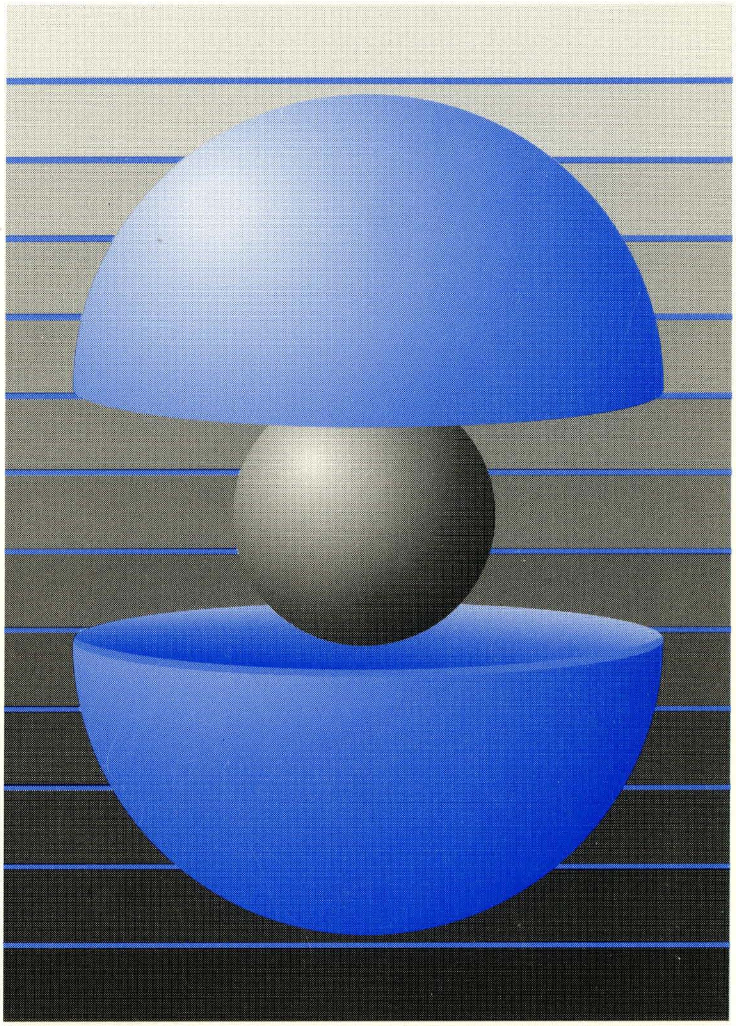
DEC OSF/1

digital

Writing TURBOchannel Device Drivers



Part Number: AA-PS3HB-TE

DEC OSF/1

Writing TURBOchannel Device Drivers

Order Number: AA-PS3HB-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This guide includes information about device drivers that operate on the TURBOchannel bus. The book is a companion volume to *Writing Device Drivers, Volume 1: Tutorial*, which discusses the general concepts and tasks associated with writing a device driver, and *Writing Device Drivers, Volume 2: Reference*, which contains reference (man) pages for interfaces and files used by driver writers.

digital equipment corporation
Maynard, Massachusetts

Contents

2 TURBOchannel Architecture

3 Structure of a TURBOchannel Device Driver

4 Kernel I/O Support Interfaces Used by TURBOchannel Device Drivers

5 Device Autoconfiguration

6 TURBOchannel Device Driver Configuration

A TURBOchannel-Specific Reference Information

Index

Figures

Tables

About This Manual

This book discusses topics associated with writing device drivers that operate on the TURBOchannel bus.

Audience

This book is intended for systems engineers who:

- Develop programs in the C language using standard library interfaces
- Know the Bourne or some other UNIX-based shell
- Understand basic DEC OSF/1 concepts such as kernel, shell, process, configuration, autoconfiguration, and so forth
- 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, memory mapping, and I/O.

This book assumes you have a strong background in UNIX-based operating systems and C programming, and that you are familiar with topics presented in *Writing Device Drivers, Volume 1: Tutorial.* This book refers to *Writing Device Drivers, Volume 1: Tutorial* when appropriate.

Organization

Chapter 1 Review of Device Driver Concepts

Reviews general device driver concepts that are discussed in detail in *Writing Device Drivers, Volume 1: Tutorial.*

Chapter 2 TURBOchannel Architecture

Discusses aspects of the TURBOchannel software architecture with which a device driver writer must be familiar.

Chapter 3	Structure of a TURBOchannel Device Driver Describes the sections that make up a TURBOchannel device driver.
Chapter 4	Kernel Interfaces Used by TURBOchannel Device Drivers Discusses the kernel interfaces developed for use with TURBOchannel device drivers.
Chapter 5	Device Autoconfiguration Discusses the sequence of events that occurs during the autoconfiguration of TURBOchannel devices.
Chapter 6	TURBOchannel Device Driver Configuration Reviews the device driver configuration models presented in *Writing Device Drivers, Volume 1: Tutorial* and discusses some driver configuration topics associated with TURBOchannel device drivers.
Appendix A	TURBOchannel-Specific Reference Information Presents, in reference (man) page style, descriptions of the header file, data structures, kernel support interfaces, and device driver interfaces that are specific to the TURBOchannel bus.

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 that requires driver writers to acquire knowledge in a variety of areas. The following categories of documentation are available to help you acquire expertise in these areas:

- Hardware documentation
- Device driver documentation
- Programming and programming tools documentation
- System management documentation
- Porting documentation
- Reference pages

Hardware Documentation

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

Device Driver Documentation

This book contains information specific to device drivers that operate on the TURBOchannel bus. It should be used in conjunction with the following books that provide general information about writing device drivers:

- *Writing Device Drivers, Volume 1: Tutorial*

  This manual provides information for systems engineers who write device drivers for hardware that runs the DEC OSF/1 operating system. Systems engineers can find 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.

- *Writing Device Drivers, Volume 2: Reference*

  This manual contains descriptions of the header files, kernel support interfaces, `ioctl` commands, global variables, data structures, device driver interfaces, and bus configuration interfaces associated with device drivers. The descriptions are formatted similar to the DEC OSF/1 reference pages.

The following book provides information about device drivers that operate on the SCSI CAM bus:

- *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.

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.

System Management Documentation

Refer to the followng book for information about building a kernel and for general information about system administration:

- *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

Reference (man) pages that are of interest to device driver writers can be found in the following documents:

- *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. 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.

Review of Device Driver Concepts 1

Before attempting to write a driver for a TURBOchannel device, you must be familiar with driver concepts in general as well as specific tasks you need to perform to successfully code the driver. An understanding of the following concepts is presumed:

- 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

1.1 Information Gathering

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. For example, you need to:

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

1.2 Device Driver Design

After you gather information about the host system and the device, you are ready to design the device