loadable drivers.

Upon successful completion, bdevsw_del returns the value zero (0).

Figure 9-10: Results of the bdevsw_del Interface


0
retval = bdevsw_del (dkip_devno) ;
returns 0 for successful completion
removes from table
In-memory bdevsw table
struct bdevsw bdevsw[MAX_BDEVSW] =
{dkipopen, nulldev, dkipstrategy, dkipdump, /*0*/
dkipsize, 0, nodev, DEV_FUNNEL_NULL
},
};

9.6.2 Adding or Deleting Entry Points in the cdevsw Table

To dynamically add entry points to the cdevsw (character device switch) table for loadable drivers, call the cdevsw_add interface. The following code fragment shows a call to cdevsw_add:

```
  •
  •
  •
struct cdevsw cb_cdevsw_entry = {
  cbopen,
  cbclose,
  cbread,
  cbwrite,
  cbioctl,
  nodev,
    nodev,
    0,
    nodev,
    0,
    0
};
    •
    •
    •
dev_t cb_cdevno;
    •
    •
    •
cdevno = cdevsw_add(cdevno,&cb_cdevsw_entry); [1]
    •
    •
    •
cb_devno = cdevno; [2]
```

[1] Shows that `cdevsw_add` takes two arguments. The first argument specifies the device number to use for the `cdevsw` device switch table entry (slot). In this example, the device number is stored in the *cdevno* variable. The device number is usually obtained in a previous call to the `makedev` interface.

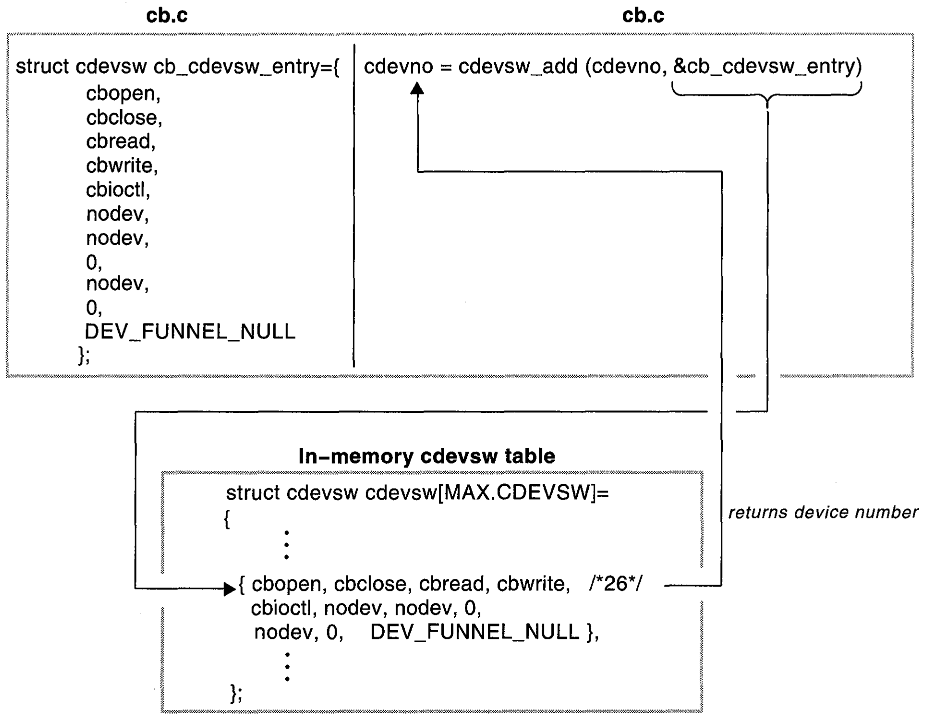
The second argument specifies the character device switch structure that contains the character device driver's entry points. In this example, the `cb_cdevsw_entry` structure contains the device drive entry points for the `/dev/cb` driver.

Figure 9-11 shows the results of `cdevsw_add`, based on the code fragment. This code would appear in the driver source, `cb.c`. As the figure shows, `cdevsw_add` identifies as 26 the device number for the `CB` device and adds the driver entry points contained in the `cb_cdevsw_entry` structure. The `cdevsw_add` interface adds the driver's entry points into the in-memory resident `cdevsw` table. It does not change the `/usr/sys/io/common/conf.c` file, which is used to build the kernel. Thus, the driver's entry points are dynamically added to the `cdevsw` table for loadable drivers.

Upon successful completion, `cdevsw_add` returns the device number associated with the entry in the `cdevsw` table. In the example, the device number is the value 26.

[2] Stores the table entry slot for this `CB` device in the *cb_cdevno* variable for use later by the `cdevsw_del` interface.

Figure 9-11: Results of the cdevsw_add Interface



cb.c
cb.c
struct cdevsw cb_cdevsw_entry={
cbopen,
cbclose,
cbread,
cbwrite,
cbioctl,
nodev,
nodev,
0,
nodev,
0,
DEV_FUNNEL_NULL
};
cdevno = cdevsw_add (cdevno, &cb_cdevsw_entry)
In-memory cdevsw table
struct cdevsw cdevsw[MAX.CDEVSW]=
{
{ cbopen, cbclose, cbread, cbwrite, /*26*/
cbioctl, nodev, nodev, 0,
nodev, 0, DEV_FUNNEL_NULL },
};
returns device number

To dynamically delete entry points from the `cdevsw` (character device switch) table, call the `cdevsw_del` interface. The following code fragment shows a call to `cdevsw_del`:

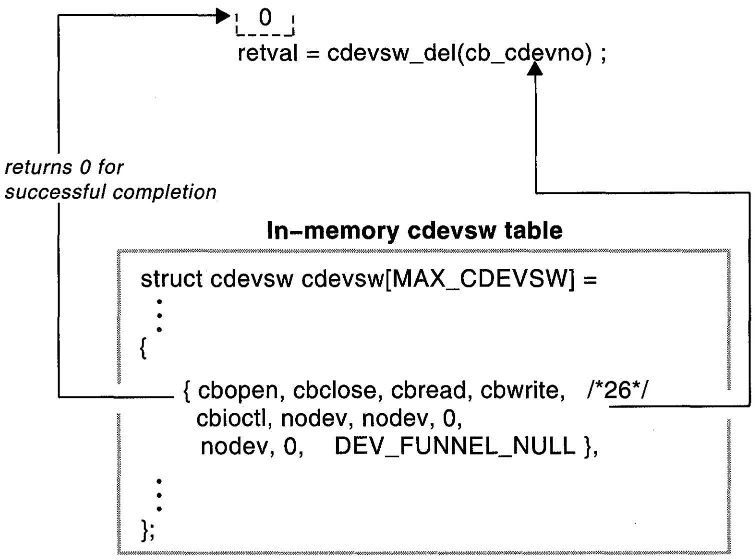
```
•
•
•
int     retval;
•
•
•
retval = cdevsw_del(cb_cdevno); [1]
•
•
•
```

[1] Shows that `cdevsw_del` takes one argument: the device number specified in a previous call to `cdevsw_add`. This number was stored in the *cb_cdevno* variable.

Figure 9-12 shows the results of `cdevsw_del`, based on the code fragment. As the figure shows, cdevsw_del identifies as 26 the device number for the CB device and deletes the driver entry points from the cdevsw table. The cdevsw_del interface deletes the driver's entry points from the in-memory resident cdevsw table. It does not change the /usr/sys/io/common/conf.c file, which is used to build the kernel. Thus, the driver's entry points are dynamically deleted from the cdevsw table for loadable drivers.

Upon successful completion, cdevsw_add returns the value zero (0).

Figure 9-12: Results of the cdevsw_del Interface



```
struct cdevsw cdevsw[MAX_CDEVSW] =
 .
 .
 .
{
    { cbopen, cbclose, cbread, cbwrite,   /*26*/
      cbioctl, nodev, nodev, 0,
      nodev, 0,    DEV_FUNNEL_NULL },
 .
 .
 .
};
```

9.6.3 Registering and Enabling a Driver's Interrupt Service Interface

An interrupt service interface is a device driver routine that handles hardware interrupts. Driver writers implementing static-only device drivers specify a device driver's interrupt service interface in the system configuration file if they are following the traditional device driver configuration model and in the config.file file fragment if they are following the third-party device driver configuration model. Chapter 11 describes both models.

Loadable device drivers, unlike static device drivers, must call a number of kernel interfaces to register, deregister, enable, and disable a device driver's interrupt service interface. This section discusses how to register and enable a driver's interrupt service interface. Section 9.6.4 discusses how to deregister and disable a driver's interrupt service interface.

To register a device driver's interrupt service interface, call the `handler_add` interface. To enable a device driver's interrupt service interface, call the `handler_enable` interface. The `handler_add` and `handler_enable` interfaces are usually called in the Autoconfiguration Support Section of the device driver. The following code fragment and discussion focus on the data structures and arguments used in the calls to these interfaces. Section 10.8.1 discusses in more detail how these interfaces are implemented for the `/dev/cb` device driver.

```
   •
   •
   •
ihandler_id_t cb_id_t[NCB]; 1
   •
   •
   •
ihandler_t handler; 2
struct tc_intr_info info; 3
int unit = ctlr->ctlr_num; 4
   •
   •
   •
info.intr = cbintr; 5
   •
   •
   •
handler.ih_bus_info = (char *)&info; 6
   •
   •
   •
cb_id_t[unit] = handler_add(&handler); 7
   •
   •
   •
if (handler_enable(cb_id_t[unit]) != 0) 8
   •
   •
   •
```

1 Declares an array of IDs that are unique numbers for identifying interrupt service interfaces to be acted on by subsequent calls to `handler_enable`, `handler_disable`, and `handler_del`. This array is filled with the return values from `handler_add`. These return values are opaque keys of type `ihandler_id_t`.

2 Declares an `ihandler_t` data structure called `handler` to contain information associated with the `/dev/cb` device driver interrupt handling. Section 7.8 describes the `ihandler_t` structure.

3 Declares a `tc_intr_info` data structure called `info`. Loadable drivers operating on the TURBOchannel bus use this interrupt handler structure. Section 7.9 describes the `tc_intr_info` structure.

4 Declares a *unit* variable and initializes it to the controller number.

5 Sets the intr member of the info data structure to the pointer to the driver's interrupt interface, cbintr.

6 Sets the ih_bus_info member of the handler data structure to the address of the bus-specific information structure, info.

7 Shows that the handler_add interface takes one argument. This argument specifies a pointer to an ihandler_t data structure. In this example, the address of the declared ihandler_t structure is passed.

The handler_add interface registers a device driver's interrupt service interface and its associated ihandler_t data structure to the bus-specific interrupt-dispatching algorithm. The ih_bus member of the ihandler_t structure specifies the parent bus structure for the bus controlling the driver being loaded. For controller devices, handler_add sets ih_bus to the address of the bus structure for the bus the controller resides on.

The handler_add interface returns an opaque ihandler_id_t key, which is a unique number used to identify the interrupt service interfaces to be acted on by subsequent calls to handler_del, handler_disable, and handler_enable. To implement this ihandler_id_t key, each call to handler_add causes the handler_key data structure to be allocated.

In the example, handler_add returns this opaque ihandler_id_t key to the array of IDs.

8 Shows that the handler_enable interface takes one argument. This argument specifies a pointer to the interrupt service interface's entry in the interrupt table. In the example, handler_enable is passed the IDs that were returned by handler_add.

The handler_enable interface marks that interrupts are enabled and can be dispatched to the driver's interrupt service interfaces, as registered in a previous call to handler_add. The *id* argument passed to handler_enable is used to call a bus-specific adp_handler_enable interface to perform the bus-specific tasks needed to enable the interrupt service interfaces.

Upon successful completion, handler_enable returns the value zero (0). Otherwise, it returns the value –1.

9.6.4 Deregistering and Disabling a Driver's Interrupt Service Interface

The deregistration of interrupt handlers consists of two calls. The first is a call to the handler_disable interface to disable any further interrupts.

The second call is to the handler_del interface to remove the interrupt service interfaces. The following code fragments show calls to these interfaces:

```
     •
     •
     •
if (handler_disable(cb_id_t[unit]) != 0) { [1]
     •
     •
     •
if (handler_del(cb_id_t[unit]) != 0) { [2]
     •
     •
     •
```

[1] Shows that the handler_disable interface takes one argument. This argument specifies a pointer to the interrupt service interface's entry in the interrupt table. In the example, handler_disable is passed the IDs that were returned by handler_add.

The handler_disable interface makes the driver's previously registered interrupt service interfaces unavailable to the system. You must call handler_disable prior to calling handler_del. The handler_disable interface uses the *id* argument to call a bus-specific adp_handler_disable interface to perform the bus-specific tasks needed to disable the interrupt service interfaces.

Upon successful completion, handler_disable returns the value zero (0). Otherwise, it returns the value –1.

[2] Shows that the handler_del interface takes the same argument as handler_disable. The handler_del interface uses the *id* argument to call a bus-specific adp_handler_del interface to remove the driver's interrupt service interface. Deregistration of an interrupt interface can consist of replacing it with the stray interface to indicate that interrupts are no longer expected from this device. The stray interface is a generic interface used as the interrupt handler when there is no corresponding interrupt service interface.

The handler_del interface deregisters a device driver's interrupt service interface from the bus-specific interrupt-dispatching algorithm. In addition, the interface unlinks the handler_key structure associated with the interrupt interface. Prior to deleting the interrupt interface, the device driver should have disabled it by calling handler_disable. If the interrupt interface was not disabled, handler_del returns an error.

Upon successful completion, handler_del returns the value zero (0). Otherwise, it returns the value –1.

9.6.5 Merging the Configuration Data

To merge the configuration data, call the `ldbl_stanza_resolver` interface. This interface is generally called in the section of the driver that implements the tasks associated with the configurable (loadable) driver. Section 10.9.2 discusses in more detail how this interface is implemented for the `/dev/cb` device driver: The following code fragment shows a call to this interface:

```
   •
   •
   •
#define CB_BUSNAME      "tc"
   •
   •
   •
struct  driver cbdriver = { cbprobe, 0, cbattach, 0, 0, 0, 0, 0,
                            "cb", cbinfo, 0, 0, 0, 0, 0,
                            cb_ctlr_unattach, 0 };
   •
   •
   •
struct tc_option cb_option_snippet [] =
{
    /*  module          driver  intr_b4 itr_aft         adpt    */
    /*  name            name    probe   attach  type    config  */
    /*  ------          ------  -------- ------- ----   ------  */
    {   "CB       ",    "cb",     0,       1,    'C',    0},
    {   "",             ""      } /* Null terminator in the table */
};
   •
   •
   •
device_config_t *indata;
   •
   •
   •
if (ldbl_stanza_resolver(indata->config_name,
                         CB_BUSNAME, &cbdriver,
                         (caddr_t *)cb_option_snippet) != 0) {
                         return(EINVAL);
} [1]
   •
   •
   •
```

[1] Shows that the `ldbl_stanza_resolver` interface takes four arguments:

- A *`driver_name`* argument

  The first argument specifies the driver name that was entered as the stanza entry in the `stanza.loadable` file fragment. The example passes the `config_name` member of the pointer to the `device_config_t` data structure. This member contains the name of the controlling device driver, which in the example is the `cb` driver. The name `cb` was entered in the `stanza.loadable` file fragment.

- A *parent_bus* argument

  The second argument specifies the name of the parent bus structure. This name is obtained from the config program. The example specifies that this is a TURBOchannel bus by passing the constant CB_BUSNAME. This constant is defined as the characters tc.

- A *driver_struct* argument

  The third argument specifies a pointer to the driver structure for the controlling device driver. The example passes the address of the cbdriver structure, which the code fragment shows was previously initialized in the driver.

- A *bus_param* argument

  The fourth argument specifies a bus-specific parameter. The example passes a snippet table, the bus-specific parameter most commonly passed for the TURBOchannel bus. The example shows that the snippet table is initialized in the driver. The snippet table adheres to the same format found in the tc_option_data.c table for static drivers.

The ldbl_stanza_resolver interface allows device drivers to merge the system configuration data specified in the stanza.loadable file fragment into the hardware topology tree created at static configuration time. This operation results in a kernel memory resident hardware topology tree that consists of both loadable and static drivers. The driver later calls the ldbl_ctlr_configure interface, which accesses this hardware topology tree. For this reason, ldbl_stanza_resolver is called prior to ldbl_ctlr_configure. Part of this operation involves calling a bus-specific configuration resolver interface that searches for devices that can be autoconfigured. The *parent_bus* argument is used to locate these bus-specific configuration resolver interfaces. The config program passes the value in the config_name member of the device_config_t data structure.

The config_resolver interface can return the following values:

ESUCCESS
: The interface successfully merged the configuration data.

ENOMEM
: The system was unable to allocate enough memory to complete the resolver operations.

LDBL_ENOBUS
: The specified parent bus structure does not exist.

9.6.6 Configuring and Unconfiguring the Specified Controller

To configure the specified controller, call the `ldbl_ctlr_configure` interface. This interface is generally called in the section of the driver that implements the tasks associated with the configurable (loadable) driver. The following code fragment shows a call to this interface. Section 10.9.2 discusses in more detail how this interface is implemented for the `/dev/cb` device driver.

```
   •
   •
   •
#define CB_BUSNAME          "tc"
   •
   •
   •
struct  driver cbdriver = { cbprobe, 0, cbattach, 0, 0, 0, 0, 0,
                            "cb", cbinfo, 0, 0, 0, 0, 0,
                            cb_ctlr_unattach, 0 };
   •
   •
   •
device_config_t *indata;
   •
   •
   •
if (ldbl_ctlr_configure(CB_BUSNAME, LDBL_WILDNUM, indata->config_name,
                        &cbdriver, 0)) {
                        return(EINVAL);
} 1
   •
   •
   •
```

1 Shows that the `ldbl_ctlr_configure` interface takes five arguments:

- The first argument specifies the bus name. The example specifies that this is a TURBOchannel bus by passing the constant `CB_BUSNAME`. This constant is defined as the characters `tc`.
- The second argument specifies the bus number. The example passes the wildcard constant `LDBL_WILDNUM`, which indicates that all instances of the controller should be configured. This constant is defined in the file `/usr/sys/include/io/common/devdriver.h`.
- The third argument specifies the name of the controlling device driver. The example passes the `config_name` member of the pointer to the `device_config_t` data structure. This member contains the name of the controlling device driver, which in the example is the `/dev/cb` driver.
- The fourth argument specifies a pointer to the `driver` structure for the controlling device driver. The example passes the address of the `cbdriver` structure, which the code fragment shows was previously

initialized in the driver.

- The fifth argument specifies miscellaneous flags contained in the file `/usr/sys/include/io/common/devdriver_loadable.h`. The example passes the value zero (0) to indicate that no flags are being passed.

If `ldbl_ctlr_configure` returns an unsuccessful status (a nonzero value), the driver aborts the configuration operation by returning `EINVAL`.

Call the `ldbl_ctlr_unconfigure` interface to unconfigure the specified controller. This interface is generally called in the section of the driver that implements the tasks associated with the unconfiguration of the loadable driver. Section 10.9.3 discusses in more detail how this interface is implemented for the `/dev/cb` device driver.

```
   •
   •
   •
#define CB_BUSNAME          "tc"
   •
   •
   •
struct  driver cbdriver = { cbprobe, 0, cbattach, 0, 0, 0, 0, 0,
                            "cb", cbinfo, 0, 0, 0, 0, 0,
                            cb_ctlr_unattach, 0 };
   •
   •
   •
if (ldbl_ctlr_unconfigure(CB_BUSNAME, LDBL_WILDNUM, &cbdriver,
                          LDBL_WILDNAME, LDBL_WILDNUM) != 0) { 1
   •
   •
   •
```

1 Shows that the `ldbl_ctlr_unconfigure` interface takes five arguments:

- The first argument specifies the bus name. The example specifies that this is a TURBOchannel bus by passing the constant `CB_BUSNAME`. This constant is defined as the characters `tc`.
- The second argument specifies the bus number. The example passes the wildcard constant `LDBL_WILDNUM` to indicate that `ldbl_ctlr_unconfigure` will match any controller.
- The third argument specifies a pointer to the `driver` structure for the controlling device driver. The example passes the address of the `cbdriver` structure, which the code fragment shows was previously initialized in the driver.
- The fourth argument specifies the controller name. You can pass the controller name as it was specified in the `stanza.loadable` file fragment, or you can pass the wildcard constant `LDBL_WILDNAME` to indicate that you want the interface to scan all `controller`

structures. The example passes the wildcard constant.

- The fifth argument specifies the controller number. You can pass the controller number as it was specified in the `stanza.loadable` file fragment, or you can pass the wildcard constant `LDBL_WILDNUM` to indicate that you want the interface to scan all `controller` structures. The example passes the wildcard constant.

The call to `ldbl_ctlr_unconfigure` indirectly results in a call to the driver's `cb_ctlr_unattach` interface, which deregisters the driver's interrupt handlers.

9.7 Input/Output (I/O) Handle-Related Interfaces

To provide device driver binary compatibility across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture, DEC OSF/1 represents I/O bus address spaces through an I/O handle. An I/O handle is a data entity that is of type `io_handle_t`. Device drivers use the I/O handle to reference bus address space (either I/O space or memory space). The bus configuration code passes the I/O handle to the device driver's *`xxprobe`* interface during device autoconfiguration. An I/O handle has similar properties as memory-mapped registers on other UNIX-based systems (for example, TURBOchannel base address regions on ULTRIX Mips).

You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle. The following list points out restrictions on the use of the I/O handle:

- You cannot add two I/O handles.
- You cannot pass an I/O handle directly to the `PHYS_TO_KSEG` interface.

Device drivers needing a local bus address can mask out the lower bits of the I/O handle.

One purpose of the I/O handle is to hide CPU-specific architecture idiosyncracies that describe how to access a device's control status registers (CSRs) and how to perform I/O copy operations. For example, rather than performing an I/O copy operation according to a specific CPU architecture, you call an I/O copy interface that uses the I/O handle. The use of the I/O handle guarantees device driver portability across those CPUs on which the I/O copy interface is implemented.

The following categories of kernel interfaces use an I/O handle:

- Control status register (CSR) I/O access interfaces
- I/O copy interfaces

The following sections discuss how the interfaces associated with each category use the I/O handle.

9.7.1 Control Status Register (CSR) Input/Output (I/O) Access Interfaces

As discussed in Section 2.4.1, one of the issues that influences device driver design is whether to read from and write to a device's control status register (CSR) addresses by directly accessing its device registers. To help you write more portable device drivers, Digital provides a number of kernel interfaces that allow you to read from and write to a device's CSR addresses without directly accessing its device registers. Specifically, these interfaces allow you to:

- Read data from a device register
- Write data to a device register

The following sections describe the kernel interfaces that perform these tasks.

9.7.1.1 Reading Data from a Device Register

To read data from a device register located in bus address space, call the `read_io_port` interface. The `read_io_port` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the read operation. Using this interface to read data from a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

The following code fragment shows a call to `read_io_port`:

```
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle; [1]
   •
   •
   •
};
   •
   •
   •
xxprobe(addr, ctlr)
 iohandle_t addr; [2]
 struct controller *ctlr;
{
  register struct xx_softc *sc; [3]
  u_long hw_id; [4]
    •
    •
    •
 sc->iohandle = addr; [5]
    •
    •
    •
xxwrite(dev, uio, flag)
 dev_t dev;
 register struct uio *uio;
 int flag;
{
 hw_id = read_io_port(sc->iohandle, 4, 0); [6]
    •
    •
    •
}
```

[1] Defines a softc structure that contains a member that stores the I/O handle. Typically, you define the softc structure in a `name_data.c` file.

[2] The bus configuration code passes an I/O handle to the example driver's `xxprobe` interface. An I/O handle is a data entity that is of type `io_handle_t`. Assume that this example driver operates on a bus that passes the I/O handle to the probe interface's first argument. The arguments you pass to the probe interface differ according to the bus on which the driver operates. See the bus-specific device driver manual, which describes the probe interface as it applies to the specific bus.

[3] Declares a pointer to a softc structure and calls it `sc`. The example device driver uses the `iohandle` member of the `sc` pointer to store the I/O handle for use in future calls to the CSR I/O access kernel interfaces.

[4] Declares a variable called *hw_id* to store the data returned by `read_io_port`.

[5] Stores the I/O handle passed in by the bus configuration code in the `iohandle` member of the `sc` pointer.

[6] Calls `read_io_port` in the `xxwrite` interface to read a longword (32 bits) from a device register located in the bus I/O address space. The `read_io_port` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates.

  You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle. In the example, the I/O handle passed to read_io_port is the one passed to *xx*probe and stored in the iohandle member of the sc pointer.

- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. The example driver passes the value 4 for the width.
- The third argument specifies flags to indicate special processing requests. Currently, no flags are used. Because there are not flags, the example driver passes the value zero (0).

Upon successful completion, read_io_port returns the requested data from the device register located in the bus address space: a byte (8 bits), a word (16 bits), a longword (32 bits), or a quadword (64 bits). This interface returns data justified to the low-order byte lane. For example, a byte (8 bits) is always returned in byte lane 0 and a word (16 bits) is always returned in byte lanes 0 and 1.

In this example, read_io_port returns a longword.

The previous example shows how a device driver reads data from a device register by directly calling read_io_port. Another way to read data from a device register is to use read_io_port to construct driver-specific macros, as in the following example:

```
   •
   •
   •
#define XX_ID      0xc80 [1]
#define XX_CSR     0xc00
#define XX_BAT     0xc04
#define XX_HIBASE  0xc08
#define XX_CONFIG  0xc0c
#define XX_ID      0xc80
#define XX_CTRL    0xc84
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle;
   •
   •
   •
};
   •
   •
   •
#define XX_READ_D8(a) read_io_port(sc->iohandle | (io_handle_t)a, 1, 0) [2]
#define XX_READ_D16(a) read_io_port(sc->iohandle | (io_handle_t)a, 2, 0)
#define XX_READ_D32(a) read_io_port(sc->iohandle | (io_handle_t)a, 4, 0)
   •
   •
   •
xxprobe(addr,ctlr)
```

```
 iohandle_t addr;
 struct controller *ctlr;
{
  register struct xx_softc *sc;
  unsigned int hw_id;
    •
    •
    •
 sc->iohandle = addr; [3]
    •
    •
    •
 hw_id = XX_READ_D32(XX_ID); [4]
    •
    •
    •
}
```

[1] Defines offsets to the device registers. Typically, you define these device register offesets in a device register header file.

[2] Constructs driver-specific macros by using the `read_io_port` interface. The macros construct the first argument by ORing the I/O handle and a device register offset for some `XX` device. To obtain the correct offset with large offsets like `0xc84`, you may need to perform an addition operation instead of an OR operation.

[3] Calls the `XX_READ_D32` macro to read a longword from the device register associated with the `XX_ID` offset.

9.7.1.2 Writing Data to a Device Register

To write data to a device register located in bus address space, call the `write_io_port` interface. The `write_io_port` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the write operation. Using this interface to write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

The following code fragment shows a call to `write_io_port` from a device driver's *`xx`*`probe` interface to write data to some device register located in the bus I/O address space:

```
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle; [1]
   •
   •
```

```
    •
};
    •
    •
    •
xxprobe(addr, ctlr)
 iohandle_t addr; [2]
 struct controller *ctlr;
{
  register struct xx_softc *sc; [3]
    •
    •
    •
 sc->iohandle = addr; [4]
    •
    •
    •
xxwrite(dev, uio, flag)
 dev_t dev;
 register struct uio *uio;
 int flag;
{
 write_io_port(sc->iohandle, 4, BUS_IO, 0x1); [5]
    •
    •
    •
}
```

[1] Defines a softc structure that contains a member that stores the I/O handle. Typically, you define the softc structure in a `*name*_data.c` file.

[2] The bus configuration code passes an I/O handle to the example driver's `*xx*probe` interface. An I/O handle is a data entity that is of type `io_handle_t`. Assume that this example driver operates on a bus that passes the I/O handle to the probe interface's first argument. The arguments you pass to the probe interface differ according to the bus on which the driver operates. See the bus-specific device driver manual, which describes the probe interface as it applies to the specific bus.

[3] Declares a pointer to a softc structure and calls it `sc`. The example device driver uses the `iohandle` member of the `sc` pointer to store the I/O handle for use in future calls to the CSR I/O access kernel interfaces.

[4] Stores the I/O handle passed in by the bus configuration code in the `iohandle` member of the `sc` pointer.

[5] Calls `write_io_port` in the `*xx*write` interface to write a longword (32 bits) to a device register located in the bus I/O address space. The `write_io_port` interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs.

You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle. In the example, the I/O handle passed to `write_io_port` is the one passed to *xxprobe* and stored in the `iohandle` member of the `sc` pointer.

- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. The example driver passes the value 4 for the width.
- The third argument specifies flags to indicate special processing requests. Currently, no flags are used. Because no flags are used, the example driver passes the value zero (0).
- The fourth argument specifies the data to be written to the specified device register in bus address space. The example driver passes the value `0x1`.

The `write_io_port` interface has no return value.

The previous example shows how a device driver writes data to a device register by directly calling `write_io_port`. Another way to write data to a device register is to use `write_io_port` to construct driver-specific macros, as in the following example:

```
   •
   •
   •
#define XX_ID     0xc80 [1]
#define XX_CSR    0xc00
#define XX_BAT    0xc04
#define XX_HIBASE 0xc08
#define XX_CONFIG 0xc0c
#define XX_ID     0xc80
#define XX_CTRL   0xc84
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle;
   •
   •
   •
};
   •
   •
   •
#define XX_WRITE_D8(a,d) write_io_port(sc->iohandle | (io_handle_t)a, 1, 0, d) [2]
#define XX_WRITE_D16(a) write_io_port(sc->iohandle | (io_handle_t)a, 2, 0, d)
#define XX_WRITE_D32(a) write_io_port(sc->iohandle | (io_handle_t)a, 4, 0, d)
   •
```

```
    •
    •
xxprobe(addr,ctlr)
 iohandle_t addr;
 struct controller *ctlr;
{
  register struct xx_softc *sc;
    •
    •
    •
 sc->iohandle = addr; 3
    •
    •
    •
XX_WRITE_D32(XX_ID, 0); 4
    •
    •
    •
}
```

1 Defines offsets to the device registers. Typically, you define these device register offesets in a device register header file.

2 Constructs driver-specific macros by using the `write_io_port` interface. Note that the macros construct the first argument by ORing the I/O handle and a device register offset for some `XX` device. To obtain the correct offset with large offsets like `0xc84`, you may need to perform an addition operation instead of an OR operation.

3 Calls the `XX_WRITE_D32` macro to write the value zero (0) to the device register associated with the `XX_ID` offset.

9.7.2 Input/Output (I/O) Copy Interfaces

DEC OSF/1 provides several generic interfaces to copy a block of memory to or from I/O space. These generic interfaces map to bus- and machine-specific interfaces that actually perform the copy operation. Using these interfaces to copy a block of memory to or from I/O space makes the device driver more portable across different CPU architectures and differenct CPU types within the same architecture. These generic interfaces allow device drivers to:

- Copy a block of memory from I/O address space to system memory
- Copy a block of byte-contiguous system memory to I/O address space
- Copy a memory block of I/O address space to another memory block of I/O address space

Each of these interfaces is discussed in the following sections.

9.7.2.1 Copying a Block of Memory from I/O Address Space to System Memory

To copy data from bus address space to system memory, call the `io_copyin` interface. The `io_copyin` interface is a generic interface that maps to a machine-specific interface that actually performs the copy from bus address space to system memory. Using `io_copyin` to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The following code fragment shows a call to `io_copyin` to copy the memory block:

```
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle; 1
   •
   •
   •
};
   •
   •
   •
xxprobe(addr, ctlr)
 iohandle_t addr; 2
 struct controller *ctlr;
{
  register struct xx_softc *sc; 3
  int ret_val; 4
   •
   •
   •
 sc->iohandle = addr; 5
   •
   •
   •
xxwrite(dev, uio, flag)
 dev_t dev;
 register struct uio *uio;
 int flag;
{
 char * buf;
 buf = (char *)kalloc(PAGE_SIZE);
 ret_val = io_copyin(sc->iohandle, buf, PAGE_SIZE); 6
   •
   •
   •
}
```

[1] Defines a softc structure that contains a member that stores the I/O handle. Typically, you define the softc structure in a `name_data.c` file.

[2] The bus configuration code passes an I/O handle to the example driver's `xxprobe` interface. An I/O handle is a data entity that is of type `io_handle_t`. Assume that this example driver operates on a bus that passes the I/O handle to the probe interface's first argument. The arguments you pass to the probe interface differ according to the bus on which the driver operates. See the bus-specific device driver manual, which describes the probe interface as it applies to the specific bus.

[3] Declares a pointer to a softc structure and calls it `sc`. The example device driver uses the `iohandle` member of the `sc` pointer to store the I/O handle for use in future calls to the CSR I/O access kernel interfaces.

[4] Declares a variable called `ret_val` to store the value returned by `io_copyin`.

[5] Stores the I/O handle passed in by the bus configuration code in the `iohandle` member of the `sc` pointer.

[6] Calls `io_copyin` in the `xxwrite` interface to copy a block of memory from I/O address space to system memory. The `io_copyin` interface takes three arguments:

– The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). For `io_copyin`, the I/O handle identifies the location in bus address space where the copy originates. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In the example, the I/O handle passed to `io_copyin` is the one passed to `xxprobe` and stored in the `iohandle` member of the `sc` pointer.

– The second argument specifies the kernel virtual address where `io_copyin` copies the data to in system memory.

  The example calls the `kalloc` interface to obtain the kernel virtual address where `io_copyin` copies the data to in system memory.

– The third argument specifies the number of bytes in the data block to be copied. The interface assumes that the buffer associated with the data block is physically contiguous.

  The example uses the `PAGE_SIZE` constant for the number of bytes in the memory block to be copied.

  Upon successful completion, `io_copyin` returns `IOA_OKAY`. It returns the value -1 on failure.

9.7.2.2 Copying a Block of Byte-Contiguous System Memory to I/O Address Space

To copy data from system memory to bus address space, call the `io_copyout` interface. The `io_copyout` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the copy to bus address space. Using `io_copyout` to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The following code fragment shows a call to `io_copyout` to copy the memory block:

```
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle; 1
   •
   •
   •
};
   •
   •
   •
xxprobe(addr, ctlr)
 iohandle_t addr; 2
 struct controller *ctlr;
{
  register struct xx_softc *sc; 3
  int ret_val; 4
   •
   •
   •
 sc->iohandle = addr; 5
   •
   •
   •
xxwrite(dev, uio, flag)
 dev_t dev;
 register struct uio *uio;
 int flag;
{
 char * buf;
 buf = (char *)kalloc(PAGE_SIZE);
 ret_val = io_copyout(buf, sc->iohandle, PAGE_SIZE); 6
   •
   •
   •
}
```

1 Defines a softc structure that contains a member that stores the I/O handle. Typically, you define the softc structure in a *name*`_data.c` file.

2 The bus configuration code passes an I/O handle to the example driver's *xx*`probe` interface. An I/O handle is a data entity that is of type `io_handle_t`. Assume that this example driver operates on a bus that passes the I/O handle to the probe interface's first argument. The arguments you pass to the probe interface differ according to the bus on which the driver operates. See the bus-specific device driver manual, which describes the probe interface as it applies to the specific bus.

3 Declares a pointer to a softc structure and calls it `sc`. The example device driver uses the `iohandle` member of the `sc` pointer to store the I/O handle for use in future calls to the CSR I/O access kernel interfaces.

4 Declares a variable called *ret_val* to store the value returned by `io_copyout`.

5 Stores the I/O handle passed in by the bus configuration code in the `iohandle` member of the `sc` pointer.

6 Calls `io_copyout` in the *xx*`write` interface to copy a block of byte-contiguous system memory to I/O address space. The `io_copyout` interface takes three arguments:

- The first argument specifies the kernel virtual address where the copy originates from in system memory.

  The example calls the `kalloc` interface to obtain the kernel virtual address where the copy originates from in system memory.

- The second argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). For `io_copyout`, the I/O handle identifies the location in bus address space where the copy occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In the example, the I/O handle passed to `io_copyout` is the one passed to *xx*`probe` and stored in the `iohandle` member of the `sc` pointer.

- The third argument specifies the number of bytes in the data block to be copied. The interface assumes that the buffer associated with the data block is physically contiguous.

  The example uses the `PAGE_SIZE` constant for the number of bytes in the memory block to be copied.

  Upon successful completion, `io_copyout` returns `IOA_OKAY`. It returns the value -1 on failure.

9.7.2.3 Copying a Memory Block of I/O Address Space to Another Memory Block of I/O Address Space

To copy data from one location in bus address space to another location in bus address space, call the `io_copyio` interface. The `io_copyio` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the copy of data from one location in bus address space to another location in bus address space. Using `io_copyio` to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The following code fragment shows a call to `io_copyio` to copy the memory block:

```
   •
   •
   •
struct xx_softc {
   •
   •
   •
    iohandle_t iohandle; [1]
   •
   •
   •
};
   •
   •
   •
xxprobe(addr, ctlr)
 iohandle_t addr; [2]
 struct controller *ctlr;
{
  register struct xx_softc *sc; [3]
  int ret_val; [4]
   •
   •
   •
 sc->iohandle = addr; [5]
   •
   •
   •
xxwrite(dev, uio, flag)
 dev_t dev;
 register struct uio *uio;
 int flag;
{
 ret_val = io_copyio(sc->iohandle, sc->iohandle, PAGE_SIZE); [6]
   •
   •
   •
}
```

1 Defines a softc structure that contains a member that stores the I/O handle. Typically, you define the softc structure in a `name_data.c` file.

2 The bus configuration code passes an I/O handle to the example driver's `xxprobe` interface. An I/O handle is a data entity that is of type `io_handle_t`. Assume that this example driver operates on a bus that passes the I/O handle to the probe interface's first argument. The arguments you pass to the probe interface differ according to the bus on which the driver operates. See the bus-specific device driver manual, which describes the probe interface as it applies to the specific bus.

3 Declares a pointer to a softc structure and calls it `sc`. The example device driver uses the `iohandle` member of the `sc` pointer to store the I/O handle for use in future calls to the CSR I/O access kernel interfaces.

4 Declares a variable called `ret_val` to store the value returned by `io_copyio`.

5 Stores the I/O handle passed in by the bus configuration code in the `iohandle` member of the `sc` pointer.

6 Calls `io_copyio` in the `xxwrite` interface to copy a memory block of I/O address space to another memory block of I/O address space. The `io_copyio` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). For `io_copyio`, this I/O handle identifies the location in bus address space where the copy originates. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In the example, the I/O handle passed to `io_copyio` is the one passed to `xxprobe` and stored in the `iohandle` member of the `sc` pointer.

- The second argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). In this case, the I/O handle identifies the location in bus address space where the copy occurs. In the example, the I/O handles passed to `io_copyio` is the one passed to `xxprobe` and stored in the `iohandle` member of the `sc` pointer.

- The third argument specifies the number of bytes in the data block to be copied. The interface assumes that the buffer associated with the data block is physically contiguous. The example uses the `PAGE_SIZE` constant for the number of bytes in the memory block to be copied.

Upon successful completion, `io_copyio` returns `IOA_OKAY`. It returns the value -1 on failure.

9.8 DMA-Related Interfaces

As discussed in Section 2.4.3, one of the issues that influences device driver design is the technique for performing direct memory access (DMA) operations. Whenever possible, you should design device drivers so that they can accommodate DMA devices connected to different buses operating on a variety of Alpha AXP CPUs. The different buses can require different methods for accessing bus I/O addresses and the different Alpha AXP CPUs can have a variety of DMA hardware support features.

To help you overcome the differences in DMA operations across the different buses and Alpha AXP CPUs, DEC OSF/1 provides a package of mapping interfaces. The mapping interfaces provide a generic abstraction to the kernel- and system-level mapping data structures and to the mapping interfaces that actually perform the DMA transfer operation. This work includes acquiring the hardware and software resources needed to map contiguous I/O bus addresses (accesses) into discontiguous system memory addresses (accesses).

Using the mapping interfaces makes device drivers more portable between major releases of the DEC OSF/1 operating system (and different hardware support for I/O buses) because it masks out any future changes in the kernel- and system-level DMA mapping data structures. Specifically, these interfaces allow you to:

- Allocate system resources for DMA data transfers
- Load and set allocated system resources for DMA data transfers
- Unload system resources for DMA data transfers
- Release and deallocate resources for DMA data transfers

The DMA mapping package also provides convenience interfaces that allow you to:

- Return a pointer to the current bus address/byte count pair
- Put a new bus address/byte count pair into the list
- Return a kernel segment (kseg) address of a DMA buffer

The following sections describe the kernel interfaces that perform these tasks and also discuss the DMA handle and the `sg_entry` data structure associated with these DMA interfaces.

9.8.1 DMA Handle

To provide device driver binary compatibility across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture, DEC OSF/1 represents DMA resources through a DMA handle. A DMA handle is a data entity that is of type `dma_handle_t`. This handle provides the information to access bus address/byte count pairs. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

9.8.2 The sg_entry Structure

The `sg_entry` data structure contains two members: `ba` and `bc`. These members represent a bus address/byte count pair for a contiguous block of an I/O buffer mapped onto a controller's bus memory space. The byte count indicates the number of bytes that the address is contiguously valid for on the controller's bus address space. Consider a list entry that has its `ba` member set to `aaaa` and its `bc` member set to `nnnn`. In this case, the device can perform a contiguous DMA data transfer starting at bus address `aaaa` and ending at bus address `aaaa+nnnn-1`.

Table 9-2 lists the members of the `sg_entry` structure along with their associated data types.

Table 9-2: Members of the sg_entry Structure

Member Name	Data Type
`ba`	`bus_addr_t`
`bc`	`u_long`

The `ba` member stores an I/O bus address.

The `bc` member stores the byte count associated with the I/O bus address. This byte count indicates the contiguous addresses that are valid on this bus.

9.8.3 Allocating System Resources for DMA Data Transfers

To allocate resources for DMA data transfers, call the `dma_map_alloc` interface. Using the `dma_map_alloc` interface makes device drivers more portable between DMA hardware mapping implementations across different hardware platforms because it masks out any future changes in the kernel- and system-level DMA mapping data structures.

The following code fragment taken from a /dev/fd device driver shows the four arguments associated with the call to dma_map_alloc. Two of the arguments passed to dma_map_alloc are defined as members of an fdcam_class data structure.

```
   •
   •
   •
#define SECTOR_SIZE 512
   •
   •
   •
struct fdcam_class {
   •
   •
   •
dma_handle_t fc_dma_handle;
   •
   •
   •
struct controller *ctlr;
};
   •
   •
   •
  struct fdcam_class* fcp = &fc_0;
   •
   •
   •
   •
   •
   •
 if (dma_map_alloc(SECTOR_SIZE, fcp->ctlr, &fcp->&fc_dma_handle,
                    DMA_SLEEP) == 0) { 1
   •
   •
   •
```

1 Shows that dma_map_alloc takes four arguments:

- The first argument specifies the maximum size (in bytes) of the data to be transferred during the DMA transfer operation. The kernel uses this size to determine the resources (mapping registers, I/O channels, and other software resources) to allocate.

  In this example, the *byte_count* is the value defined by the SECTOR_SIZE constant.

- The second argument specifies a pointer to the controller structure associated with this controller. The interface uses this pointer to obtain the bus-specific interfaces and data structures that it needs to allocate the necessary mapping resources. In this example, *ctlr_p* is the value stored in the ctlr member. Assume that the /dev/fd driver previously set the ctlr member to the controller structure pointer associated with this device at probe time.

- The third argument specifies a pointer to a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the `ba` and `bc` members of an `sg_entry` structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation. The `dma_map_alloc` interface returns to this variable the address of the DMA handle. The device driver uses this address in a call to `dma_map_load`. The `dma_map_alloc` interface returns to this variable the address of the DMA handle. The device driver uses this address in a call to `dma_map_load`.

  In this example, the DMA handle appears as a member in the `fdcam_class` data structure.

- The fourth argument specifies special conditions that the device driver wants the system to perform.

  In this example, the bit represented by the `DMA_SLEEP` constant is passed. This bit puts the process to sleep if the system cannot allocate the necessary resources to perform a data transfer of size *`byte_count`* at the time the driver called the interface.

The code fragment sets up a condition statement that performs some tasks based on the value returned by `dma_map_alloc`. Upon successful completion, `dma_map_alloc` returns a byte count (in bytes) that indicates the DMA transfer size it can map. It returns the value zero (0) to indicate a failure.

9.8.4 Loading and Setting Allocated System Resources for DMA Data Transfers

To load and set allocated system resources for DMA data transfers, call the `dma_map_load` interface. The `dma_map_load` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the loading and setting of system resources for DMA data transfers. Using this interface in DMA read and write operations makes the device driver more portable across different bus architectures, different CPU architectures, and differenct CPU types within the same CPU architecture.

The following code fragment taken from a `/dev/fd` device driver shows the seven arguments associated with the call to `dma_map_load`. Note that the arguments passed to `dma_map_load` are defined as members of an `fdcam_class` data structure. The `/dev/fd` driver shows an example of fixed preallocated DMA resources.

```
   •
   •
   •
#define SECTOR_SIZE 512
   •
   •
   •
struct fdcam_class {
   •
   •
   •
int rw_count;
unsigned char *rw_buf;
struct proc *rw_proc;
dma_handle_t dma_handle;
   •
   •
   •
struct controller *ctlr;
};
   •
   •
   •
  struct fdcam_class* fcp = fsb->fcp;
   •
   •
   •
  flags = (fcp->rw_op == OP_READ) ? DMA_IN : DMA_OUT;
  if (dma_map_load(fcp->rw_count * SECTOR_SIZE, fcp->rw_buf,
                   fcp->rw_proc, fcp->ctlr, fcp->dma_handle, 0,
                   flags) == 0) { [1]
   •
   •
   •
```

[1] Shows that `dma_map_load` takes seven arguments:

- The first argument specifies the maximum size (in bytes) of the data to be transferred during the DMA transfer operation. The kernel uses this size to determine the resources (mapping registers, I/O channels, and other software resources) to allocate, load, and set. In this example, the size is the result of the `SECTOR_SIZE` and the value stored in `rw_count`. Assume that the `/dev/fd` driver previously set the `rw_count` member to some value.
- The second argument specifies the virtual address where the DMA transfer occurs. The interface uses this address with the pointer to the `proc` structure to obtain the physical addresses of the system memory pages to load into DMA mapping resources. In this example, the virtual address is the value stored in the `rw_buf` member. Assume that the `/dev/fd` driver previously set the `rw_count` member to some value.

- The third argument specifies a pointer to the proc structure associated with the valid context for the virtual address passed in the second argument. The interface uses this pointer to retrieve the pmap that is needed to translate this virtual address to a physical address. If *proc_p* is equal to zero (0), the address is a kernel address. In this example, *proc_p* is the value stored in the rw_proc member. Assume that the /dev/fd driver previously set the rw_proc member to some value.
- The fourth argument specifies a pointer to the controller structure associated with this controller. The dma_map_load interface uses the pointer to get the bus-specific interfaces and data structures that it needs to load and set the necessary mapping resources. In this example, *ctlr_p* is the value stored in the ctlr member. Assume that the /dev/fd driver previously set the ctlr member to the controller structure pointer associated with this device at probe time.
- The fifth argument specifies a pointer to a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the ba and bc members of an sg_entry structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

  Typically, the device driver passes an argument of type dma_handle_t *. In this example, the /dev/fd device driver simply passes the address of the DMA handle that is a member of the fdcam_class structure.
- The sixth argument specifies the maximum-size byte-count value that should be stored in the bc members of the sg_entry structures. In this example, the /dev/fd driver passes the value zero (0).
- The seventh argument specifies special conditions that the device driver wants the system to perform. In this example, *flags* is DMA_IN if rw_op evaluates to TRUE (that is, this is a DMA read operation). The DMA_IN bit indicates that the system should perform a DMA write into core memory. Otherwise, *flags* is DMA_OUT if rw_op evaluates to FALSE (that is, this is a DMA write operation). The DMA_OUT bit indicates that the system should perform a DMA read from main core memory.

The code fragment sets up a condition statement that performs some tasks based on the value returned by dma_map_load. Upon successful completion, dma_map_load returns a byte count (in bytes) that indicates the DMA transfer size it can support. It returns the value zero (0) to indicate a failure.

9.8.5 Unloading System Resources for DMA Data Transfers

To unload the resources that were loaded and set up in a previous call to dma_map_load, call the dma_map_unload interface. The dma_map_unload interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the unloading of system resources associated with DMA data transfers. Using this interface in DMA read and write operations makes the device driver more portable across different bus architectures, different CPU architectures, and differenct CPU types within the same CPU architecture.

The following code fragment taken from a /dev/fd device driver shows the two arguments associated with the call to dma_map_unload. One of the arguments passed to dma_map_unload is defined as a member of an fdcam_class data structure.

```
   •
   •
   •
struct fdcam_class {
   •
   •
   •
char rw_use_dma;
   •
   •
   •
dma_handle_t dma_handle;
   •
   •
   •
};
   •
   •
   •
  struct fdcam_class* fcp = fsb->fcp;
   •
   •
   •
  if (fcp->rw_use_dma) {
      dma_map_unload(0, fcp->dma_handle);
  } [1]
   •
   •
   •
```

[1] Shows that dma_map_unload takes two arguments:

- To cause a deallocation of DMA mapping resources, set this argument to the special condition bit DMA_DEALLOC.

This bit setting is analogous to setting the *dma_handle_p* argument to the value zero (0) for dma_map_load to allocate the DMA mapping resources.

In this example, the /dev/fd driver passes the value zero (0) to indicate that it did not want to deallocate the DMA mapping resources.

– The second argument specifies a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the ba and bc members of an sg_entry structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation. In this example, the /dev/fd device driver simply passes the DMA handle that is a member of the fdcam_class structure.

The code fragment sets up a condition statement that calls dma_map_unload if the rw_use_dma evaluates to a nonzero (true) value. Upon successful completion, dma_map_unload returns the value 1. Otherwise, it returns the value zero (0).

A call to dma_map_unload does not release or deallocate the resources that were allocated in a previous call to dma_map_alloc unless the driver sets the *flags* argument to the DMA_DEALLOC bit.

9.8.6 Releasing and Deallocating Resources for DMA Data Transfers

To release and deallocate the resources that were allocated in a previous call to dma_map_alloc, call the dma_map_dealloc interface. Using the dma_map_dealloc interface makes device drivers more portable between DMA hardware mapping implementations across different hardware platforms because it masks out any future changes in the kernel- and system-level DMA mapping data structures.

The following code fragment taken from a /dev/fd device driver shows the argument associated with the call to dma_map_dealloc. This argument is defined as a member of an fdcam_class data structure.

```
   •
   •
   •
struct fdcam_class {
   •
   •
   •
char rw_use_dma;
   •
   •
   •
dma_handle_t dma_handle;
   •
   •
   •
};
   •
   •
   •
struct fdcam_class* fcp;
   •
   •
   •
  if (fcp->rw_use_dma) {
      dma_map_dealloc(fcp->dma_handle_p);
  } [1]
   •
   •
   •
```

[1] Shows that dma_map_dealloc takes one argument: a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the ba and bc members of an sg_entry structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

In this example, the /dev/fd device driver simply passes the DMA handle that is a member of the fdcam_class structure.

The code fragment sets up a condition statement that calls dma_map_dealloc if the rw_use_dma evaluates to a nonzero (true) value. Upon successful completion, dma_map_dealloc returns the value 1. Otherwise, it returns the value zero (0).

9.8.7 Returning a Pointer to the Current Bus Address/Byte Count Pair

The DMA handle provides device drivers with a tag to the allocated system resources needed to perform a direct memory access (DMA) operation. In particular, the handle provides the information for drivers to access bus address/byte count pairs. The system maintains arrays of sg_entry data structures. Some device drivers may need to traverse the arrays of sg_entry data structures to obtain specific bus address/byte count pairs. The DMA mapping package provides two convenience interfaces that allow you to traverse the discontinuous sets of sg_entry arrays:

- The dma_get_curr_sgentry interface returns a pointer to an sg_entry data structure. A device driver can use this pointer to retrieve the current bus address/byte count pair for the mapping of a block of an in-memory I/O buffer onto the controller's I/O bus.

- The dma_get_next_sgentry interface returns a pointer to an sg_entry data structure. A device driver can use this pointer to retrieve the next valid bus address/byte count pair for the mapping of a block of an in-memory I/O buffer onto the controller's I/O bus.

The following example shows the similarities and differences between the calls to dma_get_curr_sgentry and dma_get_next_sgentry:

```
   •
   •
   •
dma_handle_t dma_handle; [1]
sg_entry_t sgentry_curr; [2]
vm_offset_t address_curr; [3]
long count_curr; [4]
   •
   •
   •
sgentry_curr = dma_get_curr_sgentry(dma_handle) [5]
   •
   •
   •
address_curr = (vm_offset_t)sgentry_curr->ba; [6]
count_curr = sgentry_curr->bc - 1; [7]

dma_handle_t dma_handle; [1]
sg_entry_t sgentry_next; [2]
vm_offset_t address_next; [3]
long count_next; [4]
   •
   •
   •
sgentry_next = dma_get_next_sgentry(dma_handle) [5]
   •
   •
   •
address_next = (vm_offset_t)sgentry_next->ba; [6]
count_next = sgentry_next->bc - 1; [7]
```

[1] Declare a DMA handle called dma_handle. The example passes this DMA handle to the dma_get_curr_sgentry and dma_get_next_sgentry interfaces.

[2] Declare pointers to sg_entry structures called sgentry_curr and sgentry_next. The example uses these pointers to store the sg_entry structure pointers in the array returned by dma_get_curr_sgentry and dma_get_next_sgentry.

[3] Declare variables called *address_curr* and *address_next* to store the bus addresses associated with the sg_entry structure pointers returned by dma_get_curr_sgentry and dma_get_next_sgentry.

[4] Declare variables called *count_curr* and *count_next* to store the byte counts associated with the sg_entry structure pointers returned by dma_get_curr_sgentry and dma_get_next_sgentry.

[5] Call the dma_get_curr_sgentry and dma_get_next_sgentry interfaces passing to them the DMA handle. This argument specifies a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the ba and bc members of an sg_entry structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

[6] Set the *address_curr* and *address_next* variables to the bus addresses associated with the sg_entry structure pointers returned by dma_get_curr_sgentry and dma_get_next_sgentry. In the call to dma_get_curr_sgentry, this bus address is associated with the last sg_entry structure pointer that the system read in the array. In the call to dma_get_next_sgentry, this bus address is associated with the next valid sg_entry structure pointer in the array.

[7] Set the *count_curr* and *count_next* variables to the byte counts associated with the sg_entry structure pointers returned by dma_get_curr_sgentry and dma_get_next_sgentry. In the call to dma_get_curr_sgentry, this byte count is associated with the last sg_entry structure pointer that the system read in the array. In the call to dma_get_next_sgentry, this byte count is associated with the next valid sg_entry structure pointer in the array.

9.8.8 Putting a New Bus Address/Byte Count Pair into the List

The DMA handle provides device drivers with a tag to the allocated system resources needed to perform a direct memory access (DMA) operation. In particular, the handle provides the information for drivers to access bus address/byte count pairs. The system maintains arrays of sg_entry data structures. Some device drivers may need to traverse the arrays of sg_entry data structures to put new bus address/byte count pair values into the ba and bc members of specific sg_entry structures. The DMA mapping package provides two convenience interfaces that allow you to traverse the discontinuous sets of sg_entry arrays:

- The dma_put_curr_sgentry interface puts new bus address/byte count values into the ba and bc members for the last sg_entry structure pointer in the list read by the system. This interface patches an existing bus address/byte count pair due to an unexpected interruption in a DMA transfer.

- The dma_put_prev_sgentry interface puts a new bus address/byte count entry into the existing bus address/byte count pair pointed to by the DMA handle passed in by you. This interface enables device drivers to patch existing bus address/byte count pairs due to an unexpected interruption in a DMA transfer.

The following code fragment shows the similarities and differences between the calls to dma_put_curr_sgentry and dma_put_prev_sgentry:

```
   •
   •
   •
dma_handle_t dma_handle; 1
sg_entry_t sgentry_curr; 2
int ret_curr; 3
   •
   •
   •
Call dma_map_load 4
   •
   •
   •
Call kalloc 5
   •
   •
   •
Call dma_map_load a second time 6
   •
   •
   •
Set the ba & bc members 7
   •
   •
   •
ret_curr = dma_put_curr_sgentry(dma_handle, sgentry_curr) 8
   •
   •
   •
Call dma_map_dealloc 9

dma_handle_t dma_handle; 1
sg_entry_t sgentry_prev; 2
int ret_prev; 3
   •
   •
   •
Call dma_map_load 4
   •
   •
   •
Call kalloc 5
   •
   •
   •
Call dma_map_load a second time 6
   •
   •
```

```
    •
Set the ba & bc members [7]
    •
    •
    •
ret_prev = dma_put_prev_sgentry(dma_handle, sgentry_prev) [8]
    •
    •
    •
Call dma_map_dealloc [9]
```

[1] Declare a DMA handle called `dma_handle`. The example passes this DMA handle to the `dma_put_curr_sgentry` and `dma_put_prev_sgentry` interfaces.

[2] Declare pointers to `sg_entry` structures called `sgentry_curr` and `sgentry_prev`. The example uses these pointers to store the new bus address/byte count pairs passed to `dma_put_curr_sgentry` and `dma_put_prev_sgentry`.

[3] Declare variables to store the values returned by `dma_put_curr_sgentry` and `dma_put_prev_sgentry`.

[4] Call the `dma_map_load` interface to load and set the allocated system resources for DMA data transfers

[5] Call the `kalloc` interface to allocate a temporary storage buffer.

[6] Call `dma_map_load` a second time to load and set the allocated system resources for the temporary storage buffer. The example makes this second call to `dma_map_load` to obtain a valid map of the temporary storage buffer into the system.

[7] Set the `ba` and `bc` members of the respective `sg_entry` structures to the values returned to the `ba` and `bc` members associated with this mapped buffer.

[8] Call the `dma_put_curr_sgentry` and `dma_put_prev_sgentry` interfaces, passing to them the DMA handle and the `sg_entry` structures containing the new bus address/byte count pair values.

[9] Call the `dma_map_dealloc` interface to release and deallocate the resources for DMA data transfers associated with the temporary mapped buffer. The example makes this call after the system transfers the patched bus address/byte count pair. This call is required to keep the hardware mapping resources valid during the DMA data transfer.

9.8.9 Returning a Kernel Segment Address of a DMA Buffer

To return a kernel segment (kseg) address of a DMA buffer, call the `dma_kmap_buffer` interface. The `dma_kmap_buffer` interface takes an *`offset`* variable and returns a kernel segment (kseg) address. The device driver can use this kseg address to copy and save the data at the offset in the buffer. The following code fragment shows a call to dma_kmap_buffer:

```
   •
   •
   •
dma_handle_t dma_handle;
u_long offset;
vm_offset_t kseg_addr;
   •
   •
   •
kseg_addr = dma_kmap_buffer(dma_handle, offset) [1]
```

[1] The dma_kmap_buffer interface takes two arguments:

- The first argument specifies a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the ba and bc members of an sg_entry structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation. The example passes the DMA handle to dma_kmap_buffer.
- The second argument specifies a byte count offset from the virtual address passed as the *virt_addr* argument of the dma_map_load interface. This virtual address specifies the beginning of a process's (or kernel) buffer that a DMA transfer operation is done to/from. A device driver determines the smallest DMA transfer size by calling the dma_min_boundary interface. The *offset* specifies the number of bytes a DMA engine moved. This number is less than the number of bytes loaded by the dma_map_load interface.

Upon successful completion, dma_kmap_buffer returns a kseg address of the byte offset pointed to by the addition of the following two values:

```
virt_addr + offset
```

where:

- *virt_addr* is the virtual address passed to the *virt_addr* argument of dma_map_load
- offset is the offset passed to the *offset* argument of dma_kmap_buffer

The dma_kmap_buffer interface returns the value zero (0) to indicate failure to retrieve the kseg addess.

9.9 Miscellaneous Interfaces

The miscellaneous interfaces allow device drivers to:

- Indicate that I/O is complete
- Get the device major number
- Get the device minor number
- Implement raw I/O

The following sections describe the kernel interfaces that perform these tasks.

9.9.1 Indicating that I/O is Complete

To indicate that I/O is complete, call the `iodone` interface. The following code fragment shows a call to this interface. The code fragment verifies read or write access to the user's buffer before beginning the DMA operation.

```
  •
  •
  •
{
                bp->b_error = EACCES;
                bp->b_flags |= B_ERROR;
                iodone(bp); [1]
                return;
               }
  •
  •
  •
```

[1] Shows that `iodone` takes one argument. This argument specifies a pointer to a `buf` structure. The `iodone` interface indicates that I/O is complete and reschedules the process that initiated the I/O.

9.9.2 Getting the Device Major Number

To get the device major number, call the `major` interface. The following code fragment shows a call to this interface. This code fragment would appear in the section of the driver source that handles the tasks associated with the loadable driver.

```
  •
  •
  •
dev_t cb_devno = NODEV; [1]
  •
  •
  •
cb_devno = cdevno; [2]
  •
  •
  •
device_config_t *outdata; [3]
```

```
     •
     •
     •
                outdata->dc_cmajnum = major(cb_devno); 4
     •
     •
     •
```

1 Initializes the variable `cb_devno`, which stores the number of the device whose associated major number you want to obtain. The constant `NODEV` indicates that no device number has yet been assigned.

2 Sets the `cb_devno` variable to the device number for this device. The device number was obtained in a call to the `cdevsw_add` interface.

3 Declares a pointer to a `device_config_t` data structure.

4 Shows that the `major` interface takes one argument. This argument specifies the number of the device whose associated major device number the `major` interface will obtain. In the example, the device number is stored in the `cb_devno` variable. Upon successful completion, `major` returns the major number portion of the `dev_t` passed as the argument. In this example, the major device number is stored in the `dc_cmajnum` of the pointer to the `device_config_t` data structure.

9.9.3 Getting the Device Minor Number

To get the device minor number, call the `minor` interface. The following code fragment shows a call to this interface:

```
  •
  •
  •
int unit = minor(dev); 1
  •
  •
  •
```

1 Shows that the `minor` interface takes one argument. This argument specifies the number of the device whose associated minor device number the `minor` interface will obtain. Upon successful completion, `minor` returns the minor number portion of the `dev_t` passed as the argument.

9.9.4 Implementing Raw I/O

To implement raw I/O, call the `physio` interface. This interface maps the raw I/O request directly into the user buffer, without using `bcopy`. The memory pages in the user address space are locked while the transfer is processed.

The following code fragment shows a call to this interface:

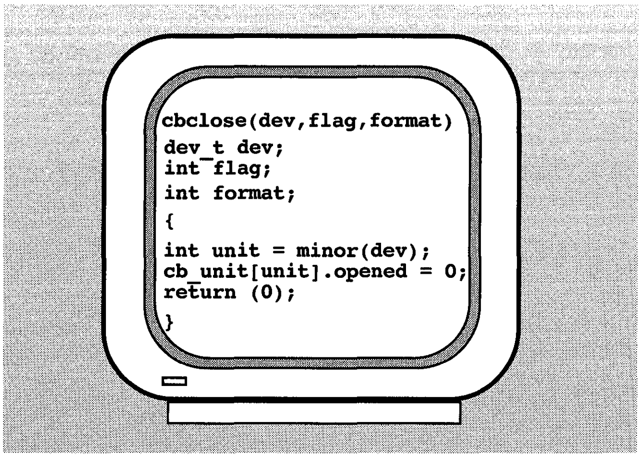
```
   •
   •
   •
return(physio(cbstrategy,cb->cbbuf,dev,B_READ,cbminphys,uio)); [1]
   •
   •
   •
```

[1] Shows that `physio` takes six arguments:

- The first argument specifies the device driver's strategy interface for the device. In this call, `cbstrategy` is the strategy interface for the `/dev/cb` device driver.
- The second argument specifies a pointer to a `buf` structure. This structure contains information such as the binary status flags, the major/minor device numbers, and the address of the associated buffer. This buffer is always a special buffer header owned exclusively by the device for handling I/O requests.

  In this call, the address of the `buf` structure associated with this `CB` device is passed.
- The third argument specifies the device number. In this call, the `CB` device's number is passed.
- The fourth argument specifies the read/write flag. In this call, the binary status flag `B_READ` is passed. This flag is set if the operation is read and cleared if the operation is write.
- The fifth argument specifies a pointer to a `minphys` interface. In this call, the `cbminphys` interface is passed.
- The sixth argument specifies a pointer to a `uio` structure. This structure describes the current position within a logical user buffer in user virtual space.

Part 5 Device Driver Example



```
cbclose(dev,flag,format)
dev_t dev;
int flag;
int format;
{
int unit = minor(dev);
cb_unit[unit].opened = 0;
return (0);
}
```

Writing a Character Device Driver 10

The `/dev/none` device driver described in Chapter 4 had no real device or bus associated with it. All real device drivers are written for some device that operates on a specific bus. This chapter provides you with an opportunity to study a real device driver called `/dev/cb`. The `/dev/cb` device driver provides a simple interface to the TURBOchannel test board. The `/dev/cb` device driver operates on a TURBOchannel bus and implements many of the character device driver interfaces described in Chapter 3. It also implements other sections needed by the TURBOchannel test board.

The chapter begins with an overview of the tasks performed by the `/dev/cb` device driver. Following this overview are sections that describe each piece of the `/dev/cb` device driver. Table 10-1 lists the parts of the `/dev/cb` device driver and the sections of the chapter where each is described.

Table 10-1: Parts of the /dev/cb Device Driver

Tasks	Section
The cbreg.h Header File	Section 10.2
Include Files Section	Section 10.3
Autoconfiguration Support Declarations and Definitions Section	Section 10.4
Loadable Driver Configuration Support Declarations and Definitions Section	Section 10.5
Local Structure and Variable Definitions Section	Section 10.6
Loadable Driver Local Structure and Variable Definitions Section	Section 10.7
Autoconfiguration Support Section	Section 10.8
Loadable Device Driver Section	Section 10.9

Table 10-1: (continued)

Tasks	Section
Open and Close Device Section	Section 10.10
Read and Write Device Section	Section 10.11
Strategy Section	Section 10.12
Start Section	Section 10.13
The ioctl Section	Section 10.14
Increment LED Section	Section 10.15
Interrupt Section	Section 10.16

The source code uses the following convention:

```
extern int hz; [1]
```

[1] Numbers appear after each line or lines of code in the `/dev/cb` device driver example. Following the example, a corresponding number appears that contains an explanation for the associated line or lines. The source code does not contain any inline comments. For those who prefer to read the `/dev/cb` driver source code in its entirety with the inline comments, see Section B.2.

10.1 Overview of the /dev/cb Device Driver

The `/dev/cb` device driver is a character driver that implements a test board on the TURBOchannel bus. The TURBOchannel test board is a minimal implementation of all the TURBOchannel hardware functions: programmed I/O, DMA read, DMA write, and I/O read/write conflict testing. The software view of the board consists of:

- An EPROM address space
- A 32-bit ADDRESS register with bits scrambled for direct use as a TURBOchannel direct memory access (DMA) address
- A 32-bit DATA register used for programmed I/O and as the holding register for DMA
- A 16-bit register used to control four light emitting diodes (LEDs) on the TURBOchannel option card, a 1-bit TEST register, and a 16-bit control status register (CSR)

All registers must be accessed as 32-bit longwords, even when they are not implemented as 32 bits. The CSR contains bits to enable option DMA read testing, conflict signal testing, I/O interrupt testing, and option DMA write testing. The CSR also contains a bit that indicates that one or more of the tests are enabled, 4-byte mask flag bits, and a DMA done bit.

The `/dev/cb` device driver:

- Reads from the data register on the test board to words in system memory
- Writes to the data register on the test board from words in system memory
- Tests the interrupt logic on the test board
- Reads one 32-bit word from the test board address (ROM/register) space into system memory
- Updates, reads, and returns the 32-bit CSR value
- Starts and stops clock-driven incrementing of the four spare LEDs on the board

10.2 The cbreg.h Header File

The `cbreg.h` file is the device register header file for the `/dev/cb` device driver. It contains public declarations and the device register offset definitions for the TURBOchannel test board (the `CB` device). The following declarations are applicable to both the loadable and static versions of the driver:

```
#define CB_REL_LOC 0x00040000 [1]
#define CB_ADR(n) ((io_handle_t)(n + CB_REL_LOC)) [2]
#define CB_SCRAMBLE(x) (((unsigned)x<<3)&~(0x1f))|(((unsigned)x>>29)&0x1f) [3]

#define CB_INTERUPT 0x0e00 [4]
#define CB_CONFLICT 0x0d00 [5]
#define CB_DMA_RD   0x0b00 [6]
#define CB_DMA_WR   0x0700 [7]
#define CB_DMA_DONE 0x0010 [8]

#define CBPIO _IO('v',0) [9]
#define CBDMA _IO('v',1) [10]
#define CBINT _IO('v',2) [11]
#define CBROM _IOWR('v',3,int) [12]
#define CBCSR _IOR('v',4,int)  [13]
#define CBINC _IO('v',5) [14]
#define CBSTP _IO('v',6) [15]

#define CB_ADDER   0x0 [16]
#define CB_DATA    0x4 [17]
#define CB_CSR     0x8 [18]
#define CB_TEST    0xC [19]
```

[1] Defines a constant called `CB_REL_LOC`.

[2] Defines a macro called `CB_ADR` that the `cbattach` interface calls to convert the register offset to the kernel virtual address. The data type specified in the type cast operation is of type `io_handle_t`. Section 10.8.2 shows how `cbattach` calls this macro.

[3] Defines a macro called `CB_SCRAMBLE` that the `cbstrategy` interface calls to discard the low-order 2 bits of the physical address while scrambling the rest of the address. Section 10.12.2.4 shows how `cbstrategy` calls this macro.

[4] Defines a constant called `CB_INTERUPT` that is used to set bits `9`, `10`, and `11` of the CSR. Section 10.14.4 shows how the `cbioctl` interface uses this constant when it performs interrupt testing.

[5] Defines a constant called `CB_CONFLICT` that is used to set bits `9`, `8`, `10`, and `11` of the CSR. This constant is not currently used.

[6] Defines a constant called `CB_DMA_RD` that is used to set bits `10`, `8`, `9`, and `11` of the CSR. Section 10.12.2.4 shows how the `cbstrategy` interface uses this constant to set up the direct memory access (DMA)

enable bits.

7 Defines a constant called CB_DMA_WR that is used to set the bits 11, 8, 9, and 10 of the CSR. Section 10.12.2.4 shows how the cbstrategy interface uses this constant when converting the buffer virtual address.

8 Defines a constant called CB_DMA_DONE for use by the cbstart interface's timeout loop. Section 10.13 shows how cbstart uses this constant in the timeout loop.

9 Uses the _IO macro to construct an ioctl macro called CBPIO. The _IO macro defines ioctl types for situations where no data is actually transferred between the application program and the kernel. For example, this could occur in a device control operation. *Writing Device Drivers, Volume 2: Reference* provides information on the _IO, _IOR, _IOW, and _IOWR ioctl macros.

The cbread, cbwrite, and cbioctl interfaces use CBPIO to set the iomode member of the cb_unit structure for this CB device to the programmed I/O (PIO) read or write code. Section 10.11.1 shows how the cbread interface uses this ioctl. Section 10.11.2 shows how the cbwrite interface uses this ioctl. Section 10.14.3 shows how the cbioctl interface uses this ioctl.

10 Uses the _IO macro to construct an ioctl macro called CBDMA. The cbread, cbwrite, and cbioctl interfaces use CBDMA to set the iomode member of the cb_unit structure for this CB device to the DMA I/O read or write code. Section 10.11.1 shows how the cbread interface uses this ioctl. Section 10.11.2 shows how the cbwrite interface uses this ioctl. Section 10.14.3 shows how the cbioctl interface uses this ioctl.

11 Uses the _IO macro to construct an ioctl macro called CBINT. Section 10.14.4 shows how the cbioctl interface uses CBINT to perform interrupt tests.

12 Uses the _IOWR macro to construct an ioctl macro called CBROM. The _IOWR macro defines ioctl types for situations where data is transferred from the user's buffer into the kernel. The driver then performs the appropriate ioctl operation and returns data of the same size back up to the user level application. Typically, this data consists of device control or status information passed to the driver from the application program. Section 10.14.5 shows how the cbioctl interface uses CBROM to perform a variety of tasks.

13 Uses the _IOR macro to construct an ioctl macro called CBCSR. The _IOR macro defines ioctl types for situations where data is transferred from the kernel into the user's buffer. Typically, this data consists of device control or status information returned to the application program. Section 10.14.5 shows how the cbioctl interface uses _IOR to

perform a variety of tasks.

14 Uses the _IO macro to construct an ioctl macro called CBINC. Section 10.14.2 shows how the cbioctl interface uses CBINC when it starts to increment the lights.

15 Uses the _IO macro to construct an ioctl macro called CBSTP. Section 10.14.5 shows how cbioctl uses CBSTP when it stops incrementing the lights.

16 Defines the device register offset definitions for the CB device. The device registers are aligned on longword (32-bit) boundaries, even when they are implemented with less than 32 bits. The CB_ADDER device register offset represents the 32-bit read/write DMA address register. The CB_ADDER and the following device register offset definitions show the new technique for defining the registers of a device. Previous versions of the /dev/cb device driver defined a device register structure called CB_REGISTERS. The /dev/cb driver used the members of this data structure to directly access the device registers of the CB device.

By defining device register offset definitions, the /dev/cb driver can use the read_io_port and write_io_port interfaces to access the device registers of the CB device. This makes the /dev/cb driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

Section 10.12.2.4 shows the use of CB_ADDER in a call to write_io_port.

17 Represents the 32-bit read/write data register. Section 10.11.1 shows the use of CB_DATA in a call to read_io_port. Section 10.11.2 shows the use of CB_DATA in a call to write_io_port.

18 Represents the 16-bit read/write CSR/LED register. Section 10.13, Section 10.15, Section 10.14.4, and Section 10.14.5 show the use of CB_CSR in calls to read_io_port and write_io_port.

19 Represents the go bit set by the cbwrite interface and cleared by the cbread interface. Section 10.13 and Section 10.14.4 show the use of CB_TEST in calls to read_io_port and write_io_port. Section 10.16 shows the use of CB_TEST in a call to read_io_port.

10.3 Include Files Section

This section is applicable to the loadable or static version of the /dev/cb device driver. It identifies the following header files needed by the /dev/cb device driver:

```
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <hal/cpuconf.h>
#include <sys/vm.h>
#include <sys/buf.h>
#include <sys/errno.h>
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/types.h>

#include <io/common/devdriver.h>
#include <sys/sysconfig.h>
#include <io/dec/tc/tc.h>

#include <kits/ESB100/cbreg.h> [1]

#define NCB TC_OPTION_SLOTS [2]
```

[1] Includes the device register header file, which contains the definitions of the device register offsets for the CB device. Section 10.2 shows the definitions of the device register offsets. The directory specification adheres to the third-party device driver configuration model discussed in Section 11.1.2. If the traditional device driver configuration model was followed, the directory specification would be <io/EasyInc/cbreg.h>. The directory specification you make here depends on where you put the device register file. *Writing Device Drivers, Volume 2: Reference* provides reference (man) page-style descriptions of the header files most frequently used by DEC OSF/1 device drivers.

[2] Defines a constant called NCB that is used to allocate data structures needed by the /dev/cb driver. The define uses the constant TC_OPTION_SLOTS, which is defined in /usr/sys/include/io/dec/tc/tc.h. There can be at most three instances of the CB controller on the system. This is a small number of instances of the driver on the system and the data structures themselves are not large, so it is acceptable to allocate for the maximum configuration. This example uses the static allocation technique model 2 described in Section 2.3.2. You could also define this constant in a *name*_data.c file.

10.4 Autoconfiguration Support Declarations and Definitions Section

This section is applicable to the loadable or static version of the /dev/cb device driver. It contains the following declarations needed by the /dev/cb device driver:

```
extern  int hz; [1]

int cbprobe(), cbattach(), cbintr(), cbopen(), cbclose();
int cbread(), cbwrite(), cbioctl(), cbstart(), cbminphys();
int cbincled(), cb_ctlr_unattach(), cbstrategy(); [2]
struct controller *cbinfo[NCB];  [3]

struct  driver cbdriver = {
        cbprobe,
        0,
        cbattach,
        0,
        0,
        0,
        0,
        0,
        "cb",
        cbinfo,
        0,
        0,
        0,
        0,
        0,
        cb_ctlr_unattach,
        0
}; [4]
```

[1] Declares the global variable *hz* to store the number of clock ticks per second. The *hz* global variable is typically used with the `timeout` kernel interface to schedule interfaces to be run at the time stored in the variable. Section 10.14.2 shows how `cbioctl` uses this global variable. Section 10.15 shows how `cbincled` uses this variable.

[2] Declares the driver interfaces for the /dev/cb device driver.

[3] Declares an array of pointers to `controller` structures and calls it `cbinfo`. The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation. Section 7.4 describes the `controller` structure.

The NCB constant is used to represent the maximum number of CB controllers. This number is used to size the array of pointers to `controller` structures. If you are writing a new device driver (as opposed to porting an existing driver, which is the case for the /dev/cb driver), dynamically allocate the data structures as needed by calling the `kalloc` interface. See Section 2.3.3 for a discussion of this technique. The structures for the `/dev/cb` driver are not dynamically allocated.

4 Declares and initializes the `driver` structure and calls it `cbdriver`. The value zero (0) indicates that the `/dev/cb` driver does not make use of a specific member of the `driver` structure. The following list describes those members initialized to a nonzero value by the `/dev/cb` device driver. Section 7.6 describes the `driver` structure.

- The driver's probe interface, `cbprobe`. Section 10.8.1 shows how to implement `cbprobe`.
- The driver's cattach interface, `cbattach`. Section 10.8.2 shows how to implement `cbattach`.
- The value `cb`, which is the name of the device.
- The value `cbinfo`, which references the array of pointers to the previously declared `controller` structures. You index this array with the controller number as specified in the `ctlr_num` member of the `controller` structure.
- The driver's controller unattach interface, `cb_ctlr_unattach`. The `cb_ctlr_unattach` interface removes the `controller` structure associated with the TURBOchannel test board from the list of `controller` structures that it handles. Section 10.8.3 shows how to implement `cb_ctlr_unattach`. Loadable drivers use the controller unattach interface.

10.5 Loadable Driver Configuration Support Declarations and Definitions Section

This section contains the following declarations used by the /dev/cb device driver when it is configured as a loadable driver:

```
extern int nodev(), nulldev(); [1]
extern ihandler_id_t handler_add(), handler_del();
extern ihandler_id_t handler_enable(), handler_disable(); [2]
ihandler_id_t cb_id_t[NCB];     [3]
#define CB_BUSNAME    "tc" [4]

int cb_is_dynamic = 0; [5]

struct tc_option cb_option_snippet [] =
{
    /*  module           driver   intr_b4  itr_aft          adpt    */
    /*  name             name     probe    attach   type    config  */
    /*  ------           ------   -------- -------- ----    ------  */
    {   "CB       ",     "cb",      0,       1,     'C',    0},
    {   "",              ""       } /* Null terminator in the table */
}; [6]

int num_cb = 0;    [7]
```

[1] Declares external references for the nodev and nulldev interfaces, which are used to initialize members of the cdevsw table under specific circumstances. The cdevsw_add kernel interface, called by the driver's cb_configure interface, initializes the cdevsw table. Section 8.2.1 provides a description of the cdevsw table and examples of the nodev and nulldev interfaces.

[2] Declares an external reference for the handler_add interface, which registers the interrupt service interface for the loadable driver. Note also the declarations of the external references for the handler_del, handler_enable, and handler_disable interfaces. Section 10.8.1 shows how the cbprobe interface calls handler_add.

[3] Declares an array of IDs used to deregister the interrupt handlers. The NCB constant represents the maximum number of CB controllers. This number sizes the array of IDs. Thus, there is one ID per CB device. Section 10.8.1 shows how cbprobe uses *cb_id_t*.

If you are writing a new device driver (as opposed to porting an existing driver, which is the case for the /dev/cb driver), dynamically allocate the data structures as needed by calling the kalloc interface.

[4] Defines a constant that represents a 2-character string that indicates this is a driver that operates on the TURBOchannel bus. This constant is passed as an argument to the ldbl_stanza_resolver, ldbl_ctlr_configure, and ldbl_ctlr_unconfigure interfaces. This bus name is used in calls to the configuration code. Other bus types can use a different name. Section 10.9.2 shows how to

call `ldbl_stanza_resolver` and `ldbl_ctlr_configure`. Section 10.9.3 shows how to call `ldbl_ctlr_unconfigure`.

5 Declares a variable called *cb_is_dynamic* and initializes it to the value zero (0). This variable is used to control any differences in the tasks performed by the `/dev/cb` device driver when it is configured as a loadable or static driver. Thus, the `/dev/cb` driver can be compiled once. The decision as to whether it is loadable or static is made at kernel configuration by the system manager and not at compile time by the driver writer.

Section 10.8.1 shows how `cbprobe` uses *cb_is_dynamic*. Section 10.8.3 shows how `cb_ctlr_unattach` uses *cb_is_dynamic*. Section 10.9.2 shows how `cb_configure` uses *cb_is_dynamic*.

6 These lines are specific to drivers written for the TURBOchannel bus. Other bus types may use a different mechanism.

Declares a `tc` option table snippet that includes an entry for the loadable version of this driver. For the static version, a similar entry is made in the `tc_option` table located in the `tc_option_data.c` file. The entry in `tc_option_data.c` is used only when the driver is configured statically; the `cb_option_snippet` entry is used only when the driver is configured dynamically.

The `tc_option` option table contains the bus-specific ROM module name for the driver. This information forms the bus-specific parameter that is passed to the `ldbl_stanza_resolver` interface to search for matches in the `tc_option` table. The `tc_option` table is used by the bus configuration interfaces associated with the TURBOchannel bus.

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

- module name

  In this column, you specify the device name in the ROM on the hardware device. The module name must be no more than seven characters in length, but you must blank-pad the name to 8 bytes. The module name can be up to 8 characters in length. You must blank-pad the name to 8 bytes for those names that are less than 8 characters in length. Thus, the entry for the `/dev/cb` driver consists of the letters ''CB'' followed by six spaces.

- driver name

  In this column, you specify the driver name as it appears in the `Module_Config_Name` field of the `stanza.loadable` file fragment. In this example, the driver name is `cb`. Because you specify the same name in the `Module_Config_Name` field and the driver name field of the `tc option` snippet table, the bus configuration code initializes the correct `controller` and `device` structures during device autoconfiguration for loadable drivers. Section 12.6.2.19 describes the `Module_Config_Name` field.

- intr_b4 probe

  In this column, you specify whether the device needs interrupts enabled during execution of the driver's `probe` interface. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled. In the example, the value zero (0) is specified to indicate that the `CB` device does not need interrupts enabled.

- itr_aft attach

  In this column, you specify whether the device needs interrupts enabled after the driver's `probe` and `attach` interfaces complete. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled. In the example, the value 1 is specified to indicate that the `CB` device needs interrupts enabled after its `cbprobe` and `cbattach` interfaces complete.

- type

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

- adpt config

  If the device in the type column is `A` (adapter), you specify the name of the interface to configure the adapter. Otherwise, you specify the value zero (0). In the example, the value zero (0) is specified because the device in the previous column is a controller. Section 10.9.2 shows how the `cb_configure` interface uses `cb_option_snippet`.

7 Declares a variable called *num_cb* to store the count on the number of controllers probed during autoconfiguration. This variable is initialized to the value zero (0) to indicate that no instances of the controller have been initialized yet. Section 10.8.1 shows how `cbprobe` uses this variable. Section 10.8.3 shows how `cb_ctlr_unattach` uses this variable. Section 10.9.2 shows how `cb_configure` uses this variable.

10.6 Local Structure and Variable Definitions Section

This section is applicable to the loadable or static version of the /dev/cb device driver. It contains the such declarations as the following cb_unit data structure:

```
struct buf cbbuf[NCB]; 1
unsigned tmpbuffer; 2

struct cb_unit {
    int  attached;
    int  opened;
    int  iomode;
    int  intrflag;
    int  ledflag;
    int  adapter;
    caddr_t cbad;
    io_handle_t cbr;
    struct buf   *cbbuf;
} cb_unit[NCB]; 3
#define MAX_XFR 4 4
```

1 Declares an array of buf structures and calls it cbbuf. Section 8.1 describes the buf structure.

The NCB constant is used to represent the maximum number of CB devices. This number is used to size the array of buf structures. Thus, there is one buf structure per CB device. Section 10.8.2 shows how cbattach references the buf structure. Section 10.11.1 and Section 10.11.2 show the buf structure passed as an argument to the physio kernel interface.

2 Declares a one-word buffer called *tmpbuffer*. Section 10.12.2.2 shows how the cbstrategy interface uses this variable to store the internal buffer virtual address.

3 Declares an array of cb_unit data structures. Again, the NCB constant is used to represent the maximum number of CB devices. This number is used to size the array of cb_unit structures. Thus, there is one cb_unit structure per CB device.

The cbattach interface initializes some of the members of the cb_unit data structure, and all of the other /dev/cb driver interfaces reference cb_unit.

The following list describes the members contained in this structure:

- attached

  Stores a value to indicate that the specified CB device is attached. Section 10.8.2 shows that the cbattach interface sets this member to a value that indicates the device is attached.

- `opened`

  Stores a value to indicate that the specified CB device is opened. Section 10.10.1 shows that the `cbopen` interface sets this member to a value that indicates the device is open; and Section 10.10.2 shows that the `cbclose` interface clears the value to indicate the device is closed.

- `iomode`

  Stores the read/write mode to one of the bits represented by these constants defined in `cbreg.h`: `CBPIO` (programmed I/O) and `CBDMA` (DMA I/O read code). Section 10.14.3 shows that the `cbioctl` interface sets this member to these constants.

- `intrflag`

  Stores a flag value used to test for an interrupt. Section 10.16 shows that `cbintr` sets the interrupt flag; and Section 10.14.4 shows that `cbioctl` clears the interrupt flag.

- `ledflag`

  Stores a flag value for the LED increment function. Section 10.9.3 shows that `cb_configure` turns off the LED increment function; and Section 10.14.2 shows that `cbioctl` sets the LED increment flag. Section 10.14.5 shows that `cbioctl` sets the flag so that it turns off the LED increment function.

- `adapter`

  Stores the TC slot number. Section 10.8.2 shows that `cbattach` sets this member to the slot number for this CB controller.

- `cbad`

  Stores the ROM base address. Section 10.8.2 shows that `cbattach` sets this member to the address of the data to be checked for read accessibility.

- `cbr`

  Stores the I/O handle for the device registers associated with this CB device. An I/O handle is a data entity that is of type `io_handle_t`. You pass this I/O handle to the `read_io_port` and `write_io_port` interfaces. Section 10.8.2 shows that `cbattach` sets this member to point to this CB device's registers. Section 10.2 shows the declaration of the device register offsets used with the I/O handle.

- `cbbuf`

  Stores a pointer to a `buf` structure. Section 10.8.2 shows that `cbattach` sets this member to point to this CB device's buffer header.

[4] Defines a constant called MAX_XFR that specifies the maximum chunk of data (in bytes) that can be transferred in read and write operations. Section 10.11.1 shows how cbread uses MAX_XFR; and Section 10.11.2 shows how cbwrite uses this constant. Section 10.12.1 shows that the cbminphys interface uses this constant to set the b_bcount member of the buf structure associated with this CB device.

10.7 Loadable Driver Local Structure and Variable Definitions Section

This section is applicable only to the loadable version of the /dev/cb device driver. It contains declarations of the driver interfaces that were previously declared in Section 10.4.

It also contains the following declaration of the /dev/cb driver's cdevsw entry that will be dynamically added to the cdevsw table:

```
int cb_config = FALSE;  [1]
dev_t cb_devno = NODEV; [2]

struct cdevsw cb_cdevsw_entry = {
        cbopen,
        cbclose,
        cbread,
        cbwrite,
        cbioctl,
        nodev,
        nodev,
        0,
        nodev,
        nodev,
        DEV_FUNNEL_NULL
}; [3]

#define CB_DEBUG [4]
#undef CB_DEBUGx
```

[1] Declares a variable called *cb_config* to store state flags that indicate whether the /dev/cb driver is configured as a loadable driver. The *cb_config* variable is initialized to the value FALSE to indicate the driver defaults to being statically configured. Section 10.9.2 shows that cb_configure sets *cb_config* to the value TRUE to indicate that the /dev/cb driver has successfully configured as a loadable driver. Section 10.9.3 shows that cb_configure sets *cb_config* to the value FALSE to indicate that the /dev/cb driver has successfully unconfigured.

[2] Declares a variable called *cb_devno* to store the cdevsw table entry slot to use. The *cb_devno* variable is initialized to the value NODEV to indicate that no major number for the device has been assigned. Section 10.9.2 shows that cb_configure sets *cb_devno* to the table entry slot.

[3] Declares and initializes the cdevsw structure called cb_cdevsw_entry. Section 10.9.2 shows that cdevsw_add uses cb_cdevsw_entry to add the /dev/cb driver interfaces to the in-memory cdevsw table.

The following list describes those members initialized to a nonzero value by the `/dev/cb` device driver:

- The driver's `open` interface, `cbopen`. Section 10.10.1 shows how to implement `cbopen`.
- The driver's `close` interface, `cbclose`. Section 10.10.2 shows how to implement `cbclose`.
- The driver's `read` interface, `cbread`. Section 10.11.1 shows how to implement `cbread`.
- The driver's `write` interface, `cbwrite`. Section 10.11.2 shows how to implement `cbwrite`.
- The driver's `ioctl` interface, `cbioctl`. Section 10.14.1 shows how to implement `cbioctl`.

Section 8.2.1 describes the `cdevsw` table.

4 Uses two of the C preprocessor statements to define and undefine debug constants. The `/dev/cb` device driver contains numerous conditional compilation debug statements. This section shows only one of these statements. However, the source code listing in Section B.2 contains all the debug code.

10.8 Autoconfiguration Support Section

This section is applicable to the loadable or static version of the `/dev/cb` device driver. Table 10-2 lists the three interfaces implemented as part of the Autoconfiguration Support Section, along with the sections in the book where each is described.

Table 10-2: Interfaces Implemented as Part of the Autoconfiguration Support Section

Interface	Section
Implementing the cbprobe Interface	Section 10.8.1
Implementing the cbattach Interface	Section 10.8.2
Implementing the cb_ctlr_unattach Interface	Section 10.8.3

10.8.1 Implementing the cbprobe Interface

The `cbprobe` interface is applicable to both the loadable and static versions of the `/dev/cb` device driver. However, there are some tasks associated only with loadable drivers. These tasks are identified by a conditional `if` statement that tests the *`cb_is_dynamic`* variable. The `cbprobe` interface's main task is to determine whether any CB devices exist on the system. For the loadable version of the driver, `cbprobe` also calls the appropriate interfaces to register the interrupt handlers for the loadable driver. The following code implements the `cbprobe` interface:

```
cbprobe(vbaddr, ctlr)
caddr_t vbaddr;            1
struct controller *ctlr;   2
{

        ihandler_t handler;         3
        struct tc_intr_info info;   4
        int unit = ctlr->ctlr_num;  5

/***************************************************
 *               DEBUG STATEMENT                   *
 ***************************************************/
#ifdef CB_DEBUG 6
printf("CBprobe @ %8x, vbaddr = %8x, ctlr = %8x\n",cbprobe,vbaddr,ctlr);
#endif /* CB_DEBUG */

        if (cb_is_dynamic) { 7

                handler.ih_bus = ctlr->bus_hd; 8
                info.configuration_st = (caddr_t)ctlr; 9
                info.config_type = TC_CTLR; 10

                info.intr = cbintr; 11

                info.param = (caddr_t)unit; 12
                handler.ih_bus_info = (char *)&info; 13

                cb_id_t[unit] = handler_add(&handler); 14
                if (cb_id_t[unit] == NULL) { 15

                        return(0);
                }
                if (handler_enable(cb_id_t[unit]) != 0) { 16
                       handler_del(cb_id_t[unit]); 17

                        return(0);
                }
        }

        num_cb++; 18

        return(1); 19
}
```

1 Declares a *vbaddr* argument that specifies the system virtual address (SVA) that corresponds to the base address of the slot. This line is applicable to the loadable or static version of the /dev/cb device driver.

2 Declares a pointer to the controller structure associated with this CB device. The controller structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation. This line is applicable to the loadable or static version of the /dev/cb device driver. Section 7.4 describes the controller structure.

3 Declares an ihandler_t data structure called handler to contain information associated with the /dev/cb device driver interrupt handling. Section 7.8 describes the ihandler_t structure. The cbprobe interface initializes two members of this data structure. This line is applicable only to the loadable version of the /dev/cb device driver.

4 Declares a tc_intr_info data structure called info. Section 7.9 describes the tc_intr_info structure.

5 Declares a *unit* variable and initializes it to the controller number. This controller number identifies the specific CB controller that is being probed.

The controller number is contained in the ctlr_num member of the controller structure associated with this CB device. Section 7.4.3 describes the ctlr_num member. This member is used as an index into a variety of tables to retrieve information about this instance of the CB device.

6 Calls the printf interface to print information useful for debugging purposes. This line is executed only during debugging of the /dev/cb driver.

Only a limited number of characters (currently 128) can be sent to the console display during each call to any section of a driver. The reason is that the characters are buffered until the driver returns to the kernel, at which time they are actually sent to the console display. If more than 128 characters are sent to the console display, the storage pointer may wrap around, discarding all previous characters; or it may discard all characters following the first 128.

If you need to see the results on the console terminal, limit the message size to the maximum of 128 whenever you send a message from within the driver. However, printf also stores the messages in an error log file. You can view the text of this error log file by using the uerf command. See the reference (man) page for this command. The messages are easier to read if you use `uerf` with the `-o terse` option.

This line is applicable to the loadable or static version of the `/dev/cb` device driver.

7 Registers the interrupt handlers if *`cb_is_dynamic`* evaluates to a nonzero value, indicating that the `/dev/cb` device driver was dynamically loaded. If the driver was statically configured, the interrupt handlers have already been registered through the `config` program.

The *`cb_is_dynamic`* variable contains a value to control any differences in tasks performed by the static and loadable versions of the `/dev/cb` device driver. This approach means that any differences are made at run time and not at compile time. The *`cb_is_dynamic`* variable was previously initialized and set by the `cb_configure` interface, discussed in Section 10.9.2.

Items 8 – 18 are applicable only if the driver was dynamically loaded. These steps accomplish the setup of the driver's interrupt handler.

8 Specifies the bus that this controller is attached to. The `bus_hd` member of the `controller` structure contains a pointer to the `bus` structure that this controller is connected to. After the initialization, the `ih_bus` member of the `ihandler_t` structure contains the pointer to the `bus` structure associated with the `/dev/cb` device driver.

9 Sets the `configuration_st` member of the `info` data structure to the pointer to the `controller` structure associated with this `CB` device. This `controller` structure is the one for which an associated interrupt will be written.

This line also performs a type-casting operation that converts `ctlr` (which is of type pointer to a `controller` structure) to be of type `caddr_t`, the type of the `configuration_st` member.

10 Sets the `config_type` member of the `info` data structure to the constant `TC_CTLR`, which identifies the `/dev/cb` driver type as a TURBOchannel controller.

11 Sets the `intr` member of the `info` data structure to `cbintr`, the `/dev/cb` device driver's interrupt service interface (ISI).

12 Sets the `param` member of the `info` data structure to the controller number for the `controller` structure associated with this `CB` device. Once the driver is operational and interrupts are generated, the `cbintr` interface is called with the controller number, which specifies which instance of the controller the interrupt is associated with.

This line also performs a type-casting operation that converts *`unit`* (which is of type `int`) to be of type `caddr_t`, the type of the `param` member.

13 Sets the ih_bus_info member of the handler data structure to the address of the bus-specific information structure, info. This setting is necessary because registration of the interrupt handlers will indirectly call bus-specific interrupt registration interfaces.

This line also performs a type-casting operation that converts info (which is of type ihandler_t) to be of type char *, the type of the ih_bus_info member.

14 Calls the .L "handler_add" interface and saves its return value for use later by the handler_del interface. The handler_add interface takes one argument: a pointer to an ihandler_t data structure, which in the example is the initialized handler structure.

This interface returns an opaque ihandler_id_t key, which is a unique number used to identify the interrupt service interfaces to be acted on by subsequent calls to handler_del, handler_disable, and handler_enable. Note that this key is stored in the *cb_id_t* array (indexed by the unit number), which was declared in Section 10.5. Section 10.8.3 shows how to call handler_del and handler_disable.

15 If the return value from handler_add equals NULL, returns a failure status to indicate that registration of the interrupt handler failed.

16 If the handler_enable interface returns a nonzero value, returns the value zero (0) to indicate that it could not enable a previously registered interrupt service interface. The handler_enable interface takes one argument: a pointer to the interrupt service interface's entry in the interrupt table. In this example, the ID associated with the interrupt entry is contained in the *cb_id_t* array.

17 If the call to handler_enable failed, removes the previously registered interrupt handler by calling handler_del prior to returning an error status.

18 Increments the number of instances of this controller found on the system.

19 The cbprobe interface simply returns the value 1 to indicate success status because the TURBOchannel initialization code already verified that the device was present.

10.8.2 Implementing the cbattach Interface

The `cbattach` interface is applicable to both the loadable and static versions of the `/dev/cb` device driver. Its main task is to perform controller-specific initialization. The following code implements the `cbattach` interface:

```
cbattach(ctlr)
struct controller *ctlr; [1]
{
        struct cb_unit *cb; [2]

        cb = &cb_unit[ctlr->ctlr_num]; [3]

        cb->attached = 1; [4]

        cb->adapter = ctlr->slot; [5]

        cb->cbad = ctlr->addr; [6]

        cb->cbr = (io_handle_t)CB_ADR(ctlr->addr); [7]
        cb->cbbuf = &cbbuf[ctlr->ctlr_num]; [8]
}
```

[1] Declares a pointer to the `controller` structure associated with this CB device. The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation. Section 7.4 describes the `controller` structure.

[2] Declares a pointer to the `cb_unit` data structure associated with this CB device and calls it `cb`. Section 10.6 shows the declaration of `cb_unit`.

[3] Sets the pointer to the `cb_unit` structure to the address of the unit data structure associated with this CB device. The `ctlr_num` member of the `controller` structure pointed to by `ctlr` holds the controller number for the controller associated with this CB device. Thus, this member is used as an index into the array of `cb_unit` structures to set the instance representing this CB device.

[4] Indicates that this CB device is attached to its associated controller by setting the `attached` member of the device's associated `cb_unit` structure to the value 1.

[5] Sets this CB device's TURBOchannel slot number by setting the `adapter` member of this CB device's `cb_unit` structure to the slot number contained in the `slot` member of the device's `controller` structure. Section 7.4.6 describes the `slot` member.

[6] Sets the base address of the device ROM. This location is the address of the data to be checked for read accessibility. The `cbattach` interface accomplishes this task by setting the `cbad` member of this CB device's `cb_unit` structure to the address contained in the `addr` member of its

controller structure.

7 Sets the pointer to the CB device's registers. The cbattach interface accomplishes this task by setting the cbr member of this device's cb_unit structure to the register address calculated by the CB_ADR macro. The cbr member is an I/O handle. An I/O handle is a data entity that is of type io_handle_t.

Note that CB_ADR takes as an argument the address of the data to be checked for read accessibility, which in this case is stored in the addr member of the controller structure. Section 10.2 shows the definition of the CB_ADR macro.

When CB_ADR completes execution, the cb->cbr pointer is set to the base address plus 20000 hexadecimal because the CB device registers are located at that offset from the base address.

8 Sets the buffer structure address (the cbbuf member of this CB device's cb_unit structure) to the address of this CB device's buf structure. Again, the ctlr_num member is used as an index into the array of buf structures associated with this CB device.

10.8.3 Implementing the cb_ctlr_unattach Interface

The `cb_ctlr_unattach` interface is a loadable driver-specific interface called indirectly from the bus code when a system manager specifies that the loadable driver is to be unloaded. In other words, this interface would never be called if the `/dev/cb` device driver were configured as a static driver because static drivers cannot be unconfigured. The `cb_ctlr_unattach` interface's main tasks are to deregister the interrupt handlers associated with the `/dev/cb` device driver and to remove the specified `controller` structure from the list of controllers the `/dev/cb` driver handles.

The following code implements the `cb_ctlr_unattach` interface:

```
int cb_ctlr_unattach(bus, ctlr)
    struct bus *bus;          [1]
    struct controller *ctlr; [2]
{
        register int unit = ctlr->ctlr_num; [3]

        if ((unit > num_cb) || (unit < 0)) { [4]
                return(1);
        }

        if (cb_is_dynamic == 0) { [5]
                return(1);
        }

        if (handler_disable(cb_id_t[unit]) != 0) { [6]
                return(1);
        }
        if (handler_del(cb_id_t[unit]) != 0) { [7]
                return(1);
        }
        return(0); [8]
}
```

[1] Declares a pointer to a `bus` structure and calls it `bus`. The `bus` structure represents an instance of a bus entity. A bus is a real or imagined entity to which other buses or controllers are logically attached. All systems have at least one bus, the system bus, even though the bus may not actually exist physically. The term controller here refers both to devices that control slave devices (for example, disk and tape controllers) and to devices that stand alone (for example, terminal or network controllers). Section 7.3 describes the `bus` structure.

[2] Declares a pointer to a `controller` structure and calls it `ctlr`. This `controller` structure is the one you want to remove from the list of controllers handled by the `/dev/cb` device driver. Section 7.4 describes the `controller` structure.

[3] Declares a *`unit`* variable and initializes it to the controller number. This controller number identifies the specific `CB` controller whose associated

controller structure is to be removed from the list of controllers handled by the /dev/cb driver. Section 7.4.3 describes the ctlr_num member.

The controller number is contained in the ctlr_num member of the controller structure associated with this CB device.

4 If the controller number is greater than the number of controllers found by the cbprobe interface or the number of controllers is less than zero, returns the value 1 to the bus code to indicate an error. This sequence of code validates the controller number. The *num_cb* variable contains the number of instances of the CB controller found by the cbprobe interface. Section 10.8.1 describes the implementation of cbprobe.

5 If *cb_is_dynamic* is equal to the value zero (0), returns the value 1 to the bus code to indicate an error. This sequence of code validates whether the /dev/cb driver was dynamically loaded. The *cb_is_dynamic* variable contains a value to control any differences in tasks performed by the static and loadable versions of the /dev/cb device driver. This approach means that any differences are made at run time and not at compile time. The *cb_is_dynamic* variable was previously initialized and set by the cb_configure interface, discussed in Section 10.9.2.

6 If the return value from the call to the handler_disable interface is not equal to the value zero (0), returns the value 1 to the bus code to indicate an error. Otherwise, the handler_disable interface makes the /dev/cb device driver's previously registered interrupt service interfaces unavailable to the system. Section 9.6.4 provides additional information on handler_disable.

7 If the return value from the call to the handler_del interface is not equal to the value zero (0), returns the value 1 to the bus code to indicate an error. Otherwise, the handler_del interface deregisters the /dev/cb device driver's interrupt service interface from the bus-specific interrupt dispatching algorithm. Section 9.6.4 provides additional information on handler_del.

The handler_del interface takes the same argument as the handler_disable interface: a pointer to the interrupt service's entry in the interrupt table.

8 Returns the value zero (0) to the bus code upon successful completion of the tasks performed by the cb_ctlr_unattach interface.

10.9 Loadable Device Driver Section

This section is applicable only to the loadable version of the `/dev/cb` device driver. It implements the `cb_configure` interface. Table 10-3 lists the tasks associated with implementing the Loadable Device Driver Section, along with the sections in the book where each task is described.

Table 10-3: Tasks Associated with Implementing the Loadable Device Driver Section

Tasks	Section
Setting Up the cb_configure Interface	Section 10.9.1
Configuring (Loading) the /dev/cb Device Driver	Section 10.9.2
Unconfiguring (Unloading) the /dev/cb Device Driver	Section 10.9.3
Querying the /dev/cb Device Driver	Section 10.9.4

10.9.1 Setting Up the cb_configure Interface

The following code shows how to set up the `cb_configure` interface:

```
cb_configure(op,indata,indatalen,outdata,outdatalen)
   sysconfig_op_t op;        [1]
   device_config_t *indata;  [2]
   size_t indatalen;         [3]
   device_config_t *outdata; [4]
   size_t outdatalen;        [5]
{
        dev_t   cdevno; [6]
        int     retval; [7]
        int     i;      [8]
        struct cb_unit *cb; [9]
        int cbincled();     [10]
```

[1] Declares an argument called *op* to contain a constant that describes the configuration operation to be performed on the loadable driver. This argument is used in a `switch` statement and evaluates to one of the following valid constants: `SYSCONFIG_CONFIGURE`, `SYSCONFIG_UNCONFIGURE`, or `SYSCONFIG_QUERY`.

[2] Declares a pointer to a `device_config_t` data structure called `indata` that consists of inputs to the `cb_configure` interface. This data structure is filled in by the device driver method of `cfgmgr`. The `device_config_t` data structure is used to represent a variety of information, including the `/dev/cb` driver's major number requirements. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of the `device_config_t` structure.

[3] Declares an argument called *`indatalen`* to store the size of this input data structure (in bytes).

[4] Declares a pointer to a data structure called `outdata` that is filled in by the `/dev/cb` driver. This data structure contains a variety of information, including the ''return values'' from the `/dev/cb` driver to `cfgmgr`. This returned information contains the major number assigned to the `CB` device.

[5] Declares an argument called *`outdatalen`* to store the size of this output data structure (in bytes).

[6] Declares a variable called *cdevno* to store the major device number for the `CB` device.

[7] Declares a variable called *`retval`* to store the return value from the `cdevsw_del` interface.

[8] Declares a variable called *`i`* to be used in the `for` loop when `cb_configure` unloads the loadable driver.

9 Declares a pointer to the `cb_unit` data structure associated with this CB device and calls it `cb`. Section 10.6 shows the declaration of `cb_unit`.

10 Declares a forward reference to the `cbincled` interface. Section 10.15 shows the implementation of `cbincled`.

10.9.2 Configuring (Loading) the /dev/cb Device Driver

The following code shows how to implement the configurable (loadable) version of the `/dev/cb` device driver:

```
switch (op) {

      case SYSCONFIG_CONFIGURE: 1

        if (indata->dc_dsflags & IH_DRV_DYNAMIC) {
                cb_is_dynamic = 1;
        } 2
        if (cb_is_dynamic) { 3

                if (strlen(indata->config_name) <= 0) {
                        printf("cb_configure, null config name.\n");
                        return(EINVAL);
                } 4

                if (ldbl_stanza_resolver(indata->config_name,
                        CB_BUSNAME, &cbdriver,
                        (caddr_t *)cb_option_snippet) != 0) {
                        return(EINVAL);
                } 5

                if (ldbl_ctlr_configure(CB_BUSNAME,
                        LDBL_WILDNUM, indata->config_name,
                        &cbdriver, 0)) {
                        return(EINVAL);
                } 6

                if (num_cb == 0) {
                     return(EINVAL);
                }
        } 7

        cdevno = makedev(indata->dc_cmajnum,
                (indata->dc_cmajnum == -1)?-1:0); 8
        cdevno = cdevsw_add(cdevno,&cb_cdevsw_entry); 9
        if (cdevno == NODEV) {

                return(ENODEV);
        } 10

        cb_devno = cdevno; 11

        outdata->dc_cmajnum = major(cb_devno); 12

        outdata->dc_begunit = 0;

        outdata->dc_numunit = num_cb;

        outdata->dc_version = DRIVER_BUILD_LEVEL;

        outdata->dc_dsflags = indata->dc_dsflags;

        outdata->dc_bmajnum = NODEV;

        outdata->dc_errcode = 0;
        outdata->dc_ihflags = 0;
        outdata->dc_ihlevel = 0;

        cb_config = TRUE; 13
        break;
```

1 Specifies the SYSCONFIG_CONFIGURE constant to indicate that this section of code implements the configure loadable driver operation. The file /usr/sys/include/sys/sysconfig.h contains the definition of this constant.

2 If the device switch configuration flag is set to the IH_DRV_DYNAMIC bit, then this is the loadable version of the driver. To indicate this state, sets the *cb_is_dynamic* variable to 1. The file /usr/sys/include/sys/sysconfig.h contains the definition of this constant.

The device switch configuration flag is set by the device driver method of cfgmgr in the dc_dsflags member of the input data structure, indata. This if statement is included because it is possible for a static driver to call the configure interface.

3 If this is the loadable version of the driver, calls the strlen kernel interface.

4 If the number of characters returned by strlen is less than or equal to zero:

- Calls printf to print a message indicating that no device driver name was specified in the stanza.loadable file fragment. The absence of a device driver name in stanza.loadable indicates that the driver name cannot be determined. The driver name is required for the configuration of the loadable driver. In this case, the interface must return an error status.

  Section 12.6 describes the stanza file format and syntax.

- Returns the constant EINVAL to indicate an invalid argument. This constant is defined in the file /usr/sys/include/sys/errno.h. This error return value gets indirectly passed back to cfgmgr.

The strlen interface takes one argument: a pointer to an array of characters terminated by a null character. In this call, the array of characters is the config_name member of the pointer to the indata input data structure. This member is set by the driver method of cfgmgr, which obtains the driver's configuration name from the Module_Config_Name field in the stanza.loadable file fragment. Because the driver's configuration name is a required field in the stanza.loadable file fragment, the driver must return an error if the name does not exist. Section 9.1.5 provides additional information on strlen. Section 12.6.2.19 describes the Module_Config_Name field.

5 If the ldbl_stanza_resolver kernel interface returns a value not equal to zero, it did not find matches in the tc_slot table. This condition indicates that no instances of the controller exist on the bus. It returns the constant EINVAL to indicate an invalid argument. Otherwise, ldbl_stanza_resolver allows the device driver to merge the system configuration data specified in the stanza.loadable file fragment into the hardware topology tree created at static configuration time.

The ldbl_stanza_resolver interface takes four arguments:

– The name of the driver specified by the driver writer in the stanza.loadable file fragment

  In this call, the driver name is obtained from the config_name member of the pointer to the indata input data structure.

– The name of the parent bus structure associated with this controller

  In this call, the constant CB_BUSNAME represents the characters ''tc'' indicating that the parent bus structure is a TURBOchannel bus. The bus structure name is obtained from the config program.

– A pointer to the driver structure for the controlling device driver

  In this call, the address of the cbdriver structure is passed.

– A bus-specific parameter

  The bus-specific parameter for a TURBOchannel bus is usually a tc_option snippet table. The snippet table is identical in format to the tc_option table defined in the tc_option_data.c file. In this call, the address of the cb_option_snippet table is passed. This table contains the appropriate entry for the loadable version of the /dev/cb device driver. Section 10.5 shows the declaration of this table.

  Note that a type-casting operation converts cb_option_snippet (which is of type struct tc_option) to be of type caddr_t *, the type of the bus-specific argument. However, ldbl_stanza_resolver does not do anything with this argument but pass it to the bus configuration code, which performs the correct type-casting operation to handle cb_option_snippet. Section 9.6.5 provides additional information on ldbl_stanza_resolver.

6 Calls the ldbl_ctlr_configure interface to cause the driver's cbprobe interface to be called once for each instance of the controller found on the system. If the call to ldbl_ctlr_configure fails, returns the constant EINVAL. The ldbl_ctlr_configure interface takes five arguments:

- The bus name

  In this call, the bus name is represented by the `CB_BUSNAME` constant, which maps to the character string `tc`.

- The bus number

  In this call, the bus number is represented by the wildcard constant `LDBL_WILDNUM`. This wild card allows for the configuration of all instances of the `CB` device present on the system. This constant is defined in the file `/usr/sys/include/io/common/devdriver.h`.

- The name of the controlling device driver

  In this call, this name is obtained from the `config_name` member of the `indata` input data structure.

- A pointer to the `driver` structure for the controlling device driver

  In this call, the controlling device driver is `cbdriver`.

- Miscellaneous flags from `/usr/sys/include/io/common/devdriver_loadable.h`

  In this call, the value zero (0) is passed to indicate that no flags are specified.

7 If the `cbprobe` interface does not find any controllers, sets the variable that keeps count of the number of controllers found to the value zero (0) and returns the constant `EINVAL` to indicate no controllers were found.

8 Calls the `makedev` interface, which makes a device number of type `dev_t` based on the specified major and minor numbers. Upon successful completion, `makedev` returns the major number for this `CB` device in the *cdevno* variable. Note that the driver configuration is performed before obtaining the device major number to prevent user-level programs from gaining access to the `/dev/cb` driver's entry points in the `cdevsw` table.

The `makedev` interface takes two arguments. The first argument is the major number for the device, which in this call is obtained from the `dc_cmajnum` member of the pointer to the `indata` input data structure associated with this device.

The second argument is the minor number for the device, which in this call is also obtained from the `dc_cmajnum` member, indicating that the major and minor numbers are identical. This interface does not make use of the minor number. *Writing Device Drivers, Volume 2: Reference* provides a reference (man) page-style description of `makedev`.

9 Calls the `cdevsw_add` interface to add the driver entry points for the `/dev/cb` driver to the `cdevsw` table. This interface takes two arguments. The first argument specifies the device switch table entry

(slot) to use. This entry represents the requested major number. In this call, the slot to use was obtained in a previous call to makedev. The second argument is the character device switch structure that contains the character device driver's entry points. In this call, this structure is called cb_cdevsw_entry. Upon successful completion, cdevsw_add returns the device number associated with the device switch table. Section 9.6.2 provides additional information on cdevsw_add.

10 If the device number associated with the device switch table is equal to the constant NODEV, returns the error constant ENODEV. The NODEV constant indicates that the requested major number is currently in use or that the cdevsw table is currently full. The NODEV constant is defined in /usr/sys/include/sys/param.h, and /usr/sys/include/sys/errno.h contains the ENODEV constant.

11 Stores the cdevsw table entry slot for this CB device in the *cb_devno* variable. Section 10.9.3 shows that cdevsw_del uses this slot value when the device is unconfigured.

12 This line and the following lines set up the pointer to the outdata output data structure to contain the returned information from the driver configuration. The cfgmgr program uses this information to determine whether the driver was successfully configured and what device special files need to be created. The cfgmgr program initializes the output data structure as follows:

- Sets dc_cmajnum to the major number for this device by calling the major interface. In this call, the number of the device is contained in the *cb_devno* variable. Section 9.9.2 provides additional information on major.
- Sets dc_begunit to the first minor device number in the range. In this case, the first minor number is zero (0).
- Sets dc_numunit to the number of instances of the controller found by the cbprobe interface. In this case, this number is contained in the *num_cb* variable, which was incremented by cbprobe upon locating each controller on the system.
- Sets dc_version to the version of the kernel interfaces that the driver was compiled to. This member is examined by cfgmgr upon driver loading to ensure compatibility. In this call, the version is specified with the constant DRIVER_BUILD_LEVEL, which is defined in /usr/sys/include/sys/sysconfig.h.
- Sets dc_dsflags to the device switch configuration flags that were passed to the input data structure by cfgmgr.
- Sets dc_bmajnum to the constant NODEV. The CB device is a character device and, therefore, has no block device major number.

– Because the `dc_errcode`, `dc_ihflags`, and `dc_ihlevel` members are not used, sets them to the value zero (0).

13 Sets the state flag to indicate that the `/dev/cb` device driver is now configured as a loadable device driver.

10.9.3 Unconfiguring (Unloading) the /dev/cb Device Driver

The following code shows how to implement the unconfiguration of the loadable version of the `/dev/cb` device driver:

```
case SYSCONFIG_UNCONFIGURE: [1]

  if (cb_config != TRUE) {
          return(EINVAL);
  } [2]

  for (i = 0; i < num_cb; i++) {
          if (cb_unit[i].opened != 0) {
                  return(EBUSY);
          } [3]
  }

  for (i = 0; i < num_cb; i++) { [4]
          cb = &cb_unit[i];
          cb->ledflag = 0;
          untimeout(cbincled, (caddr_t)cb);
  }

          retval = cdevsw_del(cb_devno);
          if (retval) {
          return(ESRCH);
          } [5]

  if (cb_is_dynamic) { [6]

          if (ldbl_ctlr_unconfigure(CB_BUSNAME,
                  LDBL_WILDNUM, &cbdriver,
                  LDBL_WILDNAME, LDBL_WILDNUM) != 0) { [7]

                  return(ESRCH);
          }
  }
  cb_config = FALSE; [8]
  break;
```

[1] Specifies the `SYSCONFIG_UNCONFIGURE` constant to indicate that this section of code implements the unconfigure operation of the loadable driver. The file `/usr/sys/include/sys/sysconfig.h` contains the definition of this constant.

[2] If the `/dev/cb` device driver is not currently configured as a loadable driver, fails the unconfiguration by returning the constant `EINVAL`. This error code is defined in `/usr/sys/include/sys/errno.h`.

[3] Prevents the system manager from unloading the device driver, if it is currently active. Check the `opened` member of this `CB` device's `cb_unit` structure to determine if the device is opened and thus active. If the device is opened, return the constant `EBUSY`. This error code is defined in `/usr/sys/include/sys/errno.h`.

4 As long as the variable *i* is less than the number of controllers found by `cbprobe`, executes the following:

- Specifies the `cb_unit` structure associated with this `CB` device.
- Turns off the LED increment function.

  This is accomplished by setting the member of the `cb_unit` structure that stores the LED increment function flag. The reason for turning off this function is to ensure that the driver is quiescent. If this function is not turned off, the `cbincled` interface could be called after its timeout interval expires. If an attempt to execute a driver interface that had already been unloaded is made, a system panic could result.
- Calls the `untimeout` kernel interface to remove the scheduled interface from the callout queues.

  When this call to `untimeout` is made, the driver does not know if there are any pending calls on the callout queues. If there are any pending calls, they are removed from the callout queues. If there are no pending calls, `untimeout` simply returns.

  The `untimeout` interface takes two arguments. The first argument is a pointer to the interface to be removed from the callout queues, which in this call is `cbincled`. The second argument is a single argument passed to the called interface, which is `cbincled`. In this call, this single argument is the pointer to the `cb_unit` structure associated with this `CB` device.

  Note that a type-casting operation converts `cb` (which is of type `cb_unit`) to be of type `caddr_t`, the type of the single argument. Section 9.5.6 provides additional information on `untimeout`.

5 Calls the `cdevsw_del` interface to delete the `/dev/cb` driver's entry points from the `cdevsw` table. This task is done prior to calling `ldbl_ctlr_unconfigure` to prevent access to the device in the middle of unconfiguring the driver. If `cdevsw_del` returns a nonzero value, it returns the error constant `ESRCH` to indicate there was no such slot in the `cdevsw` table. Otherwise, it deletes the driver's entry points. Section 9.6.1 provides additional information on the `cdevsw_del` interface.

The `cdevsw_del` interface takes one argument: the device switch table entry (slot) to use. In this call, the slot is contained in the *`cb_devno`* variable, which was set when the driver was configured.

6 If *`cb_is_dynamic`* evaluates to TRUE, calls the `ldbl_ctlr_unconfigure` interface to unconfigure the specified controller. Section 10.9.2 shows that the *`cb_is_dynamic`* variable was previously set to TRUE (the value 1) when the driver was configured.

7 If the `ldbl_ctlr_unconfigure` interface returns a nonzero value, returns the error constant `ESRCH` to indicate that it did not successfully unconfigure the specified controller. Otherwise, unconfigures the controller. A call to this interface results in a call to the driver's `cb_ctlr_unattach` interface for each instance of the controller. Section 10.8.3 describes `cb_ctlr_unattach`.

The `ldbl_ctlr_unconfigure` interface takes five arguments:

- The bus name

  In this call, the bus name is represented by the constant `CB_BUSNAME`.

- The bus number

  In this call, the wildcard constant indicates that the interface `ldbl_ctlr_unconfigure` deregisters all instances of the controllers connected to the TURBOchannel bus.

- A pointer to the `driver` structure for the controlling device driver

  In this case, the controlling device driver is `cbdriver`.

- The controller name and controller number

  In this call, the wildcard constants indicate that `ldbl_ctlr_unconfigure` scans all `controller` structures. Section 9.6.6 provides additional information on `ldbl_ctlr_unconfigure`.

8 Sets the *`cb_config`* variable to the value FALSE to indicate that the `/dev/cb` device driver is now unconfigured.

10.9.4 Querying the /dev/cb Device Driver

The following code shows how to implement the query section of a loadable device driver. This section of code executes when the system manager requests a query of information associated with the loadable version of the driver.

```
                case SYSCONFIG_QUERY: [1]

                  if (cb_config != TRUE) {
                          return(EINVAL);
                  } [2]
                  outdata->dc_cmajnum = major(cb_devno);
                  outdata->dc_bmajnum = NODEV;
                  outdata->dc_begunit = 0;
                  outdata->dc_numunit = num_cb;
                  outdata->dc_version = DRIVER_BUILD_LEVEL;
                  break;
                default:
                  return(EINVAL); [3]
        }

        return(0); [4]
}
```

[1] Specifies the `SYSCONFIG_QUERY` constant to indicate that this section of code implements the query operation of the loadable driver. The file `/usr/sys/include/sys/sysconfig.h` contains the definition of this constant.

[2] Fails the query if the driver is not currently configured as a loadable driver. Otherwise, sets the following members of the `outdata` output data structure:

- Sets `dc_cmajnum` to the major number for this device by calling the `major` interface. In this call, the number of the device is contained in the *cb_devno* variable. Section 9.9.2 provides additional information on `major`.
- Sets `dc_bmajnum` to the constant `NODEV`. The `CB` device is a character device and, therefore, has no block device major number.
- Sets `dc_begunit` to the first minor device number in the range. In this case, the first minor number is zero (0).
- Sets `dc_numunit` to the number of instances of the controller found by the `cbprobe` interface. In this case, this number is contained in the *num_cb* variable, which was incremented by `cbprobe` upon locating each controller on the system.
- Sets `dc_version` to the version of the kernel interfaces that the driver was compiled to. This member is examined by `cfgmgr` upon driver loading to ensure compatibility. In this call, the version is

specified with the constant DRIVER_BUILD_LEVEL, which is defined in /usr/sys/include/sys/sysconfig.h.

3 Defines an unknown operation type and returns the error constant EINVAL to indicate this condition. This section of code is called if the *op* argument is set to anything other than SYSCONFIG_CONFIGURE, SYSCONFIG_UNCONFIGURE, or SYSCONFIG_QUERY.

4 To indicate that the /dev/cb driver's cb_configure interface completed successfully, returns the value zero (0) is returned.

10.10 Open and Close Device Section

This section is applicable to the loadable or static version of the `/dev/cb` device driver. Table 10-4 lists the two interfaces implemented as part of the Open and Close Device Section along with the sections in the book where each is described.

Table 10-4: Interfaces Implemented as Part of the Open and Close Device Section

Interfaces	Section
Implementing the cbopen Interface	Section 10.10.1
Implementing the cbclose Interface	Section 10.10.2

10.10.1 Implementing the cbopen Interface

This section is applicable to the loadable or static version of the `/dev/cb` device driver. The following code implements the `cbopen` interface:

```
cbopen(dev, flag, format)
dev_t dev;    [1]
int flag;     [2]
int format;   [3]
{
        int unit = minor(dev); [4]
        if ((unit > NCB) || !cb_unit[unit].attached)
                return(ENXIO); [5]
        cb_unit[unit].opened = 1; [6]
        return(0);                [7]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific CB device. The minor device number is used to determine the logical unit number for the CB device that is to be opened.

[2] Declares an argument to contain flag bits from the file `/usr/sys/include/sys/file.h`. These flags indicate whether the device is being opened for reading, writing, or both.

[3] Declares an argument that specifies the format of the special device to be opened. The *format* argument is used by a driver that has both block and character interfaces and uses the same open interface in both the `bdevsw` and `cdevsw` tables. The driver uses this argument to distinguish the type of device being opened. The `cbopen` interface does not use this argument.

[4] Declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument.

[5] If the device minor number is greater than the number of CB devices configured in this system OR if this CB device is not attached, returns the error code ENXIO, which indicates no such device or address. This error code is defined in `/usr/sys/include/sys/errno.h`.

The NCB constant is used in the comparison of the *unit* variable. This constant defines the maximum number of CB devices configured on this system.

The line also checks the `attached` member of this CB device's `cb_unit` structure. Section 10.8.2 shows that the `cbattach` interface set this member to the value 1.

6 If the previous line evaluates to FALSE, sets the opened member of this CB device's cb_unit structure to the value 1 to indicate that this CB device is open and ready for operation. Section 10.6 shows the declaration of the cb_unit data structure.

7 Returns success to the open system call, indicating a successful open of this CB device.

10.10.2 Implementing the cbclose Interface

This section is applicable to the loadable or static version of the `/dev/cb` device driver. The following code implements the `cbclose` interface:

```
cbclose(dev, flag, format)
dev_t dev;  [1]
int flag;   [2]
int format; [3]
{
        int unit = minor(dev);      [4]
        cb_unit[unit].opened = 0; [5]
        return(0);                  [6]
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific CB device. The minor device number is used to determine the logical unit number for the CB device that is to be closed.

[2] Declares an argument to contain flag bits from the file `/usr/sys/include/sys/file.h`. The `cbclose` interface does not use this argument.

[3] Declares an argument that specifies the format of the special device to be closed. The *`format`* argument is used by a driver that has both block and character interfaces and uses the same close interface in both the `bdevsw` and `cdevsw` tables. The driver uses this argument to distinguish the type of device being closed.

The `cbclose` interface does not use this argument.

[4] Declares a *`unit`* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *`dev`* argument.

[5] Sets the `opened` member of this CB device's `cb_unit` structure to the value 0, indicating that this CB device is now closed. Section 10.6 shows the declaration of the `cb_unit` data structure.

[6] Returns success to the `close` system call, indicating a successful close of this CB device.

10.11 Read and Write Device Section

This section is applicable to the loadable or static version of the `/dev/cb` device driver. Table 10-5 lists the interfaces implemented as part of the Read and Write Device Section along with the sections in the book where each is described.

Table 10-5: Interfaces Implemented as Part of the Read and Write Device Section

Interfaces	Section
Implementing the cbread Interface	Section 10.11.1
Implementing the cbwrite Interface	Section 10.11.2

10.11.1 Implementing the cbread Interface

This section is applicable to the loadable or static version of the /dev/cb device driver. The following code implements the cbread interface:

```
cbread(dev, uio, flag)
dev_t dev;          [1]
struct uio *uio; [2]
int flag;
{
        unsigned tmp; [3]
        int cnt, err; [4]
        int unit = minor(dev); [5]
        struct cb_unit *cb;    [6]

        err = 0;                   [7]
        cb = &cb_unit[unit];       [8]
        if(cb->iomode == CBPIO) { [9]

                while((cnt = uio->uio_resid) && (err == 0)) { [10]
                        if(cnt > MAX_XFR)cnt = MAX_XFR; [11]
                        tmp = read_io_port(cb->cbr | CB_DATA,
                                                  4,
                                                  0);[12]


                        err = uiomove(&tmp,cnt,uio); [13]
                        }
                return(err); [14]
                }
        else if(cb->iomode == CBDMA) [15]

           return(physio(cbstrategy,cb->cbbuf,dev,B_READ,cbminphys,uio));
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific CB device. The minor device number is used to determine the logical unit number for the CB device on which the read operation is performed.

[2] Declares 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 pointer unmodified to the uiomove or physio kernel interface. This driver passes the pointer to both interfaces.

[3] Declares a variable called *tmp* to store the 32-bit read/write data register. This variable is passed as an argument to the uiomove kernel interface.

[4] Declares a variable called *cnt* to store the number of bytes of data that still need to be transferred. This variable is passed as an argument to the uiomove interface.

Declares a variable called *err* to store the return value from uiomove.

5 Declares a *unit* variable and initializes it to the device minor number. Note the use of the minor interface to obtain the device minor number.

The minor interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument.

The *unit* variable is used to select the CB board to be accessed for the read operation.

6 Declares a pointer to the cb_unit data structure associated with this CB device and calls it cb. Section 10.6 shows the declaration of cb_unit.

7 Initializes the *err* variable to the value zero (0) to indicate no error has occurred yet.

8 Sets the pointer to the cb_unit structure to the address of the unit data structure associated with this CB device. The *unit* variable contains this CB device's minor number. Thus, this argument is used as an index into the array of cb_unit structures associated with this CB device.

9 If the I/O mode bit is CBPIO, then this is a programmed I/O read operation. This bit is set in the iomode member of the pointer to the cb_unit data structure associated with this CB device.

For a programmed I/O read operation, the contents of the data register on the TURBOchannel test board are read into a 32-bit local variable. Then the contents of that variable are moved into the buffer in the user's virtual address space with the uiomove interface.

10 Sets up a while loop that allows the uiomove interface to transfer bytes from the TURBOchannel test board data register to the user's buffer until all of the requested bytes are moved or until an error occurs.

The uio_resid member of the pointer to the uio structure specifies the number of bytes that still need to be transferred.

This task must be accomplished by the while loop because the TURBOchannel test board data register can supply only a maximum of MAX_XFR bytes at a time. The MAX_XFR constant was previously defined as 4 bytes. This loop may not be required by other devices.

11 If the number of bytes that still need to be transferred is greater than 4, then forces *cnt* to contain 4 bytes of data. This code causes a read of more than 4 bytes to be divided into a number of 4-byte maximum transfers with a final transfer of 4 bytes or less.

12 Reads the 32-bit read/write data register by calling the read_io_port interface. This register value is defined by the CB_DATA device register offset associated with this CB device. The read_io_port interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the read operation. Using this interface to read data from a device register makes the device driver more portable across

different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The read_io_port interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. You can perform standard C mathematical operations on the I/O handle. In this call, the /dev/cb driver ORs the I/O handle with the 32-bit read/write data register represented by CB_DATA.
- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the /dev/cb driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the /dev/cb driver passes the value zero (0).

Upon successful completion, read_io_port returns the data read from the 32-bit read/write data register to the *tmp* variable.

13 Calls the uiomove interface to move the bytes read from the TURBOchannel data register in system virtual space to the user's buffer in user space. The maximum number of bytes moved is 4 and uio_resid is updated as each move is completed. The uiomove interface takes three arguments:

- A pointer to the kernel buffer in system virtual space

  In this call, this pointer is the 32-bit read/write data contained in the *tmp* variable.
- The number of bytes to be moved

  In this call, the number of bytes to move is contained in the *cnt* variable, which is always 4 bytes.
- A pointer to a uio structure

  This structure describes the current position within a logical user buffer in user virtual space. Section 9.3.5 provides additional information on uiomove.

14 Returns a zero (0) value whenever the user virtual space described by the uio structure is accessible and the data is successfully moved. Otherwise, it returns an EFAULT error value.

15 If the I/O mode bit is CBDMA, then this is a DMA I/O read operation. This bit is set in the iomode member of the pointer to the cb_unit data structure associated with this CB device.

For a DMA I/O read operation, the physio kernel interface and the

/dev/cb driver's cbstrategy and cbminphys interfaces are called to transfer the contents of the data register on the TURBOchannel test board into the buffer in the user's virtual address space. Because only a single word of 4 bytes can be transferred at a time, both modes of reading include code to limit the read to chunks with a maximum of 4 bytes each. Reading more than 4 bytes will propagate the contents of the data register throughout the words of the user's buffer.

The physio interface takes six arguments:

- A pointer to the driver's strategy interface

  In this call, the driver's strategy interface is cbstrategy. Section 10.12.2 shows how to set up the cbstrategy interface.

- A pointer to a buf structure

  In this call, the buf structure is the one associated with this CB device. This structure contains information such as the binary status flags, the major/minor device numbers, and the address of the associated buffer. This buffer is always a special buffer header owned exclusively by the device for handling I/O requests. Section 8.1 describes the buf structure.

- The device number, which in this call is contained in the *dev* argument.

- The read/write flag

  In this call the read/write flag is the constant B_READ.

- A pointer to the minphys interface

  In this call, the driver's minphys interface is cbminphys. Section 10.12.1 shows how to set up the cbminphys interface.

- A pointer to a uio structure

10.11.2 Implementing the cbwrite Interface

This section is applicable to the loadable or static version of the /dev/cb device driver. The following code implements the cbwrite interface:

```
cbwrite(dev, uio, flag)
dev_t dev;        [1]
struct uio *uio;  [2]
int flag;
{
        unsigned tmp; [3]
        int cnt, err; [4]
        int unit = minor(dev); [5]
        struct cb_unit *cb;    [6]

        err = 0;                  [7]
        cb = &cb_unit[unit];      [8]
        if(cb->iomode == CBPIO) { [9]

                while((cnt = uio->uio_resid) && (err == 0)) { [10]
                        if(cnt > MAX_XFR)cnt = MAX_XFR; [11]

                        err = uiomove(&tmp,cnt,uio); [12]
                        write_io_port(cb->cbr | CB_DATA,
                                                 4,
                                                 0,
                                                 tmp); [13]
                        }
                return(err); [14]
                }
        else if(cb->iomode == CBDMA) [15]

               return(physio(cbstrategy,cb->cbbuf,dev,B_WRITE,cbminphys,uio));
}
```

[1] Declares an argument that specifies the major and minor device numbers for a specific CB device. The minor device number is used to determine the logical unit number for the CB device on which the write operation is performed.

[2] Declares 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 pointer unmodified to the uiomove or physio kernel interface. This driver passes the pointer to both interfaces.

[3] Declares a variable called *tmp* to store the 32-bit read/write data register. This variable is passed as an argument to the uiomove kernel interface.

[4] Declares a variable called *cnt* to store the number of bytes of data that still need to be transferred. This variable is passed as an argument to the uiomove interface.

Declares a variable called *err* to store the return value from uiomove.

5 Declares a *unit* variable and initializes it to the device minor number. Note the use of the `minor` interface to obtain the device minor number.

The `minor` interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument.

The *unit* variable is used to select the TURBOchannel test board to be accessed for the write operation.

6 Declares a pointer to the `cb_unit` data structure associated with this CB device and calls it `cb`. Section 10.6 shows the declaration of `cb_unit`.

7 Initializes the *err* variable to the value zero (0) to indicate no error has occurred yet.

8 Sets the pointer to the `cb_unit` structure to the address of the unit data structure associated with this CB device. The *unit* variable contains this CB device's minor number. Thus, this argument is used as an index into the array of `cb_unit` structures associated with this CB device.

9 If the I/O mode bit is `CBPIO`, then this is a programmed I/O write operation. This bit is set in the `iomode` member of the pointer to the `cb_unit` data structure associated with this CB device.

For a programmed I/O write operation, the contents of one word from the buffer in the user's virtual address space are moved to a 32-bit local variable by calling the `uiomove` interface. Then the contents of that variable are moved to the data register on the CB test board.

10 Sets up a `while` loop that allows the `uiomove` interface to transfer bytes from the user's buffer to the TURBOchannel test board data register until all of the requested bytes are moved or until an error occurs.

The `uio_resid` member of the pointer to the `uio` structure specifies the number of bytes that still need to be transferred.

This task must be accomplished by the `while` loop because the TURBOchannel test board data register can accept only a maximum of `MAX_XFR` bytes at a time. The `MAX_XFR` constant was previously defined as 4 bytes. This loop may not be required by other devices.

11 If the number of bytes that still need to be transferred is greater than 4, then forces *cnt* to contain 4 bytes of data. This code causes a write of more than 4 bytes to be divided into a number of 4-byte maximum transfers with a final transfer of 4 bytes or less.

12 Calls the `uiomove` interface to move the bytes from the user's buffer to the local variable, *tmp*. The maximum number of bytes moved is 4 and `uio_resid` is updated as each move is completed. The `uiomove` interface takes the same three arguments as described for `cbread`.

[13] Writes the data to the 32-bit read/write data register by calling the `write_io_port` interface. This register value is defined by the `CB_DATA` device register offset associated with this `CB` device. The `write_io_port` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the write operation. Using this interface to write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The `write_io_port` interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. In this call, the `/dev/cb` driver ORs the I/O handle with the 32-bit read/write data register represented by `CB_DATA`.
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the `/dev/cb` driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the `/dev/cb` driver passes the value zero (0).
- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the `/dev/cb` driver passes the value stored in the *`tmp`* variable. Section 10.11.1 shows that the `read_io_port` interface stored this value in the *`tmp`* variable.

[14] Returns a zero (0) value whenever the user virtual space described by the `uio` structure is accessible and the data is successfully moved. Otherwise, it returns an `EFAULT` error value.

[15] If the I/O mode bit is `CBDMA`, then this is a DMA I/O write operation. This bit is set in the `iomode` member of the pointer to the `cb_unit` data structure associated with this `CB` device.

For a DMA I/O write operation, the `physio` kernel interface and the `/dev/cb` driver's `cbstrategy` and `cbminphys` interfaces are called to transfer the contents of the buffer in the user's virtual address space to the data register on the TURBOchannel test board. Because only a single word of 4 bytes can be transferred at a time, both modes of reading include code to limit the write to chunks with a maximum of 4 bytes. Writing more than 4 bytes has limited usefulness because all the words in the user's buffer will be written into the single data register on the test board.

This call to the `physio` interface takes the same arguments as those passed to `cbread` except the read/write flag is `B_WRITE` instead of `B_READ`.

10.12 Strategy Section

Table 10-6 lists the tasks associated with implementing the Strategy Section along with the sections in the book where each task is described.

Table 10-6: Tasks Associated with Implementing the Strategy Section

Tasks	Section
Setting Up the cbminphys Interface	Section 10.12.1
Setting Up the cbstrategy Interface	Section 10.12.2
Initializing the buf Structure for Transfer	Section 10.12.2.1
Testing the Low Order Two Bits and Using the Internal Buffer	Section 10.12.2.2
Converting the Buffer Virtual Address	Section 10.12.2.3
Converting the 32-Bit Physical Address	Section 10.12.2.4
Starting I/O and Checking for Time Outs	Section 10.12.2.5

10.12.1 Setting Up the cbminphys Interface

This section is applicable to the loadable or static version of the `/dev/cb` device driver. The following code sets up the `cbminphys` interface, whose major task is to bound the data transfer size to four bytes:

```
cbminphys(bp)
register struct buf *bp; [1]
{
        if (bp->b_bcount > MAX_XFR) [2]
                bp->b_bcount = MAX_XFR;
        return;
}
```

[1] Declares a pointer to a `buf` structure and calls it `bp`. Section 8.1 describes the `buf` structure.

[2] If the size of the requested transfer is greater than 4 bytes, sets the `b_bcount` member of `bp` to 4 bytes and returns.

The `b_bcount` member stores the size of the requested transfer (in bytes).

In the call to `physio`, the driver writer passes `cbminphys` as the interface to call to check for size limitations. Section 10.11.1 and Section 10.11.2 discuss the call to `physio`.

10.12.2 Setting Up the cbstrategy Interface

This section is applicable to the loadable or static version of the /dev/cb device driver. It sets up the cbstrategy interface, which performs the following tasks:

- Initializes the buf structure for data transfer
- Tests the low-order 2 bits and uses the internal buffer
- Performs a DMA write
- Converts the buffer virtual address
- Converts the 32-bit physical address
- Starts I/O and checks for timeouts

Section 10.11.1 and Section 10.11.2 show that cbread and cbwrite call the cbstrategy interface. The following code implements the cbstrategy interface:

```
cbstrategy(bp)
register struct buf *bp; 1
{
        register int unit = minor(bp->b_dev); 2
        register struct controller *ctlr; 3
        struct cb_unit *cb;     4
        caddr_t buff_addr;      5
        caddr_t virt_addr;      6
        unsigned phys_addr;     7
        int cmd;                8
        int err;                9
        int status;             10
        unsigned lowbits;       11
        unsigned tmp;           12
        int s;                  13

        ctlr = cbinfo[unit]; 14
```

1 Declares a pointer to a buf structure and calls it bp. Section 8.1 describes the buf structure.

2 Gets the minor device number and stores it in the *unit* variable.

The minor kernel interface is used to obtain the minor device number associated with this CB device. The minor interface takes one argument: the number of the device for which the minor device needs to be obtained. In this call, the device number is stored in the b_dev member of the buf structure associated with this CB device.

3 Declares a pointer to a controller structure and calls it ctlr. Section 7.4 describes the controller structure.

4 Declares a pointer to the cb_unit data structure associated with this CB device and calls it cb. Section 10.6 shows the declaration of cb_unit.

5 Declares a variable called *buff_addr* that stores the user buffer's virtual address. Section 10.12.2.2 shows that *buff_addr* is passed to the copyin kernel interface. Section 10.12.2.5 shows that *buff_addr* is passed to the copyout kernel interface.

6 Declares a variable called *virt_addr* that stores the user buffer's virtual address. Section 10.12.2.2 shows that *virt_addr* is passed to the copyin kernel interface. Section 10.12.2.3 shows that *virt_addr* is passed to the vtop kernel interface. Section 10.12.2.5 shows that *virt_addr* is passed to the copyout kernel interface.

7 Declares a variable called *phys_addr* that stores the user buffer's physical address. Section 10.12.2.3 shows that this variable stores the value returned by the vtop kernel interface.

8 Declares a variable called *cmd* that stores the current command for the TURBOchannel test board. Section 10.12.2.4 shows that the commands are represented by the constants CB_DMA_RD and CB_DMA_WR.

9 Declares a variable called *err* that stores the error status returned by cbstart.

10 Declares a variable called *status* to store the value associated with the 16-bit read/write CSR/LED register. This value is obtained by calling the read_io_port interface. This variable is used by cbstrategy in a CB_DEBUG statement, which is shown in Section B.2.

11 Declares a variable called *lowbits* to store the low 2 virtual address bits. Section 10.12.2.2 and Section 10.12.2.5 show how this variable is used.

12 Declares a temporary holding variable called *tmp*. Section 10.12.2.4 shows that this variable is used by the CB_SCRAMBLE macro.

13 Declares a temporary holding variable called *s*. Section 10.12.2.4 shows that the splbio kernel interface uses this variable to store its return value.

14 Sets the pointer to the controller structure to its associated CB device. Note that *unit*, which now contains this CB device's minor device number, is used as an index into the array of controller structures to obtain the controller structure associated with this device.

10.12.2.1 Initializing the buf Structure for Transfer

The following code initializes the `buf` structure for transfer:

```
bp->b_resid = bp->b_bcount; 1
bp->av_forw = 0;            2

cb = &cb_unit[unit];        3

virt_addr = bp->b_un.b_addr; 4
buff_addr = virt_addr;       5
```

1 Initializes the bytes not transferred. This is done in case the transfer fails at a later time. The `b_resid` member of the pointer to the `buf` structure stores the data (in bytes) not transferred because of some error. The `b_bcount` member stores the size of the requested transfer, in bytes.

2 Clears the buffer queue forward link. The `av_forw` member stores the position on the free list if the `b_flags` member is not set to `B_BUSY`.

3 Sets the pointer to the `cb_unit` structure to the address of the unit data structure associated with this CB device. The *`unit`* variable contains this CB device's minor number. Thus, this argument is used as an index into the array of `cb_unit` structures associated with this CB device. Section 10.12.2 shows how the `minor` interface initializes *`unit`* to the device's minor number.

4 Sets the *`virt_addr`* variable to the buffer's virtual address. The operating system software sets this address in the `b_addr` member of the union member `b_un` in the pointer to the `buf` structure.

5 Copies the buffer's virtual address into the *`buff_addr`* variable for use by the driver.

10.12.2.2 Testing the Low Order Two Bits and Using the Internal Buffer

Direct memory access (DMA) on the TURBOchannel test board can be done only with full words and must be aligned on word boundaries. Because the user's buffer can be aligned on any byte boundary, the `/dev/cb` driver code must check for and handle the cases where the buffer is not word aligned. In this context, word aligned means that the address is evenly divisible by 4, where a word is a 4-byte entity. Any address that is word aligned has its lowest order 2 bits set to zeroes. (If the TURBOchannel interface hardware included special hardware to handle nonword-aligned transfers, this checking would not have to be performed.) If the user's buffer is not word aligned, the driver can:

- Exit with an error
- Take some action to assure the words are aligned on the transfer

Because virtual-to-physical mapping is done on a page basis, the low-order 2 bits of the virtual address of the user's buffer are also the low 2 bits of the physical address of the user's buffer. The buffer alignment can be determined by examining the low 2 bits of the virtual buffer address. If these 2 bits are nonzero, the buffer is not word aligned and the driver must take the desired action.

The following code tests the low-order 2 bits:

```
if ((lowbits = (unsigned)virt_addr & 3) != 0) { [1]
        virt_addr = (caddr_t)(&tmpbuffer);  [2]

        if ( !(bp->b_flags&B_READ) ) { [3]
                tmpbuffer = 0 ;

                if (err = copyin(buff_addr,virt_addr,bp->b_resid)) { [4]
                        bp->b_error = err;        [5]
                        bp->b_flags |= B_ERROR;   [6]
                        iodone(bp);               [7]
                        return;                   [8]
                }
        }
}
```

[1] This bitwise AND operation uses the low-order 2 bits of the buffer virtual address as the word-aligned indicator for this transfer. If the result of the bitwise AND operation is nonzero, the user's buffer is not word aligned and the next line gets executed.

Section 10.12.2.1 shows that the *`virt_addr`* variable gets set to the buffer's virtual address. Because *`virt_addr`* is of type `caddr_t` and *`lowbits`* is of type `unsigned`, the code performs the appropriate type-casting operation.

[2] Because the user's buffer is not word aligned, use the internal buffer, *`tmpbuffer`*. This line replaces the current user buffer virtual address

with the internal buffer virtual address. The `physio` kernel interface updates the current user buffer virtual address as each word is transferred. Because DMA to the TUROBchannel test board can be done only a word at a time, the internal buffer needs to be only a single word.

3 If the transfer type is a write, clears the one-word temporary buffer *`tmpbuffer`*.

4 Calls the `copyin` kernel interface to copy data from the user address space to the kernel address space. The `copyin` kernel interface takes three arguments:

- The address in user space of the data to be copied

  In this call, the address of the data is stored in the *`buff_addr`* variable, which is set in Section 10.12.2.1.

- The address in kernel space to copy the data to

  In this call, the address in kernel space is stored in the *`virt_addr`* variable, which is set in Section 10.12.2.2.

- The number of bytes to copy

  In this call, the number of bytes to copy is stored in the `b_resid` member of the pointer to the `buf` structure. Section 10.12.2.1 shows the initialization of this member.

5 Upon success, `copyin` returns the value zero (0). Otherwise, it returns `EFAULT` to indicate that the address specified in *`buff_addr`* could not be accessed. This line sets the `b_error` member of the pointer to the `buf` structure to the value returned in *`err`*. Section 10.11.2 shows that *`err`* was initialized to the value zero (0) to indicate that no error has yet occurred.

6 Sets `b_flags` to the bitwise inclusive OR of the read and error bits.

7 Calls the `iodone` kernel interface to indicate that the I/O operation is complete. This interface takes one argument: a pointer to a `buf` structure. Section 9.9.1 provides additional information on the `iodone` kernel interface.

8 Returns with an error to `physio`, which was called by `cbwrite`. Section 10.11.2 shows the call to `physio`.

10.12.2.3 Converting the Buffer Virtual Address

The following code for the `cbstrategy` interface converts the buffer virtual address to a physical address for DMA by calling the `vtop` interface. In previous versions of this book, the `/dev/cb` driver made calls to the `IS_KSEG_VA`, `KSEG_TO_PHYS`, `IS_SEG0_VA`, `pmap_kernel`, and `pmap_extract` interfaces to accomplish this task. It now accomplishes this task by making one call to `vtop`.

```
phys_addr = vtop(bp->b_proc, virt_addr); [1]
```

[1] Converts the buffer virtual address to a physical address for DMA. This interface takes two arguments:

- The first argument specifies a pointer to a `proc` structure. The `vtop` interface uses the `proc` structure pointer to obtain the pmap. In this call, the `/dev/cb` driver passes the `b_proc` member of the `buf` structure pointer associated with this CB device. The `b_proc` member specifies a pointer to the `proc` structure that represents the process performing the I/O.
- The second argument specifies the virtual address that `vtop` converts to a physical address. In this call, the `/dev/cb` driver passes the value stored in *`virt_addr`*.

Upon successful completion, `vtop` returns the physical address associated with the specified kernel virtual address.

10.12.2.4 Converting the 32-Bit Physical Address

The following code uses the CB_SCRAMBLE macro to convert the 32-bit physical address to a form suitable for use in the DMA operation:

```
tmp = CB_SCRAMBLE(phys_addr); [1]
write_io_port(cb->cbr | CB_ADDER,
              4,
              0,
              tmp); [2]

if(bp->b_flags&B_READ)      [3]
        cmd = CB_DMA_WR;
else
        cmd = CB_DMA_RD;

s = splbio();               [4]
```

[1] Converts the 32-bit physical address (actually the low 32 bits of the 34-bit physical address) from the linear form to the condensed form used by DMA to pack 34 address bits onto 32 board lines. TURBOchannel DMA can be done only with full words and must be aligned on word boundaries. The CB_SCRAMBLE macro discards the low-order 2 bits of the physical address while scrambling the rest of the address. Therefore, anything that is going to be done to resolve this address must be done before calling CB_SCRAMBLE.

[2] Writes the data to the 32-bit read/write DMA address register by calling the write_io_port interface. This register value is defined by the CB_ADDER device register offset associated with this CB device. The write_io_port interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the write operation. Using this interface to write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The write_io_port interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. In this call, the /dev/cb driver ORs the I/O handle with the 32-bit read/write DMA address register represented by CB_ADDER.
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the /dev/cb driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the /dev/cb driver passes the value zero (0).

- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the `/dev/cb` driver passes the value returned by `CB_SCRAMBLE` in the *`tmp`* variable.

3 If the read bit is set, initializes the *cmd* argument to the DMA write bit, which is defined in the `cbreg.h` file. This bit indicates a write to memory.

Otherwise, if the read bit is not set, initializes the *cmd* argument to the DMA read bit, which also is defined in the `cbreg.h` file. This bit indicates a read from memory.

4 Calls the `splbio` interface to mask (disable) all controller interrupts. The value returned by `splbio` is an integer value that represents the CPU priority level that existed prior to the call. The return value stored in *`s`* becomes the argument passed to `splx`, which is discussed in Section 10.12.2.5.

10.12.2.5 Starting I/O and Checking for Time Outs

The following code starts the I/O and checks for timeouts:

```
        err = cbstart(cmd,cb);        [1]
        splx(s);                      [2]

        if(err <= 0) { [3]

                bp->b_error = EIO;
                bp->b_flags |= B_ERROR;
                iodone(bp);
                return;                       [4]
                }
        else { [5]

                if ( (lowbits)!=0 && bp->b_flags&B_READ) { [6]

                        if (err = copyout(virt_addr,buff_addr,bp->b_resid)) { [7]
                                bp->b_error = err;
                               bp->b_flags |= B_ERROR;
                        }
                }
                bp->b_resid = 0;    [8]
            }
        iodone(bp);                    [9]
        return;
}
```

[1] Starts the I/O by calling the driver's `cbstart` interface, passing to it the current command for the test board and the address of the `cb_unit` data structure associated with this CB device.

Section 10.13 describes the `cbstart` driver interface.

The *cmd* argument is set either to `CB_DMA_WR` (the write to memory bit) or to `CB_DMA_RD` (the read from memory bit).

[2] Restores the CPU priority by calling the `splx` kernel interface, passing to it the value returned in a previous call to `splbio`. This value is an integer that represents the CPU priority level that existed before the call to `splbio`.

[3] If the return value from `cbstart` is the value zero (0), the DMA operation did not complete within the timeout period. In this case, do the following:

- Set the `b_error` member in the `buf` structure pointer to indicate that an I/O error occurred. This error flag, `EIO`, is defined in `/usr/sys/include/sys/errno.h`.
- Set the `b_flags` member in the `buf` structure pointer to indicate that an error occurred on this data transfer.
- Call the `iodone` kernel interface to indicate that the I/O transfer is complete. The `iodone` interface takes one argument: a pointer to a

buf structure. The iodone interface reschedules the process that initiated the I/O.

4 Returns the error status.

5 Else, executes the following lines because the DMA completed successfully.

6 If a read was attempted to an unaligned user buffer, calls copyout to copy the bytes that were read into the user buffer.

7 If the copyout kernel interface was unable to copy data from kernel address space to user address space, do the following:

   - Set the b_error member in the buf structure pointer to the value returned by copyout. This value indicates that the kernel address specified in the first argument could not be accessed or that the number of bytes to copy specified in the third argument is invalid.
   - Set the b_flags member in the buf structure pointer to indicate an error occurred on this data transfer.

8 Sets the b_resid member in the buf structure pointer to the value zero (0) to indicate that the read or write operation has completed.

9 Calls iodone to indicate that the I/O transfer is complete and to initiate the return status.

10.13 Start Section

The `cbstart` interface is applicable to both the loadable and static versions of the `/dev/cb` device driver. The interface's main task is to load the CSR register of the TURBOchannel test board. Because the `cbincled` interface increments the LEDs in the high 4 bits of the 16-bit CSR register, `cbstart` always loads the 4 bits into whatever value it will be storing into the CSR before doing the actual storage operation. The `cbstart` interface is called with system interrupts disabled; thus, `cbincled` is not called while `cbstart` is incrementing.

The following shows the implementation of the `cbstart` interface:

```
int cbstart(cmd,cb)
int cmd;              1
struct cb_unit *cb;   2
{
        int timecnt; 3
        int status;  4

        cmd = (read_io_port(cb->cbr | CB_CSR,
                            4,
                            0)&0xf000)|(cmd&0xfff); 5

        status = read_io_port(cb->cbr | CB_TEST,
                              4,
                              0); 6


        write_io_port(cb->cbr | CB_CSR,
                      4,
                      0,
                      cmd); 7
        wbflush();            8

        write_io_port(cb->cbr | CB_TEST,
                      4,
                      0,
                      0); 9

        wbflush();                  10
        timecnt = 10;               11
        status = read_io_port(cb->cbr | CB_CSR,
                              4,
                              0); 12

        while((!(status & CB_DMA_DONE)) && timecnt > 0) { 13
                write_io_port(cb->cbr | CB_CSR,
                              4,
                              0,
                              cmd);

                wbflush();
                status = read_io_port(cb->cbr | CB_CSR,
                                      4,
                                      0);
                timecnt --;
                }

        return(timecnt); 14
}
```

1 Declares a variable to contain the current command for the test board. Section 10.12.2.4 shows that *cmd* was set to CB_DMA_WR or CB_DMA_RD.

2 Declares a pointer to the cb_unit data structure associated with this CB device and calls it cb. Section 10.6 shows the declaration of cb_unit.

3 Declares a variable to contain the timeout loop count.

4 Declares a variable to contain the CSR contents for status checking.

5 Sets the *cmd* variable to the logical or high 4 LED bits and the command to be performed by calling the read_io_port interface. The read_io_port interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the read operation. Using this interface to read data from a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. Reads the test register by calling the read_io_port interface. On the CB device, reading the test register clears the go bit. The test register is defined by the CB_TEST device register offset associated with this CB device. The read_io_port interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. You can perform standard C mathematical operations on the I/O handle. In this call, the /dev/cb driver ORs the I/O handle with the go bit represented by CB_TEST.
- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the /dev/cb driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the /dev/cb driver passes the value zero (0).

Upon successful completion, read_io_port returns the data read from the go bit register to the *status* variable.

6 Reads the test register by calling the read_io_port interface. On the CB device, reading the test register clears the go bit. This call to read_io_port is the same as the previous call except that here the /dev/cb driver ORs the I/O handle with the go bit represented by the

CB_TEST device register offset.

7 Writes the data to the specified location by calling the write_io_port interface. This location is the result of the ORing of the I/O handle with the 16-bit read/write CSR/LED register value. This register value is defined by the CB_CSR device register offset associated with this CB device. The write_io_port interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the write operation. Using this interface to write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The write_io_port interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. In this call, the /dev/cb driver ORs the I/O handle with the 16-bit read/write CSR/LED register represented by CB_CSR.
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the /dev/cb driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the /dev/cb driver passes the value zero (0).
- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the /dev/cb driver passes the value stored in the *cmd* variable.

8 Calls the wbflush kernel interface to ensure that a write to I/O space has completed.

9 Writes the value zero (0) to the specified location by calling the write_io_port interface. This call is the same as the previous call except for the values passed to the first and fourth arguments. For the first argument, the location is the result of ORing the I/O handle with the go bit device register offset. This register value is defined by the CB_TEST device register offset associated with this CB device. For the fourth argument, the data to be written is the value zero (0). Writing zero (0) to this device register has the effect of setting the go bit.

10 The wbflush interface is called a second time to ensure that a write to I/O space has completed.

11 Initializes the timeout loop counter variable, *timecnt*, to the value 10.

12 Reads the status for this CB device from the specified location by calling the read_io_port interface. This call passes the same values as a previous call. For the first argument, the location of the read operation is the result of ORing the I/O handle with the 16-bit read/write CSR/LED register represented by CB_CSR.

13 Spins until the DMA completes or the timeout loop counter expires. Then:

- Writes a value to the specified location by calling the write_io_port interface. This call is the same as previous calls. For the first argument, the location is the result of ORing the I/O handle with the 16-bit read/write CSR/LED device register offset. This register value is defined by the CB_CSR device register offset associated with this CB device. For the fourth argument, the data to be written is stored in the *cmd* variable.
- Calls wbflush a third time to ensure that a write to I/O space has completed.
- Reads the status for this CB device from the specified location by calling the read_io_port interface. This call passes the same values as a previous call. For the first argument, the location of the read operation is the result of ORing the I/O handle with the 16-bit read/write CSR/LED register represented by CB_CSR.
- Decrements the counter

14 Returns the timeout count. If the command was successful, cbstart returns a nonzero value. If the loop exits because of a timeout, cbstart returns a zero (0) value.

10.14 The ioctl Section

Table 10-7 lists the tasks associated with implementing the The ioctl Section, along with the sections in the book where each task is described.

Table 10-7: Tasks Associated with Implementing the The ioctl Section

Part	Section
Setting Up the cbioctl Interface	Section 10.14.1
Incrementing the Lights	Section 10.14.2
Setting the I/O Mode	Section 10.14.3
Performing an Interrupt Test	Section 10.14.4
Returning a ROM Word, Updating the CSR, and Stopping Increment of the Lights	Section 10.14.5

10.14.1 Setting Up the cbioctl Interface

This section is applicable to the loadable or static version of the `/dev/cb` device driver. The following code sets up the `cbioctl` interface:

```
#define CBIncSec  1  [1]

cbioctl(dev, cmd, data, flag)
dev_t dev;              [2]
unsigned int cmd;       [3]
int *data;              [4]
int flag;               [5]
{
        int tmp;                     [6]
        int *addr;                   [7]
        int timecnt;                 [8]
        int unit = minor(dev);       [9]
        struct cb_unit *cb;          [10]
        int cbincled();              [11]

        cb = &cb_unit[unit];         [12]
```

[1] Defines a constant called `CBIncSec` that indicates the number of seconds between increments of the TURBOchannel test board lights. Section 10.14.2 shows that this constant is passed to the `timeout` kernel interface.

[2] Declares an argument that specifies the major and minor device numbers for a specific `CB` device. The minor device number is used to determine the logical unit number for the `CB` device on which the `ioctl` operation is to be performed.

[3] Declares an argument that specifies the `ioctl` command in the file `/usr/sys/include/sys/ioctl.h` or in another include file defined by the device driver writer. There are two types of `ioctl` commands. One type is supported by all drivers of a given class. Another type is specific to a given device. The values of the *cmd* argument are defined by using the `_IO`, `_IOR`, `_IOW`, and `_IOWR` macros. Section 10.2 shows that the following `ioctl` commands are defined in the `cbreg.h` file: `CBPIO`, `CBDMA`, `CBINT`, `CBROM`, `CBCSR`, `CBINC`, and `CBSTP`.

[4] Declares a pointer to `ioctl` command-specific data that is to be passed to the device driver or filled in by the device driver. This argument is a kernel address. The size of this data cannot exceed the size of a page. At least 128 bytes is guaranteed. Any size between 128 bytes and the page size may fail if memory cannot be allocated. The particular `ioctl` command implicitly determines the action to be taken. The `ioctl` system call performs all the necessary copy operations to move data to and from user space.

Section 10.14.5 shows how `cbioctl` initializes the *`data`* argument.

5 Declares an argument that specifies the access mode of the device. This argument is not used by the /dev/cb driver.

6 Declares a temporary holding variable called *tmp*.

7 Declares a pointer to a variable called *addr* that is used for word access to the TURBOchannel test board. Section 10.14.5 shows that this variable is used with the *data* argument.

8 Declares a variable called *timecnt*. Section 10.14.4 shows that this variable is used in the timeout loop count.

9 Declares a *unit* variable and initializes it to the device minor number. Note the use of the minor interface to obtain the device minor number.

The minor interface takes one argument: the number of the device for which an associated device minor number will be obtained. The minor number is encoded in the *dev* argument.

The *unit* variable is used to select the TURBOchannel test board to be accessed for the ioctl operation.

10 Declares a pointer to the cb_unit data structure associated with this CB device and calls it cb. Section 10.6 shows the declaration of cb_unit.

11 Declares a forward reference to the cbincled interface. Section 10.15 shows the implementation of cbincled.

12 Sets the pointer to the cb_unit structure to the address of the unit data structure associated with this CB device.

10.14.2 Incrementing the Lights

The following code starts incrementing the lights on the TURBOchannel test board:

```
switch(cmd&0xFF) {       1
      case CBINC:           2
             if(cb->ledflag == 0) {  3
                 cb->ledflag++;  4
                 timeout(cbincled, (caddr_t)cb, CBIncSec*hz); 5
             }
             break;
```

1 Uses the *cmd* argument to perform the appropriate `ioctl` operation.

2 When *cmd* evaluates to `CBINC`, the `ioctl` operation starts incrementing the lights on the TURBOchannel test board.

3 If the increment function has not started, executes the next two lines. This line of code determines the start of the increment function by checking the `ledflag` member of the `CB` structure associated with this `CB` device.

4 Sets the flag for the LED increment function.

5 Starts the timer by calling the `timeout` kernel interface. The `timeout` kernel interface takes three arguments:

- The first argument is a pointer to the interface to call, which in this case is `cbincled`.
- The second argument is a single argument to be passed to the interface specified by the first argument when it is called. In this example, the single argument is the pointer to the `cb_unit` data structure associated with this `CB` device. Because the second argument to `timeout` is of type `caddr_t`, the code performs the appropriate type-casting operation.
- The third argument is the amount of time to delay before calling the `cbincled` interface. The constant `CBIncSec` represents some amount of time in seconds.

The `timeout` interface initializes a callout queue element. Section 9.5.5 provides additional information on `timeout`.

10.14.3 Setting the I/O Mode

The following code sets the I/O mode for the `ioctl` operations to either programmed I/O or DMA I/O:

```
case CBPIO:                          [1]
        cb->iomode = CBPIO;
        break;
case CBDMA:                          [2]
        cb->iomode = CBDMA;
        break;
```

[1] When *cmd* evaluates to `CBPIO`, the `ioctl` operation sets the I/O mode to programmed I/O for this `CB` device.

[2] When *cmd* evaluates to `CBDMA`, the `ioctl` operation sets the I/O mode to DMA I/O for this `CB` device.

10.14.4 Performing an Interrupt Test

The following code tests the interrupt operation:

```
case CBINT:                     1
        timecnt = 10;           2
        cb->intrflag = 0;       3

        tmp = read_io_port(cb->cbr | CB_TEST,
                           4,
                           0); 4

        tmp = CB_INTERUPT|(read_io_port(cb->cbr | CB_CSR,
                           4,
                           0)&0xf000); 5
        write_io_port(cb->cbr | CB_CSR,
                      4,
                      0,
                      tmp); 6
        wbflush();              7
        write_io_port(cb->cbr | CB_TEST,
                      4,
                      0,
                      1); 8
        wbflush();              9

        while ((cb->intrflag == 0) && (timecnt > 0)) { 10
                write_io_port(cb->cbr | CB_CSR,
                              4,
                              0,
                              tmp);
                wbflush();
                timecnt --;
        }
        tmp = read_io_port(cb->cbr | CB_TEST,
                           4,
                           0); 11

        return(timecnt == 0);   12
```

1 When *cmd* evaluates to `CBINT`, the `ioctl` operation performs an interrupt test.

2 Initializes the timeout loop count variable, *`timecnt`*, to the value 10.

3 Clears the interrupt flag by setting the `intrflag` member of the `cb_unit` structure associated with this `CB` device to the value zero (0). Section 10.14.1 shows the declaration of the pointer to the `cb_unit` data structure called `cb`.

4 Clears the go bit by calling the `read_io_port` interface. The `read_io_port` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the read operation. Using this interface to read data from a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The `read_io_port` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. You can perform standard C mathematical operations on the I/O handle. In this call, the `/dev/cb` driver ORs the I/O handle with the go bit device register represented by `CB_TEST`.
- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the `/dev/cb` driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the `/dev/cb` driver passes the value zero (0).

Upon successful completion, `read_io_port` returns the data read from the go bit device register to the *`tmp`* variable.

5 Performs a bitwise inclusive OR operation that assigns the 16-bit read/write CSR/LED register to the temporary holding variable. This operation uses two values. The first value is represented by the constant `CB_INTERUPT`. Section 10.2 shows that this constant is currently defined as `0x0e00`. The second value is the result of the bitwise AND operation of the value returned by `read_io_port` and the value `0xf000`. These 4 bits contain the current LED state.

The result of these operations produces the value, which is assigned to the *`tmp`* variable.

6 Calls the `write_io_port` interface to load enables and LEDs. The `write_io_port` interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. In this call, the `/dev/cb` driver ORs the I/O handle with the 16-bit read/write CSR/LED register represented by `CB_CSR`.
- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. In this call, the `/dev/cb` driver passes the value 4.
- The third argument specifies flags to indicate special processing requests. In this call, the `/dev/cb` driver passes the value zero (0).
- The fourth argument specifies the data to be written to the specified device register in bus address space. In this call, the `/dev/cb` driver

passes the value stored in the *tmp* variable. This value is the bit calculated by the `CB_INTERUPT` macro and the current LED state.

7 Calls the `wbflush` kernel interface to ensure that a write to I/O space has completed.

8 Writes the value 1 to the specified location by calling the `write_io_port` interface. This call is the same as previous calls. For the first argument, the location is the result of ORing the I/O handle with the go bit device register offset. This register value is defined by the `CB_TEST` device register offset associated with this `CB` device. For the fourth argument, the data to be written is the value 1.

9 Calls `wbflush` again to ensure that a write to I/O space has completed.

10 Note that the interrupt flag, `cb->intrflag`, is set to a nonzero value by `cbintr` if an interrupt is received. See Section 10.16 for a discussion of `cbintr`.

While the interrupt flag is equal to the value zero (0) and the timeout loop count variable is greater than zero (0), executes the following statements:

- Updates the status of the 16-bit read/write CSR/LED register by calling the `write_io_port` interface.
- Calls `wbflush` again to ensure that a write to I/O space has completed.
- Decrements the timeout loop count variable.

This section of code executes until the interrupt flag is set and the timeout loop counter expires.

11 Ensures that the go bit is cleared by calling the `read_io_port` interface.

12 Returns to the `ioctl` system call. If the interrupt was started before the timeout loop count expired, `cbioctl` returns a zero (0) value to indicate success. If the timeout count expires, `cbioctl` returns a nonzero (1) value to indicate failure.

10.14.5 Returning a ROM Word, Updating the CSR, and Stopping Increment of the Lights

The following code returns a ROM word, updates the CSR, and then stops incrementing the lights on the TURBOchannel test board:

```
                case CBROM&0xFF:              1
                       tmp = *data;           2
                       if(tmp < 0 || tmp >= 32768*4+4*4) 3
                               return(-tmp);
                       tmp <<= 1;
                       addr = (int *)&(cb->cbad[tmp]); 4
                       *data = *addr;         5
                       break;
                case CBCSR&0xFF:                        6
                       write_io_port(cb->cbr | CB_CSR,
                                     4,
                                     0,
                                     read_io_port(cb->cbr | CB_CSR,
                                                  4,
                                                  0)); 7
                        wbflush();            8
                        *data = read_io_port(cb->cbr | CB_CSR,
                                             4,
                                             0); 9
                        break;
                case CBSTP:                   10
                        cb->ledflag = 0;
                        break;
                }
        return(0); 11
}
```

1 When *cmd* evaluates to CBROM, the `ioctl` operation returns a ROM word for this CB device by executing the statements from 2 – 5.

2 Gets the specified byte offset from the argument that is a kernel address.

3 If the byte offset is not in the valid range of 32k words + 4 registers, returns the byte offset to the `ioctl` system call to indicate that it is out of range.

4 Gets the ROM base address from the `cbad` member of the CB structure associated with this CB device.

The `cbad` member provides the base address of the ROM. Because `cbad` is type cast as an `int *`, the *tmp* variable is used as an index to determine how many bytes to go into the ROM, and the resulting address is used to fetch the contents.

5 Returns the word from the TURBOchannel test board.

6 When *cmd* evaluates to CBCSR, the `ioctl` operation updates and returns the CSR for this CB device by executing the statements from 7 – 9.

7 Reads from and writes to this CB device's 16-bit read/write CSR/LED register by calling the `read_io_port` and `write_io_port` interfaces.

8 Calls the `wbflush` kernel interface to ensure that a write to I/O space has completed.

9 Returns the CSR from the TURBOchannel test board by calling the `read_io_port` interface.

10 When *cmd* evaluates to `CBSTP`, the `ioctl` operation stops incrementing the lights on the next timeout by clearing the LED increment function flag.

11 Upon successful completion, `cbioctl` returns the value zero (0) to the `ioctl` system call.

10.15 Increment LED Section

The cbincled interface is applicable to the loadable or static versions of the /dev/cb device driver. It is called by the softclock kernel interface CBIncSec seconds after the last timeout call. If the increment flag is still set, cbincled increments the pattern in the high four LEDs of the LED/CSR register and restarts the timeout to recall later.

The following code shows the implementation of the cbincled interface:

```
cbincled(cb)
struct cb_unit *cb; [1]
{
                int tmp;

                tmp = read_io_port(cb->cbr | CB_CSR,
                                   4,
                                   0);
                                   tmp -= 0x1000;
                write_io_port(cb->cbr | CB_CSR,
                              4,
                              0,
                              tmp); [2]
     if(cb->ledflag != 0) {    [3]
             timeout(cbincled, (caddr_t)cb, CBIncSec*hz);
             }
     return;
}
```

[1] Declares a pointer to the cb_unit data structure associated with this CB device and calls it cb. Section 10.6 shows the declaration of cb_unit.

This argument is specified in the callout to the timeout interface.

[2] Calls the read_io_port and write_io_port interfaces to increment the lights. Because the LEDs are on when a bit is zero (0), a subtraction is done to accomplish the increment.

[3] If the increment function flag is still set, restarts the timer by calling the timeout kernel interface and returns to softclock.

The increment function flag is stored in the ledflag member of the cb_unit data structure associated with this CB device.

The timeout kernel interface takes three arguments:

- The first argument is a pointer to the interface to call, which in this case is cbincled.
- The second argument is a single argument to be passed to the interface specified by the first argument when it is called. In this example, the single argument is the pointer to the cb_unit data structure associated with this CB device. Because the second argument to timeout is of type caddr_t, the code performs the

appropriate type-casting operation.

- The third argument is the amount of time to delay before calling the `cbincled` interface. The constant `CBIncSec` represents some amount of time in seconds.

The `timeout` interface initializes a callout queue element. Section 9.5.5 provides additional information on `timeout`.

10.16 Interrupt Section

The `cbintr` interface is applicable to the loadable or static versions of the `/dev/cb` device driver. The interface's tasks are to clear the go bit and set a flag to indicate that an interrupt occurred. The following code shows the implementation of the `cbintr` interface:

```
cbintr(ctlr)
int ctlr; [1]
{
        int tmp; [2]
        struct cb_unit *cb;     [3]
        cb = &cb_unit[ctlr];    [4]
        tmp = read_io_port(cb->cbr | CB_TEST,
                           4,
                           0); [5]
        cb->intrflag++;         [6]

        return; [7]
}
```

[1] Declares a variable to contain the controller number, which is passed in by the operating system interrupt code.

[2] Declares a temporary variable to hold the go bit.

[3] Declares a pointer to the `cb_unit` data structure associated with this CB device and calls it `cb`. Section 10.6 shows the declaration of `cb_unit`.

[4] Sets the pointer to the `cb_unit` structure to the address of the unit data structure associated with this CB device. The *`ctlr`* argument is used as an index into the array of `cb_unit` structures associated with this CB device.

[5] Calls the `read_io_port` interface to read the test register to clear the go bit. The `read_io_port` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the read operation. Using this interface to read data from a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture. The `read_io_port` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. You can perform standard C mathematical operations on the I/O handle. In this call, the `/dev/cb` driver ORs the I/O handle with the go bit device register represented by `CB_TEST`.
- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms

support all of these values. In this call, the /dev/cb driver passes the value 4.

  – The third argument specifies flags to indicate special processing requests. In this call, the /dev/cb driver passes the value zero (0).

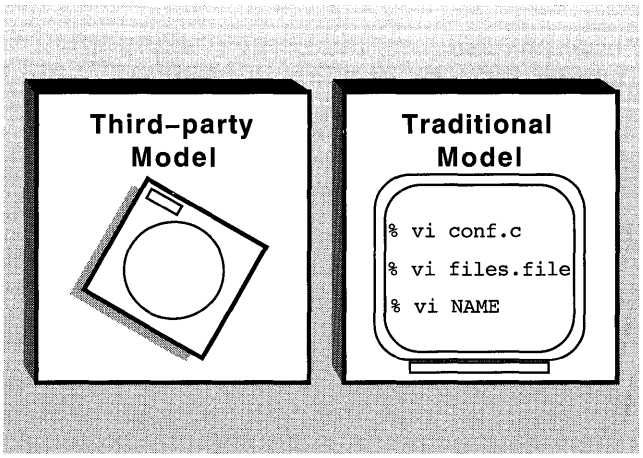
Upon successful completion, read_io_port returns the data read from the go bit device register to the *tmp* variable.

6 Sets the interrupt flag to indicate that an interrupt occurred.

  The flag value is contained in the intrflag member of the cb_unit data structure associated with this CB device.

7 Returns to the operating system interrupt code.

Part 6 Device Driver Configuration



Device Driver Configuration Models 11

Device driver configuration is the process of incorporating device drivers into the kernel and making them available to system management and other utilities. Do not confuse device driver configuration with device autoconfiguration. The former is for incorporating device drivers into the kernel; the latter is what occurs when the kernel boots. Chapter 7 describes device autoconfiguration.

The DEC OSF/1 operating system provides two models for configuring device drivers:

- The third-party device driver configuration model

  This model is recommended for third-party device driver writers who want to ship loadable and static drivers to customers running Digital's DEC OSF/1 operating system. Customers then use a variety of system management utilities that automatically configure the static and loadable drivers.

- The traditional device driver configuration model

  This model is suitable for driver writers (or system managers) who simply want to configure static and loadable drivers, without going through the kit-building process. For example, device driver writers developing a driver in a classroom setting may want to use this model. This model is also suitable for driver writers following the third-party model during the initial stages of driver development. Driver writers or system managers manually perform many of the tasks that otherwise are automated in the third-party device driver configuration model.

The following sections describe each of these models. This chapter uses a fictitious device driver development company called EasyDriver Incorporated to illustrate the approach third-party driver developers can take to configure their drivers.

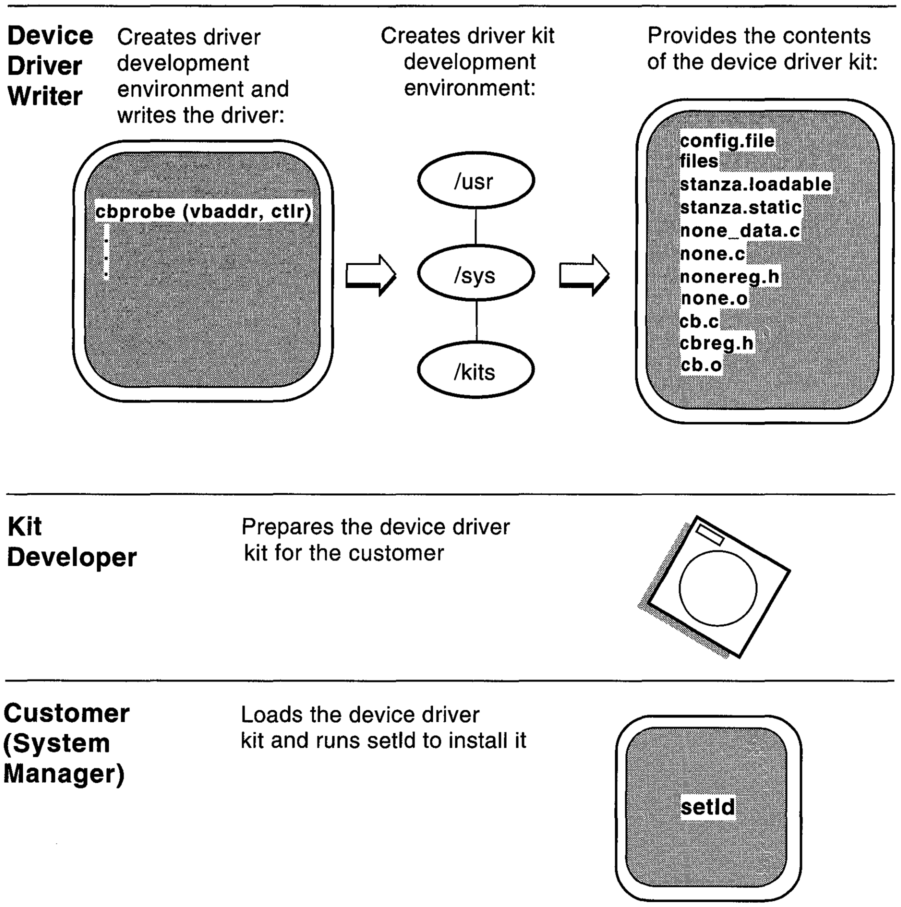
11.1 Third-Party Device Driver Configuration Model

The third-party device driver configuration model provides tools that customers use to automate the installation of third-party device drivers. These customers can use the automated mechanism to:

- Install the device driver any time after the installation of the operating system
- Install the device driver without manual edits to system files
- Install static or loadable device drivers
- Integrate into the driver configuration site-specific tasks that are not currently handled by the `config` program

This model requires that third-party driver writers provide a device driver kit to their customers. Figure 11-1 shows the tasks and groups of people involved in the third-party device driver kit delivery process.

Figure 11-1: Third-Party Device Driver Kit Delivery Process


Device Driver Writer
Creates driver development environment and writes the driver:
cbprobe (vbaddr, ctlr)
Creates driver kit development environment:
/usr
/sys
/kits
Provides the contents of the device driver kit:
config.file
files
stanza.loadable
stanza.static
none_data.c
none.c
nonereg.h
none.o
cb.c
cbreg.h
cb.o
Kit Developer
Prepares the device driver kit for the customer
Customer (System Manager)
Loads the device driver kit and runs setld to install it
setld

The figure shows that the third-party device driver kit delivery process involves at least three different audiences: the device driver writer, the kit developer, and the system manager (who, from the device driver writer's point of view, is the customer). In addition to showing the groups of people, Figure 11-1 also shows the tasks each group does to deliver the device driver:

- The device driver writer creates a driver development environment and writes and tests the driver.
- The device driver writer creates a driver kit development environment.
- The device driver writer provides the contents of the device driver kit.

- The kit developer prepares the device driver kit.
- The customer loads the device driver kit and runs `setld`.

The following sections describe these tasks in more detail. Because these tasks encompass more than one group of people, the following discussions explicitly identify the group typically associated with the specific tasks.

11.1.1 Creating a Driver Development Environment and Writing the Driver

The first task of the driver writer is to write the device driver, using the design and development techniques described in the previous chapters of this book. During the initial stages of driver development, driver writers are probably not interested in going through the kit-building process. Therefore, during the initial driver development stages driver writers can follow the traditional device driver configuration model described in Section 11.2. This model presents a development environment that makes it possible for driver writers to design, write, test, rewrite, and configure the driver without going through the kit building process.

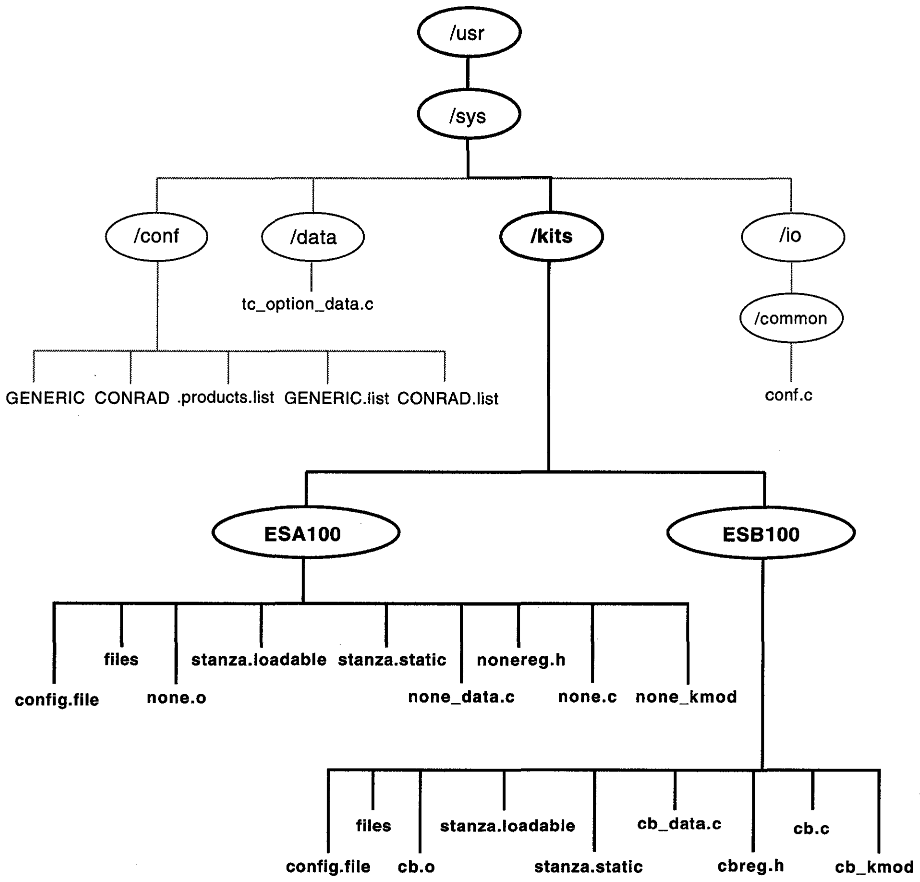
The driver writers at EasyDriver Incorporated perform their initial driver development by using the traditional model. Their development directory structure is discussed in Section 11.2.

11.1.2 Creating a Driver Kit Development Environment

When all device driver testing (following the traditional model) is complete, the driver writer works with the kit developer to create a driver kit development environment. The driver writers for EasyDriver Incorporated work with their kit developers to create the kit development environment shown in Figure 11-2.

A directory structure, such as the one shown in Figure 11-2, helps the kit developer to prepare the kit and the driver writer to test the driver under conditions similar to those experienced by customers. Figure 11-2 contrasts the directories and files related to the system with the directories and files related to the driver kit development environment.

Figure 11-2: Driver Kit Development Environment for EasyDriver Incorporated



The directories and files related to EasyDriver Incorporated's system are as follows:

- The `/usr/sys/conf` directory

  This directory contains files that define the kernel configuration for the generic (all-encompassing) and target machine kernels. In the `/usr/sys/conf` directory for EasyDriver Incorporated, the generic kernel is called `GENERIC`, and the target machine kernel is called CONRAD. This directory also contains the `.products.list` file and, optionally, a `GENERIC.list` or `CONRAD.list` file.

- A `/usr/sys/data` directory

  This directory contains the `name_data.c` files supplied by Digital. An example of one such file is `tc_option_data.c`.

- A `/usr/sys/io/common` directory

  This directory contains the `conf.c` template file, which contains the `bdevsw` and `cdevsw` tables.

The directories and files related to EasyDriver Incorporated's driver kit development environment (identified in darker type in the figure) are as follows:

- A `/usr/sys/kits` directory

  This directory contains subdirectories that represent each of the driver products developed by EasyDriver Incorporated. Figure 11-2 shows two such directories: `ESA100` and `ESB100`.

  Driver writers can follow the naming conventions described in the *Programming Support Tools* book. The driver writers at EasyDriver Incorporated adhere to these conventions by specifying directory names based on three-character product and version codes. Thus, the driver writers at EasyDriver Incorporated use the product codes `ESA` and `ESB` to represent the directories that contain the files associated with the `/dev/none` and `/dev/cb` drivers. The version code `100` indicates that this is Version 1.0 of the driver products.

- A `/usr/sys/kits/ESA100` directory

  This directory contains the configuration-related file fragments, driver load modules (for loadable drivers), driver object files (for static drivers), and, optionally, driver source code for the `/dev/none` device driver that EasyDriver Incorporated ships to their customers.

- A `/usr/sys/kits/ESB100` directory

  This directory contains the configuration-related file fragments, driver load modules (for loadable drivers), driver object files (for static drivers), and, optionally, driver source code for the `/dev/cb` device driver that EasyDriver Incorporated ships to their customers.

The following sections describe these files.

11.1.2.1 The /usr/sys/conf/NAME File

The `/usr/sys/conf/`*`NAME`* file (referred to as the system configuration file) is an ASCII text file that defines the components of the system. The system configuration file name, *NAME*, is usually the name of the system. By convention, the system configuration file name is capitalized. There can be more than one system configuration file defined in `/usr/sys/conf`, each with a capitalized name. As Figure 11-2 shows, EasyDriver Incorporated has two system configuration files: `GENERIC` and `CONRAD`. Customers, too, can have more than one system configuration file to represent different configurations.

The `GENERIC` system configuration file supplied by Digital contains all the possible software and hardware options available to DEC OSF/1 systems and includes all supported Digital devices. The `GENERIC` system configuration file is used to build a kernel that represents all possible combinations of statically configured drivers that Digital supports. This kernel is booted during the operating system installation and is often referred to as the generic kernel. While running the generic kernel, the installation software determines which subset of all possible device drivers should be used to build a target kernel to match the hardware attached to the system being installed.

The installation software builds a tailored system configuration file to match the hardware present by calling the `sizer` program. This tailored system configuration file is later used by `doconfig` to create a tailored kernel.

Device driver writers following the third-party model do not supply complete system configuration files to their customers. Rather, they supply entries in the `config.file` file fragment that is located in the vendor-specific directory and is a logical extension to the system configuration file found in `/usr/sys/conf`. For the `/dev/cb` driver product, the vendor-specific directory is `/usr/sys/kits/ESB100`. Section 12.2 describes the syntaxes driver writers use to specify the necessary information in the `config.file` file fragment.

11.1.2.2 The .products.list and NAME.list Files

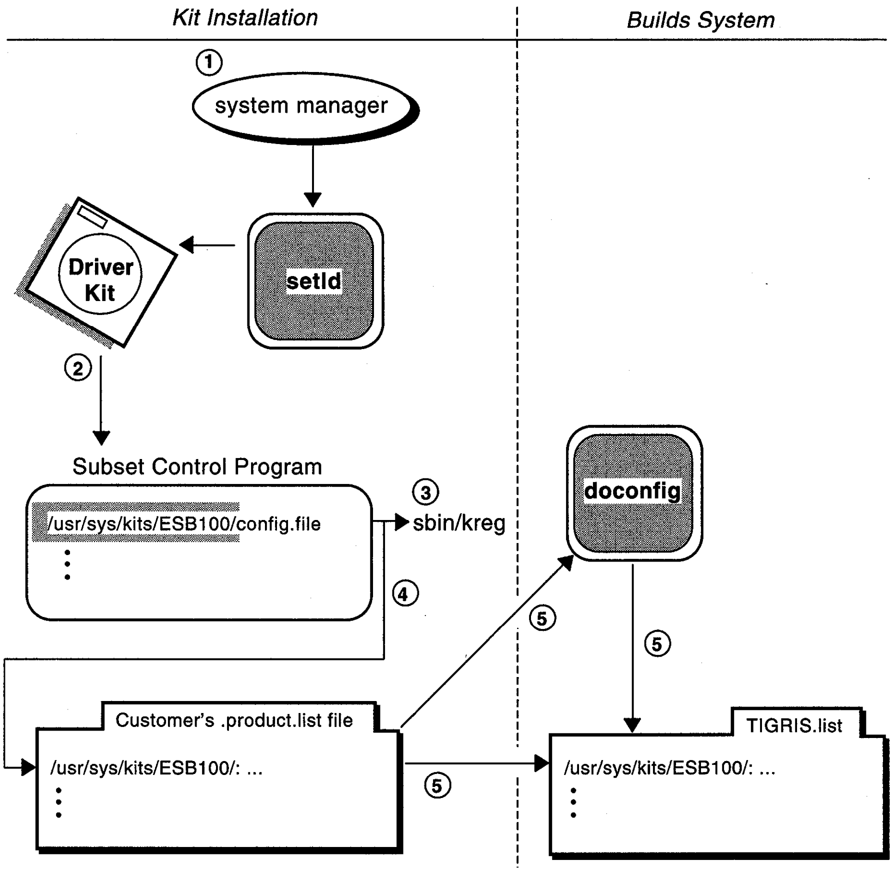
The `/usr/sys/conf/.products.list` file (for static drivers) stores information about static device driver layered products. The *`NAME`*`.list` file is a copy of the `.products.list` file that is created when the system manager installs the device driver kit supplied by a third-party vendor. Device driver writers and kit developers do not supply either of these files; however, an understanding of these files can help during third-party kit development and testing.

Figure 11-3 shows the relationship between these two files during kit installation:

1. The system manager loads the device driver kit and runs the `setld` utility.
2. The `setld` utility reads the device driver kit and calls the subset control program (SCP) provided by the kit developer. The SCP contains path specifications for all of the files related to the driver product. For example, Figure 11-3 shows the path and one file associated with the `/dev/cb` driver product.

3. The SCP calls the `/sbin/kreg` utility, which registers the product on the customer's system. This action makes the device driver product available to system management-related programs such as `doconfig` and `config`.

Figure 11-3: Comparison of .products.list File and NAME.list



4. As the figure shows, the SCP calls `/sbin/kreg` and supplies it with the driver path to where the files related to the `/dev/cb` driver product are located. The `/sbin/kreg` utility registers the path along with other supporting information into the customer's `.products.list` file. The three dots in the figure indicate that the supporting information is contained in other fields in the entry. The key piece of information for the driver writer and the kit developer is the location of the files associated with the driver product.

5. The doconfig program, run by the system manager, reads the .products.list file and copies it to a file of the form *NAME*.list . The *NAME* variable usually specifies the name of the system configuration file. For example, TIGRIS.list would be the name for the file that contains information about static device drivers for the customer system described by the system configuration file called TIGRIS.

The fields contained in the /usr/sys/conf/.products.list file are described in Section 12.4.

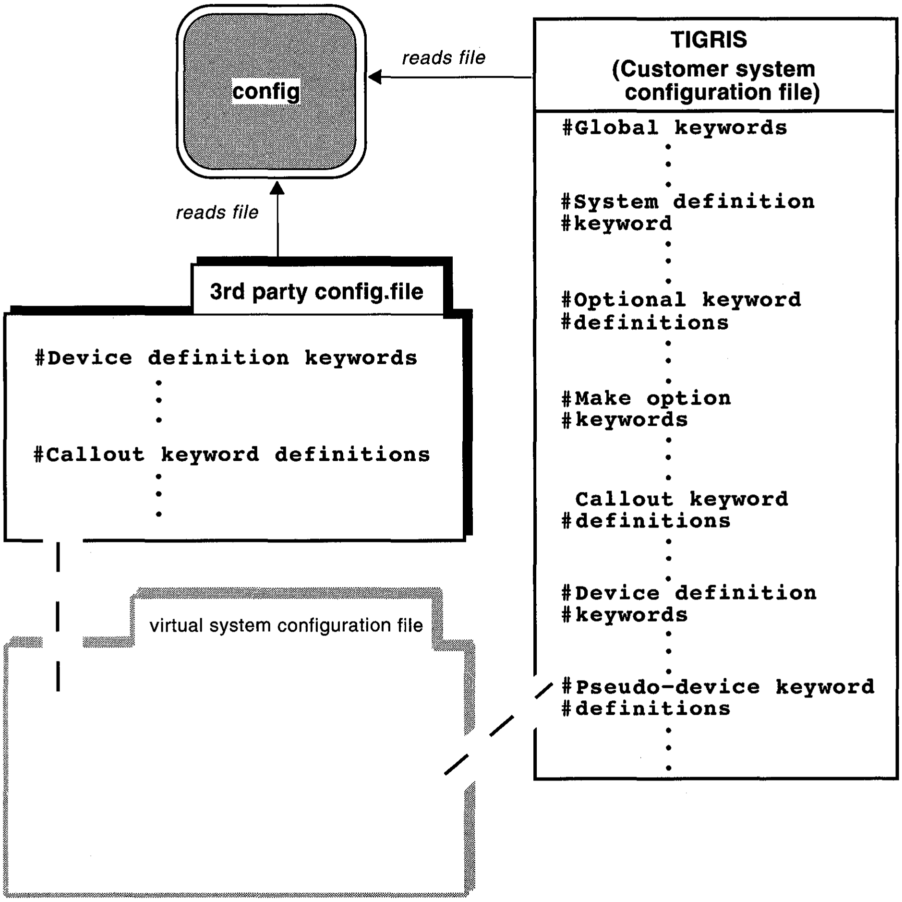
Customers can edit the *NAME*.list file to exclude a driver entry, thus removing the entry's associated functionality from the rebuilt kernel. Otherwise, customers always get the driver products as they are specified in the /usr/sys/conf/.products.list file. Customers should never edit the /usr/sys/conf/.products.list file directly. Instead, customers make required changes by using the kreg utility or by editing the *NAME*.list file.

11.1.2.3 The config.file File Fragment

The config.file file fragment (for static drivers) can be viewed as a ''mini'' system configuration file, as shown in Figure 11-4. The figure shows the relationship between a config.file file fragment from EasyDriver Incorporated and a customer system configuration file called TIGRIS. The config.file file fragment from EasyDriver Incorporated contains device definition keywords and callout keyword definitions for the static driver products. The customer's system configuration file contains not only these categories of keywords, but also additional ones. Although config.file can also contain these other keywords, the driver writers at EasyDriver Incorporated specify only keywords related to their device driver product and needed by their customers.

Furthermore, Figure 11-4 shows that the config program run by the customer reads the information contained in both files and makes the options specified in them available to the system. The figure shows that config does not append the information in config.file to the system configuration file, but rather creates a virtual system configuration file of the entries contained in both files. The config program does not alter the supplied third-party config.file file fragment or the customer's system configuration file.

Figure 11-4: Comparison of config.file File Fragment and System Configuration File



However, driver writers must be particularly careful about choosing appropriate naming conventions for such items as the device connectivity information to avoid name conflicts with other third-party driver vendors. One suggestion is to use extensions to names that minimize the chances for conflict. These extensions could include any combination of the company initials, product name, version number, and release number. For example, the driver writers at EasyDriver Incorporated might specify `edcb` for an internally developed device. The prefix `ed` represents the company name and `cb` is the name of the device.

Section 12.2 discusses the syntaxes used by driver writers to populate the `config.file` file fragment with device- and callout-related information.

11.1.2.4 The files File

The `files` file fragment (for static drivers) can be viewed as a ''mini'' `files` file, as shown in Figure 11-5. The figure shows the relationship between a `files` file fragment from EasyDriver Incorporated and a customer's `files` file. The `files` file fragment (for static drivers) from EasyDriver Incorporated contains the following information about the static driver products:

- The location of the source code associated with the product drivers
- Tags indicating when the product drivers are to be loaded into the kernel
- Whether the source code or the binary form of the product drivers is supplied to the customer

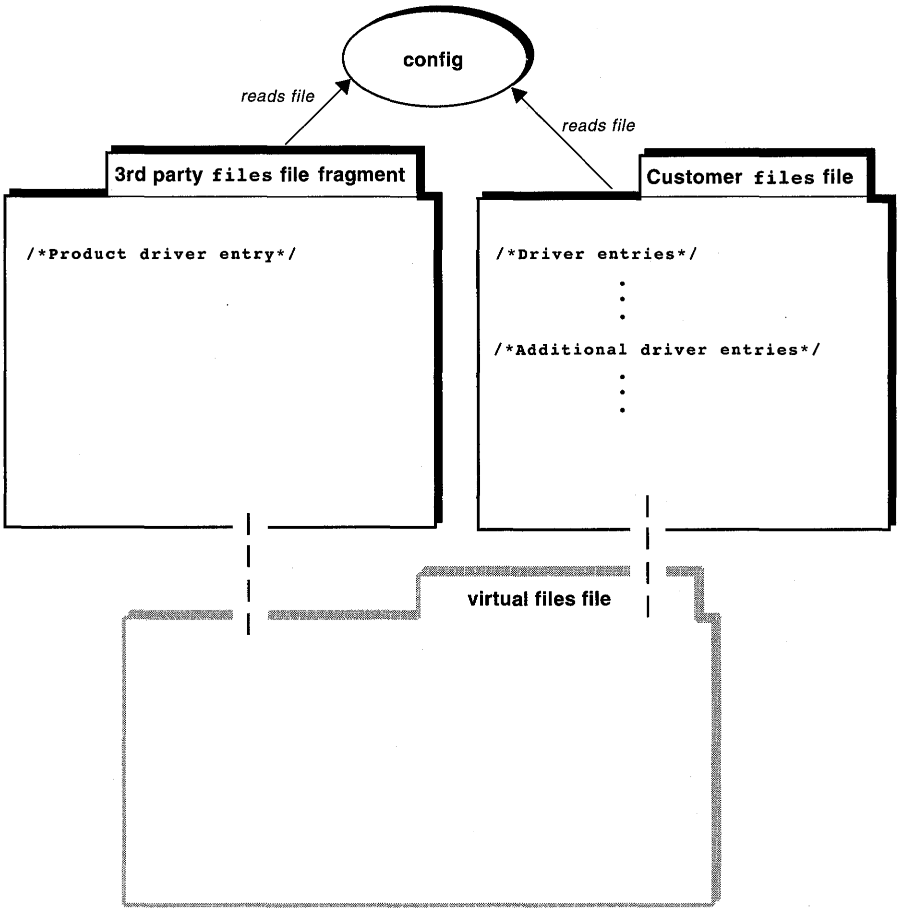
The customer's `files` file contains similar entries for other device drivers. To have the `config` program automatically generate the appropriate rules to build the drivers, the driver writer must specify the appropriate information in the `files` file fragment.

Figure 11-5 shows that the `config` program run by the customer reads both `files` files and makes the information specified in them available to the system. The figure also shows that `config` does not append the information in the supplied `files` file fragment to the customer's `files` file, but rather creates a virtual `files` file of the entries contained in both files. It does not alter the supplied third-party `files` file fragment or the customer's `files` file.

This automated mechanism relieves the customer from having to make tedious and potentially error-prone changes to the `files` file. Driver writers must be particularly careful about choosing unique path and file names in this file.

Section 12.5 describes the syntaxes used to specify the previously described information.

Figure 11-5: Comparison of files File Fragment and Customer's files File



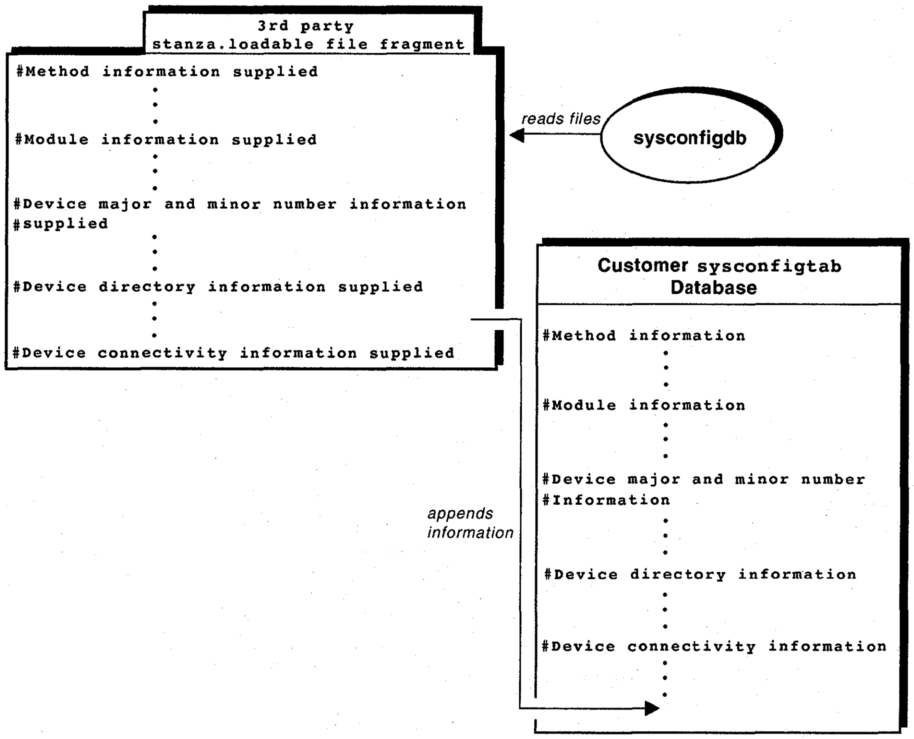
11.1.2.5 The stanza.loadable File Fragment

The `stanza.loadable` file fragment (for loadable drivers) can be viewed as a ‘‘mini’’ `sysconfigtab` database, as shown in Figure 11-6. The figure shows the relationship between a `stanza.loadable` file fragment from EasyDriver Incorporated and a customer `/etc/sysconfigtab` database. The `stanza.loadable` file fragment from EasyDriver

Incorporated contains such items as device connectivity information, the driver's major number requirements, the names and minor numbers of the device special files, and the permissions and directory name where the device special files reside. The customer's `/etc/sysconfigtab` database contains not only these categories of information, but also additional ones that can represent other loadable drivers. The driver writers at EasyDriver Incorporated specify only options related to their device driver product and needed by their customers.

Furthermore, Figure 11-6 shows that the `sysconfigdb` utility appends the information contained in the `stanza.loadable` file fragment to the customer's `/etc/sysconfigtab` database.

Figure 11-6: Comparison of stanza.loadable File Fragment and sysconfigtab Database



3rd party
stanza.loadable file fragment
#Method information supplied
#Module information supplied
#Device major and minor number information
#supplied
#Device directory information supplied
#Device connectivity information supplied
reads files
sysconfigdb
Customer sysconfigtab Database
#Method information
#Module information
#Device major and minor number
#Information
#Device directory information
#Device connectivity information
appends information

This automated mechanism relieves the customer from having to make tedious and potentially error-prone changes to the `/etc/sysconfigtab` database. However, driver writers must be particularly careful about choosing appropriate naming conventions for such items as the device connectivity information to avoid name conflicts with other third-party driver vendors. One suggestion is to use extensions to names that minimize the chances for conflict. These extensions could include any combination of the company initials, product name, version number, and release number. For example, the driver writers at EasyDriver Incorporated might specify `edcb` for an internally developed device. The prefix `ed` represents the company name and `cb` is the name of the device.

Section 12.6 discusses the syntaxes that driver writers use to populate the `stanza.loadable` file fragment.

11.1.2.6 The stanza.static File Fragment

The `stanza.static` file fragment (for static drivers) contains such items as the driver's major number requirements, the names and minor numbers of the device special files, the permissions and directory name where the device special files reside, and the driver interface names to be added to the `bdevsw` and `cdevsw` tables.

The `kmknod` utility uses the information in this file fragment to dynamically create device special files for static device drivers.

Section 12.6 discusses the syntaxes that driver writers use to populate the `stanza.static` file fragment.

11.1.2.7 The name_data.c File

The *name*`_data.c` file (for static drivers) provides a convenient place to size the data structures and data structure arrays that static device drivers use. In addition, this file can contain definitions that customers can change. The *name* argument is usually based on the device name. For example, the `none` device's *name*`_data.c` file is called `none_data.c`. The `CB` device's *name*`_data.c` file is called `cb_data.c`. The `edgd` device's *name*`_data.c` file would be called `edgd_data.c`. See Section 3.1.5 for more information on this file and how it relates to loadable drivers.

11.1.2.8 Files Related to the Device Driver Product

The driver writer also supplies files related to the device driver product. For the static device driver product, these can include the driver header files, which have `.h` extensions; source files, which have `.c` extensions; and the driver object files, which have `.o` extensions.

The device driver object files are the actual executables associated with the static device driver. To supply the device driver objects and the device driver source code, driver writers specify the valid syntaxes in a `files` file fragment. Section 12.5 discusses these syntaxes.

For the loadable device driver product the driver writer supplies the driver load modules, which have `_kmod` extensions.

11.1.2.9 The conf.c File

The `/usr/sys/io/common/conf.c` file contains the `bdevsw` and `cdevsw` tables. The method for adding device driver interfaces to these tables differs according to whether the driver is static or loadable:

- For static drivers

  The driver writer provides the driver interfaces in the `stanza.static` file fragment. The `config` program reads the `stanza.static` file fragment to obtain these interface names and then automatically adds them to these tables for the driver product. Section 12.8 describes how to provide this information for static drivers.

- For loadable drivers

  Loadable drivers use the `devsw` interfaces to add the interfaces to the `bdevsw` and `cdevsw` tables. Section 9.6 describes how to use these interfaces.

- For drivers implemented as static and loadable

  If the driver is implemented as both a loadable and a static driver, the driver writer implements the `devsw` interfaces and specifies the driver interface names in the `stanza.static` file fragment. During the installation of the product drivers, an appropriate prompt would request that the customer specify whether the driver is to be installed as a loadable or static driver. The device driver kit can provide a software subset for the static device driver product and a software subset for the loadable driver product. The customer can then choose to install either the static or the loadable driver product.

11.1.3 Providing the Contents of the Device Driver Kit

The driver writer provides specific items that the kit developer uses as the contents of the device driver kit. These items consist of the files and file fragments discussed in the previous sections. These files and file fragments contain information necessary for system managers (customers) to configure the loadable or static drivers into their systems. Table 11-1 summarizes the files and file fragments the kit developer needs to supply on the device driver kit. The table has the following columns:

- File

  Contains the name of the file or file fragment.

- Static drivers (source)

  A ''Yes'' appears in this column if the file or file fragment should be supplied with static drivers supplied as source. Otherwise, a ''No'' appears.

- Static drivers (binary)

  A ''Yes'' appears in this column if the file or file fragment should be supplied with static drivers supplied as binary. Otherwise, a ''No'' appears.

- Loadable drivers (binary)

  A ''Yes'' appears in this column if the file or file fragment should be supplied with loadable drivers supplied as binary. Otherwise, a ''No'' appears.

Note

The third-party device driver configuration model does not currently support shipping loadable drivers as source and building them at the customer site. Therefore, the table omits a column for loadable drivers (source).

Table 11-1: Contents of Device Driver Kit

File	Static Drivers (Source)	Static Drivers (Binary)	Loadable Drivers (Binary)
`config.file` file fragment	Yes	Yes	No
`files` file fragment	Yes	Yes	No
`stanza.loadable` file fragment	No	No	Yes
`stanza.static` file fragment	Yes	Yes	No
`name``_data.c`	Yes	Yes	No
driver headers	Yes	No*	No
driver sources	Yes	No	No
driver objects	No	Yes	No
driver load modules	No	No	Yes

Table 11-1: (continued)

File	Static Drivers (Source)	Static Drivers (Binary)	Loadable Drivers (Binary)
Subset Control Program (SCP)	Yes	Yes	Yes

* But Yes if needed by data.c

11.1.4 Preparing the Device Driver Kit

The kit developer prepares the distribution medium, which in Figure 11-1 is a CDROM. See the *Programming Support Tools* book for a discussion of the media supported by Digital's `setld` architecture.

In this book, the distribution medium is referred to as the device driver kit. It consists of a hierarchical group of the files and directories provided by the device driver writer. It is the responsibility of the driver writer to work with the kit developer to determine how the files and directories are grouped within the hierarchy. The driver writers for EasyDriver Incorporated worked with their kit developers to create the directory structure shown in Figure 11-2.

The kit developer performs the appropriate tasks for preparing the device driver kit to be used with the `setld` utility. One of these tasks is to write a subset control program (SCP) that installs and manages software subsets. In addition, this SCP calls `/sbin/kreg` (for static drivers), which is the utility that registers a subset as a kernel build module and fills the `/usr/sys/conf/.products.list` file. For loadable drivers, the SCP calls the `sysconfigdb` utility, which maintains and manages the `/etc/sysconfigtab` database.

The kit developer should refer to the *Programming Support Tools* book, which contains information about preparing software distribution kits that are compatible with the `setld` utility. The `setld` utility installs and manages DEC OSF/1 software kits and layered product kits.

Section 12.9 provides an example SCP written by EasyDriver Incorporated.

11.1.5 Loading the Distribution Medium and Running setld

To install the device driver kit at the customer site, the system manager (customer) loads the device driver kit and runs `setld`. This utility transfers the contents of the kit to the customer's file system at a known location, as determined by the device driver kit. The `setld` utility then calls the SCP supplied by the third-party vendor.

If the driver can be both statically configured and dynamically loaded, the SCP must prompt the system manager to choose. If the system manager chooses static, the SCP calls the `kreg` utility. The `kreg` utility creates a `/usr/sys/conf/.products.list` file or updates an existing one. If the system manager chooses loadable, the SCP calls the `sysconfigdb` utility. The `sysconfigdb` utility maintains and manages the `/etc/sysconfigtab` database.

Both the `kreg` and `sysconfigdb` utilities examine the files that were transferred from the device driver kit to determine if the system has the subset loaded or if it has previously been registered.

Table 11-2 summarizes the system management tools that customers use to automatically configure device drivers. Third-party driver writers may want to understand the events that occur when the system manager configures the driver as a static or loadable driver. The sections following the table are designed to promote an understanding of these events.

Table 11-2: Summary of System Management Tools

Tool	Description
`doconfig`	Creates a new or modifies an existing system configuration file, copies `.products.list` to *NAME*`.list`, creates the device special files for static drivers, and builds a new DEC OSF/1 kernel. (The `doconfig` program calls `config`, which actually performs many of the tasks mentioned.)
`cfgmgr`	Works with `kloadsrv`, the kernel load server, to manage loadable device drivers.
`kmknod`	Uses the information from the `stanza.static` file fragment to dynamically create device special files for static device drivers at boot time.
`kreg`	Maintains the `/sys/conf/.product.list` system file, which registers static device driver products.
`sysconfig`	Modifies the loadable subsystem configuration. This modification provides a user interface to the `cfgmgr` daemon.
`sysconfigdb`	Maintains the `sysconfigdb` database. The driver stanza entries in the `stanza.loadable` file fragment are appended to this database.

11.1.5.1 Installing Static Device Drivers

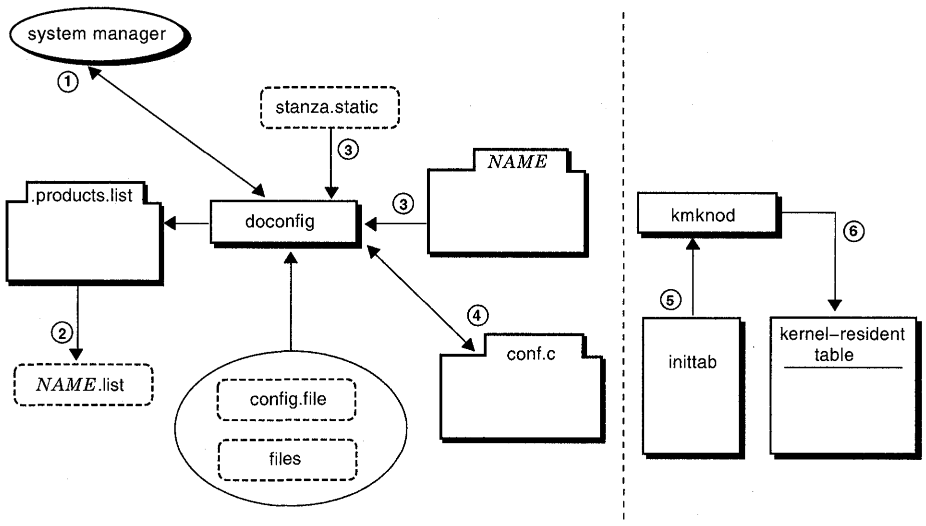
Figure 11-7 shows the events that occur when the system manager installs a static device driver. The events start after the system manager loads the device driver kit and runs `setld`.

1. The system manager runs the `doconfig` program (which calls `config`) to generate a system configuration file (shown as *NAME* in the figure) and to generate a kernel that contains the statically linked drivers.
2. The `config` program reads the `/usr/sys/conf/.products.list` file and copies it to the *NAME*`.list` file. It uses the entries in *NAME*`.list` to locate and use the supplied entries contained in the `config.file` and `files` file fragments and any source and/or object files.
3. The `config` program reads the `stanza.static` file fragment to determine the driver's major and minor number requirements and the system configuration file, *NAME*, to determine the device options and other system parameters. It also saves the device special file characteristics.
4. After determining the major number requirements, `config` automatically edits the `/usr/sys/io/common/conf.c` file to place the driver's entry points in the `bdevsw` and/or `cdevsw` tables. The assigned major number is saved in a kernel-resident table for later use.
5. The `config` program completes the kernel `makefile` to include the new static device driver.

   After `doconfig` builds the kernel and the system manager boots the system, the third-party static device driver is configured into the kernel. There is an entry in the `inittab` file that is referenced at boot time to cause the `kmknod` utility to run. This utility creates the device special files for this driver so that the customer's utilities can access this third-party device driver.
6. The `kmknod` utility references the kernel-resident table of assigned major numbers to determine what major number has been assigned to this driver. It also gets the device characteristics from this table. All of the other information needed to do the `mknods` for the device special file was specified in the `stanza.static` file fragment.

   At this point, the driver is fully configured, the device special files have been created, and the kernel is running.

Figure 11-7: Sequence of Events for Installation of Static Drivers



11.1.5.2 Installing Loadable Device Drivers

Figure 11-8 shows the events that occur when the system manager installs a loadable device driver. The events start after the system manager loads the device driver kit. The SCP requests that the system manager specify the driver be installed as loadable. The following events occur:

1. The SCP calls the `sysconfigdb` utility.

   The SCP calls `sysconfigdb`, which adds the entries contained in the `stanza.loadable` file fragment to the `/etc/sysconfigtab` database.

   The `sysconfigdb` utility can be called to add the driver to the list of drivers that are automatically configured each time the system boots. At this point, the loadable driver has been installed on the system.

2. The SCP calls the `sysconfig` utility.

   Although the loadable driver has been installed on the system, it is not currently loaded into the kernel and is therefore not usable by user level programs. To load the installed loadable driver, the system manager (or the SCP) runs the `sysconfig` utility.

3. The `sysconfig` utility calls the `cfgmgr` daemon.

   The `sysconfig` utility passes the name of the driver to load to the cfgmgr daemon through a socket connection. The cfgmgr daemon searches the global stanza database, /etc/sysconfigtab, to fetch the stanza entry for the driver. One of the fields of the stanza entry identifies the loadable module as a device driver. This information causes the cfgmgr daemon to use the device driver-specific portion (referred to as the driver method) to perform the specific loading tasks (such as, creating the necessary device special files).

4. The cfgmgr daemon calls kloadsrv.

   Another field in the stanza entry lists the pathname of the loadable object (the driver itself). The cfgmgr daemon calls the kernel loader utility kloadsrv to load the driver's object into the kernel's address space.

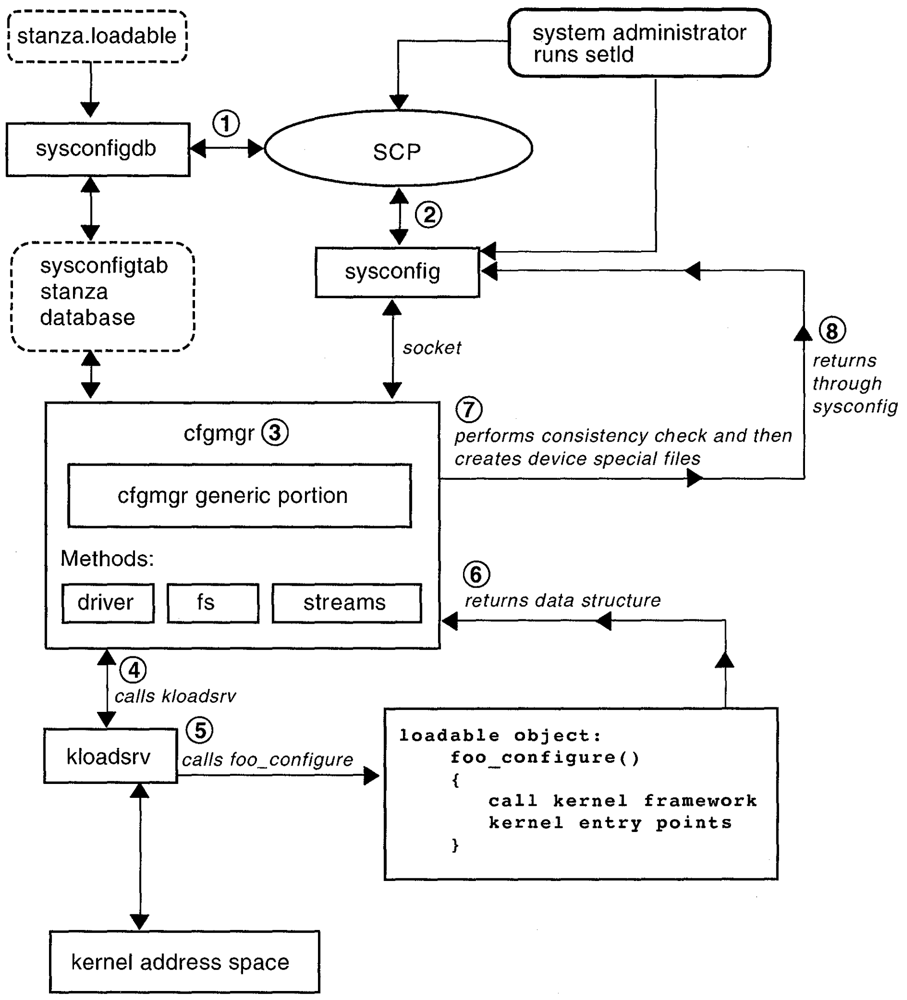
5. The kloadsrv utility calls the driver's configure interface.

   After loading the driver's object, kloadsrv calls the device driver's configure interface. The driver's configure interface causes the driver to be linked into the autoconfiguration-related data structures: bus, controller, and device. The driver's interrupt interface is registered. The configure interface also calls a bdevsw_add or cdevsw_add interface to provide a major number for this driver. The assignment of a major number includes adding the driver's entry points into the bdevsw and/or cdevsw tables.

6. The driver's configure interface returns a data structure

   The driver's configure interface returns a data structure that contains, among other things, the major number or numbers assigned to the driver. This return information is passed back to the cfgmgr daemon.

Figure 11-8: Sequence of Events for Installation of Loadable Drivers



7. The `cfgmgr` daemon performs a consistency check and then creates device special files.

   Once the driver has been loaded into the kernel's address space and prior to creating the device special files for this driver, a consistency check is performed to delete any device special files of the same name or the same driver type and major number.

The `cfgmgr` daemon then creates the device special files associated with the device by taking the major number returned from the driver's configure interface and the information contained in the driver's stanza entry to do the appropriate `mknod` calls.

8. The `cfgmgr` daemon returns through `sysconfig`

   The `cfgmgr` daemon returns through `sysconfig` and the driver loading is complete. User level utilities can now access the driver.

11.2 Traditional Device Driver Configuration Model

The traditional device driver configuration model provides a manual mechanism for driver writers or system managers to configure device drivers into the kernel. One advantage of the traditional model is that no device driver kit is needed. Thus, the driver writer can configure the driver almost immediately. The traditional model is particularly suited to a classroom situation, where the goal of the class is to learn how to write device drivers. It is also suited to driver writers following the third-party model during the initial stages of driver development.

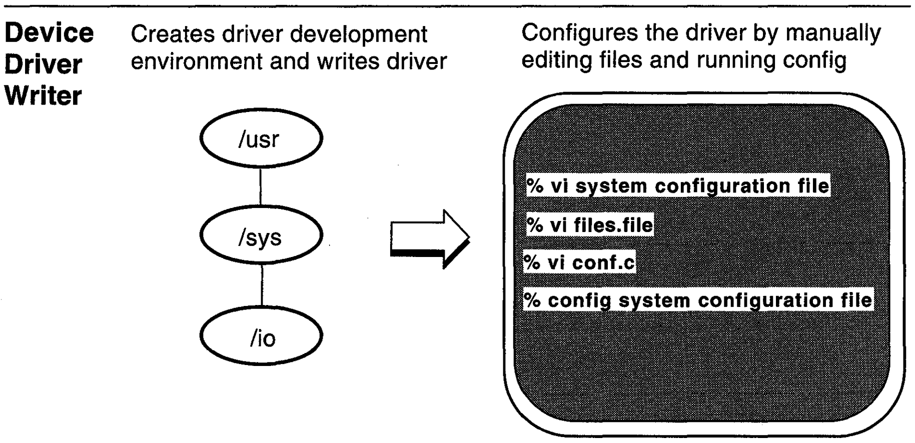
The end result of the traditional and third-party models is the same. The disadvantage of the traditional model is that it does not take advantage of the automated driver installation interfaces. Thus, the traditional model is potentially error prone.

This model requires that driver writers perform the tasks shown in Figure 11-9. The figure shows that the traditional driver configuration model involves only the device driver writer. The third-party driver configuration model is different in that it involves at least three groups of people. Figure 11-9 also shows the following tasks performed by the driver writer:

- Creating a driver development environment and writing the driver
- Configuring the device driver

The sections following Figure 11-9 describe these tasks in detail.

Figure 11-9: Tasks for Traditional Model

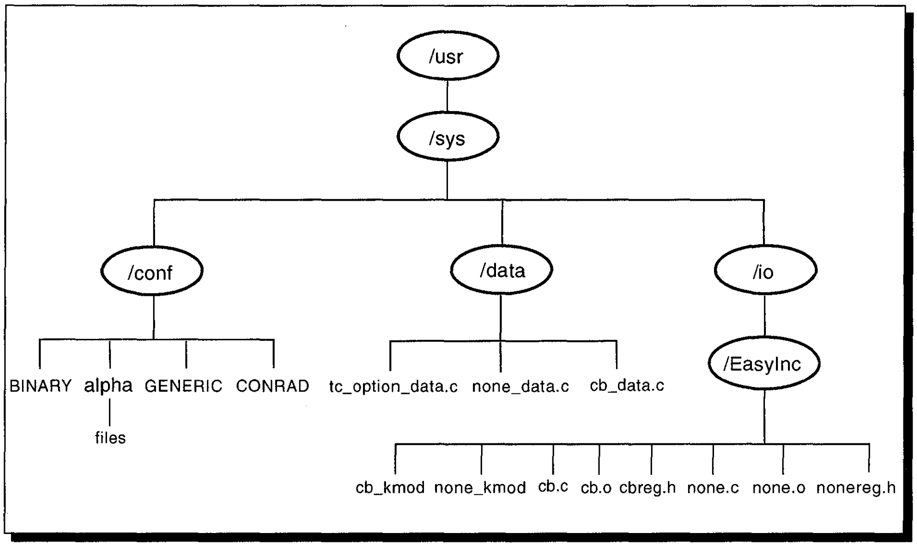

Device Driver Writer
Creates driver development environment and writes driver
Configures the driver by manually editing files and running config
/usr
/sys
/io

```
% vi system configuration file
% vi files.file
% vi conf.c
% config system configuration file
```

11.2.1 Creating a Driver Development Environment and Writing the Driver

The device driver writer creates a driver development environment and writes the device driver by using the information and techniques described in the previous chapters of this book. Figure 11-10 shows the development environment for EasyDriver Incorporated, using the traditional driver configuration model.

Figure 11-10: Driver Development Environment for EasyDriver Incorporated Using the Traditional Model



The figure shows the following directory structure:

- The `/usr/sys/conf` directory

  This directory contains files that define the kernel configuration for the generic (all-encompassing) and target machine kernels. In the `/usr/sys/conf` directory for EasyDriver Incorporated, the generic kernel is called `GENERIC`, and the target machine kernel is called CONRAD. The `/usr/sys/conf/BINARY` system configuration file is shown here because the traditional model requires that entries be added to accommodate loadable device drivers.

- A `/usr/sys/conf/alpha` directory

  This directory contains the `files` file.

- A `/usr/sys/data` directory

  This directory contains the *name*`_data.c` files supplied by Digital. An example of one such file is `tc_option_data.c`. The figure also shows that EasyDriver Incorporated uses this directory to contain the *name*`_data.c` files associated with their device drivers.

- A `/usr/sys/io/EasyInc` directory

  This directory contains the source code, object, header, and load module files for the drivers developed by EasyDriver Incorporated.

The following sections describe each of these files. Some of these files were discussed in the sections related to the third-party driver configuration model. The following sections discuss these same files from the traditional model point of view.

11.2.1.1 The /usr/sys/conf/NAME File

The `/usr/sys/conf/`*NAME* file (referred to as the system configuration file) is an ASCII text file that defines the components of the system. The system configuration file name, *NAME*, is usually the name of the system. By convention, the system configuration file name is capitalized. A device driver writer can have more than one system configuration file defined in `/usr/sys/conf`, each with a capitalized name. As Figure 11-10 shows, EasyDriver Incorporated has two system configuration files: `GENERIC` and `CONRAD`. In addition, there is a system configuration file called `/usr/sys/conf/BINARY`.

The `GENERIC` system configuration file supplied by Digital contains all the possible software and hardware options available to DEC OSF/1 systems and includes all supported Digital devices. The `GENERIC` system configuration file is used to build a kernel that represents all possible combinations of statically configured drivers that Digital supports. This kernel is booted during the operating system installation and is often referred to as the generic kernel. While running the generic kernel, the installation software determines which subset of all possible device drivers should be used to build a target kernel to match the hardware attached to the system being installed.

The installation software builds a tailored system configuration file to match the hardware present by calling the `sizer` program. This tailored system configuration file is later used by `doconfig` to create a tailored kernel.

The `BINARY` system configuration file supplied by Digital contains a subset of all the possible software and hardware options available to DEC OSF/1 systems.

In the traditional model, device driver writers manually edit the system configuration file to specify device connectivity information associated with the static device driver. For loadable drivers, device driver writers manually edit the `BINARY` system configuration file to cause an independent load module (for example, the `cb_kmod` module) to be created. This step contrasts with static device drivers where the driver itself is incorporated into the tailored kernel. Thus, driver writers modify the `BINARY` system configuration file for loadable drivers and the system configuration file for static drivers. Section 12.2 describes the syntaxes driver writers use to specify the necessary information in the system configuration file.

11.2.1.2 The files File

The device driver writer manually edits the `files` file, which contains the following information:

- Driver source code location
- Under what conditions the driver is to be statically configured
- Whether the device driver sources are supplied

Section 12.5 describes the syntaxes used to specify this information. Note that in the traditional device driver configuration model, the device driver writer specifies information in the `files` file for both loadable and static device drivers. This information is necessary to construct the driver object file for static drivers and the kernel load module for loadable drivers.

The `files` file can reside in one of several directories, based on the architecture. For example:

```
conf/alpha/files
conf/mips/files
```

There may also be one `files` file for each kernel built. In this case, the `files` file takes the name of the system configuration file. For example:

```
conf/files.TIGRIS
conf/files.EUPHRATES
```

The driver writers at EasyDriver Incorporated chose to locate their files in the `/usr/sys/conf/alpha` directory.

11.2.1.3 The name_data.c File

The *name*`_data.c` file provides a convenient place to size the data structures and data structure arrays used by static device drivers. The *name* argument is usually based on the device name. For example, the `none` device's *name*`_data.c` file is called `none_data.c`. The `CB` device's *name*`_data.c` file is called `cb_data.c`. The `edgd` device's *name*`_data.c` file would be called `edgd_data.c`. See Section 3.1.5 for more information on this file and how it relates to static drivers.

11.2.1.4 Driver Header, Source, Object, and Load Module Files

The driver writer creates a directory to contain the files related to the device driver. For the static device driver product, these can include the driver header files, which have `.h` extensions; source files, which have `.c` extensions; and the driver object files, which have `.o` extensions. The device driver object files are the actual driver executables. Figure 11-10 shows that driver writers for EasyDriver Incorporated place these files in the `/usr/sys/io/EasyInc` directory.

For the loadable device driver product, there are driver load modules, which have `_kmod` extensions.

Device Driver Configuration Syntaxes and Mechanisms 12

The previous chapter described the third-party and traditional device driver configuration models, which included brief descriptions of the file fragments the driver writer needs to supply on the device driver kit for the third-party model and the files the driver writer manually edits for the traditional model. This chapter reviews these files and describes the syntaxes and mechanisms used to populate them.

Specifically, the chapter discusses how to:

- Review device driver configuration-related files
- Specify information in the `config.file` file fragment and system configuration file
- Understand the format of the *NAME*`.list` file
- Specify information in the `files` file fragment and `files` file
- Specify information in the `stanza` files
- Specify information in the *name*`_data.c` file
- Specify information in the `conf.c` file
- Supply the subset control program (SCP)

12.1 Reviewing Device Driver Configuration-Related Files

Table 12-1 reviews the files and file fragments related to device driver configuration and provides a summary of how the driver writer populates these files with the necessary information. Subsequent sections in the chapter describe the actual syntaxes and mechanisms used to populate these files and file fragments. The table also indicates whether the file is applicable to the third-party or traditional model.

Table 12-1: Contents of Device Driver Configuration-Related Files

File	Information/Mechanisms	Third-Party Model	Traditional Model
system configuration file	For static drivers, specify a valid syntax for the device connectivity information contained in this file. This file is not applicable to loadable drivers.	No	Yes
`config.file` file fragment	For static drivers, specify a valid syntax for the device connectivity and callout information contained in this file. This file is not applicable to loadable drivers.	Yes	No
`BINARY` system configuration file	For loadable drivers, specify a valid syntax to ensure that loadable drivers build successfully. This file is not applicable to static drivers.	No	Yes
`NAME``.list` file	For static drivers, contains such information as where the files for the driver product are located. This file is not supplied on the device driver kit. This file is not applicable to loadable drivers.	Yes	No
`files` file	For static drivers, specify a valid syntax for when the driver is to be loaded into the kernel, the location of the driver source code, and whether the driver sources are supplied. For loadable drivers using the traditional model, specify a valid syntax for when the driver source is to be compiled into a dynamic load module and the location of the driver source code. This file is not applicable to loadable drivers shipped in binary form.	No	Yes
`files` file fragment	For static drivers shipped in source form, specify a valid syntax for when the driver is to be loaded into the kernel, the location of the driver source code, and whether the driver sources are supplied. This file is not applicable to loadable drivers shipped in binary form.	Yes	No

Table 12-1: (continued)

File	Information/Mechanisms	Third-Party Model	Traditional Model
`stanza.loadable` file fragment	For loadable drivers, specify a valid syntax for device connectivity information, the driver's major number requirements, the names and minor numbers of the device special files, and the permissions and directory name where the device special files are created. This file fragment is not applicable to static drivers.	Yes	Yes
`stanza.static` file fragment	For static drivers, specify a valid syntax for the driver's major number requirements, the names and minor numbers of the device special files, the permissions and directory name where the device special files reside, and the driver interfaces to be added to the `bdevsw` and `cdevsw` tables. This file is not applicable to loadable drivers.	Yes	No
A *name*`_data.c` file	For static drivers, specify data structure sizes in this file. Loadable drivers should dynamically allocate memory to accommodate data structure sizes.	Yes	Yes
driver sources and driver objects	The driver writer supplies the `.c` (optional), `.h` (optional), `.o` (for static drivers), and `_kmod` (for loadable drivers) driver files.	Yes	Yes
`conf.c`	For static drivers using the traditional model, edit the `conf.c` file with the device driver entry points in the appropriate `devsw` table. For static drivers using the third-party model, use a valid syntax in the `stanza.static` file fragment to automatically add device driver entry points to the `bdevsw` and `cdevsw` tables. This file is not used for loadable drivers. Instead, for loadable drivers, use the `bdevsw_add` and `cdevsw_add` interfaces to dynamically add device driver entry points to the in-memory `bdevsw` and `cdevsw` tables.	No	Yes

Table 12-1: (continued)

File	Information/Mechanisms	Third-Party Model	Traditional Model
Subset Control Program (SCP)	The kit developer writes an SCP that installs and manages the files and file fragments supplied to customers.	Yes	No

12.2 Specifying Information in the config.file File Fragment and the System Configuration File

The `config.file` file fragment uses the same syntax as the system configuration file to specify device connectivity information and command callout options. For the third-party model, driver writers specify in `config.file` only the information needed for their driver product. For the traditional model, driver writers manually edit the system configuration file to include the necessary information. The following sections discuss the syntaxes for the device options and the callout options available to device driver writers. For descriptions of the other options and definitions that could be specified in these files, see the *System Administration* guide.

12.2.1 Specifying Device Definitions

The device definition keywords, part of the `config.file` file fragment and the system configuration file, contain descriptions of each current or planned device on the system. That is, these definitions describe such things as bus, controller, disk, and tape mnemonics and logical unit numbers for devices connected to the system. When the system is initially configured, the `doconfig` and `sizer` utilities identify all of the devices supplied by Digital that are attached to the system and place their associated entries in the specified system configuration file.

For the third-party driver configuration model, device driver writers must make their device definitions available to their customer's system through the `config.file` file fragment. For the traditional model, driver writers must manually edit the system configuration file to make the appropriate device definitions. In either case, the keywords and values for making these entries are identical.

Device driver writers must know the syntax for making:

- A bus specification
- A controller specification

- A device specification

The syntaxes for each category are discussed in the following sections. Section 12.2.1.4 provides examples of the device options syntaxes for the `CB` and `none` devices.

12.2.1.1 Bus Specification

The following is the syntax for specifying a bus in the `config.file` file fragment and the system configuration file:

bus *bus_name#* **at** *bus_connection#*

`bus`
: The keyword that precedes a bus name and its associated unit number.

`bus_name#`
: Specifies the name and number of the bus. The bus name can be any string. For buses supported by Digital, the bus name is one of the valid strings listed in the *System Administration* guide. For example, the string `tc` represents a TURBOchannel bus.

  Third-party driver writers who write drivers that operate on non-Digital buses can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgb` for an internally developed bus.

  The number for a bus is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the bus connection.

`bus_connection#`
: Specifies the name and number of the bus that this bus is connected to. The keyword `nexus` indicates that the specified bus is the system bus.

  A wildcard syntax for the bus specification is also allowed, as shown in the following example:

```
bus tc?   1
bus tc? at nexus 2
```

1 Specifies any TURBOchannel bus.

2 Specifies the top-level or system bus.

12.2.1.2 Controller Specification

The following is the syntax for specifying a controller in the `config.file` file fragment and the system configuration file:

controller *ctlr_name#* **at** *bus_name#* [**csr** *addr*] [**flags** *flag_value*] [**slot** *slot#*] **vector** *vec...*

`controller`
: The keyword that precedes a controller name and its associated logical unit number. A controller identifies either a physical or logical connection with zero or more slaves (that is, disk and tape drives) attached to it.

`ctlr_name#`
: Specifies the controller's name and associated logical unit number. The controller name can be any string. For controllers supplied by Digital, the controller name is one of the valid strings listed in the *System Administration* guide. For example, the string `hsc` represents an HSC controller.

Third-party driver writers who write drivers for non-Digital controllers can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgc` for an internally developed controller.

The logical unit number for a controller is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the bus to which the controller is connected.

`bus_name#`
: Specifies the name and number of the bus the controller is connected to. The bus name can be any string. For buses supported by Digital, the bus name is one of the valid strings listed in the *System Administration* guide. For example, the string `tc` represents a TURBOchannel bus.

Third-party driver writers who write drivers that operate on non-Digital buses can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgc` for an internally developed controller.

A wildcard syntax for the bus specification is also allowed, as shown in the following example:

```
controller sii0 at tc? 1
controller sii0 at *   2
```

1 Specifies an instance of a controller of type `sii` attached to any TURBOchannel bus.

2 Specifies an instance of a controller of type `sii` attached to any bus type (that is, not restricted to a TURBOchannel bus).

`csr`
: Specifies an optional keyword that precedes a control status register value for some device.

`addr`
: Specifies the address of the control status register for the device. This address is required if the `csr` keyword was specified.

`flags`
: An optional keyword that precedes some value that directs the system to perform some request.

`flag_value`
: 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.

`slot`
: The keyword that precedes the bus slot or node number.

`slot#`
: Specifies the bus slot or node number.

`vector`
: Specifies the keyword that precedes the name or names of the interrupt handlers for the device. This keyword is for static drivers. Loadable drivers register their interrupt handlers through the `handler_add` and `handler_enable` interfaces. Section 9.6.3 shows how to register the interrupt service interface for the `/dev/cb` driver by calling `handler_add` and `handler_enable`.

`vec...`
: Specifies the name or names of the interrupt handlers for the device.

12.2.1.3 Device Specification

The following is the syntax for specifying a device (for example, disk or tape) in the `config.file` file fragment and the system configuration file:

device *device_spec device_name#* **at** *ctlr_name#* **drive** *phys#*

`device`
: The keyword that precedes the string that identifies the device.

`device_spec`
: Specifies a keyword that precedes a device name and its logical unit number. This keyword can be any string. Digital uses the keywords `disk` and `tape` to represent disk and tape devices.

`device_name#`
: Specifies the device's name and associated logical unit number. The device name can be any string. For devices supported by Digital, the device name is one of the valid strings listed in the *System Administration* guide. For example, the strings `ra` and `tz` represent SCSI disk and tape drives supported by Digital.

  Third-party driver writers who write drivers that operate on non-Digital devices can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgd` for an internally developed disk device.

  The logical unit number for a disk drive is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the controller to which the device is attached.

`ctlr_name#`
: Specifies the name and logical unit number of the controller to which the device is attached. The controller name can be any string. For controllers supplied by Digital, the controller name is one of the valid strings listed in the *System Administration* guide. For example, the string `hsc` represents an HSC controller.

  Third-party driver writers who write drivers for non-Digital controllers can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgc` for an internally developed controller.

  The logical unit number for a controller is any positive integer, for example, 0 or 1.

`drive`
: The keyword that precedes the physical unit number of the device.

`phys#`
: Specifies the physical unit number of the device, if required.

12.2.1.4 Device Options Syntaxes Example

The driver writers from EasyDriver Incorporated supply the following `config.file` file fragment to their customers. This file fragment shows entries for the `none` and `CB` device controllers, which operate on the customer's TURBOchannel bus:

```
# Entries in config.file for none and CB devices

controller none0 at tc? vector noneintr [1]
controller cb0   at tc? vector cbintr   [2]
```

[1] Indicates that the `CB` controller is connected to the customer's TURBOchannel bus. Note the use of the wildcard character to indicate that the `none` controller can be connected to any TURBOchannel bus.

The interrupt handler for the `/dev/none` device driver controlling the `none` controller is called `noneintr`. The interrupt handler is placed here for the static version of the `/dev/none` driver. The loadable version of the `/dev/none` driver registers the interrupt handler through the `handler_add` and `handler_enable` interfaces. Section 4.1.6.1 shows how to register `noneintr` by calling `handler_add` and `handler_enable`.

[2] Indicates that the `/dev/cb` controller is connected to the customer's TURBOchannel bus. The interrupt handler, `cbintr`, is placed here for the static version of the `/dev/cb` driver. The loadable version of the `/dev/cb` driver registers the interrupt handler through the `handler_add` and `handler_enable` interfaces. Section 10.8.1 shows how to register `cbintr` by calling `handler_add` and `handler_enable`.

Following the traditional device driver configuration model, the driver writers at EasyDriver Incorporated edit their system configuration file with the following entries:

```
   •
   •
   •
controller none0 at tc0 slot? vector noneintr
   •
   •
   •
controller cb0 at tc0 slot? vector cbintr
```

•
•
•

Note that the `slot` keyword is used because these controllers are connected to the TURBOchannel bus.

12.2.2 Specifying Callouts

The `callout` keyword provides driver writers with a mechanism for handling customer-specific tasks related to driver configuration that are not currently handled by the `config` program. The `config.file` file fragment and the system configuration file can contain one or more callout keyword definitions that invoke a subprocess. While the subprocess executes, `config` suspends its execution. It resumes execution when the subprocess completes.

The driver writer can invoke any subprocess with a `callout` keyword. The following list describes issues to consider when using the callout mechanism:

- Ensure that the command is in the search path or specify the full pathname.
- Ensure that system resources, such as memory, disks, or tapes are available.
- The subprocess must handle all error conditions because the `config` program behaves as if the subprocess always succeeds.
- If more than one callout is used with the same keyword value, the order of execution is determined by the order in the system configuration file from top to bottom.

The `callout` keyword specifies the point in the configuration sequence at which to invoke the subprocess. The subprocess that is called out has the `CONFIG_NAME` environment variable set to specify the system configuration file that called it.

Table 12-2 describes the `callout` keywords and the times at which they are invoked by `config`.

Table 12-2: The callout Keywords

callout Keyword	Usage
`at_start`	After `config` has parsed the system configuration file syntax, but before processing any other input such as `stanza.static` file fragments

Table 12-2: (continued)

callout Keyword	Usage
`at_exit`	Immediately before `config` exits, regardless of its exit status
`at_success`	Before the `at_exit` process, if specified, and only if `config` exits with a success exit status
`before_h`	Before `config` creates any `*.h` files
`after_h`	After `config` creates any `*.h` files
`before_c`	Before `config` creates any `*.c` files
`after_c`	After `config` creates any `*.c` files
`before_makefile`	Before `config` creates the `makefile`
`after_makefile`	After `config` creates the `makefile`
`before_conf`	Before `config` creates the `/usr/sys/`*NAME*`/conf.c` file
`after_conf`	After `config` creates the `/usr/sys/`*NAME*`/conf.c` file

The following example shows one possible entry in a `config.file` file fragment and system configuration file:

```
    •
    •
    •
bus tc0 at nexus?
callout after_c "../bin/mktcdata" [1]
    •
    •
    •
```

[1] In this example, the `callout` keyword invokes a subprocess (C program) called `mktcdata`. The purpose of `mktcdata` is to automatically add third-party devices registered through the `kreg` utility to the `tc_option` table.

12.3 Specifying Information in the BINARY System Configuration File

To build a loadable driver load module, using the traditional device driver configuration model, the `BINARY` system configuration file must be edited. The following syntax specifies a loadable driver entry in the `BINARY` system configuration file:

pseudo-device *driver_name* **dynamic** *driver_name*

`pseudo-device`
: Specifies a keyword that causes the appropriate `makefile` to be generated for the specified device driver.

`driver_name`
: Specifies the name of the loadable driver. This name is the *`entry_name`* that was specified in the `stanza.loadable` file fragment.

`dynamic`
: The keyword that specifies that the device driver is built as a dynamic load module (loadable driver).

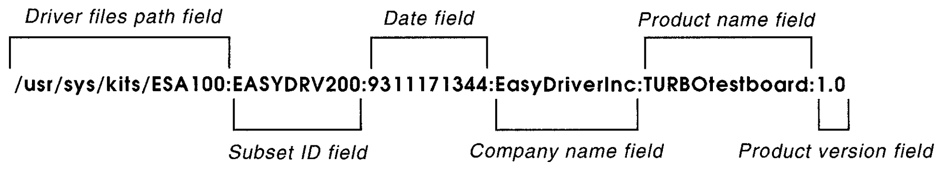
The following example shows an entry for the `/dev/cb` device driver:

```
pseudo-device    cb    dynamic    cb
```

12.4 Understanding the Format of the NAME.list File

In the third-party device driver configuration model, the `kreg` utility sets up the `.products.list` and *`NAME`*`.list` files on the customer's system. Although third-party driver writers and kit developers do not supply these files, it is useful to understand the files' format. Note that these files are not used by driver writers following the traditional device driver configuration model. Figure 12-1 shows the format of such a file for EasyDriver Incorporated.

Figure 12-1: A NAME.list File for EasyDriver Incorporated



The figure shows that the *`NAME`*`.list` file has fields separated by colons (:), as follows:

- Driver files path field

  The driver files path field contains the path that points to the location of the files associated with the driver product, which in the figure is `/usr/sys/kits/ESA100`. The value in the driver files path field must be unique.

- Subset ID field

  The subset ID field contains the subset ID associated with the `setld` utility, which in the figure is `EASYDRV200`.

- Date field

  The date field contains the date when the product is ready for distribution. The date has the form *yymmddhhmm*, where *yy* represents the year, *mm* represents the month, *hh* represents the hour (24-hour time), and *mm* represents the minutes. For EasyDriver Incorporated, the date field translates to:

  ```
  November 17, 1993 1:44 PM
  ```

- Company name field

  The company name field contains the company's name, which in the figure is abbreviated to EasyDriverInc.

- Product name field

  The product name field contains the name of the product, which in the figure is `TURBOtestboard`.

- Product version field

  The product version field contains the version number of the product, which in the figure is 1.0.

  Note that the configuration software (namely, the `config` program) uses the information only in the path field. Although editing the *NAME*`.list` file is not supported, it is possible for driver writers to add an entry to this file. Doing so would be useful if for some reason the driver writer did not want to use the `setld` utility for kit processing.

12.5 Specifying Information in the files File Fragment and files File

The `files` file fragment contains the same kind of information that appears in the `files` file. For the third-party model, driver writers specify in the `files` file fragment only the information needed for their driver product. For the traditional model, driver writers manually edit the `files` file to include the necessary information.

The following syntax specifies an entry in the `files` file fragment and the `files` file for static device drivers:

path_name **optional** *key_string* **| standard device-driver Binary | Notbinary**

The following syntax specifies an entry in the `files` file for loadable device drivers:

path_name **optional** *key_string* | **standard device-driver if_dynamic** *key_string* **Binary**

`path_name`
: Specifies the file specification indicating where the device driver sources reside.

`optional`
: This keyword indicates that this software module will be included in those kernels whose system configuration files specify the *`key_string`* that follows the keyword `optional`.

`standard`
: This keyword indicates that this software module will be included in every kernel.

`device-driver`
: The keyword `device-driver` directs the `config` program to create the `makefile` entry that builds the kernel object so that the C compiler builds the object code unoptimized. This keyword is mandatory for all device driver entries.

`if_dynamic`
: The `if_dynamic` keyword marks the specified device driver source files such that the resulting object files can be built as either static or loadable.

`Binary`
: The `Binary` keyword causes symbolic links to be made to existing object modules.

`Notbinary`
: The `Notbinary` keyword causes the `config` program to create a `makefile` that compiles the object from source. Static device drivers written by third-party vendors can use either keyword, depending on whether they want to supply the driver sources or to compile their *`name`*`_data.c` files, if shipped.

  Loadable drivers written by third-party vendors must use the `Binary` keyword, as shown in the syntax.

The following examples show how the driver writers at EasyDriver Incorporated specify entries in the `files` file fragment (third-party model) and the `files` file (traditional model) for the static versions of the

/dev/none and /dev/cb drivers.

```
# This example illustrates the third-party model
# by showing a files file fragment for static
# drivers developed by EasyDriver Incorporated.

ESA100/none.c standard none device-driver if_dynamic none Binary [1]

ESB100/cb.c optional cb device-driver Binary [2]


# This example illustrates the traditional model
# by showing a files file for static drivers
# developed by  EasyDriver Incorporated.

io/EasyInc/none.c standard none device-driver Notbinary [1]
io/EasyInc/cb.c optional cb device-driver Binary [2]
```

[1] In the third-party model example, shows that the `/dev/none` device driver object is located in the directory `ESA100`. The `kreg` utility puts the path `/usr/sys/kits/ESA100` in the customer's `.products.list` file.

The corresponding line in the traditional model example shows that the `/dev/none` device driver object is located in the directory `/usr/sys/io/EasyInc`. In both cases, the name of the driver source is `none.c`.

In the third-party model example, the `standard` keyword indicates that the `/dev/none` driver will be included in every customer kernel. In the traditional model example, the `standard` keyword indicates that the `/dev/none` driver will be included in every kernel at EasyDriver Incorporated.

The `device-driver` keyword used in both model examples indicates to `config` that the driver is built with compiler optimization turned off.

The `Notbinary` keyword indicates that the `none.c` file will be supplied to customers.

[2] In both the third-party and traditional models, shows the syntax for the `/dev/cb` driver. The syntax is identical to that specified for the `/dev/none` driver except that in both models it uses the `optional` and `Binary` keywords.

The `optional` keyword indicates that the `/dev/cb` driver will be included in those kernels whose system configuration files specify `cb` as a device entry.

The `Binary` keyword indicates that the `cb.c` file will not be compiled and need not exist but that the kernel links in a supplied `cb.o` object file.

The following example show how the driver writers at EasyDriver Incorporated specify entries in the `files` file (traditional model) for the loadable versions of the /dev/none and /dev/cb drivers.

```
# This example illustrates the traditional model
# by showing a files file for loadable drivers
# developed by EasyDriver Incorporated.

io/EasyInc/none.c optional none device-driver if_dynamic none Binary [1]
io/EasyInc/cb.c optional cb device-driver if_dynamic cb Binary[2]
```

[1] In the traditional model example, shows that the /dev/none device driver object is located in the directory io/EasyInc.

The optional keyword indicates that the /dev/none device driver driver will be included in those kernels whose system configuration files specify none as a device entry.

The device-driver keyword used in both model examples indicates to config that the driver is built with compiler optimization turned off.

The if_dynamic keyword marks the /dev/none device driver source files such that the resulting object files can be built as either static or loadable.

The Binary keyword causes symbolic links to be made to existing load modules.

[2] Specifies the keywords for the /dev/cb driver. They are the same as the ones used for the /dev/none driver.

12.6 Specifying Information in the stanza Files

Device driver writers must understand the following topics associated with the stanza files:

- Stanza file format
- Stanza file syntax

Section 12.6.3 provides examples of the syntaxes for the stanza.static and stanza.loadable file fragments.

12.6.1 Stanza File Format

Figure 12-2 shows the format associated with the fields in the stanza files separated by colons (:). It also shows that the syntax for a stanza entry contains:

- *entry_name*

  Specifies the name of the device driver. The figure shows one sample driver entry called tdc.

  Typically, each driver contains a separate stanza entry. If more than one stanza entry is supplied in a single `stanza.loadable` or `stanza.static` file fragment, separate them with one or more blank lines.

Third-party driver writers who write drivers for non-Digital controllers can select a name that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other third-party drivers. For example, the driver writers at EasyDriver Incorporated might specify `edgd` for a device driver developed for an internally developed device.

- Comments

  A number sign (#) at the beginning of a line indicates a comment. You can include comments at the beginning or the end of a driver stanza entry. The figure shows an example of a comment at the beginning of the entry for the `tdc` driver. Comments are not allowed within the body of the stanza entry.

- Trailing blanks

  Tabs are allowed at the beginning or end of lines and, as the figure shows, trailing blanks are allowed at the end of lines.

- *`Attribute_name`* and *`Attribute_value`*

  Specifies a valid stanza field. The figure shows that a valid stanza entry consists of a keyword that identifies the field, an equal sign (=) separator, and one or more values. The values can be strings (as in the figure) or valid keywords described in this book.

  Driver writers interested in learning about the other valid stanza keywords should see `sysconfigtab` in *Reference Pages Sections 4, 5, and 7.*

- New lines

  The figure also shows that a new line terminates an attribute name and value pair.

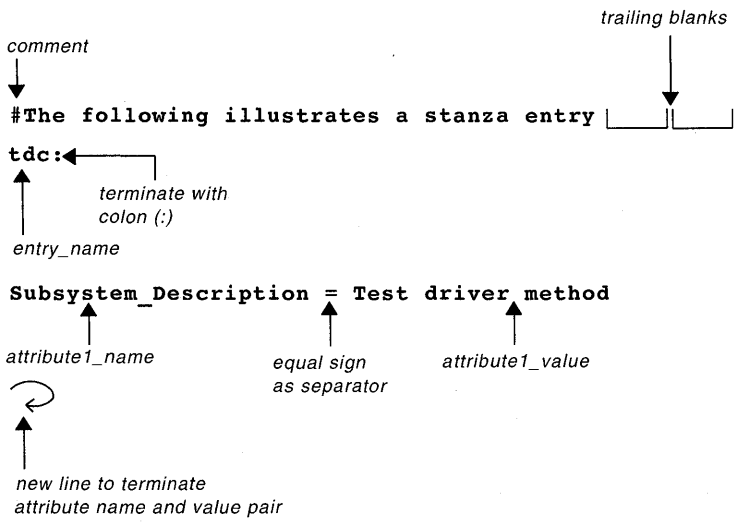
The following list describes restrictions associated with a `stanza.loadable` and `stanza.static` file fragment:

- An individual stanza entry can be a maximum of 40960 bytes in length. The system ignores all bytes in excess of this limit.
- An individual line (attribute) within a stanza entry cannot exceed 500 bytes.
- An individual stanza entry cannot consist of over 2048 lines (attributes).
- At least one blank line is required between stanza entries.

Note

At least one blank line is required at the end of the `stanza.loadable` and `stanza.static` files.

Figure 12-2: Format of the Stanza Files


trailing blanks
comment
#The following illustrates a stanza entry
tdc:
terminate with colon (:)
entry_name
Subsystem_Description = Test driver method
attribute1_name
equal sign as separator
attribute1_value
new line to terminate attribute name and value pair

12.6.2 Stanza File Syntax

Table 12-3 lists the stanza file fields driver writers must be familiar with. The table has the following columns:

- Field

  This column lists the stanza field.

- `stanza.loadable`

  A Yes appears in this column if the field is applicable to the `stanza.loadable` file fragment. Otherwise, a No appears.

- `stanza.static`

  A Yes appears in this column if the field is applicable to the `stanza.static` file fragment. Otherwise, a No appears.

Table 12-3: The stanza File Fields

Field	stanza.loadable	stanza.static
Subsystem_Description	Yes	Yes
Method_Name	Yes	No
Method_Type	Yes	No
Method_Path	Yes	No
Module_Type	Yes	No
Module_Path	Yes	No
Device_Dir	Yes	Yes
Device_Subdir	Yes	Yes
Device_Block_Subdir	Yes	Yes
Device_Char_Subdir	Yes	Yes
Device_Major_Req	Yes	Yes
Device_Block_Major	Yes	Yes
Device_Block_Minor	Yes	Yes
Device_Block_Files	Yes	Yes
Device_Char_Major	Yes	Yes
Device_Char_Minor	Yes	Yes
Device_Char_Files	Yes	Yes
Device_User	Yes	Yes
Device_Group	Yes	Yes
Device_Mode	Yes	Yes
Module_Config_Name	Yes	Yes
Module_Config	Yes	Yes
Device_Block_Open	No	Yes
Device_Block_Close	No	Yes
Device_Block_Strategy	No	Yes
Device_Block_Dump	No	Yes
Device_Block_Psize	No	Yes
Device_Block_Flags	No	Yes
Device_Block_Ioctl	No	Yes
Device_Block_Funnel	No	Yes
Device_Char_Open	No	Yes
Device_Char_Close	No	Yes

Table 12-3: (continued)

Field	stanza.loadable	stanza.static
`Device_Char_Read`	No	Yes
`Device_Char_Write`	No	Yes
`Device_Char_Ioctl`	No	Yes
`Device_Char_Stop`	No	Yes
`Device_Char_Reset`	No	Yes
`Device_Char_Ttys`	No	Yes
`Device_Char_Select`	No	Yes
`Device_Char_Mmap`	No	Yes
`Device_Char_Funnel`	No	Yes
`Device_Char_Segmap`	No	Yes
`Device_Char_Flags`	No	Yes

Each of the fields from `Subsystem_Description` to `Module_Config` is discussed in the following sections. The remainder of the fields are related to the `conf.c` file; therefore, they are discussed in Section 12.8.2. Note that you can include fields that are not used because the system ignores them.

12.6.2.1 Subsystem_Description Field

The following is the syntax for specifying the `Subsystem_Description` field in the `stanza.loadable` or `stanza.static` file fragment:

Subsystem_Description = *description*

The `Subsystem_Description` field is optional, and it specifies a short literal string used as a description. If specified in the `stanza.static` file, the description is used as the comment in the entry to the `bdevsw` or `cdevsw` table.

12.6.2.2 Method_Name Field

The following syntax specifies the `Method_Name` field in a `stanza.loadable` file fragment:

Method_Name = device

The `Method_Name` field is required, and it specifies a unique logical name of the configuration method associated with the subsystem. The method name is used by the `cfgmgr` daemon to dispatch to the appropriate subsystem a specific method to perform subsystem loading. The only valid value for the loadable device driver subsystem is `device`.

12.6.2.3 Method_Type Field

The following syntax specifies the `Method_Type` field in a `stanza.loadable` file fragment:

Method_Type = Static

The `Method_Type` field is required. Because the driver method is statically built into the `cfgmgr` daemon, this field must be specified as `Static`.

12.6.2.4 Method_Path Field

The following syntax specifies the `Method_Path` field in a `stanza.loadable` file fragment:

Method_Path = None

The `Method_Path` field specifies the file pathname for the configuration method object module. Because the driver method specified for `Method_Type` is `Static`, this field must be specified as `None`.

12.6.2.5 Module_Type Field

The following syntax specifies the `Module_Type` field in a `stanza.loadable` file fragment:

Module_Type = Dynamic | Static

The `Module_Type` field is required, and it specifies a subsystem type of `Dynamic` or `Static`. The `Dynamic` type indicates that the driver subsystem is a loadable module. Loadable device drivers should specify this value type.

The `Static` type indicates that the driver subsystem is statically configured into the kernel. This type enables statically configured subsystems to be included in the global stanza database, `/etc/sysconfigtab`, and to be controlled by the `cfgmgr` daemon.

12.6.2.6 Module_Path Field

The following syntax specifies the `Module_Path` field in a `stanza.loadable` file fragment:

Module_Path = *path_name*

The `Module_Path` field specifies the full pathname where the loadable driver subsystem resides. This field is required if `Dynamic` is specified in the `Module_Type` field.

12.6.2.7 Device_Dir Field

The following syntax specifies the `Device_Dir` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Dir = *directory*

The `Device_Dir` field is optional, and it specifies a valid directory specification for the location of the device special files. This directory is typically `/dev` for both block and character devices. If you do not specify a directory for this field, it defaults to `/dev` for both block and character devices.

12.6.2.8 Device_Subdir Field

The following syntax specifies the `Device_Subdir` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Subdir = *subdirectory*

The `Device_Subdir` field is optional, and it is appended to the directory specified or defaulted to in the `Device_Dir` field. The `Device_Subdir` field specifies a single directory location for the placement of the device special files associated with both character and block drivers. If you do not specify a directory for this field, the device special files are placed in the directory specified or defaulted to in the `Device_Dir` field.

If the device special files for both block and character drivers should not reside in a single directory, use the `Device_Block_Subdir` and `Device_Char_Subdir` fields.

12.6.2.9 Device_Block_Subdir Field

The following syntax specifies the `Device_Block_Subdir` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Block_Subdir = *subdirectory*

The `Device_Block_Subdir` field is optional, and it specifies a subdirectory for the directory specified in `Device_Dir`. The `Device_Block_Subdir` field overrides any directory specification made in the `Device_Subdir` field for device special files associated with block device drivers. This directory is used to place the device special files for block drivers to keep them separate from the device special files for character drivers. If you do not specify a directory for this field, the device special files are placed in the directory specified or defaulted to in the Device_Dir field and the Device_Subdir field, if specified.

If the device special files for block device drivers should reside in the same directory as the device special files for character drivers, use the Device_Subdir field.

12.6.2.10 Device_Char_Subdir Field

The following syntax specifies the Device_Char_Subdir field in the stanza.loadable and stanza.static file fragments:

Device_Char_Subdir = *subdirectory*

The Device_Char_Subdir field is optional, and it specifies a subdirectory for the directory specified in Device_Dir. The Device_Char_Subdir field overrides any directory specification made in the Device_Subdir field for device special files associated with character device drivers. This directory is used to place the device special files for character drivers to keep them separate from the device special files for block drivers. If you do not specify a directory for this field, the device special files are placed in the directory specified or defaulted to in the Device_Dir field and the Device_Subdir field, if specified.

If the device special files for character device drivers should reside in the same directory as the device special files for block drivers, use the Device_Subdir field.

12.6.2.11 Device_Major_Req Field

The following syntax specifies the Device_Major_Req field in the stanza.loadable and stanza.static file fragments:

Device_Major_Req = None | Same

The Device_Major_Req field is optional, and it specifies the requirements that relate to major number assignment. Specify None or omit this field if there are no major number requirements.

Specify Same if the driver needs the same major number in both the bdevsw and cdevsw tables. If Same is specified, the values of the Device_Block_Major and Device_Char_Major fields must be identical. Otherwise, the attempt to load the driver fails.

12.6.2.12 Device_Block_Major Field

The following syntax specifies the `Device_Block_Major` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Block_Major = Any | *major#*

The `Device_Block_Major` field is required only if the driver is a block device driver, and it specifies the block major number for the device driver. Specify the value `Any`, which indicates that the system dynamically assigns the next available block device major number; or, specify a number (for example, 24) and the system assigns this major number to the driver if it is not currently in use. If the specified number is in use, the attempt to dynamically load the driver fails.

12.6.2.13 Device_Block_Minor Field

The following syntax specifies the `Device_Block_Minor` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Block_Minor = *minor#*

The `Device_Block_Minor` field is optional, and it specifies the minor numbers used to create the device special files for the driver. Each minor number must be paired with a file name specified in the `Device_Block_Files` field.

The following example shows four ways to specify the block device minor numbers:

```
# One device minor number [1]
Device_Block_Minor = 1

# More than one device minor number [2]
Device_Block_Minor = 0,1,2,3

# A range of device minor numbers [3]
Device_Block_Minor = [0-10]

# More than one range of device minor numbers [4]
Device_Block_Minor = [0-10],[21-30]
```

[1] To specify a single device minor number, simply specify the number. The device minor number must be a positive integer that cannot exceed 99999. A maximum of 512 device special files can be created for a device major number, unless a driver is both a block and character device. In this case, the maximum is 1024 device special files (512 for the block and 512 for the character drivers).

[2] To specify more than one device minor number, specify the numbers separated by commas. Each device minor number must be greater than the one previously specified.

3 To specify a range of device minor numbers, enclose the range within brackets ([ ]) and separate the beginning and ending values with a dash (–). The following rules apply to numbers specified in a range:

- The ending number must be greater than the beginning number, as in the example. Thus, `[10–0]` is an invalid range specification.
- The numbers specified in the range must be greater than the value zero (0). Thus, `[–1–10]` is also an invalid range specification because the first number is less than the value zero (0).
- The largest allowable number in the range is 99999.
- A maximum of 512 device special files can be created for any device major number. Thus, `[0–511]` and `[1000–1500]` are valid range specifications while `[0–600]` is invalid.

4 To specify more than one range of device minor numbers, enclose the first range within brackets ([]) and separate the range with a dash (–). Follow the first range with a comma (,). Specify the second range in the same manner as the first and omit the comma (unless there are additional ranges).

12.6.2.14 Device_Block_Files Field

The following syntax specifies the `Device_Block_Files` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Block_Files = *devspecialfilename*

The `Device_Block_Files` field is optional, and it specifies the device special files to be created. Each device special file name must be paired with a minor number specified in the `Device_Block_Minor` field. If a driver is both a block and character driver, specify device special files in both the `Device_Block_Files` and `Device_Char_Files` fields.

The following example shows four ways to specify the block device special file names:

```
# One device special file name 1
Device_Block_Files = rz1

# More than one device special file name 2
Device_Block_Files = rz1,rz2,rz3

# A range of device special file names 3
Device_Block_Files = rz1[a-h]

# More than one range of device special file names 4
Device_Block_Files = rz1[a-h],rz2[a-h]
```

1 To specify a single file name, simply specify the name.

2 To specify more than one file name, specify the names separated by commas.

3 To specify a range of file names, enclose the range within brackets ([]) and separate the beginning and ending names with a dash (–). The following rules apply to letters specified in a range:

- Both letters must be either lowercase or uppercase. Thus, `[a-h]` or `[A-H]` are valid range specifications while `[A-h]` and `[a-H]` are invalid.
- Only one letter is allowed in the range specification, as in the example. Thus, `[aa-hh]` is not a valid range specification.
- The ending letter must be greater than the beginning letter. Thus, `[z-a]` is not a valid range specification.

4 To specify more than one range of file names, enclose the first range within brackets ([]) and separate the range with a dash (–). Follow the first range with a comma (,). Specify the second range in the same manner as the first and omit the comma (unless there are additional ranges).

As stated previously, the `Device_Block_Files` field must be paired with corresponding minor numbers specified in the `Device_Block_Minor` field. The ranges for each must represent an equal number of file-to-minor number associations. The following example shows a correct match:

```
Device_Block_Files = rz0[a–h]
Device_Block_Minor = [0–7]
```

The following shows an invalid match:

```
Device_Block_Files = rz0[a–c]
Device_Block_Minor = [0–7]
```

The following shows a match using more than one range. Note that the gap in the minor numbers is valid:

```
Device_Block_Files = rz0[a–h],rz1[a–h],rz2[a–h],rt5[a–h]
Device_Block_Minor = [0–7],[8–15],[16–23],[40–47]
```

The number of files must equal the number of minor numbers, just as in a single range specification.

In addition, the maximum number of devices is 512. Thus, the following range specifications are invalid:

```
Device_Block_Files = rz0[0–300],rz1[301–600]
Device_Block_Minor = [0–300],[301–600]
```

If you violate any of the previously discussed rules, none of the device special files will be created.

12.6.2.15 Device_Char_Major Field

The following syntax specifies the `Device_Char_Major` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Char_Major = Any | *major#*

The `Device_Char_Major` field is required only if the driver is a character driver and it specifies the character major number for the device driver. Specify the value `Any`, which indicates that the system dynamically assigns the next available character device major number. or, specify a number (for example, 24) and the system assigns this major number to the driver if it is not currently in use. If the specified number is in use, the attempt to dynamically load the driver fails.

12.6.2.16 Device_Char_Minor Field

The following syntax specifies the `Device_Char_Minor` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Char_Minor = *minor#*

The `Device_Char_Minor` field is optional, and it specifies the minor numbers used to create the device special files for the driver. Each minor number must be paired with a file name specified in the `Device_Char_Files` field.

The following example shows four ways to specify the character device minor numbers:

```
# One device minor number [1]
Device_Char_Minor = 1

# More than one device minor number [2]
Device_Char_Minor = 0,1,2,3

# A range of device minor numbers [3]
Device_Char_Minor = [0-10]

# More than one range of device minor numbers [4]
Device_Char_Minor = [0-10],[21-30]
```

[1] To specify a single device minor number, simply specify the number. The device minor number must be a positive integer that cannot exceed 99999. A maximum of 512 device special files can be created for a device major number, unless a driver is both a block and character device. In this case, the maximum is 1024 device special files (512 for the block and 512 for the character drivers).

[2] To specify more than one device minor number, specify the numbers separated by commas. Each device minor number must be greater than the one previously specified.

3 To specify a range of device minor numbers, enclose the range within brackets ([]) and separate the beginning and ending values with a dash (–). The following rules apply to numbers specified in a range:

- The ending number must be greater than the beginning number, as in the example. Thus, `[10–0]` is an invalid range specification.
- The numbers specified in the range must be greater than the value zero (0). Thus, `[–1–10]` is also an invalid range specification because the first number is less than the value zero (0).
- The largest allowable number in the range is 99999.
- A maximum of 512 device special files can be created for any device major number. Thus, `[0–511]` and `[1000–1500]` are valid range specifications while `[0–600]` is invalid.

4 To specify more than one range of device minor numbers, enclose the first range within brackets ([]) and separate the range with a dash (–). Follow the first range with a comma (,). Specify the second range in the same manner as the first and omit the comma (unless there are additional ranges).

12.6.2.17 Device_Char_Files Field

The following syntax specifies the `Device_Char_Files` field in the `stanza.loadable` and `stanza.static` file fragments:

Device_Char_Files = *devspecialfilename*

The `Device_Char_Files` field is optional, and it specifies the device special files to be created. Each device special file name must be paired with a minor number specified in the `Device_Char_Minor` field. If a driver is both a block and character driver, specify device special files in both the `Device_Block_Files` and `Device_Char_Files` fields.

The following example shows four ways to specify the character device special file names:

```
# One device special file name 1
Device_Char_Files = rrz1

# More than one device special file name 2
Device_Char_Files = rrz1,rrz2,rrz3

# A range of device special file names 3
Device_Char_Files = rrz1[1-10]

# More than one range of device special file names 4
Device_Char_Files = rrz1[a-h],rrz2[i-t]
```

1 To specify a single file name, simply specify the name.

2 To specify more than one file name, specify the names separated by commas.

3 To specify a range of file names, enclose the range within brackets ([]) and separate the beginning and ending names with a dash (–). The following rules apply to letters specified in a range:

- Both letters must be either lowercase or uppercase. Thus, `[a-h]` or `[A-H]` are valid range specifications while `[A-h]` and `[a-H]` are invalid.
- Only one letter is allowed in the range specification, as in the example. Thus, `[aa-hh]` is not a valid range specification.
- The ending letter must be greater than the beginning letter. Thus, `[z-a]` is not a valid range specification.

4 To specify more than one range of file names, enclose the first range within brackets ([]) and separate the range with a dash (–). Follow the first range with a comma (,). Specify the second range in the same manner as the first and omit the comma (unless there are additional ranges).

As stated previously, the `Device_Char_Files` field must be paired with corresponding minor numbers specified in the `Device_Char_Minor` field. The ranges for each must represent an equal number of file-to-minor number associations. The following example shows a correct match:

```
Device_Char_Files = rrz0[a–h]
Device_Char_Minor = [0–7]
```

The following shows an invalid match:

```
Device_Char_Files = rrz0[a–c]
Device_Char_Minor = [0–7]
```

The following shows a match using more than one range. Note that the gap in the minor numbers is valid:

```
Device_Char_Files = rrz0[a–h],rrz1[a–h],rrz2[a–h],rrt5[a–h]
Device_Char_Minor = [0–7],[8–15],[16–23],[40–47]
```

The number of files must equal the number of minor numbers, just as in a single range specification.

In addition, the maximum number of devices is 512. Thus, the following range specifications are invalid:

```
Device_Char_Files = rrz0[0–300],rrz1[301–600]
Device_Char_Minor = [0–300],[301–600]
```

If you violate any of the previously discussed rules, none of the device special files will be created.

12.6.2.18 Device_User, Device_Group, and Device_Mode Fields

The following is the syntax for specifying the `Device_User`, `Device_Group`, and `Device_Mode` fields for block and character device special files in `stanza.loadable` and `stanza.static` file fragments:

Device_User = *name*
Device_Group = *group*
Device_Mode = *mode*

If these optional fields are omitted, the system uses the values specified in the default stanza entry of the `/etc/sysconfigtab` database to create the device special files for loadable drivers. The system does not use the values specified in `/etc/sysconfigtab` for static drivers.

The `Device_User` field is optional, and it specifies the user ID (UID) that owns the device special files for block and character devices. You can specify a decimal number or a string of alphabetic characters. The default is the value zero (0).

The `Device_Group` field is optional, and it specifies the group ID (GID) to which the device special files associated with the block and character devices belong. You can specify a decimal number or a string of alphabetic characters. The default is the value zero (0).

The `Device_Mode` field is optional, and it specifies the protection mode for the device special files. You must specify an octal number. The default is the octal number 0664.

12.6.2.19 Module_Config_Name Field

The following is the syntax for specifying the `Module_Config_Name` field in `stanza.loadable` and `stanza.static` file fragments:

Module_Config_Name = *name*

The `Module_Config_Name` field specifies the driver name. This field can be set to the same name that was specified for the *`entry_name`* field in the `stanza.loadable` and `stanza.static` file fragments. The following list describes the interpretation of this field for loadable and static drivers:

- For loadable drivers

  This field is used to identify the driver's configuration *NAME* and is required only if the `Module_Config` field is specified.

- For static drivers

  This field is required because it indicates that this stanza entry is to be added through the use of the automated configuration tools. If no driver name is specified in this field, the entire stanza entry is skipped. Ignoring the entire stanza entry is not considered an error, and no error message will be generated.

12.6.2.20 Module_Config Field for Bus Specification

The following syntax specifies the `Module_Config` field for buses in the `stanza.loadable` file fragment. This field is not used for static drivers because device connectivity information for static drivers is specified in the `config.file` file fragment or the system configuration file. Section 12.2 describes the syntax for specifying the device connectivity information in the `config.file` file fragment and the system configuration file.

Module_Config*n* = *device_connectivity_info*

The `Module_Config` field is optional, and it specifies the device connectivity information. The *n* argument is a number in the range of 0 to 499, for example, `Module_Config2`. The *`device_connectivity_info`* argument has the following format for bus specification. Note that this format is a subset of those used in the `config.file` file fragment and the system configuration file.

bus *bus_name#* **at** *bus_connection#*

`bus`
: The keyword that precedes a bus name and its associated unit number.

`bus_name#`
: Specifies the name and number of the bus. The bus name can be any string. For buses supported by Digital, the bus name is one of the valid strings listed in the *System Administration* guide. For example, the string `tc` represents a TURBOchannel bus.

  Third-party driver writers who write drivers that operate on non-Digital buses can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors.

  The number for a bus is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the bus connection.

`bus_connection#`
: Specifies the name and number of the bus that this bus is connected to. The bus name can be any string. For buses supported by Digital, the bus name is one of the valid strings listed in the *System Administration* guide. For example, the string `tc` represents a TURBOchannel bus.

  Third-party driver writers who write drivers that operate on non-Digital buses can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors.

The number for a bus is any positive integer, for example, 0 or 1.

A wildcard syntax for the bus specification is also allowed, as shown in the following example:

```
bus tc?  [1]
bus tc0 at bus tc1     [2]
```

[1] Specifies any TURBOchannel bus.

[2] Specifies TURBOchannel bus number 1 is connected to TURBOchannel bus number 0.

12.6.2.21 Module_Config Field for Controller Specification

The following syntax specifies the `Module_Config` field for controllers in the `stanza.loadable` file fragment. This field is not used for static drivers because device connectivity information for static drivers is specified in the `config.file` file fragment or the system configuration file. Section 12.2 describes the syntax for specifying the device connectivity information in the `config.file` file fragment and the system configuration file.

Module_Config*n* = *device_connectivity_info*

The `Module_Config` field is optional, and it specifies the device connectivity information. The *n* argument is a number in the range of 0 to 499, for example, `Module_Config2`. The *`device_connectivity_info`* argument has the following format for controller specification. Note that this format is a subset of those used in the `config.file` file fragment and the system configuration file.

controller *ctlr_name#* **at** *bus_name* [**slot** *slot#*]

`controller`
: The keyword that precedes a controller name and its associated logical unit number. A controller identifies either a physical or logical connection with zero or more slaves (that is, disk and tape drives) attached to it.

`ctlr_name#`
: Specifies the controller's name and associated logical unit number. The controller name can be any string. For controllers supplied by Digital, the controller name is one of the valid strings listed in the *System Administration* guide. For example, the string `hsc` represents an HSC controller.

Third-party driver writers who write drivers for non-Digital controllers can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgc` for an internally developed controller.

The logical unit number for a controller is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the bus to which the controller is connected.

`bus_name#`
: Specifies the name and number of the bus the controller is connected to. The bus name can be any string that matches the previously specified string for the bus entry as describe in Section 12.6.2.20. For buses supported by Digital, the bus name is one of the valid strings listed in the *System Administration* guide. For example, the string `tc` represents a TURBOchannel bus.

Third-party driver writers who write drivers that operate on non-Digital buses can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors.

A wildcard syntax for the bus specification is also allowed, as shown in the following example:

```
controller sii0 at tc? 1
controller sii0 at *   2
```

1 Specifies an instance of a controller of type `sii` attached to any TURBOchannel bus.

2 Specifies an instance of a controller of type `sii` attached to any bus type (that is, not restricted to a TURBOchannel bus).

`slot`
: The keyword that precedes the bus slot or node number.

`slot#`
: Specifies the bus slot or node number.

12.6.2.22 Module_Config Field for Device Specification

The following syntax specifies the `Module_Config` field for devices. This field is not used for static drivers because device connectivity information for static drivers is specified in the `config.file` file fragment or the system configuration file. Section 12.2 describes the syntax for specifying the device connectivity information in the `config.file` file fragment and the system configuration file.

Module_Config*n* = *device_connectivity_info*

The `Module_Config` field is optional, and it specifies the device connectivity information. The *n* argument is a number in the range of 0 to 499, for example, `Module_Config2`. The *`device_connectivity_info`* argument has the following format for device specification. Note that this format is a subset of those used in the `config.file` file fragment and the system configuration file.

device *device_spec device_name#* **at** *ctlr_name#* **drive** *phys#*

`device`
: The keyword that precedes the string that identifies the device.

`device_spec`
: Specifies a keyword that precedes a device name and its logical unit number. This keyword can be any string. Digital uses the keywords `disk` and `tape` to represent disk and tape devices.

`device_name``#`
: Specifies the device's name and associated logical unit number. The device name can be any string. For devices supported by Digital, the device name is one of the valid strings listed in the *System Administration* guide. For example, the strings `ra` and `tz` represent SCSI disk and tape drives supported by Digital.

  Third-party driver writers who write drivers that operate on non-Digital devices can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify `edgd` for an internally developed disk device.

  The logical unit number for a disk drive is any positive integer, for example, 0 or 1.

`at`
: The keyword that precedes the controller to which the device is attached.

`ctlr_name``#`
: Specifies the name and logical unit number of the controller to which the device is attached. The controller name can be any string that matches the previously specified string for the controller entry as described in Section 12.6.2.21. For controllers supplied by Digital, the controller name is one of the valid strings listed in the *System*

Administration guide. For example, the string hsc represents an HSC controller.

Third-party driver writers who write drivers for non-Digital controllers can select a string that might include the vendor and product names. The string could also include version and release numbers. This type of naming scheme reduces the chance of name conflicts with other vendors. For example, the driver writers at EasyDriver Incorporated might specify edgc for an internally developed controller.

The logical unit number for a controller is any positive integer, for example, 0 or 1.

drive
: The keyword that precedes the physical unit number of the device.

phys#
: Specifies the physical unit number of the device, if required.

12.6.3 Stanza File Fragment Examples

The following example shows a stanza.loadable file fragment for the /dev/none device driver. The driver writers at EasyDriver Incorporated create this file in the directory /usr/sys/kits/ESA100, as discussed in Section 11.1.2.

```
none:
        Subsystem_Description = none device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESA100/none_kmod
        Module_Config_Name = none
        Module_Config1 = controller none0 at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = none
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

The following example shows a stanza.loadable file fragment for the /dev/cb device driver. The driver writers at EasyDriver Incorporated create this file in the directory /usr/sys/kits/ESB100, as discussed in

Section 11.1.2.

```
cb:
        Subsystem_Description = cb device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESB100/cb_kmod
        Module_Config_Name = cb
        Module_Config1 = controller cb at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = cb
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

The following example shows a `stanza.static` file fragment for the `/dev/none` device driver. The driver writers at EasyDriver Incorporated create this file in the directory `/usr/sys/kits/ESA100`, as discussed in Section 11.1.2. For descriptions of the fields that contain the driver interfaces, see Section 12.8.2.

```
none:
        Subsystem_Description = none device driver
        Module_Config_Name = none
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = none
        Device_Char_Open = noneopen
        Device_Char_Close = noneclose
        Device_Char_Read = noneread
        Device_Char_Write = nonewrite
        Device_Char_Ioctl = noneioctl
        Device_Char_Stop = nodev
        Device_Char_Reset = nodev
        Device_Char_Ttys = 0
        Device_Char_Select = nodev
        Device_Char_Mmap = nodev
        Device_Char_Funnel = DEV_FUNNEL_NULL
```

The following example shows a `stanza.static` file fragment for the `/dev/cb` device driver. The driver writers at EasyDriver Incorporated create this file in the directory `/usr/sys/kits/ESB100`, as discussed in Section 11.1.2. For descriptions of the fields that contain the driver interfaces, see Section 12.8.2.

```
cb:
        Subsystem_Description = cb device driver
        Module_Config_Name = cb
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = cb
        Device_Char_Open = cbopen
        Device_Char_Close = cbclose
        Device_Char_Read = cbread
        Device_Char_Write = cbwrite
        Device_Char_Ioctl = cbioctl
        Device_Char_Stop = nodev
        Device_Char_Reset = nodev
        Device_Char_Ttys = 0
        Device_Char_Select = nodev
        Device_Char_Mmap = nodev
        Device_Char_Funnel = DEV_FUNNEL_NULL
```

12.7 Specifying Information in the name_data.c File

A *name*`_data.c` file is compiled when the kernel is made and it is usually used to size data structures for static-only drivers. The *name*`_data.c` file is typically not used for loadable drivers. Driver writers are encouraged to dynamically allocate data structures, as described in Section 2.3.

The *name* argument is usually based on the device name. For example, the `none` device's *name*`_data.c` file is called `none_data.c`. The `CB` device's *name*`_data.c` file is called `cb_data.c`. The `edgd` device's *name*`_data.c` file would be called `edgd_data.c`.

12.8 Specifying Information in the conf.c File

The `conf.c` file contains two device switch tables called `bdevsw` and `cdevsw`. Section 8.2.1 and Section 8.2.2 describe in detail each of the members contained in these tables. This section is concerned with the mechanisms used by device driver writers to populate the device switch tables with the device driver entry points associated with their device drivers. The mechanisms are as follows:

- For static drivers

  For the third-party model, specify the device driver entry points in the `stanza.static` file fragment. For the traditional model, edit the `bdevsw` and/or `cdevsw` tables in the `conf.c` file with the driver entry points.

- For loadable drivers

  For block device drivers, add the entry points to the bdevsw table by calling the bdevsw_add interface in the configure section of the device driver.

  For character device drivers, add the entry points to the cdevsw table by calling the cdevsw_add interface in the configure section of the device driver.

The following sections show you how to populate the cdevsw table by using the cdevsw_add interface for loadable drivers and the stanza.static file fragment for static drivers.

12.8.1 Using cdevsw_add to Add Entries to the cdevsw Table

The following code fragment shows how a loadable driver uses the cdevsw_add interface to add a character device driver's entry points into the cdevsw table. This code fragment would appear in the configure section of the device driver.

```
/* Device switch structure for dynamic configuration */
   •
   •
   •
#include <sys/conf.h>
   •
   •
   •
struct cdevsw cb_cdevsw_entry = { [1]
  cbopen,           /* d_open */
  cbclose,          /* d_close */
  cbread,           /* d_read */
  cbwrite,          /* d_write */
  cbioctl,          /* d_ioctl */
  nodev,            /* d_stop */
  nodev,            /* d_reset */
  0,                /* d_ttys */
  nodev,            /* d_select */
  nodev,            /* d_mmap */
  DEV_FUNNEL_NULL   /* d_funnel */
};
   •
   •
   •
cdevno = cdevsw_add(cdevno,&cb_cdevsw_entry); [2]
   •
   •
   •
```

[1] Declares a cdevsw structure called cb_cdevsw_entry and initializes it to the device driver entry points associated with the /dev/cb device driver.

The cdevsw and bdevsw structures are defined in /usr/sys/include/sys/conf.h.

2 Calls the cdevsw_add interface to register the device driver entry points indicated in the comments in the cb_cdevsw_entry structure. The code fragment shows that the first argument is the device switch table entry (slot) to use. This slot would be obtained in a previous call to the makedev interface. Section 10.9.2 provides more details on how this is accomplished in the /dev/cb driver. The first argument represents the device driver's major number requirements. For most drivers, this argument is set in the stanza entry in such a way that the next available major number gets assigned.

The second argument is the address of the previously initialized cb_cdevsw_entry data structure. This structure is dynamically added to the cdevsw table by the cdevsw_add interface. The cdevsw_add interface adds the driver's entry points to the in-memory resident cdevsw table.

Upon return from the call to cdevsw_add, the *cdevno* variable is set to the assigned device major number. If this value is NODEV, the call to cdevsw_add failed. This prevents the driver from being configured as a loadable driver.

Using the bdevsw_add interface is the same as using cdevsw_add, except that the entries get filled in the in-memory resident bdevsw table instead of the in-memory resident cdevsw table. The device driver writer would initialize a structure of type bdevsw with the block driver's entry points.

12.8.2 Using the stanza.static File Fragment to Add Entries to the Device Switch Tables

The following describes the syntax used for adding static device driver interfaces to the bdevsw table.

Device_Block_Open = *d_open*
Device_Block_Close = *d_close*
Device_Block_Strategy = *d_strategy*
Device_Block_Dump = *d_dump*
Device_Block_Psize = *d_psize*
Device_Block_Flags = *d_flags*
Device_Block_Ioctl = *d_ioctl*
Device_Block_Funnel = *d_funnel*

`Device_Block_Open`	Specifies the block driver's open interface.
`Device_Block_Close`	Specifies the block driver's close interface.
`Device_Block_Strategy`	Specifies the block driver's strategy interface.
`Device_Block_Dump`	Specifies the block driver's dump interface.
`Device_Block_Psize`	Specifies the block driver's psize interface.
`Device_Block_Flags`	Specifies whether this is an SVR4 DDI/DKI compliant device driver. Set this field to the constant `B_DDIDKI` to indicate that this is an SVR4 DDI/DKI compliant device driver.
`Device_Block_Ioctl`	Specifies the block driver's `ioctl` interface.
`Device_Block_Funnel`	Schedules a device driver onto a CPU in a multiprocessor configuration. Because multiprocessor configurations are not supported on DEC OSF/1, set this field to the constant `DEV_FUNNEL_NULL`.

The following syntax is used to add static device driver interfaces to the `cdevsw` table. Any field not present is filled in with the `nulldev` interface. No particular field is required, but at least one must be present.

Device_Char_Open = *d_open*
Device_Char_Close = *d_close*
Device_Char_Read = *d_read*
Device_Char_Write = *d_write*
Device_Char_Ioctl = *d_ioctl*
Device_Char_Stop = *d_stop*
Device_Char_Reset = *d_reset*
Device_Char_Ttys = *d_ttys*
Device_Char_Select = *d_select*
Device_Char_Mmap = *d_mmap*
Device_Char_Funnel = *d_funnel*
Device_Char_Segmap = *d_segmap*
Device_Char_Flags = *d_flags*

`Device_Char_Open`	Specifies the character driver's open interface.
`Device_Char_Close`	Specifies the character driver's close interface.
`Device_Char_Read`	Specifies the character driver's read interface.

`Device_Char_Write`	Specifies the character driver's write interface.
`Device_Char_Ioctl`	Specifies the character driver's `ioctl` interface.
`Device_Char_Stop`	Specifies the character driver's stop interface.
`Device_Char_Reset`	Specifies the character driver's reset interface.
`Device_Char_Ttys`	Specifies the character driver's private data. Only terminal drivers use this field.
`Device_Char_Select`	Specifies the character driver's select interface.
`Device_Char_Mmap`	Specifies the character driver's memory map interface.
`Device_Char_Funnel`	Schedules a device driver onto a CPU in a multiprocessor configuration. Because multiprocessor configurations are not supported on DEC OSF/1, set this field to the constant `DEV_FUNNEL_NULL`.
`Device_Char_Segmap`	Specifies the segmap entry point.
`Device_Char_Flags`	Specifies whether this is an SVR4 DDI/DKI compliant device driver. Set this field to the constant `C_DDIDKI` to indicate that this is an SVR4 DDI/DKI compliant device driver.

12.9 Supplying the Subset Control Program

Digital provides the tools for creating kits for device driver products as part of the standard operating system distribution. These tools are described in the *Programming Support Tools* book. The kit developer at EasyDriver Incorporated reads the *Programming Support Tools* book to learn how to create a kit for the device driver products developed by EasyDriver Incorporated. The following list summarizes the steps in the kit-building process:

1. Understand the syntax of the `setld` utility.
2. Understand the files used by the `setld` utility.
3. Become familiar with the steps the `setld` utility performs when a system manager loads, configures, verifies, and removes the software subset or subsets on the device driver kit by invoking the appropriate option.

4. Learn how to effectively use the file system.
5. Create a kit for the `setld` utility. This step involves creating a subset control program (SCP) that performs special tasks beyond the basic installation managed by `setld` and building the kit.

The SCP is the program used with the `setld` utility to register third-party device driver subsets. After reading the information on creating and managing software product kits in the *Programming Support Tools* book, the kit developer at EasyDriver Incorporated decides on the strategy to follow for writing the SCPs associated with the driver products. The strategy you choose depends on how you want to market your device driver products. The following list presents some strategies for writing the SCP:

- Write one SCP for a kit that contains the software subset associated with the static `/dev/none` device driver.
- Write one SCP for a kit that contains the software subset associated with the loadable `/dev/none` device driver.
- Write one SCP for a kit that contains the software subset associated with the static `/dev/cb` device driver.
- Write one SCP for a kit that contains the software subset associated with the loadable `/dev/cb` device driver.
- Write one SCP for a kit that contains the software subsets associated with both the static and loadable versions of the `/dev/none` device driver.
- Write one SCP for a kit that contains the software subsets associated with both the static and loadable versions of the `/dev/cb` device driver.

The following example provides the SCP for the kit that contains the software subsets associated with both the static and loadable versions of the `/dev/cb` device driver:

```
#!/sbin/sh [1]
#
#
#    CB.scp - install files associated with the loadable and static
#             cb device driver product [2]
#

echo "*********** CB Device Driver Product Installation Menu ***********"
echo "***********                                             ***********"
echo "1. Install the static cb device driver subset."
echo "2. Delete the static cb device driver subset."
echo "3. Install the loadable cb device driver subset."
echo "4. Delete the loadable cb device driver subset."

echo" Type the number corresponding to your choice [] " [3]

read answer
case ${answer} in
   1)
   case "$ACT" in [4]
```

```
    POST_L) 5

     # Register the files associated with the static CB
     # device driver product.
     kreg -l EasyDriverInc EASYDRVCBSTATIC100 /usr/sys/kits/ESB100 6

     # Reminder
     echo "The CB device driver is installed on your system."
     echo "Before your utilities can make use of the driver, you"
     echo "must build a new kernel by running doconfig." 7
     ;;
    2)
    POST_D) 8
    kreg -d EASYDRVCBSTATIC100 9

    echo "The CB device driver is no longer on the system." 10
    echo "Remember to build a new kernel to remove the CB driver"
    echo "functionality."

    ;;

    3)
    POST_L) 11
    # Add the files associated with the loadable CB device
    # driver product to the customer's sysconfigtab database
    sysconfigdb -a -f /usr/sys/kits/ESB100/stanza.loadable cb 12

    # Cause the CB device driver to be automatically loaded each time
    # the system reboots
    sysconfigdb -on cb 13

    # Load the CB device driver and create the device special files
    sysconfig -c cb 14

    echo "The CB device driver was added to your global stanza"
    echo "database (sysconfigtab) and will automatically be loaded"
    echo "each time the system reboots." 15

    4)
    POST_D) 16
    # Make sure the CB device driver is not currently loaded
    sysconfig -u cb 17

    # Remove the CB device driver from the automatic startup list
    sysconfigdb -off cb 18

    # Delete the CB device driver's stanza entry from the global
    # stanza database
    sysconfigdb -d cb 19
    ;;
esac
exit 0
```

1 The kit developer for EasyDriver Incorporated follows Digital's recommendation to write the SCP as a script for `/sbin/sh`. Note that the kit developer supplies a menu of choices for installing or deleting the subsets associated with the `/dev/cb` device driver product. You would probably also want to supply an installation booklet that walks the customer through the installation and the deletion of the subsets.

2 The name of this SCP is `CB.scp` and it copies the files associated with the static and loadable versions of the `/dev/cb` device driver to a specific directory on the customer's system. It can also delete the subsets after they have been installed.

3 To install the software subsets associated with the `/dev/cb` driver product, the system manager enters the value `1`.

4 The `ACT` environment variable is set by `setld` when it invokes the SCP. In this SCP, the `ACT` environment variable can take the value `POST_L` or `POST_D`.

5 Specifies an `ACT` environment variable setting that indicates the tasks to be performed after loading the software subset. For the static `/dev/cb` device driver, the `kreg` utility performs these tasks.

6 The `kreg` utility registers a device driver product by creating the `/usr/sys/conf/.products.list` file on the customer's system. This file contains registration information associated with the static device driver product. In this call to `kreg`, the following flag and arguments are passed:

- The `-l` flag

  This flag indicates that the subset was loaded and it directs `kreg` to register this device driver layered product as a new kernel extension.

- The company name

  The company name is `EasyDriverInc`. The `kreg` utility places this name in the company name field of the customer's `/usr/sys/conf/.products.list` file.

- The software subset name

  The software subset name for this device driver product is EASYDRVCBSTATIC100. The subset name consists of the product code `EASYDRV`, the subset mnemonic `CBSTATIC`, and the 3-digit version code `100`. The `kreg` utility extracts information from the specified subset data and loads it into the customer's `/usr/sys/conf/.products.list` file.

- The directory name

  The directory on the customer's system where `kreg` copies the files associated with this driver product is `/usr/sys/kits/ESB100`. The `kreg` utility places this directory in the driver files path field of the customer's `/usr/sys/conf/.products.list` file.

7 This message displays on the console terminal after the files contained on the kit have been copied to the appropriate directory.

8 To delete the software subsets associated with the `/dev/cb` driver product that were previously installed, the system manager enters the

value 2 at the prompt `Type the number corresponding to your choice [ ]`. The ACT environment variable is set to `POST_D` by `setld` when it invokes the SCP. The `POST_D` variable indicates the tasks to be performed when deleting the software subset. The `kreg` utility performs these tasks.

9 In this call to `kreg`, the following flag and argument are passed:

- The `-d` flag

  This flag deletes the entry for the specified layered product from the customer's `/usr/sys/conf/.products.list` file when the customer removes the subset from the system.

- The software subset name

  The software subset name, `EASYDRVCBSTATIC100`, indicates that the static `/dev/cb` device driver product is to be removed from the customer's `/usr/sys/conf/.products.list` file.

10 This message displays on the console terminal after the entry for the device driver product has been removed from the customer's `/usr/sys/conf/.products.list` file.

11 To add the software subsets associated with the loadable `/dev/cb` driver product, the system manager enters the value 3 at the prompt `Type the number corresponding to your choice [ ]`. The ACT environment variable is set to `POST_L` by `setld` when it invokes the SCP. The `POST_L` variable indicates the tasks to be performed after loading the software subset.

For the loadable `/dev/cb` device driver, the `sysconfigdb` utility performs these tasks.

12 The `sysconfigdb` utility maintains and manages the `/etc/sysconfigtab` data base. For each driver product, this data base contains such items as device connectivity information, the driver's major number requirements, the names and minor numbers of the device special files, and the permissions and directory name where the device special files reside. In this call to `sysconfigdb` the following flags and arguments are passed:

- The `-a` flag

  Specifies that `sysconfigdb` add the device driver entry to the customer's `/etc/sysconfigtab` database.

- The `-f` flag

  Specifies the flag that precedes the name of the `stanza.loadable` file fragment whose device driver entry is to be added to the `/etc/sysconfigtab` database. This flag is used with the `-a` flag.

– The `stanza.loadable` file fragment

  The kit developer at EasyDriver Incorporated specifies `/usr/sys/kits/ESB100/stanza.loadable` to indicate the location of the `stanza.loadable` file fragment for the `/dev/cb` driver.

– The device driver name

  The kit developer at EasyDriver Incorporated specifies `cb` as the name of the driver whose associated information is added to the `/etc/sysconfigtab` database. Note that this name is obtained from the *`entry_name`* field of the `stanza.loadable` file fragment, as described in Section 12.6.1.

13 The kit developer at EasyDriver Incorporated calls `sysconfigdb` a second time with the `-on` flag, which causes the loadable `/dev/cb` device driver to be automatically loaded each time the customer reboots the system. The name of the driver as specified in the `stanza.loadable` file fragment follows the flag.

14 The kit developer at EasyDriver Incorporated calls the `sysconfig` utility with the `-c` flag, which configures the loadable `/dev/cb` device driver into the running system and creates the device special files. The name of the driver as specified in the `stanza.loadable` file follows the flag.

15 This message displays on the console terminal after `sysconfigdb` and `sysconfig` have performed their tasks.

16 To delete the software subsets associated with the loadable `/dev/cb` driver product that were previously installed, the system manager enters the value 4 at the prompt `Type the number corresponding to your choice [ ]`. The `ACT` environment variable is set to `POST_D` by `setld` when it invokes the SCP. The `POST_D` variable indicates the tasks to be performed when deleting the software subset.

These tasks are performed by the `sysconfig` and `sysconfigdb` utilities.

17 The kit developer at EasyDriver Incorporated calls the `sysconfig` utility with the `-u` flag, which unconfigures the loadable `/dev/cb` device driver from the running system. The name of the driver as specified in the `stanza.loadable` file fragment follows the flag.

18 The kit developer at EasyDriver Incorporated calls the `sysconfigdb` utility with the `-off` flag, which causes the loadable `/dev/cb` device driver to not be automatically configured during an early phase of system startup. It removes the `/dev/cb` driver from the `automatic` entry in the customer's `/etc/sysconfigtab` database.

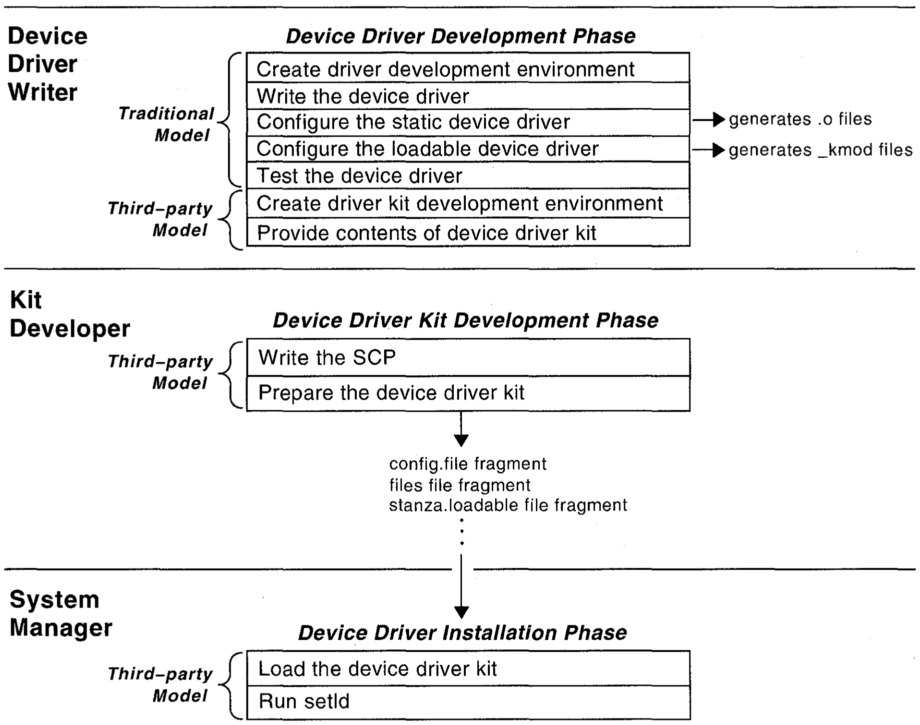
The name of the driver as specified in the `stanza.loadable` file fragment follows the flag.

19 The kit developer at EasyDriver Incorporated calls the `sysconfigdb` utility with the `-d` flag, which deletes the loadable `/dev/cb` device driver from the customer's `/etc/sysconfigtab` database. The name of the driver as specified in the `stanza.loadable` file fragment follows the flag.

Device Driver Configuration Examples 13

This chapter ties together the device driver configuration models presented in Chapter 11 and the device driver syntaxes and mechanisms presented in Chapter 12 by walking you through device driver configuration as it is accomplished by EasyDriver Incorporated. You can choose to follow this model or devise an alternate one that matches your device driver development environment. Figure 13-1 shows the steps the driver writer, kit developer, and system manager perform at EasyDriver Incorporated to configure the `/dev/none` and `/dev/cb` device drivers.

Figure 13-1: Device Driver Configuration as Done by EasyDriver Incorporated


Device Driver Writer
Device Driver Development Phase
Traditional Model
Create driver development environment
Write the device driver
Configure the static device driver
Configure the loadable device driver
Test the device driver
generates .o files
generates _kmod files
Third-party Model
Create driver kit development environment
Provide contents of device driver kit
Kit Developer
Device Driver Kit Development Phase
Third-party Model
Write the SCP
Prepare the device driver kit
config.file fragment
files file fragment
stanza.loadable file fragment
System Manager
Device Driver Installation Phase
Third-party Model
Load the device driver kit
Run setld

As the figure shows, EasyDriver Incorporated organizes its driver development into a:

- Device driver development phase
- Kit development phase
- Driver installation phase

Note that the figure identifies the audiences expected to complete each of the tasks. The figure also identifies which tasks are associated with the traditional and third-party device driver configuration models. The tasks associated with each phase are discussed in the following sections.

13.1 Device Driver Development Phase

The device driver writers at EasyDriver Incorporated perform the following tasks during the device driver development phase:

- Create the device driver development environment
- Write the device driver
- Configure the static device driver
- Configure the loadable device driver
- Test the device driver
- Create the device driver kit development environment
- Provide the contents of the device driver kit

13.1.1 Creating the Device Driver Development Environment

The driver writers at EasyDriver Incorporated create a device driver development environment following the traditional device driver configuration model discussed in Section 11.2. This section discusses one possible directory structure for locating the files associated with the device driver.

Follow the guidelines presented in Section 11.2 to create a driver development environment suitable to your needs.

13.1.2 Writing the Device Driver

The driver writers at EasyDriver Incorporated write their device drivers, using the techniques described in this book. They use the guidelines presented in Section 2.1.2.1 for naming the device driver source file. For the `/dev/none` device driver, the source file is called `none.c`. For the `/dev/cb` device driver, the source file is called `cb.c`.

Follow the guidelines presented in Section 2.1.2.1 for naming your device driver source files.

13.1.3 Configuring the Static Device Driver

The driver writers at EasyDriver Incorporated configure the static versions of the `/dev/none` and `/dev/cb` device drivers by following the steps provided by the traditional device driver configuration model:

1. Make an entry in the `tc_option_data.c` table (TURBOchannel specific).
2. Compile and link the static device driver.

3. Back up the new kernel.

The steps described in the following sections apply to device drivers written for the TURBOchannel bus. These steps might differ for drivers written for other buses. See the bus-specific device driver manual on how to configure drivers for the specific bus, using the traditional model.

13.1.3.1 Step 1: Make an Entry in the tc_option_data.c Table

The driver writers at EasyDriver Incorporated make an entry in the `tc_option` table, located in the `/usr/sys/data/tc_option_data.c` file. The `tc_option` table provides a mapping between the device name in the read-only memory (ROM) on the hardware device module and the driver in the DEC OSF/1 kernel. This step is specific to drivers written for the TURBOchannel bus. Other buses may require some other task to be performed.

The following shows the `tc_option` table in the Digital-provided `tc_option_data.c` file:

```
struct tc_option tc_option [] =
{
/* module      driver     intr_b4     itr_aft             adpt    */
/* name        name       probe       attach     type     config  */
/* ------      ------     -------     -------    ----     ------  */

{ "PMTNV-AA", "nvtc",    0,          1,         'C',     0}, /* TCNVRAM */
{ "PMAP-AA ", "nvtc",    0,          1,         'C',     0}, /* TCNVRAM temp */
{ "PMAD-AA ", "ln",      0,          1,         'C',     0}, /* Lance */
{ "PMAF-AA ", "fza",     1,          1,         'C',     0}, /* FDDI */
{ "PMAF-FA ", "fta",     0,          1,         'C',     0}, /* FDDI */
{ "PMAZ-AA ", "asc",     0,          1,         'C',     0}, /* SCSI */
{ "PMAZ-DS ", "asc",     1,          1,         'A',     tscsiconf}, /* TCDS */
{ "PMAZB-AA", "asc",     1,          1,         'A',     tscsiconf}, /* TCDS */
{ "PMAZB-AB", "asc",     1,          1,         'A',     tscsiconf}, /* TCDS */
{ "PMAG-BA ", "fb",      0,          0,         'C',     0}, /* CFB */
{ "PMAG-AA ", "fb",      0,          0,         'C',     0}, /* MFB */
{ "PMAGB-BA", "fb",      0,          1,         'C',     0}, /* SFB */
{ "PMAG-RO ", "fb",      0,          0,         'C',     0}, /* RO*/
{ "PMAG-JA ", "fb",      0,          0,         'C',     0}, /* RO*/
{ "PMAG-CA ", "px",      0,          1,         'C',     0}, /* 2DA */
{ "PMAG-DA ", "px",      0,          1,         'C',     0}, /* LM-3DA */
{ "PMAG-FA ", "px",      0,          1,         'C',     0}, /* HE-3DA */
{ "PMAG-FB ", "px",      0,          1,         'C',     0}, /* HE+3DA */
{ "PMAGB-FA", "px",      0,          1,         'C',     0}, /* HE+3DA */
{ "PMAGB-FB", "px",      0,          1,         'C',     0}, /* HE+3DA */
{ "PMAGZ-PV", "pv",      0,          1,         'C',     0}, /* PV+3DA */
#ifdef mips
{ "PMABV-AA", "vba",     1,          1,         'A',     xviaconf}, /* VME */
{ "CITCA-AA", "ci",      0,          1,         'A',     tcciconf},/* CI */
#endif /* mips */


/*
* 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.
*/

    /*
%%%Used by mktcdata as placemarker for automatic installation
   */

/*
* 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. You must blank-pad the names to 8 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 driver's `probe` interface. A zero (0) value indicates that the device does not need interrupts enabled; a value of 1 indicates that the device needs interrupts enabled.

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

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

adpt config
: If the device in the type column is `A` (adapter), you specify the name of the interface to configure the adapter. Otherwise, you specify the value zero (0).

The entries for the /dev/none and /dev/cb drivers are as follows:

```
{  "NONE    ", "none", 0, 1, 'C', 0}, /* None */
{  "CB      ", "cb", 0, 1, 'C', 0}, /* cb */
```

Make similar entries in this table if your device driver operates on the TURBOchannel bus.

13.1.3.2 Step 2: Compile and Link the Static Device Driver

To compile and link the static device driver, the driver writers at EasyDriver Incorporated perform the following tasks:

- Back up files.
- Make an entry in the system configuration file.
- Add the driver source to the `files` file.
- Declare the device driver entry points in `/usr/sys/io/common/conf.c`.
- Modify the `bdevsw` or `cdevsw` table.
- Run the `config` program.
- Create a new kernel.

Each of these tasks is discussed in the following sections.

Step 2a: Back Up Files

The driver writers at EasyDriver Incorporated use the traditional device driver configuration model to configure their device drivers during the initial stages of development. Because they will later test the third-party device driver configuration model, the driver writers do not want to make permanent edits to their system configuration file, `files` file, and `conf.c` file. If your driver development environment resembles that of EasyDriver Incorporated, you will probably want to back up these files. The following shows one way to accomplish this task:

```
%cd /usr/sys/conf/CONRAD
%cp CONRAD CONRAD.save
%cd /usr/sys/conf/Alpha
%cp files files.save
%cd /usr/sys/io/common
%cp conf.c conf.c.save
```

Step 2b: Make an Entry in the System Configuration File

Make an entry in `/usr/sys/conf/`*NAME*, the system configuration file, to add the device to the system. The *NAME* variable represents the name of the system you want to configure, for example, CONRAD. Because the

/dev/none and /dev/cb drivers are developed for use on the TURBOchannel bus, the device entry must follow the syntaxes associated with the TURBOchannel, as follows:

```
controller none0 at tc0 slot? vector noneintr
controller cb0 at tc0 slot? vector cbintr
```

Step 2c: Add the Driver Source to the files File

Make an entry in /usr/sys/conf/Alpha/files as either Binary (no driver sources are supplied) or Notbinary (driver sources are supplied). The following example shows the entries for the /dev/none and /dev/cb device drivers without source code:

```
io/EasyInc/none.c  optional  none device-driver Binary
io/EasyInc/cb.c  optional  cb device-driver Binary
```

The following example shows the entries for the /dev/none and /dev/cb device drivers with source code:

```
io/EasyInc/none.c  optional  none device-driver Notbinary
io/EasyInc/cb.c  optional  cb device-driver Notbinary
```

Step 2d: Declare the Device Driver Entry Points in conf.c

Declare the device driver entry points for your device by editing the /usr/sys/io/common/conf.c file. The following example shows the device driver interface declarations for the /dev/none and /dev/cb device drivers:

```
#include <none.h>
#if NNONE > 0
int     noneopen(),noneclose(),noneread(),nonewrite(),noneioctl();
int     nonereset();
#else
#define noneopen          nodev
#define noneclose         nodev
#define noneread          nodev
#define nonewrite         nodev
#define noneioctl         nodev
#define nonereset         nodev

   •
   •
   •
#include <cb.h>
#if NCB > 0
int     cbopen(),cbclose(),cbread(),cbwrite(),cbioctl();
int     cbselect(),cbmmap();
#else
#define cbopen           nodev
#define cbclose          nodev
#define cbread           nodev
```

```
#define cbwrite         nodev
#define cbioctl         nodev
#define cbselect        nodev
#define cbmmap          nodev
```

First, you include the device driver header file that was created by `config`. The `config` program 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 files are `none.h` and `cb.h`. These names indicate that the characters ''none'' and ''cb'' were previously specified for these devices in the system configuration file.

Next, you declare the device driver interfaces that were defined in the `bdevsw` or `cdevsw` table if the device constant (or constants) are greater than zero, which indicates that the device was 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 constants are `NNONE` and `NCB` and the `none` and `cb` interfaces defined in the `cdevsw` table are declared to be of type `int`. Otherwise, if the device is not actually in the system configuration file, you declare the entry points as `nodev`.

Step 2e: Modify the bdevsw or cdevsw Table

To modify the `bdevsw` or `cdevsw` table, edit the `/usr/sys/io/common/conf.c` file and search for `struct bdevsw` or `struct cdevsw`. 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 your comment is your device major number. Keep this number for use in a subsequent step. The following example shows the entries for the `/dev/none` and `/dev/cb` device drivers along with the entry that precedes them:

```
struct cdevsw   cdevsw[MAX_CDEVSW] =
{
   •
   •
   •
/* STREAMS clone device */
{clone_open,    nodev,    nodev,          nodev, /*32*/
 nodev,         nodev,    nodev,          0,
 nodev,         nodev,    DEV_FUNNEL_NULL },
   •
   •
   •
/* Example none device */
{noneopen,    noneclose,    noneread,    nonewrite, /*33*/
 noneioctl,   nodev,        nodev,   0,
 nodev,       nodev,      DEV_FUNNEL_NULL },
/* cb device */
{cbopen,    cbclose,    cbread,    cbwrite, /*34*/
 cbioctl,   nodev,      nodev      0,
 nodev,  nodev,     DEV_FUNNEL_NULL },
   •
   •
   •
```

Step 2f: Run config on the System Configuration File

Run `config` on the system configuration file from the `/usr/sys/conf` directory.

In the following example, `config` is run on the system called CONRAD:

```
%cd /sys/conf
%./config CONRAD
```

Step 2g: Create a Device Special File

Create a device special for your device, using the `mknod` command. The following example shows the entries for the devices associated with the `/dev/none` and `/dev/cb` device drivers:

```
% mknod /dev/none c 33 0 [1]
% mknod /dev/cb c 34 1 [2]
```

[1] The first entry describes the `none` device. The letter `c` represents character device, as opposed to `b` for block device. The number 33 is the major number you were told to record when you added the device to the `cdevsw` table. The value zero (0) is the minor number associated with this device.

[2] The second entry describes the `/dev/cb` device. The number 34 is the major number and 1 is the minor number.

Step 2h: Create a New Kernel

Create a new kernel by going to the `/usr/sys/NAME` directory, which was created by `config`.

The driver writers at EasyDriver Incorporated specify the following; you would specify something similar:

```
%cd /usr/sys/CONRAD
%make depend
%make
```

Some common errors encountered when creating a new kernel are coding errors, especially if the driver was defined as `Notbinary`. In addition, you may obtain errors from the system configuration file, the `files` file, and the `conf.c` file.

13.1.3.3 Step 3: Back Up the New Kernel

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`, as in the following example:

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

Use the following steps in Section 13.1.3.2 for specific modifications:

- After modifying any driver source code, start with step 2g.
- To add a new device, start with step 2e.
- To change vectors, perform steps 2a, 2d, 2g, and 3.
- To add entry points, perform steps 2c, 2d, 2e, 2g, and 3.

13.1.4 Configuring the Loadable Device Driver

The driver writers at EasyDriver Incorporated now configure the loadable versions of the `/dev/none` and `/dev/cb` device drivers by following the steps provided by the traditional device driver configuration model:

1. Create a `stanza.loadable` file fragment.
2. Compile and link the device driver.
3. Run the `sysconfigdb` utility.
4. Run the `sysconfig` utility

The steps described in the following sections apply to device drivers written for the TURBOchannel bus. These steps might differ for drivers written for other buses. See the bus-specific device driver manual on how to configure drivers for the specific bus, using the traditional model.

13.1.4.1 Step 1: Create a stanza.loadable File Fragment

Create a `stanza.loadable` file fragment for each of your driver products. Section 12.6 discusses the format and syntaxes associated with the `stanza.loadable` file fragment. The driver writers at EasyDriver

Incorporated create the following `stanza.loadable` file fragment for the `/dev/none` device driver:

```
none:
        Subsystem_Description = none device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESA100/none_kmod
        Module_Config_Name = none
        Module_Config1 = controller none0 at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = none
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

They also create the following `stanza.loadable` file fragment for the `/dev/cb` device driver:

```
cb:
        Subsystem_Description = cb device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESB100/cb_kmod
        Module_Config_Name = cb
        Module_Config1 = controller cb at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = cb
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

13.1.4.2 Step 2: Compile and Link the Device Driver

To compile and link the loadable device driver, perform the following tasks:

- Add the driver source to the `files` file.
- Make an entry in the `BINARY` system configuration file.
- Run the `config` program.

- Rebuild the BINARY makefile.
- Build the driver load module.

Each of these tasks is discussed in the following sections.

Step 2a: Add the Driver Source to the files File

Make an entry in /usr/sys/conf/alpha/files as Binary (no driver sources are supplied). The following example shows the entries for the /dev/none and /dev/cb device drivers without source code:

```
io/EasyInc/none.c optional none device-driver if_dynamic none
Binary
io/EasyInc/cb.c optional cb device-driver if_dynamic cb Binary
```

Note that the specification for a loadable driver in the files file is identical to that for a static driver except for the use of the keyword if_dynamic followed by the *key_string*. This keyword marks the specified device driver source files such that the resulting object files can be built as either static or loadable.

Step 2b: Make an Entry in the BINARY System Configuration File

For the static versions of the /dev/none and /dev/cb device drivers, you made appropriate entries in the system configuration file that added their associated devices to the system. The config program uses these entries to create a system configuration tree and interrupt handler description for the static kernel. You do not describe loadable device drivers in the system configuration file in the same way as static drivers. The reason for this is loadable drivers are dynamically added to the system configuration tree and their interrupt handlers are dynamically registered. In addition, entries for loadable drivers are specified in the BINARY system configuration file. The following shows the entries in the BINARY system configuration file for the /dev/none and /dev/cb device drivers:

```
pseudo-device none dynamic none
pseudo-device cb dynamic cb
```

The use of the keyword pseudo-device ensures that the appropriate makefile is generated. The dynamic keyword specifies that the device driver is built as a dynamic load module (loadable driver).

Step 2c: Run config on the System Configuration File

Run `config` on the `BINARY` system configuration file from the `/usr/sys/conf` directory, as follows:

```
%cd /sys/conf
%./config BINARY
```

Step 2d: Construct the BINARY makefile

This step rebuilds the `makefile` in the `/usr/sys/BINARY` directory. This directory was created by the `config` program. This `makefile` specifies the operations needed to build the load module for the driver. These operations include the syntax used to compile and link the driver.

To build the `makefile` for the `/dev/none` and `CB` device drivers, the driver writers at EasyDriver Incorporated specify the following commands:

```
%cd /usr/sys/BINARY
%make depend
```

Use this example as a guide to specify the correct instruction and command for rebuilding your `makefile`.

Step 2e: Build the Driver Load Module

After the `makefile` has been rebuilt in `/usr/sys/BINARY`, you build the driver load module by calling `make` followed by the device driver name, as specified in the `stanza.loadable` file fragment. To build the driver load modules for the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated specify the following commands:

```
%cd /usr/sys/BINARY
%make none
%make cb
```

Upon successfully completing the previously described steps, the driver writers at EasyDriver Incorporated notice two device driver load modules in `/usr/sys/BINARY`: `none_kmod` and `cb_kmod`.

Use this example as a guide to specify the correct commands for building the driver load modules for your device drivers. Note the resulting load modules, which are of the form *xx*`_kmod` , where *xx* represents the name of your device driver and `_kmod` is the extension that identifies the load module.

13.1.4.3 Step 3: Run the sysconfigdb Utility

You use the `sysconfigdb` utility to manage and maintain `/etc/sysconfigtab`, the global stanza database. This database contains the information specified in the `stanza.loadable` file fragment for the loadable driver. To add the information contained in the `stanza.loadable` file fragments for the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated specify the following commands:

```
%sysconfigdb -a -f /usr/sys/kits/ESA100/stanza.loadable none
%sysconfigdb -a -f /usr/sys/kits/ESB100/stanza.loadable cb
```

The `-a` and `-f` flags cause `sysconfigdb` to add the information contained in the `stanza.loadable` file fragments for the `/dev/none` and `/dev/cb` device drivers to EasyDriver Incorporated's `/etc/sysconfigtab` database. See the *Reference Pages Section 8* for additional information on the `sysconfigdb` utility and its associated flags.

Use this example as a guide to specify the correct command for adding the informaton contained in the `stanza.loadable` file fragments associated with your device drivers to your `/etc/sysconfigtab` database.

13.1.4.4 Step 4: Run the sysconfig Utility

To dynamically configure the device driver and create the corresponding device special files, use the `sysconfig` utility. To dynamically configure the `/dev/none` and `/dev/cb` device drivers and create their corresponding device special files, the driver writers at EasyDriver Incorporated specify the following commands:

```
%sysconfig -c none
%sysconfig -c cb
```

As the example shows, the `-c` flag causes `sysconfig` to configure the `/dev/none` and `/dev/cb` device drivers into the running system and create the device special files. See the *Reference Pages Section 8* for additional information on the `sysconfig` utility and its associated flags.

Use this example as a guide to specify the correct command for configuring your device drivers into the running system and creating the device special files.

To verify that their device drivers are currently configured, the driver writers at EasyDriver Incorporated specify the following command:

```
%sysconfig -q none
%sysconfig -q cb
```

The `-q` flag causes `sysconfig` to display information about the `/dev/none` and `/dev/cb` device drivers. Use this example as a guide to specify the correct command for querying your device drivers.

To iteratively develop the loadable driver, you can unload the driver, make changes to the device driver source, compile and link the driver, and then load the driver again. You can unload the driver by using the `sysconfig` utility as follows:

```
%sysconfig -u none
```

To rebuild the loadable driver, follow Step 2e: Build the Driver Load Module. To reload the loadable driver, follow Step 4: Run the sysconfig Utility.

13.1.5 Testing the Device Driver

The driver writers at EasyDriver Incorporated test the device driver to ensure that it works with the associated utilities on the DEC OSF/1 operating system. They repeat the previously described tasks until the device driver works. This ends the device driver development phase.

You will probably perform similar testing and will most likely repeat the previous tasks just like the device driver writers at EasyDriver Incorporated.

13.1.6 Creating the Device Driver Kit Development Environment

The driver writers at EasyDriver Incorporated are now ready to create the device driver kit development environment that was discussed in Section 11.1.2. Create your driver kit development environment by following the recommendations provided in that section.

13.1.7 Providing the Contents of the Device Driver Kit

The driver writers at EasyDriver Incorporated plan to supply their customers with the static and loadable binary versions of the `/dev/none` and `/dev/cb` device drivers. Thus, they provide to their kit developers the following file fragments and files that become the contents of the device driver kit:

- `config.file` file fragment
- `files` file fragment
- `stanza.loadable` file fragment
- `stanza.static` file fragment
- Device driver objects
- Device driver load modules

Because the driver writers at EasyDriver Incorporated are shipping the binary versions of the device drivers, they do not provide the header or source files for the `/dev/none` and `/dev/cb` device drivers. Note, also, that they are not supplying any *name*`_data.c` files because they did not dynamically allocate any data structures. See Section 11.1.3 for the table that summarizes the files and file fragments supplied to the kit developer, based on whether the driver is shipped in source or binary versions.

The following sections show the contents of these files as they apply to the driver products for EasyDriver Incorporated.

13.1.7.1 Providing the Contents of the config.file File Fragment

For the static binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following `config.file` file fragment to their kit developers:

```
# Entries in config.file for none and CB devices

controller none0 at tc? vector noneintr
controller cb0   at tc? vector cbintr
```

13.1.7.2 Providing the Contents of the files File Fragment

For the static binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following `files` file fragment to their kit developers:

```
# This example illustrates the third-party model
# by showing a files file fragment for static
# drivers developed by EasyDriver Incorporated.

ESA100/none.c standard none device-driver if_dynamic none Binary

ESB100/cb.c standard cb device-driver if_dynamic cb Binary
```

13.1.7.3 Providing the Contents of the stanza.loadable File Fragment

For the loadable binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following `stanza.loadable` file fragment to their kit developers:

```
none:
        Subsystem_Description = none device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESA100/none_kmod
        Module_Config_Name = none
        Module_Config1 = controller none0 at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = none
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

```
cb:
        Subsystem_Description = cb device driver
        Method_Name = device
        Method_Type = Static
        Method_Path = None
        Module_Type = Dynamic
        Module_Path = /usr/sys/kits/ESB100/cb_kmod
        Module_Config_Name = cb
        Module_Config1 = controller cb at tc*
        Device_Dir = /dev
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = cb
        Device_User = root
        Device_Group = 0
        Device_Mode = 666
```

13.1.7.4 Providing the Contents of the stanza.static File Fragment

For the static binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following `stanza.static` file fragment to their kit developers:

```
none:
        Subsystem_Description = none device driver
        Module_Config_Name = none
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = none
        Device_Char_Open = noneopen
        Device_Char_Close = noneclose
        Device_Char_Read = noneread
        Device_Char_Write = nonewrite
        Device_Char_Ioctl = noneioctl
        Device_Char_Stop = nodev
        Device_Char_Reset = nodev
        Device_Char_Ttys = 0
        Device_Char_Select = nodev
        Device_Char_Mmap = nodev
        Device_Char_Funnel = DEV_FUNNEL_NULL
```

```
cb:
        Subsystem_Description = cb device driver
        Module_Config_Name = cb
        Device_Char_Major = Any
        Device_Char_Minor = 0
        Device_Char_Files = cb
        Device_Char_Open = cbopen
        Device_Char_Close = cbclose
        Device_Char_Read = cbread
        Device_Char_Write = cbwrite
        Device_Char_Ioctl = cbioctl
        Device_Char_Stop = nodev
        Device_Char_Reset = nodev
        Device_Char_Ttys = 0
        Device_Char_Select = nodev
        Device_Char_Mmap = nodev
        Device_Char_Funnel = DEV_FUNNEL_NULL
```

13.1.7.5 Providing the Device Driver Object Files

For the static binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following device driver object files. These object files were created as a result of following the steps beginning in Section 13.1.3.1.

- `none.o`
- `cb.o`

13.1.7.6 Providing the Device Driver Load Modules

For the loadable binary versions of the `/dev/none` and `/dev/cb` device drivers, the driver writers at EasyDriver Incorporated provide the following device driver load modules. These load modules were created as a result of following the steps beginning in Section 13.1.4.1.

- `none_kmod`
- `cb_kmod`

13.2 Device Driver Kit Development Phase

The kit developer at EasyDriver Incorporated performs the following tasks during the device driver kit development phase:

- Writes the SCP.
- Prepares the device driver kit.

13.2.1 Writing the SCP

As part of the kit development phase, the kit developer at EasyDriver Incorporated writes a subset control program (SCP) such as the one described in Section 12.9. Your kit developers can use that example as a guide for writing their own SCPs. In addition, they can refer to the *Programming Support Tools* book for details on writing an SCP.

13.2.2 Preparing the Device Driver Kit

As part of the kit development phase, the kit developer at EasyDriver Incorporated prepares the device driver kit, following the guidelines presented in Section 11.1.4. This section refers to the *Programming Support Tools* book, which provides complete details about preparing software distribution kits that are compatible with the `setld` utility.

Your kit developers can also follow the guidelines presented in that section to prepare their device driver kits.

13.3 Device Driver Installation Phase

The system manager at EasyDriver Incorporated performs the following tasks to install the `/dev/none` and `/dev/cb` device drivers:

- Restores the backed up files
- Loads the device driver kit.
- Runs the `setld` utility.

13.3.1 Restoring the Backed Up Files

The driver writers at EasyDriver Incorporated previously used the traditional device driver configuration model to configure their device drivers during the initial stages of development. They backed-up their system configuration file, `files` file, and `conf.c` file, to avoid making permanent edits. Before loading the device driver kit, the system manager at EasyDriver Incorporated restores the previously backed-up files. If you previously backed up these files, you will probably want to restore them at this time. The following shows one way to accomplish this task:

```
%cd /usr/sys/conf/CONRAD
%mv CONRAD.save CONRAD
%cd /usr/sys/conf/Alpha
%mv files.save files.
%cd /usr/sys/io/common
%mv conf.c.save conf.c
```

13.3.2 Loading the Device Driver Kit

The system manager at EasyDriver Incorporated loads the device driver kit, following instructions provided by the device driver writers and kit developer. The system manager ensures that the instructions are clear and concise and that the installation of the device drivers is successful.

You can perform similar testing by having your system manager install the device driver kit. You can also provide instructions on how to install the kit.

13.3.3 Running the setld Utility

The system manager at EasyDriver Incorporated is instructed to type the following command:

```
setld -l /dev/rmt0h
```

The `setld` utility invokes the subset control program (SCP) that copies the driver-related files from the kit to the customer's system. The driver writers at EasyDriver Incorporated created the SCP that was discussed in Section 12.9. That SCP displays a prompt that asks whether to install the static or loadable version of the `/dev/cb` device driver. If the static version is selected, the SCP calls the `kreg` utility, which does the following:

- Registers the device driver product by creating the `/usr/sys/conf/.products.list` file on the system. This file contains registration information associated with the static device driver product.
- Loads the data that controls how to include the device driver product in the kernel build process.

After these tasks are complete, the SCP instructs you to run `doconfig` to build a new kernel and thus make the `/dev/cb` driver available to the system utilities. The driver writers at EasyDriver Incorporated run `doconfig`, which automatically does all of the tasks described in the traditional model.

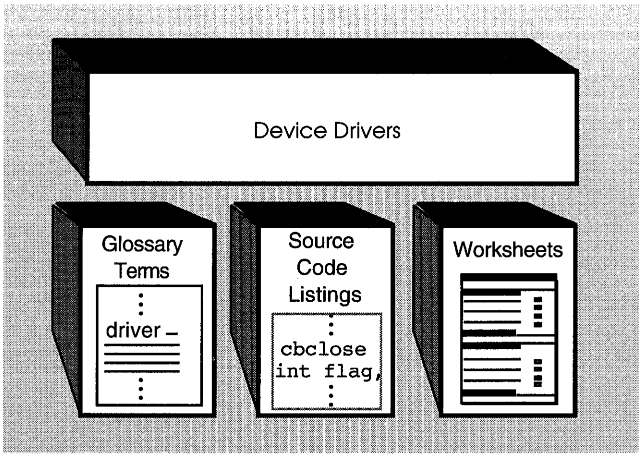
If the loadable version is selected, the SCP calls the `sysconfigdb` utility, which does the following:

- Adds the files associated with the loadable `/dev/cb` device driver product to the `/etc/sysconfigtab` database.
- Causes the `/dev/cb` device driver to be automatically loaded each time the system reboots.

The SCP also calls the `sysconfig` utility, which does the following:

- Loads the `/dev/cb` device driver and creates the device special files.
- Displays a prompt indicating that the `/dev/cb` device driver was added to the `/etc/sysconfigtab` database and that the loadable driver will automatically be loaded each time the system reboots.

Part 7 Appendixes



Summary Tables A

This appendix presents tables that summarize:

- Header files
- Kernel interfaces
- ioctl commands
- Global variables
- Data structures
- Device driver interfaces
- Bus configuration interfaces

A.1 List of Header Files

Table A-1 lists the header files related to device drivers, along with short descriptions of their contents. For convenience, the files are listed in alphabetical order. Note that device drivers should include header files that use the relative pathname instead of the explicit pathname. For example, although `buf.h` resides in `/usr/sys/include/sys/buf.h`, device drivers should include it as:

```
<sys/buf.h>
```

Chapter 1 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the header files listed in the table.

Table A-1: Summary Descriptions of Header Files

Header File	Contents
`buf.h`	Defines the `buf` structure.
`conf.h`	Defines the `bdevsw` (block device switch) and `cdevsw` (character device switch) tables.
`cpu.h`	Defines structures and constants related to the CPU.
`devdriver.h`	Defines the structures, constants, and external interfaces that device drivers and the autoconfiguration software use.

Table A-1: (continued)

Header File	Contents
devdriver_loadable.h	Defines constants and declares external functions associated with loadable drivers.
devio.h	Defines common structures and definitions for device drivers and ioctl requests.
disklabel.h	Defines structures and macros that operate on DEC OSF/1 disk labels.
errno.h	Defines the error codes returned to a user process by a device driver.
fcntl.h	Defines I/O mode flags supplied by user programs to open and fcntl system calls.
ioctl.h	Defines commands for ioctl interfaces in different device drivers.
iotypes.h	Defines constants used for 64-bit conversions.
kernel.h	Defines global variables that the kernel uses.
map.h	Defines structures associated with resource allocation maps.
mman.h	Defines constants associated with the mmap kernel interface.
mode.h	Defines constants that driver interfaces use.
mtio.h	Defines commands and structures for magnetic tape operations.
param.h	Defines constants and interfaces that the kernel uses.
poll.h	Defines polling bit masks.
proc.h	Defines the proc structure, which defines a user process.
sched_prim.h	Defines scheduling interfaces.
security.h	Defines structures, constants, and data types that UNIX security software uses.
sysconfig.h	Defines operation codes and data structures used in loadable device driver configuration.
systm.h	Defines generic kernel global variables.
time.h	Contains structures and symbolic names that time-related interfaces use.
types.h	Defines system data types and major and minor device interfaces.
uio.h	Contains the definition of the uio structure.
user.h	Defines the user structure.
vm.h	Contains a sequence of include statements that includes all of the virtual memory-related files.
vmmac.h	Contains definitions for byte conversions.

A.2 List of Kernel Support Interfaces

Table A-2 lists the kernel interfaces used by device drivers. Chapter 2 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the kernel interfaces listed in the table.

Note

Device drivers use the following header files most frequently:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <io/common/devdriver.h>
#include <sys/uio.h>
#include <machine/cpu.h>
```

Table A-2: Summary Descriptions of Kernel Support Interfaces

Kernel Interface	Summary Description
BADADDR	Probes the address during device autoconfiguration.
bcmp	Compares two byte strings.
bcopy	Copies a series of bytes with a specified limit.
bdevsw_add	Adds entry points in the block device switch table.
bdevsw_del	Deletes entry points from the block device switch table.
blkclr	Zeros a block of memory.
btop	Converts a virtual address to a kernel page frame number.
BUF_LOCK	Locks the specified I/O buffer.
BUF_UNLOCK	Unlocks the specified I/O buffer.
bzero	Zeros a block of memory.
cdevsw_add	Adds entry points in the character device switch table.
cdevsw_del	Deletes entry points from the character device switch table.
copyin	Copies data from a user address space to a kernel address space.
copyinstr	Copies a null-terminated string from a user address space to a kernel address space.
copyout	Copies data from a kernel address space to a user address space.
copyoutstr	Copies a null-terminated string from a kernel address space to a user address space.
copystr	Copies a null-terminated character string with a specified limit.
copy_to_phys	Copies data from a virtual address to a physical address.
DELAY	Delays the calling interface a specified number of microseconds.

Table A-2: (continued)

Kernel Interface	Summary Description
`disable_option`	Disables a device's interrupt line to the processor.
`dma_get_curr_sgentry`	Returns a pointer to the current `sg_entry`.
`dma_get_next_sgentry`	Returns a pointer to the next `sg_entry`.
`dma_get_private`	Gets a data element from the DMA private storage space.
`dma_kmap_buffer`	Returns a kernel segment (kseg) address of a DMA buffer.
`dma_map_alloc`	Allocates resources for DMA data transfers.
`dma_map_dealloc`	Releases and deallocates resources for DMA data transfers.
`dma_map_load`	Loads and sets allocated system resources for DMA data transfers.
`dma_map_unload`	Unloads system DMA resources.
`dma_min_boundary`	Returns system-level information.
`dma_put_curr_sgentry`	Puts a new bus address/byte count in the linked list of `sg_entry` structures.
`dma_put_prev_sgentry`	Puts a new bus address and byte count.
`dma_put_private`	Stores a data element in the DMA private storage space.
`do_config`	Initializes board to its assigned configuration.
`drvr_register_shutdown`	Registers or deregisters a shutdown interface.
`dualdevsw_add`	Adds entry points to the block device switch and character device switch tables.
`dualdevsw_del`	Deletes entry points from the block device switch and character device switch tables.
`enable_option`	Enables a device's interrupt line to the processor.
`ffs`	Finds the first set bit in a mask.
`fubyte`	Returns a byte from user data address space.
`fuibyte`	Returns a byte from user data address space.
`fuiword`	Returns a word from user data address space.
`fuword`	Returns a word from user data address space.
`get_config`	Returns assigned configuration data for a device.
`gsignal`	Sends a signal to a process group.
`handler_add`	Registers a device driver's interrupt service interface.
`handler_del`	Deregisters a device driver's interrupt service interface.
`handler_disable`	Disables a previously registered interrupt service interface.
`handler_enable`	Enables a previously registered interrupt service interface.
`htonl`	Converts longword values from host-to-network byte order.
`htons`	Converts word values from host-to-network byte order.

Table A-2: (continued)

Kernel Interface	Summary Description
`insque`	Adds an element to the queue.
`io_copyin`	Copies data from bus address space to system memory.
`io_copyio`	Copies data from bus address space to bus address space.
`io_copyout`	Copies data from system memory to bus address space.
`io_zero`	Zeros a block of memory in bus address space.
`iodone`	Indicates that I/O is complete.
`IS_KSEG_VA`	Determines if the specified address is located in the kernel unmapped address space.
`kalloc`	Allocates a variable-sized section of kernel virtual memory.
`kfree`	Returns previously allocated memory.
`kget`	Performs nonblocking allocation of variable-size kernel memory.
`KSEG_TO_PHYS`	Converts a kernel unmapped virtual address to a physical address.
`ldbl_ctlr_configure`	Configures the specified controller.
`ldbl_ctlr_unconfigure`	Unconfigures the specified controller.
`ldbl_stanza_resolver`	Merges the configuration data.
`major`	Returns the device major number.
`makedev`	Returns a `dev_t`.
`mb`	Performs a memory barrier.
`minor`	Returns the device minor number.
`minphys`	Bounds the data transfer size.
`ntohl`	Converts longword values from network-to-host byte order.
`ntohs`	Converts word values from network-to-host byte order.
`ovbcopy`	Copies a byte string with a specified limit.
`panic`	Causes a system crash.
`physio`	Implements raw I/O.
`PHYS_TO_KSEG`	Converts a physical address to a kernel unmapped virtual address.
`pmap_extract`	Extracts a physical page address.
`pmap_kernel`	Returns the physical map handle for the kernel.
`pmap_set_modify`	Sets the modify bits of the specified physical page.
`printf`	Prints text to the console and the error logger.
`privileged`	Checks for proper privileges.
`psignal`	Sends a signal to a process.
`remque`	Removes an element from the queue.

Table A-2: (continued)

Kernel Interface	Summary Description
`read_io_port`	Reads data from a device register.
`rmalloc`	Allocates size units from the given resource map.
`rmfree`	Frees space previously allocated into the specified resource map.
`rmget`	Allocates size units from the given resource map.
`rminit`	Initializes a resource map.
`round_page`	Rounds the specified address.
`select_dequeue`	Removes the last thread waiting for an event.
`select_dequeue_all`	Removes all kernel threads waiting for an event.
`select_enqueue`	Adds the current kernel thread.
`select_wakeup`	Wakes up a kernel thread.
`sleep`	Puts a calling process to sleep.
`spl`	Sets the processor priority to mask different levels of interrupts.
`strcmp`	Compares two null-terminated character strings.
`strcpy`	Copies a null-terminated character string.
`strlen`	Returns the number of characters in a null-terminated string.
`strncmp`	Compares two strings, using a specified number of characters.
`strncpy`	Copies a null-terminated character string with a specified limit.
`subyte`	Writes a byte into user data address space.
`suibyte`	Writes a byte into user data address space.
`suiword`	Writes a word into user data address space.
`suser`	Checks whether the current user is the superuser.
`suword`	Writes a word into user data address space.
`svatophys`	Converts a system virtual address to a physical address.
`swap_lw_bytes`	Performs a longword byte swap.
`swap_word_bytes`	Performs a short word byte swap.
`swap_words`	Performs a word byte swap.
`timeout`	Initializes a callout queue element.
`trunc_page`	Truncates the specified address.
`uiomove`	Moves data between user and system virtual space.
`untimeout`	Removes the scheduled interface from the callout queues.
`uprintf`	Nonsleeping kernel printf function.
`vm_map_pageable`	Sets pageability of the specified address range.
`vtop`	Converts a kernel pmap/virtual address pair to a physical address.

Table A-2: (continued)

Kernel Interface	Summary Description
`wakeup`	Wakes up all processes sleeping on a specified address.
`wbflush`	Ensures a write to I/O space has completed.
`write_io_port`	Writes data to a device register.

A.3 List of Global Variables that Device Drivers Use

Table A-3 summarizes the global variables used by device drivers. Chapter 2 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the global variables listed in the table.

Table A-3: Summary Descriptions of Global Variables

Global Variable	Summary Description
`cpu`	Provides a unique logical processor type family identifier.
`hz`	Stores the number of clock ticks per second.
`lbolt`	Periodic wakeup mechanism.
`page_size`	Virtual page size.

A.4 List of Data Structures

Table A-4 summarizes the structures that device drivers use. Chapter 3 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the structures listed in the table.

Table A-4: Summary Descriptions of Data Structures

Structure Name	Meaning
`bdevsw`	Defines a device driver's entry points in the block device switch table.
`buf`	Describes arbitrary I/O.
`bus`	Represents an instance of a bus entity.

Table A-4: (continued)

Structure Name	Meaning
`cdevsw`	Defines a device driver's entry points in the character device switch table.
`controller`	Represents an instance of a controller entity.
`device`	Represents an instance of a device entity.
`device_config_t`	Contains device configuration information.
`driver`	Defines driver entry points and other driver-specific information.
`handler_key`	Contains handler-specific information.
`ihandler_t`	Contains information associated with device driver interrupt handling.
`port`	Contains information about a port.
`sg_entry`	Contains bus address/byte count pairs.
`tc_intr_info`	Contains interrupt handler information.
`uio`	Describes I/O, either single vector or multiple vectors.

A.5 List of Device Driver Interfaces

Table A-5 summarizes the interfaces that device drivers use. Chapter 4 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the driver interfaces listed in the table. The table has the following columns:

- Interface

  This column lists the driver interface name.

- Entry point

  This column lists the structure (or file) where the driver writer defines the entry point for the device driver interface.

- Character

  A Yes appears in this column if the interface is applicable to a character device. Otherwise, N/A (not applicable) appears.

- Block

  A Yes appears in this column if the interface is applicable to a block device. Otherwise, N/A (not applicable) appears.

Table A-5: Summary of Block and Character Device Driver Interfaces

Interface	Entry Point	Character	Block
cattach	driver	Yes	Yes
dattach	driver	Yes	Yes
close	bdevsw, cdevsw	Yes	Yes
configure	N/A	Yes	Yes
ctrl_unattach	driver	Yes	Yes
dev_unattach	driver	Yes	Yes
dump	bdevsw	N/A	Yes
intr	system configuration file (static drivers) handler_add and handler_enable (loadable drivers)	Yes	Yes
ioctl	bdevsw, cdevsw	Yes	Yes
mmap	cdevsw	Yes	N/A
open	bdevsw, cdevsw	Yes	Yes
probe	driver	Yes	Yes
psize	bdevsw	N/A	Yes
read	cdevsw	Yes	N/A
reset	cdevsw	Yes	N/A
select	cdevsw	Yes	N/A
slave	driver	Yes	Yes
stop	cdevsw	Yes	N/A
strategy	bdevsw	N/A	Yes
write	cdevsw	Yes	N/A

A.6 List of Bus Configuration Interfaces

Table A-6 summarizes the bus configuration interfaces related to device drivers. Chapter 4 of *Writing Device Drivers, Volume 2: Reference* provides reference (man) page descriptions of the bus configuration interfaces listed in the table.

Table A-6: Summary Description of Bus Configuration Interfaces

Bus Interface	Summary Description
`adp_handler_add`	Registers a driver's interrupt service interface.
`adp_handler_del`	Deregisters a driver's interrupt service interface.
`adp_handler_disable`	Signifies that the driver's interrupt service interface is not callable.
`adp_handler_enable`	Signifies that the driver's interrupt service interface is now callable.
`bus_search`	Searches a `bus` structure for specified values.
`config_resolver`	Configures TURBOchannel buses.
`conn_bus`	Connects bus structures.
`conn_ctlr`	Connects a controller structure to a bus structure.
`conn_device`	Connects a device structure to a controller structure.
`ctlr_configure`	Performs the tasks necessary to make the controller available.
`ctlr_search`	Searches for a `controller` structure connected to a specific bus.
`ctlr_unconfigure`	Performs the tasks necessary to make the controller unavailable.
`get_bus`	Searches the static `bus_list` array for a `bus` structure.
`get_ctlr`	Searches the static `ctlr_list` array for a `controller` structure.
`get_ctlr_num`	Gets a `controller` structure that matches the specified controller name and number.
`get_device`	Searches for the next device in a `controller` structure.
`get_sys_bus`	Returns a pointer to a system `bus` structure.
`perf_init`	Initializes the performance structure for a disk device.
`xxconfl1`	Configures the specified bus.
`xxconfl2`	Configures the specified bus.

Device Driver Example Source Listings B

This appendix contains source listings for the following:

- The /dev/none device driver
- The /dev/cb device driver

B.1 Source Listing for the /dev/none Device Driver

```
/***************************************************
 *                                                 *
 *            Copyright (c) 1993 by                *
 *    Digital Equipment Corporation, Maynard, MA   *
 *             All rights reserved.                *
 *                                                 *
 * This software is furnished under the terms and  *
 * conditions of the TURBOchannel Technology       *
 * license and may be used and copied only in      *
 * accordance with the terms of such license and   *
 * with the inclusion of the above copyright       *
 * notice.  No title to and ownership of the       *
 * software is hereby transferred.                 *
 *                                                 *
 * The information in this software is subject to  *
 * change without notice and should not be         *
 * construed as a commitment by Digital Equipment  *
 * Corporation.                                    *
 *                                                 *
 * Digital assumes no responsibility for the use   *
 * or reliability of its software on equipment     *
 * which is not supplied by Digital.               *
 ***************************************************/


/***************************************************
 * nonereg.h   Header file for none.c 13-Apr-1993  *
 *                                                 *
 *                                                 *
 * Define ioctl macros for the none driver.        *
 ***************************************************/


#define DN_GETCOUNT    _IOR(0,1,int)
#define DN_CLRCOUNT    _IO(0,2)

/***************************************************
```

```
 *                                                 *
 * Device register structure for a none device.    *
 *                                                 *
 ***************************************************/
#define NONE_CSR 0 /* 64-bit read/write CSR/LED register */


/***************************************************
 * none.c  Driver for none device    13-Apr-1993   *
 *                                                 *
 * The /dev/none device driver is an example       *
 * driver that supports a fictitious ''none''      *
 * device.                                         *
 *                                                 *
 ***************************************************/
/***************************************************
 * Tim Burke, Mark Parenti, and Al Wojtas          *
 * Digital Device Driver Project                   *
 *                                                 *
 ***************************************************/
/***************************************************
 *             Include Files Section               *
 ***************************************************/


/***************************************************
 * Common driver header files                      *
 ***************************************************/
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/ioctl.h>
#include <sys/tty.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/map.h>
#include <sys/buf.h>
#include <sys/vm.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/devio.h>
#include <hal/cpuconf.h>
#include <sys/exec.h>
#include <io/common/devdriver.h>
#include <sys/sysconfig.h>
#include <io/dec/tc/tc.h>
#include <machine/cpu.h>
#include <kits/ESA100/nonereg.h>  /* Device register header file */

/***************************************************
```

```
 *      Data structure sizing approach                 *
 *****************************************************
 *                                                     *
 * The following define will be used to allocate      *
 * data structures needed by the /dev/none driver.     *
 * There can be at most 4 instances of the none        *
 * controller on the system.  This is a small          *
 * number of instances of the driver and the data      *
 * structures themselves are not large, so it is       *
 * acceptable to allocate for the maximum              *
 * configuration.                                      *
 *****************************************************/


#define NNONE 4


/*****************************************************
 * Autoconfiguration Support Declarations and          *
 * Definitions Section                                 *
 *****************************************************/

/*****************************************************
 *          Device register address                    *
 *****************************************************/


/*****************************************************
 *        Bits for csr member                          *
 *****************************************************/

#define DN_RESET 0001 /* Device ready for data transfer */
#define DN_ERROR 0002 /* Indicate error */

/*****************************************************
 *        Defines for softc structure                  *
 *****************************************************/

#define DN_OPEN  1 /* Device open bit */
#define DN_CLOSE 0 /* Device close bit */

/*****************************************************
 *     Forward declarations of driver interfaces       *
 *****************************************************/

int noneprobe(), nonecattach(), noneintr();
int noneopen(),  noneclose(),  noneread(), nonewrite();
int noneioctl(), none_ctlr_unattach();

/*****************************************************
 *      controller and driver Structures               *
 *****************************************************/

/*****************************************************
 * Declare an array of pointers to controller          *
 * structures                                          *
 *****************************************************/

struct controller *noneinfo[NNONE];
```

```
/***************************************************
 * Declare and initialize driver structure        *
 ***************************************************/

struct driver nonedriver = {
        noneprobe,              /* probe */
        0,                      /* slave */
        nonecattach,            /* cattach */
        0,                      /* dattach */
        0,                      /* go */
        0,                      /* addr_list */
        0,                      /* dev_name */
        0,                      /* dev_list */
        "none",                 /* ctlr_name */
        noneinfo,               /* ctlr_list */
        0,                      /* xclu */
        0,                      /* addr1_size */
        0,                      /* addr1_atype */
        0,                      /* addr2_size */
        0,                      /* addr2_atype */
        none_ctlr_unattach,     /* ctlr_unattach */
        0                       /* dev_unattach */
};
/***************************************************
 * Declare softc structure                        *
 ***************************************************/

struct none_softc {
     int sc_openf; /* Open flag */
     int sc_count; /* Count of characters written to device */
     int sc_state; /* Device state, not currently used */
} none_softc[NNONE];


/***************************************************
 * Loadable Driver Configuration Support          *
 * Declarations and Definitions Section           *
 ***************************************************/


/***************************************************
 * External function references.  These are needed *
 * for the cdevsw declaration.  The handler_add    *
 * interface is used to register the interrupt     *
 * service interface for the loadable driver.  The *
 * none_id_t array contains "id's" that are used   *
 * to deregister the interrupt handlers.           *
 ***************************************************/


extern int nodev(), nulldev();
extern ihandler_id_t handler_add(), handler_del();
extern ihandler_id_t handler_enable(), handler_disable();
ihandler_id_t none_id_t[NNONE];
#define DN_BUSNAME    "tc" /* This is a TURBOchannel driver */
```

```
/***************************************************
 * The variable none_is_dynamic will be used to    *
 * control any differences in functions performed  *
 * by the static and loadable versions of the      *
 * driver.  In this manner any differences are     *
 * made on a run-time basis and not on a           *
 * compile-time basis.                             *
 ***************************************************/

int none_is_dynamic = 0;

/***************************************************
 * When the driver is loadable it may not have an  *
 * entry in the statically built tc_option         *
 * table (tc_option_data.c).  It is not an error   *
 * if this entry already existed in the table.     *
 * The entry in tc_option_data.c is only used when *
 * the driver is configured statically; the entry  *
 * below is only used when the driver is           *
 * configured dynamically.                         *
 *                                                 *
 * This table contains the bus specific ROM module *
 * name for the driver.  This information forms    *
 * the bus-specific parameter that is passed to    *
 * the ldbl_stanza_resolver interface to look for  *
 * matches in the tc_slot table.                   *
 ***************************************************/

struct tc_option none_option_snippet [] =
/***************************************************/
{
/*  module       driver  intr_b4 itr_aft       adpt    */
/*  name         name    probe   attach  type  config  */
/*  ------       ------  ------- ------- ----  ------  */
{   "NONE    ",  "none",    0,      1,    'C',   0},
{   "",          ""      } /* Null terminator in the */
                           /* table */
};
int num_none = 0;   /* Count on the number of controllers probed */

/***************************************************
 * Loadable Driver Local Structure and Variable    *
 * Definitions Section                             *
 ***************************************************/

int none_config = FALSE;  /* State flags indicating driver configured
dev_t none_devno = NODEV; /* No major number assigned yet. */

/***************************************************
 * Device switch structure for dynamic             *
 * configuration.  The following is a definition   *
 * of the cdevsw entry that will be dynamically    *
 * added for the loadable driver.  For this reason *
 * the loadable driver does not need to have its   *
 * entry points statically configured into conf.c. *
 ***************************************************/


struct cdevsw none_cdevsw_entry = {
        noneopen,         /* d_open */
        noneclose,        /* d_close */
        noneread,         /* d_read */
        nonewrite,        /* d_write */
        noneioctl,        /* d_ioctl */
        nodev,            /* d_stop */
        nodev,            /* d_reset */
        0,                /* d_ttys */
        nodev,            /* d_select */
        0,                /* d_mmap */
        DEV_FUNNEL_NULL   /* d_funnel */
};


/***************************************************
 *            Autoconfiguration Support Section    *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *--------------- noneprobe -----------------------*
 ***************************************************/



/***************************************************
 *              Probe Interface                    *
 ***************************************************
 *                                                 *
 * The noneprobe interface is called from the      *
 * operating system configuration code during boot *
 * time.  The noneprobe interface calls the        *
 * BADADDR interface to determine if the device is *
 * present.  If the device is present, noneprobe   *
 * returns the size of the device register         *
 * structure.  If the device is not present,       *
 * noneprobe returns 0.                            *
 ***************************************************/


noneprobe(addr1, ctlr)
io_handle_t addr1; /* I/O handle passed to the /dev/none */
/* driver's probe interface */
struct controller *ctlr; /* Pointer to controller structure */
{

/***************************************************
*                                                  *
* These data structures will be used to register   *
* the interrupt handler for the loadable driver.   *
*                                                   *
****************************************************/

     ihandler_t handler;
     struct tc_intr_info info;
     int unit = ctlr->ctlr_num;
     register io_handle_t reg = addr1;

/***************************************************
 * If the driver has been statically configured,   *
 * then the interrupt handlers have already been   *
 * registered via the config generated scb_vec.    *
 * Otherwise, the driver has been loaded and it    *
 * is necessary to register the interrupt handlers *
 * here.                                           *
 ***************************************************/

     if (none_is_dynamic) {

/***************************************************
 * Specify the bus that this controller is         *
 * attached to.                                    *
 ***************************************************/

             handler.ih_bus = ctlr->bus_hd;

/***************************************************
 * Set up the fields of the TC specific bus info   *
 * structure and specify the controller number     *
 ***************************************************/

             info.configuration_st = (caddr_t)ctlr;

/***************************************************
 * Specifies the driver type as a controller       *
 ***************************************************/

             info.config_type = TC_CTLR;

/***************************************************
 * Specifies the interrupt service interface (ISI) *
 ***************************************************/

             info.intr = noneintr;

/***************************************************
 * This parameter will be passed to the ISI        *
 ***************************************************/

             info.param = (caddr_t)unit;

/***************************************************
```

```
 * The address of the bus specific info structure. *
 **************************************************/

            handler.ih_bus_info = (char *)&info;

/**************************************************
 * Save off the return id from handler_add.  This  *
 * id will be used later to deregister the         *
 * handler.                                        *
 **************************************************/

            none_id_t[unit] = handler_add(&handler);
            if (none_id_t[unit] == NULL) {
                    return(0); /* Return failure status */
            }
            if (handler_enable(none_id_t[unit]) != 0) {
                    handler_del(none_id_t[unit]);
                    return(0); /* Return failure status */
            }
    }
/**************************************************
 * Determine if the device is present by calling   *
 * the BADADDR interface.  If the device is        *
 * present, return 0.  Otherwise, reset the        *
 * device and assure that a write to I/O           *
 * space completes.                                *
 **************************************************/
    else {
         if (BADADDR( (caddr_t) reg + NONE_CSR, sizeof(long)) !=0)
         {
                    return (0);
          }
             }
    write_io_port(reg + NONE_CSR, 8, 0, DN_RESET); /* Reset the device */
    wbflush();                  /* Ensure a write to I/O space completes */
/**************************************************
 * If the error bit is set, noneprobe returns 0 to *
 * the configuration code.  Otherwise, it          *
 * calls write_io_port and calls wbflush to assure *
 * that a write to I/O space completes.            *
 **************************************************/
    if(read_io_port(reg + NONE_CSR, 8, 0) & DN_ERROR)
    {
        return (0);
    }
    write_io_port(reg + NONE_CSR, 8, 0, 0); /* Write to the CSR/LED register */
    wbflush();    /* Ensure a write to I/O space completes */
/**************************************************
 * Return a nonzero value.  The device is present. *
 **************************************************/
    return (1);
}
```

```
/***************************************************
 *              Attach Interface                   *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *--------------- nonecattach ---------------------*
 *                                                 *
 *                                                 *
 * The nonecattach interface does not currently    *
 * perform any tasks.  It is provided here as a    *
 * stub for future development.                    *
 ***************************************************/


nonecattach(ctlr)
struct controller *ctlr; /* Pointer to controller struct */
{
      /* Attach interface goes here. */
      return;
}



/***************************************************
 *------------ none_ctlr_unattach ---------------- *
 *                                                 *
 * loadable driver specific interface called       *
 * indirectly from the bus code when a driver is   *
 * being unloaded.                                 *
 *                                                 *
 * Returns 0 on success, non-zero (1) on error.    *
 ***************************************************/


int none_ctlr_unattach(bus, ctlr)
    struct bus *bus;          /* Pointer to bus structure */
    struct controller *ctlr; /* Pointer to controller structure */
{
        register int unit = ctlr->ctlr_num;

/***************************************************
 * Validate the unit number                        *
 ***************************************************/


        if ((unit > num_none) || (unit < 0)) {
                return(1); /* Return error status */
        }

/***************************************************
 * This interface should never be called for a     *
 * static driver.  The reason is that the static   *
 * driver does not do a handler_add in the first   *
 * place.                                          *
 ***************************************************/
```

```
        if (none_is_dynamic == 0) {
                return(1); /* Return error status */
        }


/************************************************
 * The deregistration of interrupt handlers      *
 * consists of a call to handler_disable to      *
 * disable any further interrupts.  Then, call   *
 * handler_del to remove the ISI.                *
 ************************************************/


        if (handler_disable(none_id_t[unit]) != 0) {
                return(1); /* Return error status */
        }
        if (handler_del(none_id_t[unit]) != 0) {
                return(1); /* Return error status */
        }
        return(0); /* Return success status */
}


/************************************************
 * Loadable Device Driver Section                *
 ************************************************/
/************************************************
 *------------------ none_configure ----------------*
 ************************************************/
/************************************************
 * The none_configure interface is called to     *
 * configure a loadable driver.  This interface is *
 * also called to configure, unconfigure, and    *
 * query the driver.  These operations are       *
 * differentiated by the "op" parameter.         *
 ************************************************/


none_configure(op,indata,indatalen,outdata,outdatalen)
   sysconfig_op_t op;         /* Configure operation */
   device_config_t *indata;   /* Input data structure */
   size_t indatalen;          /* Size of input data structure */
   device_config_t *outdata;  /* Output data structure */
   size_t outdatalen;         /* Size of output data structure */
{
        dev_t   cdevno;
        int     retval;
        int     i;
        switch (op) {

/**************************************************
 *            Configure (load) the driver.        *
 *                                                *
 **************************************************/

            case SYSCONFIG_CONFIGURE:

/**************************************************
 *          The configure interface could be      *
 *          called for either a static or loadable *
 *          driver.  For this reason it is not    *
 *          possible to conclude that the driver  *
 *          is being dynamically loaded merely    *
 *          because the configure interface has   *
 *          been entered.  To see if the driver   *
 *          is dynamically configured check the   *
 *          flags field.  If this is set, then    *
 *          set a driver global variable to       *
 *          indicate the driver is loaded.        *
 **************************************************/

                if (indata->dc_dsflags & IH_DRV_DYNAMIC) {
                        none_is_dynamic = 1;
                }
                if (none_is_dynamic) {

/**************************************************
 *              Sanity check on the config name.  *
 *              If it is null the resolver and    *
 *              configure code won't know what to *
 *              look for.                         *
 **************************************************/

                 if (strlen(indata->config_name) <= 0) {
                     printf("none_configure, null config name.\n");
                     return(EINVAL);
                        }

/**************************************************
 *             Call the resolver to look for      *
 *             matches to the module's rom name   *
 *             in the tc_slot table.  This will   *
 *             add the controller data structure  *
 *             into the topology tree.            *
 **************************************************/

                 if (ldbl_stanza_resolver(indata->config_name,
                     DN_BUSNAME, &nonedriver,
                     (caddr_t *)none_option_snippet) != 0) {
                     return(EINVAL);
                        }

/**************************************************
```

```
 *             Call the configuration code to     *
 *             cause the driver's probe interface  *
 *             to be called once for each instance *
 *             of the controller found on the      *
 *             system.                             *
 **************************************************/

                       if (ldbl_ctlr_configure(DN_BUSNAME,
                               LDBL_WILDNUM, indata->config_name,
                               &nonedriver, 0)) {
                               return(EINVAL);
                       }

/**************************************************
 *            The above call should have called    *
 *            the driver's probe interface for     *
 *            each instance of the controller.     *
 *            If there were no controllers found   *
 *            then fail the driver configure       *
 *            operation.                           *
 **************************************************/

                       if (num_none == 0) {
                            return(EINVAL);
                       }
               }

/**************************************************
 *           Perform the driver configuration      *
 *           above prior to getting the major      *
 *           number so that user level programs do *
 *           not have access to the driver's       *
 *           entry points in cdevsw prior to the   *
 *           completion of the topology and        *
 *           interrupt configuration.              *
 *                                                 *
 *           Register the driver's cdevsw entry    *
 *           points and obtain the major number    *
 **************************************************/

               cdevno = makedev(indata->dc_cmajnum,
                       (indata->dc_cmajnum == -1)?-1:0);
               cdevno = cdevsw_add(cdevno,&none_cdevsw_entry);
               if (cdevno == NODEV) {

/**************************************************
 *            The call to cdevsw_add could fail if *
 *            the driver is requesting a specific  *
 *            major number and that number is      *
 *            currently in use, or if the cdevsw   *
 *            table is currently full.             *
 **************************************************/

                       return(ENODEV);
               }
```

```
/***************************************************
 *            Stash away the dev_t so that it can  *
 *            be used later to unconfigure the     *
 *            device.  Save off the minor number   *
 *            information.  This will be returned  *
 *            by the query call.                   *
 ***************************************************/

                none_devno = cdevno;

/***************************************************
 *            Set up the "outdata" structure to    *
 *            contain the returned information     *
 *            from driver configuration.  This     *
 *            will be used by cfgmgr to determine  *
 *            what device special files need to be *
 *            created.                             *
 *                                                 *
 *            This member specifies the major      *
 *            number that was assigned to this     *
 *            driver.                              *
 ***************************************************/

                outdata->dc_cmajnum = major(none_devno);

/***************************************************
 *            This member indicates that the       *
 *            beginning minor number will be zero. *
 ***************************************************/

                outdata->dc_begunit = 0;

/***************************************************
 *            Specifies the number of instances of *
 *            the controller that were located.    *
 ***************************************************/

                outdata->dc_numunit = num_none;

/***************************************************
 *           This member specifies the revision of *
 *           kernel interfaces that the driver was *
 *           compiled to.  The member will be      *
 *           examined upon driver loading to       *
 *           ensure compatibility.                 *
 ***************************************************/

                outdata->dc_version = DRIVER_BUILD_LEVEL;

/***************************************************
 *           This flags member is unused.  Return  *
 *           the flags that were passed as input   *
 *           parameters.                           *
 ***************************************************/
```

```
                outdata->dc_dsflags = indata->dc_dsflags;

/************************************************
 *          This is a character driver.  For this *
 *          reason no block major number is      *
 *          assigned.                            *
 ************************************************/

                outdata->dc_bmajnum = NODEV;

/************************************************
 *          The following members are not used by  *
 *          this driver.  Set them to zero to that *
 *          they will have defined values.        *
 ************************************************/


                outdata->dc_errcode = 0;
                outdata->dc_ihflags = 0;
                outdata->dc_ihlevel = 0;

/************************************************
 *          Set this state field to indicate that  *
 *          the driver has successfully           *
 *          configured.                           *
 ************************************************/

                none_config = TRUE;
                break;


/************************************************
 * Unconfigure (unload) the driver.               *
 ************************************************/


              case SYSCONFIG_UNCONFIGURE:

/************************************************
 *              DEBUG STATEMENT                   *
 ************************************************/
#ifdef NONE_DEBUG
printf("none_configure: SYSCONFIG_UNCONFIGURE.\n");
#endif /* NONE_DEBUG */
/************************************************
 *          Fail the unconfiguration if the driver *
 *          is not currently configured.          *
 ************************************************/


                if (none_config != TRUE) {
                        return(EINVAL);
                }

/************************************************
 *          Do not allow the driver to be unloaded *
 *          if it is currently active.  To see if  *
 *          the driver is active look to see if    *
 *          any users have the device open.          *
 ***************************************************/

                for (i = 0; i < num_none; i++) {
                        if (none_softc[i].sc_openf != 0) {
                                return(EBUSY);
                        }
                }

/***************************************************
 *        Call cdevsw_del to remove the driver      *
 *        entry points from the in-memory resident *
 *        cdevsw table.  This is done prior to      *
 *        deleting the loadable configuration       *
 *        and handlers to prevent users from        *
 *        accessing the device in the middle of     *
 *        deconfigure operation.                    *
 ***************************************************/

                retval = cdevsw_del(none_devno);
                if (retval) {
                        return(ESRCH);
                }

/***************************************************
 *       Deregister the driver's configuration      *
 *       data structures from the hardware          *
 *       topology and cause the interrupt handlers *
 *       to be deleted.                             *
 ***************************************************/

                if (none_is_dynamic) {

/***************************************************
 *       The bus number is wildcarded to            *
 *       deregister on all instances of the tc      *
 *       bus.  The controller name and number are  *
 *       wildcarded.  This causes all instances     *
 *       that match the specified driver structure *
 *       to be deregistered.  Through the bus       *
 *       specific code, this interface results      *
 *       in a call to the none_ctlr_unattach        *
 *       interface for each instance of the         *
 *       controller.                                *
 ***************************************************/

                        if (ldbl_ctlr_unconfigure(DN_BUSNAME,
                                LDBL_WILDNUM, &nonedriver,
                                LDBL_WILDNAME, LDBL_WILDNUM) != 0) {

/***************************************************
 *              DEBUG STATEMENT                     *
 ***************************************************/
#ifdef NONE_DEBUG
printf("none_configure:ldbl_ctlr_unconfigure failed.\n");
#endif /* NONE_DEBUG */
```

```
                                return(ESRCH);
                        }
                }
                none_config = FALSE;
                break;


/***************************************************
 *              Driver Query.  Return configuration *
 *              information.  For a query           *
 *              operation, the indata members are   *
 *              not looked at.  Rather, the outdata *
 *              members are filled in similarly to  *
 *              what was done at the end of the     *
 *              configure interface.                *
 ***************************************************/

                case SYSCONFIG_QUERY:

/***************************************************
 *                Fail the query if the driver is   *
 *                not currently configured.         *
 ***************************************************/

                  if (none_config != TRUE) {
                          return(EINVAL);
                  }
                  outdata->dc_cmajnum = major(none_devno);
                  outdata->dc_bmajnum = NODEV;
                  outdata->dc_begunit = 0;
                  outdata->dc_numunit = num_none;
                  outdata->dc_version = DRIVER_BUILD_LEVEL;
                  break;
                default: /* Unknown operation type */
                  return(EINVAL);
        }

/***************************************************
 *      The driver's configure interface has        *
 *      completed successfully.  Return a success   *
 *      status.                                     *
 ***************************************************/

        return(0);
}
```

```
/***************************************************
 *            Open and Close Device Section        *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *                                                 *
 *----------------- noneopen -------------------- *
 *                                                 *
 *                                                 *
 *                                                 *
 * The noneopen interface is called as the result  *
 * of an open system call.  The noneopen interface *
 * checks to ensure that the open is unique,       *
 * marks the device as open, and returns the       *
 * value zero (0) to the open system call to       *
 * indicate success.                               *
 ***************************************************/


noneopen(dev, flag, format)
dev_t dev;  /* Major/minor device number */
int flag;   /* Flags from /usr/sys/h/file.h */
int format; /* Format of special device */
{
/***************************************************
 * Perform the following initializations:          *
 *                                                 *
 *  (1) Initialize unit to the minor device number *
 *  (2) Initialize the pointer to the controller   *
 *      structure associated with this none device *
 *  (3) Initialize the pointer to the none_softc   *
 *      structure associated with this none        *
 *      device                                     *
 ***************************************************/

     register int unit = minor(dev);

     struct controller *ctlr = noneinfo[unit];

     struct none_softc *sc = &none_softc[unit];

/***************************************************
 * If the device does not exist, return no such    *
 * device.                                         *
 ***************************************************/

     if(unit >= NNONE)
          return ENODEV;

/***************************************************
 * Make sure the open is unique                    *
 ***************************************************/

     if (sc->sc_openf == DN_OPEN)
          return (EBUSY);

/***************************************************
 * If device is initialized, set sc_openf and      *
 * return 0 to indicate success.  Otherwise, the   *
 * device does not exist.  Return an error code.   *
 ***************************************************/
     if ((ctlr !=0) && (ctlr->alive & ALV_ALIVE))
     {
          sc->sc_openf = DN_OPEN;
          return(0);
     }
/***************************************************
 * Return an error code to indicate device does    *
 * not exist.                                      *
 ***************************************************/
     else return(ENXIO);
}

/***************************************************
 * Close Interface                                 *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *--------------- noneclose -----------------------*
 *                                                 *
 *                                                 *
 *                                                 *
 * The noneclose interface uses the same arguments *
 * as noneopen; gets the device minor number in    *
 * the same way; and initializes the device and    *
 * none_softc structures identically. The purpose  *
 * of noneclose is to turn off the open flag for   *
 * the specified none device.                      *
 ***************************************************/
noneclose(dev, flag, format)
dev_t dev;  /* Major/minor device number */
int flag;   /* Flags from /usr/sys/h/file.h */
int format; /* Format of special device */
{
/***************************************************
 * Perform the following initializations:          *
 *                                                 *
 *  (1) Initialize unit to the minor device number *
 *  (2) Initialize the pointer to the controller   *
 *      structure associated with this none device *
 *  (3) Initialize the pointer to the none_softc   *
 *      structure associated with this none        *
 *      device                                     *
 *  (4) Initialize the pointer to the device       *
 *      register structure.                        *
 ***************************************************/
     register int unit = minor(dev);
     struct controller *ctlr = noneinfo[unit];
     struct none_softc *sc = &none_softc[unit];
```

```
          register io_handle_t reg =
          (io_handle_t) ctlr->addr;
/***************************************************
 * Turn off the open flag for the specified device *
 ***************************************************/

     sc->sc_openf = DN_CLOSE;
/***************************************************
 * Turn off interrupts                             *
 ***************************************************/

        write_io_port(reg + NONE_CSR, 8, 0, 0);
/***************************************************
 * Assure that write to I/O space completes        *
 ***************************************************/

     wbflush();
/***************************************************
 * Return success                                  *
 ***************************************************/

     return(0);
}


/***************************************************
 *            Read and Write Device Section        *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *--------------- noneread ------------------------*
 *                                                 *
 *                                                 *
 *                                                 *
 * The noneread interface simply returns success   *
 * to the read system call because the /dev/none   *
 * driver always returns EOF on read operations.   *
 ***************************************************/


noneread(dev, uio, flag)
dev_t dev;       /* Major/minor device number */
struct uio *uio; /* Pointer to uio structure  */
int flag; /* Access mode of device */
{
     return (0); /* Return success */
}
```

```
/***************************************************
 *          Write Interface                        *
 *                                                 *
 *                                                 *
 *--------------- nonewrite -----------------------*
 *                                                 *
 *                                                 *
 *                                                 *
 * The nonewrite interface takes the same formal   *
 * parameters as the noneread interface.  The      *
 * nonewrite interface, however, copies data from  *
 * the address space pointed to by the uio         *
 * structure to the device.  Upon a successful     *
 * write, nonewrite returns the value zero (0) to  *
 * the write system call.                          *
 ***************************************************/


nonewrite(dev, uio, flag)
dev_t dev;        /* Major/minor device number */
struct uio *uio;  /* Pointer to uio structure  */
int flag; /* Access mode of device */
{
/***************************************************
 * Perform the following initializations and      *
 * declarations:                                   *
 *                                                 *
 *  (1) Initialize unit to the minor device number *
 *  (2) Initialize the pointer to the controller   *
 *      structure associated with this none device *
 *  (3) Initialize the pointer to the none_softc   *
 *      structure associated with this none device *
 *  (4) Declare a count variable to store thei     *
 *      size of the write request                  *
 *  (5) Declare a pointer to an iovec structure    *
 ***************************************************/
     int unit = minor(dev);
     struct controller *ctlr = noneinfo[unit];
     struct none_softc *sc = &none_softc[unit];
     unsigned int count;
     struct iovec *iov;
/***************************************************
 * While true, get the next I/O vector             *
 ***************************************************/
     while(uio->uio_resid > 0) {
          iov = uio->uio_iov;
          if(iov->iov_len == 0) {
               uio->uio_iov++;
               uio->uio_iovcnt--;
               if(uio->uio_iovcnt < 0)
                    panic("none write");
```

```
                continue;
          }
/***************************************************
 * Figure out how big the write request is         *
 ***************************************************/

     count = iov->iov_len;

/***************************************************
 * Note that the data is consumed                  *
 ***************************************************/

     iov->iov_base += count;
     iov->iov_len -= count;
     uio->uio_offset += count;
     uio->uio_resid -= count;

/***************************************************
 * Count the bytes written                         *
 ***************************************************/

     sc->sc_count +=count;
     }
     return (0);
}


/***************************************************
 *            Interrupt Section                    *
 *                                                 *
 *                                                 *
 *--------------- noneintr ------------------------*
 *                                                 *
 *                                                 *
 *                                                 *
 * The noneintr interface does not currently       *
 * perform any tasks.  It is provided here as a    *
 * stub for future development.  The noneintr      *
 * interface does nothing because there is no      *
 * real physical device to generate an interrupt.  *
 ***************************************************/


noneintr(unit)
int unit; /* Logical unit number for device */

{

/***************************************************
 * Declare and initialize structures               *
 ***************************************************/

     struct controller *ctlr = noneinfo[unit];
     struct none_softc *sc = &none_softc[unit];

/* Code to perform the interrupt */

}
```

```
/***************************************************
 *            ioctl Section                        *
 *                                                 *
 *                                                 *
 *--------------- noneioctl -----------------------*
 *                                                 *
 *                                                 *
 *                                                 *
 * The noneioctl interface obtains and clears the  *
 * count of bytes that was previously written by   *
 * nonewrite.  When a user program issues the      *
 * command to obtain the count, the /dev/none      *
 * driver returns the count through the data       *
 * pointer passed to the noneioctl interface.      *
 * When a user program asks to clear the count,    *
 * the /dev/none driver does so.                   *
 ***************************************************/


noneioctl(dev, cmd, data, flag)
dev_t dev;              /* Major/minor device number */
unsigned int cmd;       /* The ioctl command */
caddr_t data;           /* ioctl command-specified data */
int flag;               /* Access mode of the device */
{
/***************************************************
 * Perform the following initializations and       *
 * declarations:                                   *
 *  (1) Initialize unit to the minor device number *
 *  (2) Declare a pointer to variable that stores  *
 *      the character count.                       *
 *  (3) Initialize the pointer to the none_softc   *
 *      structure associated with this none device *
 ***************************************************/
     int unit = minor(dev);
     int *res;
     struct none_softc *sc = &none_softc[unit];
/***************************************************
 * For GETCOUNT operations, set the res variable   *
 * to point to the kernel memory allocated by the  *
 * ioctl system call.  The ioctl system call       *
 * copies the data to and from user address space. *
 ***************************************************/
     res = (int *) data;
/***************************************************
 * Save the count, if necessary                    *
 ***************************************************/
     if(cmd == DN_GETCOUNT)
          *res = sc->sc_count;
/***************************************************
 * Clear the count, if necessary                   *
 ************************************************/
     if(cmd == DN_CLRCOUNT)
          sc->sc_count = 0;
/************************************************
 * Success                                      *
 ************************************************/
     return (0);
}
```

B.2 Source Listing for the /dev/cb Device Driver

```
/***************************************************
 * cbreg.h   Header file for cb.c 17-Nov-1993      *
 *                                                 *
 ***************************************************/
/***************************************************
 *                                                 *
 *           Copyright (c) 1993 by                 *
 *   Digital Equipment Corporation, Maynard, MA    *
 *            All rights reserved.                 *
 *                                                 *
 * This software is furnished under the terms and  *
 * conditions of the TURBOchannel Technology       *
 * license and may be used and copied only in      *
 * accordance with the terms of such license and   *
 * with the inclusion of the above copyright       *
 * notice.  No title to and ownership of the       *
 * software is hereby transferred.                 *
 *                                                 *
 * The information in this software is subject to  *
 * change without notice and should not be         *
 * construed as a commitment by Digital Equipment  *
 * Corporation.                                    *
 *                                                 *
 * Digital assumes no responsibility for the use   *
 * or reliability of its software on equipment     *
 * which is not supplied by Digital.               *
 ***************************************************/


/***************************************************
 *                                                 *
 * Define an offset of registers from base address *
 * of option; a macro to convert register offset   *
 * to kernel virtual address; and a macro to       *
 * scramble physical address to TC DMA address.    *
 ***************************************************/


#define CB_REL_LOC 0x00040000
#define CB_ADR(n) ((io_handle_t)(n + CB_REL_LOC))
#define CB_SCRAMBLE(x) (((unsigned)x<<3)&~(0x1f))|(((unsigned)x>>29)&0x1f)


/***************************************************
 * TURBOchannel test board CSR Enable and Status   *
 * bits                                            *
 *                                                 *
 ***************************************************/


#define CB_INTERUPT 0x0e00 /* Bits: 8 = 0; 9, 10 & 11 = 1 */
#define CB_CONFLICT 0x0d00 /* Bits: 9 = 0; 8, 10 & 11 = 1 */
#define CB_DMA_RD   0x0b00 /* Bits: 10 = 0; 8, 9 & 11 = 1 */
#define CB_DMA_WR   0x0700 /* Bits: 11 = 0; 8, 9 & 10 = 1 */
#define CB_DMA_DONE 0x0010 /* Use in timeout loop */


/***************************************************
 * Define ioctl macros for the cb driver.          *
 *                                                 *
 ***************************************************/

#define CBPIO _IO('v',0) /* Set Read/Write mode to PIO */
#define CBDMA _IO('v',1) /* Set Read/Write mode to DMA */
#define CBINT _IO('v',2) /* Perform Interrupt test */
#define CBROM _IOWR('v',3,int) /* Return specified word */
#define CBCSR _IOR('v',4,int)  /* Update & return CSR word */
#define CBINC _IO('v',5) /* Start incrementing lights */
#define CBSTP _IO('v',6) /* Stop incrementing lights */

/***************************************************
 * Register offset definitions for a CB device.    *
 * The registers are aligned on longword (32-bit)  *
 * boundaries, even when they are implemented with *
 * less than 32 bits.                              *
 *                                                 *
 ***************************************************/

#define CB_ADDER    0x0 /* 32-bit read/write DMA address register */
#define CB_DATA     0x4 /* 32-bit read/write data register */
#define CB_CSR      0x8 /* 16-bit read/write CSR/LED register */
#define CB_TEST     0xC /* Go bit: Write sets and Read Clears */

/***************************************************
 *                                                 *
 *          Copyright (c) 1993 by                  *
 *   Digital Equipment Corporation, Maynard, MA    *
 *           All rights reserved.                  *
 *                                                 *
 * This software is furnished under the terms and  *
 * conditions of the TURBOchannel Technology       *
 * license and may be used and copied only in      *
 * accordance with the terms of such license and   *
 * with the inclusion of the above copyright       *
 * notice.  No title to and ownership of the       *
 * software is hereby transferred.                 *
 *                                                 *
 * The information in this software is subject to  *
 * change without notice and should not be         *
 * construed as a commitment by Digital Equipment  *
 * Corporation.                                    *
 *                                                 *
 * Digital assumes no responsibility for the use   *
 * or reliability of its software on equipment     *
 * which is not supplied by Digital.               *
 ***************************************************/
/***************************************************
 *                                                 *
 * The /dev/cb device driver operates on a         *
 * TURBOchannel (TC) bus.  The device it controls  *
 * is called a TURBOchannel test board.  The       *
 * TURBOchannel test board is a minimal            *
 * implementation of all TURBOchannel hardware     *
 * functions:                                      *
 *                                                 *
 *    o Programmed I/O (PIO)                       *
 *    o Direct Memory Access (DMA) read            *
 *    o DMA write                                  *
 *    o Input/Output (I/O) read/write conflict     *
 *      testing                                    *
 *                                                 *
 * The software view of the board consists of:     *
 *                                                 *
 *    o An EPROM address space                     *
 *    o A 32-bit ADDRESS register with bits        *
 *      scrambled for direct use as a TC           *
 *      DMA address                                *
 *    o A 32-bit DATA register used for            *
 *      programmed I/O and as the holding          *
 *      register for DMA                           *
 *    o A 16-bit Light Emitting Diode (LED)/       *
 *      Control Status Register (CSR)              *
 *    o A 1-bit TEST register                      *
 *                                                 *
 * All registers MUST be accessed as 32-bit        *
 * longwords, even when they are not implemented   *
 * as 32 bits.  The CSR contains bits to enable    *
 * option DMA read testing, conflict signal        *
 * testing, I/O interrupt testing, and option      *
 * DMA write testing.  It also contains a bit to   *
 * indicate that one or more of the tests are      *
 * enabled, 4 byte mask flag bits, and a DMA       *
 * done bit.                                       *
 ***************************************************/
/***************************************************
 * This example DEC OSF/1 driver provides a     *
 * simple interface to the TURBOchannel test       *
 * board.  It:                                     *
 *                                                 *
 *     (a) Reads from the data register on the     *
 *         test board to words in system memory    *
 *     (b) Writes to the data register on the test *
 *         board from words in system memory       *
 *     (c) Tests the interrupt logic on the        *
 *         test board                              *
 *     (d) Reads one 32-bit word from the test     *
 *         board address (ROM/register) space into *
 *         system memory                           *
 *     (e) Updates, reads, and returns the 32-bit  *
 *         CSR value                               *
 *     (f) Starts and stops clock-driven           *
 *         incrementing of the four spare LEDs on  *
 *         the board.                              *
 *                                                 *
 * ioctl calls are used to:                        *
 *                                                 *
 *     (a) Set the I/O mode to Programmed I/O      *
 *         (the default)                           *
 *     (b) Set the I/O mode to DMA I/O             *
 *     (c) Enable a single interrupt test          *
 *     (d) Read one 32-bit word from the test      *
 *         board address (ROM/register) space      *
 *     (e) Start clock-driven incrementing of the  *
 *         4 spare LEDs on the board or            *
 *     (f) Stop clock-driven incrementing of the 4 *
 *         spare LEDs on the board.                *
 *                                                 *
 * Standard read and write calls are used to       *
 * perform the data register reads and writes.     *
 ***************************************************/

/***************************************************
 * Larry Robinson and Jim Crapuchettes, Digital    *
 * TRIADD Program.  Ported to DEC OSF/1 by Mark *
 * Parenti, Digital.  Made loadable on DEC OSF/1*
 * by Tim Burke, Digital, and Jeff Anuszczyk,      *
 * formerly of Digital.                            *
 ***************************************************/

/***************************************************
 *            Include Files Section                *
 *                                                 *
 ***************************************************/

/***************************************************
 * Define a constant called NCB that is used to    *
 * allocate the data structures needed by the      *
 * /dev/cb driver.  Note that the define uses the  *
 * TC_OPTION_SLOTS constant, which is defined in   *
 * tc.h.  There can be at most three instances of  *
 * the CB controller on the system.  This is a     *
 * small number of instances of the device on the  *
 * system and the data structures themselves are   *
 * not large, so it is acceptable to allocate for  *
 * the maximum configuration.  This is an example  *
 * of the static allocation technique model 2.     *
 ***************************************************/



/***************************************************
 * The following include files assume that the     *
 * current directory is a subdirectory of          *
 * /usr/sys.                                       *

 ***************************************************/
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <hal/cpuconf.h>
#include <sys/vm.h>
#include <sys/buf.h>
#include <sys/errno.h>
#include <sys/conf.h>
#include <sys/file.h>
#include <sys/uio.h>
#include <sys/types.h>

#include <io/common/devdriver.h>
#include <sys/sysconfig.h>
#include <io/dec/tc/tc.h>

#include <kits/ESB100/cbreg.h> /* Device register header file */

#define NCB TC_OPTION_SLOTS
```

```
/***************************************************
 * Autoconfiguration Support Declarations and      *
 * Definitions Section                             *
 ***************************************************/


extern  int hz; /* System clock ticks per second */

/***************************************************
 * Do forward declaration of driver entry points   *
 * and define information structures for driver    *
 * structure definition and initialization below.  *
 ***************************************************/


int cbprobe(), cbattach(), cbintr(), cbopen(), cbclose();
int cbread(), cbwrite(), cbioctl(), cbstart(), cbminphys();
int cbincled(), cb_ctlr_unattach(), cbstrategy();

/***************************************************
 * Declare an array of pointers to controller      *
 * structures                                      *
 ***************************************************/

struct controller *cbinfo[NCB];

/***************************************************
 * Define and initialize the driver structure for  *
 * this driver.  It is used to connect the driver  *
 * entry points and other information to the       *
 * DEC OSF/1 code.  The driver structure is     *
 * used primarily during Autoconfiguration.  Note  *
 * that the "slave" and "go" entry points do not   *
 * exist in this driver and that a number of the   *
 * members of the structure are not used because   *
 * this is a driver that operates on the           *
 * TURBOchannel bus (not on the VMEbus or some     *
 * other bus).                                     *
 ***************************************************/


struct  driver cbdriver = {
        cbprobe,            /* probe */
        0,                  /* slave */
        cbattach,           /* cattach */
        0,                  /* dattach */
        0,                  /* go */
        0,                  /* addr_list */
        0,                  /* dev_name */
        0,                  /* dev_list */
        "cb",               /* ctlr_name */
        cbinfo,             /* ctlr_list */
        0,                  /* xclu */
        0,                  /* addr1_size */
        0,                  /* addr1_atype */
        0,                  /* addr2_size */
        0,                  /* addr2_atype */
        cb_ctlr_unattach,   /* ctlr_unattach */
        0                   /* dev_unattach */
};
```

```
/***************************************************
 * Loadable Driver Configuration Support           *
 * Declarations and Definitions Section            *
 ***************************************************/

/***************************************************
 * External function references.  These are needed *
 * for the cdevsw declaration.  The handler_add    *
 * interface is used to register the interrupt     *
 * service interface for the loadable driver.  The *
 * cb_id_t array contains "id's" that are used to  *
 * deregister the interrupt handlers.              *
 ***************************************************/

extern int nodev(), nulldev();
extern ihandler_id_t handler_add(), handler_del();
extern ihandler_id_t handler_enable(), handler_disable();
ihandler_id_t cb_id_t[NCB];
#define CB_BUSNAME    "tc" /* This is a TURBOchannel driver */

/***************************************************
 * The variable cb_is_dynamic will be used to      *
 * control any differences in functions performed  *
 * by the static and loadable versions of the      *
 * driver.  In this manner any differences are     *
 * made on a run-time basis and not on a compile   *
 * time basis.                                     *
 ***************************************************/

int cb_is_dynamic = 0;

/***************************************************
 * When the driver is loadable it may not have an  *
 * entry in the statically built tc_option         *
 * table (located in tc_option_data.c).  It is not *
 * an error if this entry already existed in the   *
 * table.  The entry in tc_option_data.c is used   *
 * only when the driver is configured statically.  *
 * The entry below is used only when the driver is *
 * configured dynamically.                         *
 *                                                 *
 * This table contains the bus specific ROM module *
 * name for the driver.  This information forms    *
 * the bus-specific parameter that is passed to    *
 * the ldbl_stanza_resolver interface to look for  *
 * matches in the tc_slot table.                   *
 ***************************************************/

struct tc_option cb_option_snippet [] =
{
    /*  module          driver  intr_b4 itr_aft         adpt    */
    /*  name            name    probe   attach  type    config  */
    /*  ------          ------  ------- ------- ----    ------  */
    {   "CB      ",     "cb",     0,       1,    'C',     0},
    {   "",                 ""        } /* Null terminator in the table */
};
int num_cb = 0;    /* Count on the number of controllers probed */


/***************************************************
 * Local Structure and Variable Definitions        *
 * Section                                         *
 ***************************************************/

/***************************************************
 * Declare an array of buffer headers, 1 per       *
 * CB unit                                         *
 ***************************************************/

struct buf cbbuf[NCB];
unsigned tmpbuffer; /* Temporary one-word buffer for cbstart */


/***************************************************
 * Structure declaration for a CB unit. It         *
 * contains status, pointers, and I/O mode for a   *
 * single CB device.                               *
 ***************************************************/


struct cb_unit { /* All items are "for this unit": */
    int  attached;        /* An attach was done */
    int  opened;          /* An open was done */
    int  iomode;          /* Read/write mode (PIO/DMA) */
    int  intrflag;        /* Flag for interrupt test */
    int  ledflag;         /* Flag for LED increment function */
    int  adapter;         /* TC slot number */
    caddr_t cbad;         /* ROM base address */
    io_handle_t cbr;    /* I/O handle for device registers */
    struct buf    *cbbuf; /* Buffer structure address */
} cb_unit[NCB];
#define MAX_XFR 4 /* Maximum transfer chunk in bytes */


/***************************************************
 * Loadable Driver Local Structure and Variable    *
 * Definitions Section                             *
 ***************************************************/


int cb_config = FALSE;  /* State flags indicating driver configured */
dev_t cb_devno = NODEV; /* No major number assigned yet. */


/***************************************************
 * Device switch structure for dynamic             *
 * configuration. The following is a definition of *
 * the cdevsw entry that will be dynamically added *
 * for the loadable driver.  For this reason the   *
 * loadable driver does not need to have its entry *
 * points statically configured into conf.c.       *
 ***************************************************/


struct cdevsw cb_cdevsw_entry = {
        cbopen,                 /* d_open */
        cbclose,                /* d_close */
        cbread,                 /* d_read */
        cbwrite,                /* d_write */
        cbioctl,                /* d_ioctl */
        nodev,                  /* d_stop */
        nodev,                  /* d_reset */
        0,                      /* d_ttys */
        nodev,                  /* d_select */
        nodev,                  /* d_mmap */
        DEV_FUNNEL_NULL         /* d_funnel */
};

/***************************************************
 * WARNING ON USE OF printf FOR DEBUGGING          *
 *                                                 *
 * Only a limited number of characters (system     *
 * release dependent; currently, seems to be 128)  *
 * can be sent to the "console" display during     *
 * each call to any section of a driver.  This is  *
 * because the characters are buffered until the   *
 * driver returns to the kernel, at which time     *
 * they are actually sent to the "console".  If    *
 * more than this number of characters are sent to *
 * the "console", the storage pointer may wrap     *
 * around, discarding all previous characters, or  *
 * it may discard all following characters! (Also  *
 * system release dependent.)  Limit "console"     *
 * output from within the driver if you need to    *
 * see the results in the console window.          *
 * However, 'printf' from within a driver also     *
 * puts the messages into the error log file.      *
 * The text can be viewed with 'uerf'.  See the    *
 * 'uerf' man page for more information.  The      *
 * "-o terse" option makes the messages easier to  *
 * read by removing the time stamp information.    *
 *                                                 *
 * WARNING ON USE OF printf FOR DEBUGGING          *
 ***************************************************/

#define CB_DEBUG /* Define debug constants */
#undef CB_DEBUGx /* Disable xtra debug */


/***************************************************
 * Autoconfiguration Support Section               *
 ***************************************************/
/***************************************************
 *                                                 *
 *                                                 *
 *--------------- cbprobe -------------------------*
 ***************************************************/

cbprobe(vbaddr, ctlr)
caddr_t vbaddr;            /* Virtual base address of slot */
struct controller *ctlr;   /* controller structure for this unit */
{
```

```
/***************************************************
 *                                                 *
 * These data structures will be used to register  *
 * the interrupt handler for the loadable driver.  *
 *                                                 *
 ***************************************************/


        ihandler_t handler;
        struct tc_intr_info info;
        int unit = ctlr->ctlr_num;


/***************************************************
 * Call printf during debug                        *
 ***************************************************/


/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("CBprobe @ %8x, vbaddr = %8x, ctlr = %8x\n",cbprobe,vbaddr,ctlr);
#endif /* CB_DEBUG */


/***************************************************
 * If the driver has been statically configured,   *
 * then the interrupt handlers have already been   *
 * registered via the config generated scb_vec.    *
 * Otherwise, the driver has been loaded and it    *
 * is necessary to register the interrupt handlers *
 * here.                                           *
 ***************************************************/


        if (cb_is_dynamic) {

/***************************************************
 *  DEBUG STATEMENT                                *
 ***************************************************/
#ifdef CB_DEBUG
printf("CBprobe: perform loadable driver config of unit %d\n",unit);
#endif /* CB_DEBUG */



/***************************************************
 * Specify the bus that this controller is         *
 * attached to.                                    *
 ***************************************************/

                handler.ih_bus = ctlr->bus_hd;

/***************************************************
 * Set up the fields of the TC specific bus info   *
 * structure and specify the controller number     *
 ***************************************************/

                info.configuration_st = (caddr_t)ctlr;
```

```
/***************************************************
 * Specifies the driver type as a controller      *
 ***************************************************/

                info.config_type = TC_CTLR;

/***************************************************
 * Specifies the interrupt service interface (ISI) *
 ***************************************************/

                info.intr = cbintr;

/***************************************************
 * This parameter will be passed to the ISI        *
 ***************************************************/

                info.param = (caddr_t)unit;

/***************************************************
 * The address of the bus specific info structure. *
 ***************************************************/

                handler.ih_bus_info = (char *)&info;

/***************************************************
 * Save off the return id from handler_add.  This  *
 * id will be used later to deregister the         *
 * handler.                                        *
 ***************************************************/

                cb_id_t[unit] = handler_add(&handler);
                if (cb_id_t[unit] == NULL) {

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("CBprobe: handler_add failed.\n");
#endif /* CB_DEBUG */

                        return(0); /* Return failure status */
                }
                if (handler_enable(cb_id_t[unit]) != 0) {
                       handler_del(cb_id_t[unit]);

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("CBprobe: handler_enable failed.\n");
#endif /* CB_DEBUG */

                        return(0); /* Return failure status */
                }
        }

        else {
/***********************************************
 *              DEBUG STATEMENT                 *
 ***********************************************/
#ifdef CB_DEBUG
printf("CBprobe: driver not loadable!\n");
#endif /* CB_DEBUG */
        }



/***********************************************
 * Increment the number of instances of this    *
 * controller.                                  *
 ***********************************************/


        num_cb++;

/***********************************************
 *              DEBUG STATEMENT                 *
 ***********************************************/
#ifdef CB_DEBUG
printf("CBprobe: return success.\n");
#endif /* CB_DEBUG */


        return(1); /* Assume ok since TC ROM probe worked */
}



/***********************************************
 *-------------- cbattach -------------------------*
 ***********************************************/


cbattach(ctlr)
struct controller *ctlr; /* controller structure for this unit */
{
     struct cb_unit *cb; /* Pointer to unit data structure */
/***********************************************
 * Set up per-unit data structure for this device  *
 ***********************************************/

     cb = &cb_unit[ctlr->ctlr_num]; /* Point to this device's structure */

     cb->attached = 1; /* Indicate device is attached */

     cb->adapter = ctlr->slot; /* Set the adapter (slot) number */

     cb->cbad = ctlr->addr; /* Set base of device ROM */

     cb->cbr = (io_handle_t)CB_ADR(ctlr->addr); /* Point to device's registers */
     cb->cbbuf = &cbbuf[ctlr->ctlr_num]; /* Point to device's
                                              buffer header */
}
```

```
/***************************************************
 *------------ cb_ctlr_unattach ------------------*
 *                                                 *
 * loadable driver specific interface called       *
 * indirectly from the bus code when a driver is   *
 * being unloaded.                                 *
 *                                                 *
 * Returns 0 on success, non-zero (1) on error.    *
 ***************************************************/


int cb_ctlr_unattach(bus, ctlr)
    struct bus *bus;         /* Pointer to bus structure */
    struct controller *ctlr; /* Pointer to controller structure */
{
        register int unit = ctlr->ctlr_num;

/***************************************************
 * Validate the unit number                        *
 ***************************************************/


        if ((unit > num_cb) || (unit < 0)) {
                return(1); /* Return error status */
        }

/***************************************************
 * This interface should never be called for a     *
 * static driver.  The reason is that the static   *
 * driver does not do a handler_add in the first   *
 * place.                                          *
 ***************************************************/


        if (cb_is_dynamic == 0) {
                return(1); /* Return error status */
        }

/***************************************************
 * The deregistration of interrupt handlers        *
 * consists of a call to handler_disable to        *
 * disable any further interrupts.  Then, call     *
 * handler_del to remove the ISI.                  *
 ***************************************************/


        if (handler_disable(cb_id_t[unit]) != 0) {
                return(1); /* Return error status */
        }
        if (handler_del(cb_id_t[unit]) != 0) {
                return(1); /* Return error status */
        }
        return(0); /* Return success status */
}
```

```
/***************************************************
 * Loadable Device Driver Section                  *
 ***************************************************/

/***************************************************
 *------------------ cb_configure -----------------*
 ***************************************************/

/***************************************************
 * The cb_configure interface is called to         *
 * configure a loadable driver.  This interface    *
 * is also called to configure, unconfigure, and   *
 * query the driver.  These operations are         *
 * differentiated by the "op" parameter.           *
 ***************************************************/


cb_configure(op,indata,indatalen,outdata,outdatalen)
   sysconfig_op_t op;          /* Configure operation */
   device_config_t *indata;    /* Input data structure */
   size_t indatalen;           /* Size of input data structure */
   device_config_t *outdata;   /* Output data structure */
   size_t outdatalen;          /* Size of output data structure */
{
        dev_t   cdevno;
        int     retval;
        int     i;
        struct cb_unit *cb; /* Pointer to unit data structure */
        int cbincled();     /* Forward reference function */


        switch (op) {

/***************************************************
 *           Configure (load) the driver.          *
 *                                                 *
 ***************************************************/


              case SYSCONFIG_CONFIGURE:
/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("cb_configure: SYSCONFIG_CONFIGURE.\n");
#endif /* CB_DEBUG */


/***************************************************
 *         The configure interface could be        *
 *         called for either a static or loadable *
 *         driver.  For this reason it is not      *
 *         possible to conclude that the driver    *
 *         is being dynamically loaded merely      *
 *         because the configure interface has     *
 *         been entered.  To see if the driver     *
 *         is dynamically configured check the     *
 *         flags field.  If this is set, then      *
 *         set a driver global variable to         *
 *         indicate the driver is loaded.          *
 ***************************************************/
```

```
                if (indata->dc_dsflags & IH_DRV_DYNAMIC) {
                        cb_is_dynamic = 1;
                }
                if (cb_is_dynamic) {

/***************************************************
 *              Sanity check on the config name.   *
 *              If it is null the resolver and     *
 *              configure code won't know what to  *
 *              look for.                          *
 ***************************************************/

                        if (strlen(indata->config_name) <= 0) {
                                printf("cb_configure, null config name.\n");
                                return(EINVAL);
                        }

/***************************************************
 *              Call the resolver to look for      *
 *              matches to the module's rom name   *
 *              in the tc_slot table.  This will   *
 *              add the controller data structure  *
 *              into the topology tree.            *
 ***************************************************/

                        if (ldbl_stanza_resolver(indata->config_name,
                                CB_BUSNAME, &cbdriver,
                                (caddr_t *)cb_option_snippet) != 0) {
                                return(EINVAL);
                        }

/***************************************************
 *             Call the configuration code to      *
 *             cause the driver's probe interface  *
 *             to be called once for each instance *
 *             of the controller found on the      *
 *             system.                             *
 ***************************************************/

                        if (ldbl_ctlr_configure(CB_BUSNAME,
                                LDBL_WILDNUM, indata->config_name,
                                &cbdriver, 0)) {
                                return(EINVAL);
                        }

/***************************************************
 *            The above call should have called    *
 *            the driver's probe interface for     *
 *            each instance of the controller.     *
 *            If there were no controllers found   *
 *            then fail the driver configure       *
 *            operation.                           *
 ***************************************************/

                        if (num_cb == 0) {
```

```
/***************************************************
 *              DEBUG STATEMENT                   *
 ***************************************************/
#ifdef CB_DEBUG
printf("cb_configure: no controllers found.\n");
#endif /* CB_DEBUG */

                              return(EINVAL);
                      }
              }


/***************************************************
 *           Perform the driver configuration     *
 *           above prior to getting the major     *
 *           number so that user level programs do *
 *           not have access to the driver's      *
 *           entry points in cdevsw prior to the  *
 *           completion of the topology and       *
 *           interrupt configuration.             *
 *                                                *
 *           Register the driver's cdevsw entry   *
 *           points and obtain the major number   *
 ***************************************************/

                cdevno = makedev(indata->dc_cmajnum,
                        (indata->dc_cmajnum == -1)?-1:0);
                cdevno = cdevsw_add(cdevno,&cb_cdevsw_entry);
                if (cdevno == NODEV) {

/***************************************************
 *            The call to cdevsw_add could fail if *
 *            the driver is requesting a specific  *
 *            major number and that number is      *
 *            currently in use, or if the cdevsw   *
 *            table is currently full.             *
 ***************************************************/

                        return(ENODEV);
                }

/***************************************************
 *            Stash away the dev_t so that it can  *
 *            be used later to unconfigure the     *
 *            device.  Save off the minor number   *
 *            information.  This will be returned  *
 *            by the query call.                   *
 ***************************************************/

                cb_devno = cdevno;

/***************************************************
 *            Set up the "outdata" structure to    *
 *            contain the returned information     *
 *            from driver configuration.  This     *
 *            will be used by cfgmgr to determine  *
 *            what device special files need to be *
 *            created.                          *
 *                                              *
 *            This member specifies the major   *
 *            number that was assigned to this  *
 *            driver.                           *
 ***********************************************/

                outdata->dc_cmajnum = major(cb_devno);

/***********************************************
 *            This member indicates that the    *
 *            beginning minor number will be zero. *
 ***********************************************/

                outdata->dc_begunit = 0;

/***********************************************
 *            Specifies the number of instances of *
 *            the controller that were located.  *
 ***********************************************/

                outdata->dc_numunit = num_cb;

/***********************************************
 *           This member specifies the revision of *
 *           kernel interfaces that the driver was *
 *           compiled to.  The member will be    *
 *           examined upon driver loading to     *
 *           ensure compatibility.               *
 ***********************************************/

                outdata->dc_version = DRIVER_BUILD_LEVEL;

/***********************************************
 *           This flags member is unused.  Return *
 *           the flags that were passed as input  *
 *           parameters.                          *
 ***********************************************/

                outdata->dc_dsflags = indata->dc_dsflags;

/***********************************************
 *           This is a character driver.  For this *
 *           reason no block major number is      *
 *           assigned.                            *
 ***********************************************/

                outdata->dc_bmajnum = NODEV;

/***********************************************
 *          The following members are not used by  *
 *          this driver.  Set them to zero to that *
 *          they will have defined values.         *
 ***********************************************/
```

```
                outdata->dc_errcode = 0;
                outdata->dc_ihflags = 0;
                outdata->dc_ihlevel = 0;


/***************************************************
 *          Set this state field to indicate that  *
 *          the driver has successfully            *
 *          configured.                            *
 ***************************************************/

                cb_config = TRUE;
                break;



/***************************************************
 * Unconfigure (unload) the driver.                *
 ***************************************************/


              case SYSCONFIG_UNCONFIGURE:


/***************************************************
 *               DEBUG STATEMENT                   *
 ***************************************************/
#ifdef CB_DEBUG
printf("cb_configure: SYSCONFIG_UNCONFIGURE.\n");
#endif /* CB_DEBUG */
/***************************************************
 *          Fail the unconfiguration if the driver *
 *          is not currently configured.           *
 ***************************************************/

                if (cb_config != TRUE) {
                        return(EINVAL);
                }

/***************************************************
 *          Do not allow the driver to be unloaded *
 *          if it is currently active.  To see if  *
 *          the driver is active look to see if    *
 *          any users have the device open.        *
 ***************************************************/



/***************************************************
 *         Do not allow the driver to be unloaded  *
 *         if it is currently active.  To see if   *
 *         the driver is active look to see if any *
 *         users have the device open.             *
 ***************************************************/

                for (i = 0; i < num_cb; i++) {
                        if (cb_unit[i].opened != 0) {
                                return(EBUSY);
                        }
                }
```

```
/***************************************************
 *        Turn off the LED increment function.     *
 *        This is needed to ensure that the driver *
 *        is quiescent.  If this was not done, the *
 *        cbincled interface could be called later *
 *        after its timeout interval had expired.  *
 *        This could then try to execute an        *
 *        interface of this driver which had       *
 *        already been unloaded, resulting in a    *
 *        system panic.                            *
 ***************************************************/


                for (i = 0; i < num_cb; i++) {
                        cb = &cb_unit[i];
                        cb->ledflag = 0;
                        untimeout(cbincled, (caddr_t)cb);
                }

/***************************************************
 *        Call cdevsw_del to remove the driver     *
 *        entry points from the in-memory resident *
 *        cdevsw table.  This is done prior to     *
 *        deleting the loadable configuration      *
 *        and handlers to prevent users from       *
 *        accessing the device in the middle of    *
 *        deconfigure operation.                   *
 ***************************************************/


                        retval = cdevsw_del(cb_devno);
                        if (retval) {
                        return(ESRCH);
                        }

/***************************************************
 *       Deregister the driver's configuration     *
 *       data structures from the hardware         *
 *       topology and cause the interrupt handlers *
 *       to be deleted.                            *
 ***************************************************/


                if (cb_is_dynamic) {


/***************************************************
 *       The bus number is wildcarded to           *
 *       deregister on all instances of the tc     *
 *       bus.  The controller name and number is   *
 *       wildcarded.  This causes all instances    *
 *       that match the specified driver structure *
 *       to be deregistered.  Through the bus      *
 *       specific code, this interface call will   *
 *       result in a call to the cb_ctlr_unattach  *
 *       interface for each instance of the        *
 *       controller.                               *
 ***************************************************/


                        if (ldbl_ctlr_unconfigure(CB_BUSNAME,
                                LDBL_WILDNUM, &cbdriver,
                                LDBL_WILDNAME, LDBL_WILDNUM) != 0) {

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("cb_configure:ldbl_ctlr_unconfigure failed.\n");
#endif /* CB_DEBUG */

                                return(ESRCH);
                        }
                }
                cb_config = FALSE;
                break;


/***************************************************
 *             Driver Query.  Return configuration *
 *             information.  For a query           *
 *             operation, the indata members are   *
 *             not looked at.  Rather, the outdata *
 *             members are filled in similarly to  *
 *             what was done at the end of the     *
 *             configure interface.                *
 ***************************************************/

              case SYSCONFIG_QUERY:

/***************************************************
 *               Fail the query if the driver is   *
 *               not currently configured.         *
 ***************************************************/

                if (cb_config != TRUE) {
                        return(EINVAL);
                }
                outdata->dc_cmajnum = major(cb_devno);
                outdata->dc_bmajnum = NODEV;
                outdata->dc_begunit = 0;
                outdata->dc_numunit = num_cb;
                outdata->dc_version = DRIVER_BUILD_LEVEL;
                break;
              default: /* Unknown operation type */
                return(EINVAL);
        }

/***************************************************
 *       The driver's configure interface has      *
 *       completed successfully.  Return a success *
 *       status.                                   *
 ***************************************************/

        return(0);
}
```

```
/***************************************************
 * Open and Close Device Section                   *
 ***************************************************/

/***************************************************
 *---------------- cbopen ------------------------*
 ***************************************************/


cbopen(dev, flag, format)
dev_t dev;   /* Major/minor device number */
int flag;    /* Flags from /usr/sys/h/file.h */
int format;  /* Format of special device */
{
/***************************************************
 *       Get device (unit) number                  *
 ***************************************************/
        int unit = minor(dev);

/***************************************************
 *      Error if unit number too big or if unit    *
 *      not attached                               *
 ***************************************************/
        if ((unit > NCB) || !cb_unit[unit].attached)
                return(ENXIO);
        cb_unit[unit].opened = 1; /* All ok, indicate device opened */
        return(0);                /* Return success! */
}



/***************************************************
 *---------------- cbclose -----------------------*
 ***************************************************/


cbclose(dev, flag, format)
dev_t dev;  /* Major/minor device number */
int flag;   /* Flags from /usr/sys/h/file.h */
int format; /* Format of special device */
{
        int unit = minor(dev);    /* Get device (unit) number */
        cb_unit[unit].opened = 0; /* Indicate device closed */
        return(0);                /* Return success! */
}

/***************************************************
 * Read and Write Device Section                   *
 ***************************************************/
/***************************************************
 *---------------- cbread ------------------------*
 ***************************************************/


cbread(dev, uio, flag)
dev_t dev;       /* Major/minor device numbers */
struct uio *uio; /* I/O descriptor structure */
int flag; /* Access mode of device */
{
        unsigned tmp;
        int cnt, err;
        int unit = minor(dev); /* Get device (unit) number */
        struct cb_unit *cb;    /* Pointer to unit data structure */
```

```
/***************************************************
 * To do the read, the device index (unit number)  *
 * is used to select the TC test board to be       *
 * accessed and the mode setting within the        *
 * controller structure for that unit is tested    *
 * to determine whether to do a programmed read or *
 * a DMA read.  For a programmed read, the         *
 * contents of the data register on the test board *
 * are read into a 32-bit local variable and then  *
 * the contents of that variable are moved into    *
 * the buffer in the user's virtual address space  *
 * with the uiomove interface.                     *
 *                                                 *
 * For a DMA read, the system's physio interface   *
 * and the driver's strategy and minphys           *
 * interfaces are used to transfer the contents of *
 * the data register on the test board into the    *
 * buffer in the user's virtual address space.     *
 *                                                 *
 * Note that since only a single word of 4         *
 * (the constant MAX_XFR) bytes can be transferred *
 * at a time, both modes of reading include code   *
 * to limit the read to chunks with a maximum of   *
 * MAX_XFR bytes each and that reading more than   *
 * MAX_XFR bytes will propagate the contents of    *
 * the data register throughout the words of the   *
 * user's buffer.                                  *
 ***************************************************/

       err = 0;                   /* Initialize for no error (yet) */
       cb = &cb_unit[unit];       /* Set pointer to unit's structure */
       if(cb->iomode == CBPIO) {  /* Programmed I/O read code */

/***************************************************
 * Transfer bytes from the test board data         *
 * register to the user's buffer until all         *
 * requested bytes are moved or an error occurs.   *
 * This must be done as a loop because the source  *
 * (the board data register) can supply only       *
 * MAX_XFR bytes at a time.  The loop may not be   *
 * required for other devices.                     *
 ***************************************************/

               while((cnt = uio->uio_resid) && (err == 0)) {

/***************************************************
 * Force count for THIS "section" to be less than  *
 * or equal to MAX_XFR bytes (the size of the data *
 * buffer on the test board).  This causes a read  *
 * of more than MAX_XFR bytes to be chopped up     *
 * into a number MAX_XFR-byte transfers with a     *
 * final transfer of MAX_XFR bytes or less.        *
 ***************************************************/

                       if(cnt > MAX_XFR)cnt = MAX_XFR;
                       tmp = read_io_port(cb->cbr | CB_DATA,
                                              4,
                                              0);/* Read data */
```

```
                                                      /* register */

/************************************************
 * Move bytes read from the data register to the   *
 * user's buffer. Note that:                       *
 *                                                 *
 *     (a) The maximum number of bytes moved is    *
 *         MAX_XFR for each call due to the code   *
 *         above.                                  *
 *     (b) The uio structure is updated as each    *
 *         move is done.                           *
 *                                                 *
 * Thus, uio->uio_resid will be updated for the    *
 * "while" statement above.                        *
 ************************************************/

                        err = uiomove(&tmp,cnt,uio);
                        }
                return(err);
                }
        else if(cb->iomode == CBDMA) /* DMA I/O read code */

/************************************************
 * Transfer bytes from the test board data         *
 * register to the user's buffer until all         *
 * requested bytes are moved or an error occurs.   *
 * The driver's strategy and minphys interfaces    *
 * account for the fact that the source (the board *
 * data register) can supply only 4 bytes at a     *
 * time and the physio interface loops as required *
 * to transfer all requested bytes.                *
 ************************************************/

          return(physio(cbstrategy,cb->cbbuf,dev,B_READ,cbminphys,uio));
}


/************************************************
 *---------------- cbwrite -----------------------*
 ************************************************/


cbwrite(dev, uio, flag)
dev_t dev;         /* Major/minor device numbers */
struct uio *uio;   /* I/O descriptor structure */
int flag; /* Access mode of device */
{
        unsigned tmp;
        int cnt, err;
        int unit = minor(dev); /* Get device (unit) number */
        struct cb_unit *cb;    /* Pointer to unit data structure */

/************************************************
 * To do the write, the device index (unit number) *
 * is used  to select the TC test board to be      *
 * accessed and the mode setting within the        *
 * controller structure for that unit is tested to *
 * determine whether to do a programmed write or a *
 * DMA write.                                       *
 *                                                  *
 * For a programmed write, the contents of one     *
 * word from the buffer in the user's virtual      *
 * address space are moved to a 32-bit local       *
 * variable with the uiomove interface and the     *
 * contents of that variable are moved to the data *
 * register on the test board.                      *
 *                                                  *
 * For a DMA write, the  system's physio interface *
 * and the driver's strategy and minphys            *
 * interfaces are used to transfer the contents of *
 * the buffer in the user's virtual address space  *
 * to the data register on the test board.         *
 *                                                  *
 * Note that since only a single word of 4         *
 * (MAX_XFR) bytes can be transferred at a time,   *
 * both modes of reading include code to limit the *
 * write to chunks with a maximum of MAX_XFR       *
 * bytes.  Note that writing more than MAX_XFR     *
 * bytes has limited usefulness since all the      *
 * words of the user's buffer will be written into *
 * the single data register on the test board.     *
 ***************************************************/

       err = 0;                   /* Initialize for no error (yet) */
       cb = &cb_unit[unit];       /* Set pointer to unit's structure */
       if(cb->iomode == CBPIO) { /* Programmed I/O write code */

/***************************************************
 * Transfer bytes from the user's buffer to the    *
 * test board data register until all requested    *
 * bytes are moved or an error occurs.  This must  *
 * be done as a loop because the destination       *
 * (the board data register) can accept only 4     *
 * bytes at a time.  The loop may not be required  *
 * for other devices.                               *
 ***************************************************/

               while((cnt = uio->uio_resid) && (err == 0)) {
                       if(cnt > MAX_XFR)cnt = MAX_XFR; /* Copy data register */

/***************************************************
 * Move bytes to write from the user's buffer to   *
 * the local variable. Note that:                   *
 *                                                  *
 *   (a) The maximum number of bytes moved is      *
 *       MAX_XFR for each call due to the above    *
 *       code.                                      *
 *   (b) The uio structure is updated as each move *
 *       is done.  Thus, uio->uio_resid will be    *
 *       updated for the above "while" statement.  *
 ***************************************************/

                       err = uiomove(&tmp,cnt,uio);
                       write_io_port(cb->cbr | CB_DATA,
                                                    4,
                                                    0,
                                                    tmp); /* Write data
                                                          to register */
                        }
                return(err);
                }
        else if(cb->iomode == CBDMA) /* DMA I/O write code */

/***************************************************
 * Transfer bytes from the user's buffer to the    *
 * test board data register until all requested    *
 * bytes are moved or an error occurs.  The        *
 * driver's strategy and minphys interfaces        *
 * account for the fact that the destination       *
 * (the board data register) can take only MAX_XFR *
 * bytes at a time and the physio interface loops  *
 * as required to transfer all requested bytes.    *
 ***************************************************/

                return(physio(cbstrategy,cb->cbbuf,dev,B_WRITE,cbminphys,uio));
}


/***************************************************
 *                  Strategy  Section              *
 ***************************************************/
/***************************************************
 *---------------- cbminphys ----------------------*
 ***************************************************/


cbminphys(bp)
register struct buf *bp; /* Pointer to buf structure */
{
        if (bp->b_bcount > MAX_XFR)
                bp->b_bcount = MAX_XFR; /* Maximum transfer
                                        is 4 bytes */
        return;
}


/***************************************************
 *---------------- cbstrategy ---------------------*
 ***************************************************/


cbstrategy(bp)
register struct buf *bp; /* Pointer to buf structure */
{
        register int unit = minor(bp->b_dev); /* Get device
                                              (unit) number */
        register struct controller *ctlr; /* Pointer to
                                          controller struct */
        struct cb_unit *cb;    /* Pointer to unit data structure */
        caddr_t buff_addr;     /* User buffer's virtual address */
        caddr_t virt_addr;     /* User buffer's virtual address */
        unsigned phys_addr;    /* User buffer's physical address */
        int cmd;               /* Current command for test board */
        int err;                /* Error status from uiomove */
        int status;             /* CSR contents for status checking */
        unsigned lowbits;       /* Low 2 virtual address bits */
        unsigned tmp;           /* Temporary holding variable */
        int s;                  /* Temporary holding variable */


#ifdef CB_DEBUG
        char *vtype;    /* String pointer for debug */
#endif /* CB_DEBUG */


        ctlr = cbinfo[unit]; /* Set pointer to unit's structure */



/***************************************************
 * The buffer is accessible, initialize buffer     *
 * structure for transfer.                         *
 ***************************************************/


        bp->b_resid = bp->b_bcount; /* Initialize bytes not xferred */
        bp->av_forw = 0;            /* Clear buffer queue forward link */
        cb = &cb_unit[unit];        /* Set pointer to unit's structure */
        virt_addr = bp->b_un.b_addr; /* Get buffer's virtual address */
        buff_addr = virt_addr;       /* and copy it for internal use */

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("\n"); /* Line between */
              /* cbstrategy calls */
#endif /* CB_DEBUG */

/***************************************************
 *                   NOTE                          *
 ***************************************************
 * TURBOchannel DMA can ONLY be done with FULL     *
 * WORDS and MUST be aligned on WORD boundaries!   *
 * Since the user's buffer can be aligned on any   *
 * byte boundary, the driver code MUST check for   *
 * and handle the cases where the buffer is NOT    *
 * word aligned (unless, of course, the            *
 * TURBOchannel interface hardware includes        *
 * special hardware to handle non-word-aligned     *
 * transfers.  The test board does NOT have any    *
 * such hardware).  If the user's buffer is NOT    *
 * word-aligned, the driver can:                   *
 *                                                 *
 *   (a) Exit with an error or                     *
 *   (b) Take some action to word-align the        *
 *       transfer.                                 *
 *                                                 *
 * Since virtual to physical mapping is done on a  *
 * page basis, the low 2 bits of the virtual       *
 * address of the user's buffer are also the low 2 *
 * bits of the physical address of the user's      *
 * buffer and the buffer alignment can be          *
 * determined by examining the low 2 bits of the   *
 * virtual buffer address.  If these 2 bits are    *
 * nonzero, the buffer is not word-aligned and the *
 * driver must take the desired action.            *
 ***************************************************/



/***************************************************
 * Use the low-order 2 bits of the buffer virtual  *
 * address as the word-aligned indicator for this  *
 * transfer.  If they are non-zero, the user's     *
 * buffer is not word-aligned and an internal      *
 * buffer must be used, so replace the current     *
 * user buffer virtual address (it is updated by   *
 * physio as each word is transferred) with the    *
 * internal buffer virtual address.  Since DMA to  *
 * the board can only be done a word at a time,    *
 * the internal buffer only needs to be a single   *
 * word.                                           *
 ***************************************************/

        if ((lowbits = (unsigned)virt_addr & 3) != 0) { /* Test low
                                                           2 bits */
                virt_addr = (caddr_t)(&tmpbuffer);  /* Use internal
                                                       buffer */

/***************************************************
 *               DEBUG STATEMENT                   *
 ***************************************************/
#ifdef CB_DEBUG
printf("Bd %8x (%d)\n",buff_addr,bp->b_resid);
#endif /* CB_DEBUG */



/***************************************************
 * If the tranfer type is a "write"                *
 * (program => device), then clear the local       *
 * one-word temporary buffer (in case less         *
 * than 4 bytes), move the user's data bytes to    *
 * the local temporary buffer and return error     *
 * status if an error occurs.  The DMA "write"     *
 * will be done from the temporary buffer.         *
 *                                                 *
 *                    NOTE                         *
 ***************************************************
 * Don't use  B_WRITE to test for a write.  It is  *
 * defined as 0 (zero).  You MUST use the          *
 * complement of test for B_READ!                  *
 ***************************************************/

                if ( !(bp->b_flags&B_READ) ) { /* Move now for "write" */
                        tmpbuffer = 0 ;        /* Clear the whole word */

/***************************************************
 *               DEBUG STATEMENT                   *
 **************************************************/
#ifdef CB_DEBUG
printf("Ci\n");
#endif /* CB_DEBUG */

                        if (err = copyin(buff_addr,virt_addr,bp->b_resid)) {
                                bp->b_error = err;       /* See cbwrite */
                                bp->b_flags |= B_ERROR; /* error code */
                                iodone(bp);             /* Signal I/O done */
                                return;                 /* Return error. */
                        }
                }
        }


/**************************************************
 * Convert the buffer virtual address to a        *
 * physical address for DMA by calling the vtop   *
 * interface.                                     *
 **************************************************/

    phys_addr = vtop(bp->b_proc, virt_addr);


/**************************************************
 * Convert the 32-bit physical address (actually  *
 * the low 32 bits of the 34-bit physical address) *
 * from the linear form to the condensed form     *
 * used by DMA to pack 34 address bits onto 32    *
 * board lines.                                   *
 *                                                *
 **************************************************/
/**************************************************
 *              WARNING NOTE                      *
 **************************************************
 *                                                *
 * TURBOchannel DMA can ONLY be done with FULL    *
 * WORDS and MUST be aligned on WORD boundaries!  *
 * The CB_SCRAMBLE macro DISCARDS the low-order   *
 * 2 bits of the physical address while scrambling *
 * the rest of the address!  Therefore, anything  *
 * that is going to be done to resolve this issue *
 * must be done BEFORE CB_SCRAMBLE is used.       *
 **************************************************/

        tmp = CB_SCRAMBLE(phys_addr);
        write_io_port(cb->cbr | CB_ADDER,
                      4,
                      0,
                      tmp);

/**************************************************
 *             DEBUG STATEMENT                    *
 **************************************************/
#ifdef CB_DEBUG
printf("%s %8x= %4s\n",vtype,virt_addr,virt_addr);
printf("ph %8x sc %8x Pr %8x\n",phys_addr,tmp,bp->b_proc);
```

```
#endif /* CB_DEBUG */

        if(bp->b_flags&B_READ)      /* Set up the DMA enable bits: */
                cmd = CB_DMA_WR;    /* Read = "Write to memory" */
        else
                cmd = CB_DMA_RD;    /* Write = "Read from memory" */
        s = splbio();               /* Raise priority */


/***************************************************
 * Although not required in this driver since it   *
 * is only called from the following line, the     *
 * "start I/O" interface is called as a function   *
 * to separate its functionality from the strategy *
 * interface.  This is the typical form it will    *
 * have in other drivers.                          *
 ***************************************************/

        err = cbstart(cmd,cb);      /* Start I/O operation */
        splx(s);                    /* Restore priority */

/***************************************************
 * If the cbstart "timed out" ("err" count not     *
 * positive), return error.  If the loop did not   *
 * "time out", set the bytes remaining to zero to  *
 * return with success.                            *
 ***************************************************/

        if(err <= 0) { /* Check return value from cbstart */

/***************************************************
 *             DEBUG STATEMENT                     *
 ***************************************************/
#ifdef CB_DEBUG
printf("err %2d CSR %4x\n",err,
       (read_io_port(cb->cbr | CB_CSR, 4, 0))&0xffff);
#endif /* CB_DEBUG */

                bp->b_error = EIO;      /* Set "I/O error on device", */
                bp->b_flags |= B_ERROR; /* return access error & error flag */
                iodone(bp);             /* Signal I/O done */
                return;                 /* Return with error. */
                }
        else {

/***************************************************
 *             DEBUG STATEMENT                     *
 ***************************************************/
#ifdef CB_DEBUG
                tmp = read_io_port(cb->cbr | CB_DATA, /* Get data */
                                   4,          /* register */
                                   0);         /* to display */
                status = read_io_port(cb->cbr |  CB_CSR, /* Get */
                                      4,          /* status */
                                      0);         /* from */
```

```
                                                                 /* CSR */
                printf("%2d CSR %4x d %8x= %4s\n",err,
                        status&0xffff,tmp,&tmp);
#endif /* CB_DEBUG */


/***************************************************
 * Did not time out: DMA transfer worked. Test the *
 * low-order 2 bits of the buffer virtual address  *
 * and the transfer mode gain.  If the low 2 bits  *
 * are nonzero, then the user's buffer was not     *
 * word-aligned and the internal buffer was used.  *
 * If the tranfer type is a "read"                 *
 * (device => program), then move the user's data  *
 * from the local one-word temporary buffer and    *
 * return error status if an error occurs.  The    *
 * DMA "read" has been done into the temporary     *
 * buffer.                                         *
 ***************************************************/

                if ( (lowbits)!=0 && bp->b_flags&B_READ) { /* Move if
                                                              "read" */

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
                        printf("Co\n");
#endif /* CB_DEBUG */

                        if (err = copyout(virt_addr,buff_addr,bp->b_resid)) {
                                bp->b_error = err;      /* See cbread */
                               bp->b_flags |= B_ERROR;  /* error code */
                        }
                }
                bp->b_resid = 0;   /* DMA complete, clear remainder */
        }
        iodone(bp);                  /* Indicate done on this buffer */
        return;
}


/***************************************************
 *                   Start  Section                *
 ***************************************************/
/***************************************************
 *---------------- cbstart ------------------------*
 *                                                 *
 * NOTE on CSR usage: cbstart, cbioctl, and        *
 * cbincled are the only interfaces that load the  *
 * CSR register of the board.  Since cbincled      *
 * increments the LEDs in the high 4 bits of the   *
 * 16-bit CSR register, cbstart and cbioctl always *
 * load the 4 bits into whatever value they will   *
 * be storing into the CSR before they do the      *
 * actual store.  Note also that cbstart is called *
 * with system interrupts disabled, so cbincled    *
 * should not be called while cbstart is          *
 * incrementing.                                  *
 **************************************************/

int cbstart(cmd,cb)
int cmd;              /* Current command for test board */
struct cb_unit *cb;   /* Pointer to unit data structure */
{
        int timecnt; /* Timeout loop count */
        int status;  /* CSR contents for status checking */

/***************************************************
 *              DEBUG STATEMENT                   *
 **************************************************/
#ifdef CB_DEBUGx
        printf("\n"); /* Blank line */
                      /* between cbstart */
                      /* calls */
#endif CB_DEBUGx

        cmd = (read_io_port(cb->cbr | CB_CSR,
                            4,
                            0)&0xf000)|(cmd&0xfff); /* High 4 LED bits */
                                                    /* into cmd */
        status = read_io_port(cb->cbr | CB_TEST,
                              4,
                              0); /* Read "test" reg to */
                                  /* clear "go" bit */

        write_io_port(cb->cbr | CB_CSR,
                      4,
                      0,
                      cmd); /* Load CSR with enable bit(s) */
        wbflush();          /* Synchronize with CSR write */

/***************************************************
 *              DEBUG STATEMENT                   *
 **************************************************/
#ifdef CB_DEBUGx
        printf("Chk CSR %4x\n",(read_io_port(cb->cbr | CB_CSR,
                                                        4,
                                                        0))&0xffff);
#endif CB_DEBUGx

        write_io_port(cb->cbr | CB_TEST,
                      4,
                      0,
                      0); /* Write "test" reg */
                          /* to set "go" bit */
        wbflush();                  /* Synchronize with test
                                    reg write */
        timecnt = 10;               /* Initialize timeout
                                    loop counter */
        status = read_io_port(cb->cbr | CB_CSR,
                              4,
                              0); /* Get status from CSR */
/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
        printf("%2d CSR %4x\n",timecnt,
               (read_io_port(cb->cbr | CB_CSR, 4, 0))&0xffff);
#endif /* CB_DEBUG */



/***************************************************
 * Wait for DMA done bit set or timeout loop       *
 * counter to expire.  This driver has a very      *
 * short timeout period because the board          *
 * should respond within a few machine cycles if   *
 * it is not broken.  Thus, the simple timeout     *
 * loop below takes less time than calling a       *
 * system interface.  In most drivers, where       *
 * timeout periods are greater than a few cycles,  *
 * timeout is done using the sleep, wakeup,        *
 * timeout, and untimeout kernel interfaces.  See  *
 * the CBINC (increment LED) code in cbioctl and   *
 * cbincled below for an example of using timeout  *
 * for repetitive timing.                          *
 ***************************************************/

        while((!(status & CB_DMA_DONE)) && timecnt > 0) {
                write_io_port(cb->cbr | CB_CSR,
                              4,
                              0,
                              cmd); /* Write to */
                                    /* update status */
                wbflush();              /* Synchronize with
                                        CSR write */
                status = read_io_port(cb->cbr | CB_CSR,
                                      4,
                                      0); /* Get status from CSR again */
                timecnt --;             /* Decrement counter */
                }

/***************************************************
 * Return "timeout" count as function result.      *
 ***************************************************/


        return(timecnt);
}

/***************************************************
 *           ioctl Section                         *
 ***************************************************/
/***************************************************
 *---------------- cbioctl ------------------------*
 ***************************************************/


/***************************************************
 * See NOTE on CSR usage at beginning of cbstart.  *
 ***************************************************/
```

```
#define CBIncSec  1  /* Number of seconds between
                     increments of lights */

cbioctl(dev, cmd, data, flag)
dev_t dev;           /* Major/minor device number */
unsigned int cmd;    /* The ioctl command */
int *data;           /* ioctl command-specified data */
int flag;            /* Access mode of the device */
{
        int tmp;                /* A destination word for throw-aways */
        int *addr;              /* Pointer for word access to board */
        int timecnt;            /* Timeout loop count */
        int unit = minor(dev); /* Get device (unit) number */
        struct cb_unit *cb;     /* Pointer to unit data structure */
        int cbincled();         /* Forward reference interface */

        cb = &cb_unit[unit];    /* Set pointer to unit's structure */


        switch(cmd&0xFF) {       /* Determine operation to do: */
               case CBINC:         /* Start incrementing lights */

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("\nCBioctl: CBINC ledflag = %d\n",cb->ledflag);
#endif /* CB_DEBUG */

                        if(cb->ledflag == 0) {  /* If not started, */
                             cb->ledflag++;  /* Set flag & start timer */
                             timeout(cbincled, (caddr_t)cb, CBIncSec*hz);
                        }
                        break;


                       case CBPIO:                      /* Set mode: programmed I/O */
                               cb->iomode = CBPIO;      /* Just set I/O mode for unit */
                               break;

                       case CBDMA:                      /* Set mode: DMA I/O */
                               cb->iomode = CBDMA;      /* Just set I/O mode for unit */
                               break;


                    case CBINT:                      /* Do interrupt test */
                            timecnt = 10;            /* Initialize timeout counter */
                            cb->intrflag = 0;        /* Clear interrupt flag */

                            tmp = read_io_port(cb->cbr | CB_TEST,
                                               4,
                                               0); /* Clear "go" bit */

                            tmp = CB_INTERUPT|(read_io_port(cb->cbr | CB_CSR,
                                               4,
                                               0)&0xf000); /* New value */
                            write_io_port(cb->cbr | CB_CSR,
                                          4,
                                          0,
                                          tmp); /* Load enables & LEDs */
                            wbflush();               /* Synch. with CSR write */

                            write_io_port(cb->cbr | CB_TEST,
                                          4,
                                          0,
                                1); /* Set the "go" bit */
                  wbflush();            /* Synch. with test write */
/***************************************************
 *                     Wait for interrupt flag    *
 *                     to set or timeout loop     *
 *                     counter to expire.         *
 ***************************************************/
                  while ((cb->intrflag == 0) && (timecnt > 0)) {
                          write_io_port(cb->cbr | CB_CSR,
                                        4,
                                        0,
                                        tmp); /* Write to update status */
                          wbflush();      /* Synch. with CSR write */
                          timecnt --;     /* Decrement counter */
                  }
                  tmp = read_io_port(cb->cbr | CB_TEST,
                                     4,
                                     0); /* Be sure "go" bit is clear */


/***************************************************
 *               DEBUG STATEMENT                  *
 ***************************************************/
#ifdef CB_DEBUG
                     printf("\nCBioctl: CBINT timecnt = %d\n",timecnt);
#endif /* CB_DEBUG */


                  return(timecnt == 0);   /* Success if non-zero count */


          case CBROM&0xFF:              /* Return a ROM word */
                 tmp = *data;           /* Get specified byte offset */
                 if(tmp < 0 || tmp >= 32768*4+4*4) /* 32k wrds + 4 regs */
                         return(-tmp);  /* Offset is out of range */
                 tmp <<= 1; /* Double address offset */
                 addr = (int *)&(cb->cbad[tmp]); /* Get byte address */
                 *data = *addr;         /* Return word from board */
                 break;
          case CBCSR&0xFF:                        /* Update and return CSR */
                 write_io_port(cb->cbr | CB_CSR,
                               4,
                               0,
                               read_io_port(cb->cbr | CB_CSR,
                                            4,
                                            0)); /* Read/write to update */
                  wbflush();              /* Synch. with CSR write */
                  *data = read_io_port(cb->cbr | CB_CSR,
                                       4,
                                       0); /* Return CSR from board */
                  break;
          case CBSTP:                     /* Stop incrementing lights */

/***************************************************
 *               DEBUG STATEMENT                  *
 ***************************************************/
#ifdef CB_DEBUG
                     printf("\nCBioctl: CBSTP called\n");
#endif /* CB_DEBUG */
```

```
                    cb->ledflag = 0;        /* Stop on next timeout */
                    break;

             }
     return(0);
}

/***********************************************
 *                  Increment LED  Section        *
 ***********************************************/
/***********************************************
 *                                                *
 *---------------- cbincled ----------------------*
 ***********************************************/



/***********************************************
 * This interface is called by the system         *
 * softclock interface CBIncSec seconds after the *
 * last call to the timeout interface.  If the    *
 * increment flag is still set, increment the     *
 * pattern in the high 4 LEDs of the LED/CSR      *
 * register and restart the timeout to recall     *
 * later.                                         *
 *                                                *
 *              NOTE                              *
 ***********************************************
 * Because the LEDs are on when a bit is 0, use a *
 * subtract to do the increment.                  *
 *                                                *
 * Also, see NOTE on CSR usage at the             *
 * beginning of cbstart.                          *
 ***********************************************/

cbincled(cb)
struct cb_unit *cb; /* Pointer to unit data structure */
{
                int tmp;

                tmp = read_io_port(cb->cbr | CB_CSR,
                                   4,
                                   0);
                                   tmp -= 0x1000;
                write_io_port(cb->cbr | CB_CSR,
                              4,
                              0,
                              tmp); /* "Increment" lights */
     if(cb->ledflag != 0) {   /* If still set, */
            timeout(cbincled, (caddr_t)cb, CBIncSec*hz); /* restart timer */
            }
     return;
}
```

```
/***************************************************
 *       Device Interrupt Handler Section          *
 ***************************************************/
/***************************************************
 *---------------- cbintr -------------------------*
 ***************************************************/


cbintr(ctlr)
int ctlr; /* Index for controller structure */
{
        int tmp;
        struct cb_unit *cb;     /* Pointer to unit data structure */
        cb = &cb_unit[ctlr];    /* Point to this device's structure */
        tmp = read_io_port(cb->cbr | CB_TEST,
                           4,
                           0); /* Read test reg to clear "go" bit */
        cb->intrflag++;         /* Set flag to tell it happened */

/***************************************************
 *              DEBUG STATEMENT                    *
 ***************************************************/
#ifdef CB_DEBUG
printf("\nCBintr interrupt, ctlr = %d\n",ctlr); /* Show interrupt */
#endif /* CB_DEBUG */


        return;
}
```

Device Driver Development Worksheets C

This appendix provides worksheets that you can use to help you gather information about designing, coding, installing, and testing a device driver. Chapter 2 explains how to fill out these worksheets. You may make copies of these worksheets.

HOST SYSTEM WORKSHEET

Specify the Host CPU

Alpha-based CPUs:

DEC 3000 Model 400 AXP Workstation	☐
DEC 3000 Model 500 AXP Workstation	☐
DEC 4000 Model 600 AXP Distributed/ Departmental Server	☐
DEC 7000 Model 600 AXP Server	☐
DEC 10000 Model 600 AXP Server	☐

Other Alpha-based CPUs: ______________

MIPS-based CPUs:

DECstation 5000 Model 100	☐
DECstation 5000 Model 200	☐
DECstation 5000 Model 300	☐
DECstation 5400	☐
DECstation 5500	☐
DECstation 5900	☐

Other MIPS-based CPUs: ______________

HOST SYSTEM WORKSHEET (Cont.)

Other CPU Architectures:

______________________ ☐

______________________ ☐

______________________ ☐

______________________ ☐

______________________ ☐

Specify the host operating system:

DEC OSF/1	☐
ULTRIX	☐

Other operating systems ______________________

____

Specify the bus or buses you plan to connect to the driver:

TURBOchannel	☐
VMEbus	☐
Q–bus	☐
UNIBUS	☐
SCSI	☐
Pseudodevice drivers	☐

Other buses ______________________

____

DEVICE DRIVER CONVENTIONS WORKSHEET

Describe the naming scheme you are following for

Device driver interfaces:

Device driver structures:

Device driver constants:

Device connectivity information:

DEVICE DRIVER CONVENTIONS WORKSHEET (Cont.)

Describe the approach to writing comments in the device driver:

Describe the approach to writing device driver documentation:

DEVICE CHARACTERISTICS WORKSHEET

Specify the following about the device:

	YES	NO
1. The device is capable of block I/O	☐	☐
2. The device will support a file system	☐	☐
3. The device supports byte stream access	☐	☐

Specify the actions that need to be taken if the device generates interrupts:

DEVICE CHARACTERISTICS WORKSHEET (Cont.)

Specify how the device should be reset:

Use the remainder of the worksheet to specify any other device characteristics:

DEVICE USAGE WORKSHEET

List the documentation you have on the device (the device documentation can help you answer subsequent questions):

Answer the following questions about the usage of the device:

1. How many of this device type can reside on the system?

2. What will the device be used for?

DEVICE REGISTER WORKSHEET

Describe or sketch the layout of the device registers. Include a short description of the purpose of each register:

DEVICE REGISTER WORKSHEET (Cont.)

Specify which memory address the registers are associated with:

Device Register	Memory Address

DEVICE DRIVER SUPPORT WORKSHEET

Specify one of the following about the device driver:

	YES	NO
1. There is no driver for this device. You are going to write it from scratch.	☐	☐
2. The driver for this device was previously written for an ULTRIX system and the code is available.	☐	☐
3. The driver for this device was previously written for a UNIX system and the code is available.	☐	☐
4. The driver for this device was previously written for another operating system and the code is available.	☐	☐
5. The existing device driver has documentation.	☐	☐

If the answer is yes, specify the title and location of documentation:

Title: ______________________________

Location: ______________________________

6. If the source code is available, specify the location:

Location: ______________________________

7. Identify any experts available whose experience you can draw on for:

	Expert's Name, Number, Location:
The device:	
The design:	
The coding:	
The installation:	
The debugging:	
The testing:	

DEVICE DRIVER TYPE WORKSHEET

Specify the type of driver:

Character	☐
Block	☐
Character and Block	☐
Network	☐
Loadable	☐
Static	☐

DEVICE DRIVER ENTRY POINTS WORKSHEET

Block driver entry points:

Entry point:	Name:
probe	
slave	
cattach	
dattach	
configure	
open	
close	
strategy	
ioctl	
interrupt	
psize	
dump	

Character driver entry points:

Entry point:	Name:
probe	
slave	
cattach	
dattach	
configure	
open	
close	
strategy	
ioctl	
stop	
reset	
read	
write	
mmap	
interrupt	

DEVICE DRIVER TESTING WORKSHEET

Specify the scope of the driver test program:

	YES	NO
1. The test program checks all entry points	☐	☐
2. The test program checks all ioctl requests separately	☐	☐
3. The test program checks multiple devices	☐	☐
4. The test program was run with multiple users using the device	☐	☐
5. The test program includes debug code to check for impossible situations	☐	☐
6. The test program tests which entry points are available through the system call interface	☐	☐

Glossary

autoconfiguration

Autoconfiguration is a process that determines what hardware actually exists during the current instance of the running kernel.

autoconfiguration software

The autoconfiguration software consists of the programs that accomplish the tasks associated with the events that occur during the autoconfiguration of devices. In most cases, it is not necessary for device driver writers to know the specific programs that execute during autoconfiguration.

bdevsw table

The block device switch, or `bdevsw`, table is an array of data structures that contains pointers to 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 entry points not used by a device driver. See also **cdevsw table** and **device switch table**.

block device

A block device is a device that is designed to operate in terms of the block I/O supported by DEC OSF/1. It is accessed through the buffer cache. A block device has a block device driver associated with it.

block device driver

A block device driver is a driver that performs I/O by using file system block-sized buffers from a buffer cache supplied by the kernel. Block device drivers are particularly well-suited for disk drives, the most common block devices.

buffer cache

A buffer cache is supplied by the kernel and contains file system block-sized buffers. Block device drivers use these buffers in I/O operations.

buf structure

The `buf` structure describes arbitrary I/O, but is usually associated with block I/O and `physio`.

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. See also **OPENbus**.

bus structure

The `bus` structure represents an instance of a bus entity. A bus is a real or imagined entity to which other buses or controllers are logically attached. All systems have at least one bus, the system bus, even though the bus may not actually exist physically. The term controller here refers both to devices that control slave devices (for example, disk and tape controllers) and to devices that stand alone (for example, terminal or network controllers).

bus support subsystem

The bus support subsystem contains all of the bus adapter-specific code. Isolating the bus-specific code and data structures into a bus support subsystem makes it easier for independent software vendors to implement different bus adapters.

The bus support subsystem communicates with the hardware-dependent and device driver subsystems of the hardware-independent model.

busy wait time

Busy wait time is the amount of CPU time expended on waiting for a simple lock to become free.

cdevsw table

The character device switch, or `cdevsw`, table is an array of data structures that contains pointers to device driver entry points for each character 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 entry points not used by a device driver. See also **bdevsw table** and **device switch table**.

central processing unit

The central processing unit (CPU) is the main computational unit in a computer and the one that executes instructions. The CPU is of interest to device driver writers because its associated architecture influences the design of the driver. For example, CPUs can have different mechanisms for handling memory mapping.

cfgmgr daemon

The `cfgmgr` daemon is a system management process that works with `kloadsrv`, the kernel load server, to manage loadable device drivers.

character device

A character device is any device that can have streams of characters read from or written to it. A character device has a character device driver associated with it.

character device driver

A character device driver is a driver that can use a variety of approaches to handle I/O. A character device driver can accept or supply a stream of data based on a request from a user process. A character device driver can be used for a device such as a line printer that handles one character at a time. However, character drivers are not limited to performing I/O one character at a time (despite the name ''character'' driver). For example, tape drivers frequently perform I/O in 10K chunks. You can also use a character device driver when it is necessary to copy data directly to or from a user process.

Because of their flexibility in handling I/O, many drivers are character drivers. Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

compile time variable

The compile time variable defines how many devices exist on the system and is created by the `config` program for use by static device drivers.

config.file file fragment

The `config.file` file fragment can be viewed as a ''mini'' system configuration file. It is the mechanism by which third-party driver writers supply device connectivity, callout keywords, and other information related to their static device driver product and needed by their customers.

config program

The `config` program is a system management tool that `doconfig` calls. The `config` program either creates a new or modifies an existing system configuration file, copies `.products.list` to *`NAME`*`.list`, creates the device special files for static drivers, and builds a new DEC OSF/1 kernel.

control status register (CSR)

See **device register**.

controller

A device controller is the hardware interface between the computer and a peripheral device. Sometimes a controller handles several devices. In other cases, a controller is integral to the device.

controller structure

The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation.

daemon

A daemon is a system management process that controls a variety of kernel tasks.

See also **cfgmgr daemon**.

data structure

Data structures are the mechanism used to pass information between the DEC OSF/1 kernel and device driver interfaces.

device autoconfiguration

See **autoconfiguration**.

device controller

See **controller**.

device driver

A device driver is a software module that resides within the DEC OSF/1 kernel and is the software interface to a hardware device or devices. The purpose of a device driver is to handle requests made by the kernel with regard to a particular type of device. See also **block device driver**, **character device driver**, and **network device driver**.

device driver configuration

Device driver configuration is the process of incorporating device drivers into the kernel and making them available to system management and other utilities. There are two configuration models: the third-party and the traditional device driver configuration models.

See also **third-party device driver configuration model** and **traditional device driver configuration model**.

device driver header file

The device driver header file contains `#define` statements for as many devices as are configured into the system. This file is generated by the `config` program during static configuration of the device driver. This file need not be included if you configure the driver as a loadable driver.

device driver kit

The device driver kit contains the files associated with a device driver product. These files contain information necessary for system managers (customers) to configure loadable or static drivers into their systems.

device driver subsystem

The device driver subsystem contains all of the driver-specific code. The device driver subsystem communicates with the hardware-dependent, bus support, and hardware-independent subsystems of the hardware-independent model.

See also **bus support subsystem**, **hardware-dependent subsystem**, and **hardware-independent subsystem**.

device register

A device register is commonly referred to as a control status register, or CSR. The device register can be used to:

- Control what a device does
- Report the status of a device
- Transfer data to or from the device

device register header file

The device register header file contains any public declarations used by the device driver. This file usually contains the device register structure associated with the device.

device register structure

A device register structure is a C structure whose members map to the registers of some device. These registers are often referred to as the device's control status register or CSR addresses. The device register structure is usually defined in the device register header file.

device structure

The `device` structure represents an instance of a device entity. A device is an entity that connects to and is controlled by a controller. A device does not connect directly to a bus.

device switch table

The device switch tables, `bdevsw` for block devices and `cdevsw` for character devices, have the following characteristics:

- They are arrays of structures that contain device driver entry points. These entry points are actually the addresses of the specific interfaces within the drivers.
- They may contain stubs for device driver entry points for devices that do not exist on a specific machine.
- The location in the table corresponds to the device major number.

See also **bdevsw table** and **cdevsw table**.

direct memory access

Direct memory access (DMA) describes the ability of a device to directly access (read from and write to) CPU memory, without CPU intervention.

direct memory access (DMA) device

A direct memory access (DMA) device is one that can directly access (read from and write to) CPU memory, without CPU intervention. Non-DMA devices cannot directly access CPU memory.

DMA handle

Specifies a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the `ba` and `bc` members of an `sg_entry` structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

doconfig

The `doconfig` program is a system management tool that calls `config`. See **config program**.

driver structure

The `driver` structure defines driver entry points and other driver-specific information. You declare and initialize an instance of this structure in the device driver.

GENERIC system configuration file

The `GENERIC` system configuration file supplied by Digital contains all the possible software and hardware options available to DEC OSF/1 systems and includes all supported Digital devices. The `GENERIC` system configuration file is used to build a kernel that represents all possible combinations of statically configured drivers that Digital supports.

hardware device

See **peripheral device**.

hardware-dependent subsystem

The hardware-dependent subsystem contains all of the hardware-dependent pieces of an operating system with the exception of device drivers. This subsystem provides the code that supports a specific CPU platform and, therefore, is implemented by specific vendors.

The hardware-dependent subsystem communicates with the hardware-independent subsystem, device driver subsystem, and bus support subsystem of the hardware-independent model.

hardware-independent model

The hardware-independent model describes the hardware and software components that make up an open systems environment. Specifically, these hardware and software components are contained in a hardware-independent subsystem, hardware-dependent subsystem, bus support subsystem, and device driver subsystem.

hardware-independent subsystem

The hardware-independent subsystem contains all of the hardware-independent pieces of an operating system, including the hardware-independent kernel interfaces, user programs, shells, and utilities. The

Open Software Foundation (OSF) provides the hardware-independent subsystem. This subsystem can contain extensions and enhancements made by vendor companies, including Digital Equipment Corporation.

The hardware-independent subsystem communicates with the hardware-dependent subsystem, bus support subsystem, and device driver subsystem of the hardware-independent model.

ihandler_id_t key

The `ihandler_id_t` key is a unique number used to identify interrupt service interfaces to be acted on by subsequent calls to the `handler_enable`, `handler_disable`, and `handler_del` kernel interfaces.

ihandler_t structure

The `ihandler_t` structure contains information associated with device driver interrupt handling. Loadable drivers use this data structure.

interrupt service interface (ISI)

An interrupt service interface is a device driver routine that handles hardware interrupts. Driver writers implementing static-only device drivers specify a device driver's interrupt service interface in the system configuration file if they are following the traditional device driver configuration model and in the `config.file` file fragment if they are following the third-party device driver configuration model.

Loadable device drivers, unlike static device drivers, must call a number of kernel interfaces to register, deregister, enable, and disable a device driver's interrupt service interface.

ioconf.c file

The `ioconf.c` file contains the `bus_list`, `controller_list`, and `device_list` arrays created by the `config` program for static device drivers during the autoconfiguration process.

I/O handle

An I/O handle is a data entity that is of type `io_handle_t`. This I/O handle provides device drivers with bus address information. The bus configuration code passes the I/O handle to the device driver's *xx*`probe` interface during device autoconfiguration.

kernel

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

kernel framework

See **subsystem**.

kloadsrv
The kernel loader daemon is used with the `cfgmgr` daemon to load the specified loadable device driver into the kernel address space and to resolve external references. See also **cfgmgr daemon**.

kmknod
The `kmknod` utility is a system management tool that uses the information from the `stanza.static` file fragment to dynamically create device special files for static device drivers at boot time.

kreg utility
The `kreg` utility is a system management tool that maintains the `/sys/conf/.product.list` system file, which registers static device driver products.

loadable device driver
A loadable device driver is a driver (block or character) that is linked dynamically into the kernel at run time. This type of device driver is installed without having to rebuild the kernel, shut down the system, and reboot. See also **static device driver**.

load module
A load module is the executable image of a loadable device driver.

method
A method is a subsystem specific portion of the `cfgmgr` daemon. For example, the device method is the portion of the `cfgmgr` daemon that handles loadable device drivers.

name_data.c file
The *name*`_data.c` file provides a convenient place to size the data structures and data structure arrays that device drivers use. In addition, the file can contain definitions that third-party driver writers might want their customers to change. This file is particularly convenient for third-party driver writers who do not want to ship device driver sources.

NAME.list file
The *NAME*`.list` file is a copy of the `.products.list` file that is created when the system manager installs the device driver kit supplied by a third-party vendor.

network device
A network device is any device associated with network activities and is responsible for both transmitting and receiving frames to and from the network medium. Network devices have network device drivers associated with them.

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.

nexus
The `nexus` keyword indicates the top of the system configuration tree.

OPENbus
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 TURBOchannel and the VMEbus, for example, can be classified as having OPENbus architectures.

open systems
The term open systems refers to an environment with hardware and software platforms that promote the use of standards. By adhering to a set of standard interfaces, these platforms make it easier for third-party programmers to write applications that can run on a variety of operating systems and hardware. This open systems environment can also make it easier for systems engineers to write device drivers for numerous peripheral devices that operate on this same variety of operating systems and hardware.

peripheral device
A peripheral device is hardware, such as a disk controller, 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.

port structure
The `port` structure contains information about a port.

.products.list file
The `/usr/sys/conf/.products.list` file (for static drivers) stores information about static device driver layered products.

pseudodevice driver
A pseudodevice driver, such as the `pty` terminal driver, is structured like any other driver. The difference is that a pseudodevice driver does not operate on a bus.

resource
A resource, from the device driver's point of view, is data or a code block that can be manipulated by more than one thread. If the resource is data, it can be stored in variables (global and local) and data structure members.

setld
The `setld` utility allows the transfer of the contents of the device driver kit to a customer's system on DEC OSF/1.

stanza.loadable file fragment

The `stanza.loadable` file fragment can be viewed as a ''mini'' `sysconfigtab` database because it contains some of the same information. The `stanza.loadable` file fragment contains an entry for each device driver, providing such information as the driver's name, location of the loadable object, device connectivity information, and device special file information. Parts of the `stanza.loadable` file fragment are functionally similar to the system configuration file in that it uses a subset of the syntaxes used in the system configuration file to specify each current or planned device on the system.

stanza.static file fragment

The `stanza.static` file fragment (for static drivers) contains such items as the driver's major number requirements, the names and minor numbers of the device special files, the permissions and directory name where the device special files reside, and the driver interface names to be added to the `bdevsw` and `cdevsw` tables.

static device driver

A static device driver is a driver (block, character, or network) that is linked directly into the kernel. This type of device driver must be installed by completing tasks that include rebuilding the kernel, shutting down the system, and rebooting. See also **loadable device driver**.

subset control program (SCP)

A subset control program (SCP) is a program written by the kit developer that contains path specifications for all of the files related to the driver product. The SCP is invoked by `setld` during the installation of the device driver kit.

subsystem

A subsystem is a kernel module that defines a set of kernel framework interfaces that allow for the dynamic configuration and unconfiguration (adding and removal) of subsystem functionality. Examples of subsystems include (but are not restricted to) device drivers, file systems, and network protocols. The ability to dynamically add subsystem functionality is utilized by loadable drivers to allow the driver to be configured and unconfigured without the need for kernel rebuilds and reboots.

sysconfig

The `sysconfig` utility is a system management tool that modifies the loadable subsystem configuration. This modification provides a user interface to the `cfgmgr` daemon.

sysconfigdb

The `sysconfigdb` utility is a system management tool that maintains the `sysconfigdb` database. The driver stanza entries in the `stanza.loadable` file fragment are appended to this database.

sysconfigtab database
The `sysconfigtab` database contains the information provided in the `stanza.loadable` file fragments. This information is appended to the `sysconfigtab` database during installation of the device driver kit.

system configuration file
The system configuration file is an ASCII text file that defines the components of the system. These components are described using valid keywords such as those that identify device definitions, callout definitions, and pseudodevice definitions.

system configuration tree
The system configuration tree represents the result of the autoconfiguration process, after the autoconfiguration software reads the entries in the system configuration file (for static drivers) and the `sysconfigtab` database (for loadable drivers). For static drivers, the result is a correctly linked list of `bus`, `controller`, and `device` structures. As loadable drivers are dynamically loaded, their `bus`, `controller`, and `device` structures are linked into the system configuration tree.

tc_intr_info structure
The `tc_intr_info` structure contains interrupt handler information for device controllers connected to the TURBOchannel bus. Loadable drivers initialize the members of the `tc_intr_info` structure, usually in the driver's `probe` interface.

terminal device
A terminal device is a special type of character device that can have streams of characters read from or written to it. Terminal devices have terminal (character) device drivers associated with them.

terminal device driver
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. It cannot be mounted as a file system and, therefore, does not use data caching.

third-party device driver configuration model
This model is recommended for third-party device driver writers who want to ship loadable and static drivers to customers running Digital's DEC OSF/1 operating system. The third-party device driver configuration model provides tools that customers use to automate the installation of third-party device drivers. This model requires that

third-party driver writers provide a device driver kit to their customers.

traditional device driver configuration model

The traditional device driver configuration model provides a manual mechanism for driver writers or system managers to configure device drivers into the kernel. One advantage of the traditional model is that no device driver kit is needed. The disadvantage is that the traditional model is not automated and potentially error prone.

TURBOchannel test board

The TURBOchannel test board is a minimal implementation of all the TURBOchannel hardware functions: programmed I/O, DMA read, DMA write, and I/O read/write conflict testing. The `/dev/cb` device driver provides a simple interface to the functions provided by the TURBOchannel test board.

uio structure

The `uio` structure describes I/O, either single vector or multiple vectors. Typically, device drivers do not manipulate the members of this structure.

user program

A user program is a software module that allows a user of the DEC OSF/1 operating system to perform some task. For example, the `ls` user program allows users to list the files contained in a specific directory. User programs make system calls to the kernel that result in the kernel making requests of a device driver. A user program never directly calls a device driver.

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) 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-bps 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[a]	————	SSB Order Processing – NQO/V19 *or* U. S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

DEC OSF/1
Writing Device Drivers, Volume 1: Tutorial
AA-PUBVB-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 PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here --

Cut
Along
Dotted
Line

Reader's Comments

DEC OSF/1
Writing Device Drivers, Volume 1: Tutorial
AA-PUBVB-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 PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3–3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062–9987

--- Do Not Tear – Fold Here --

**Cut
Along
Dotted
Line**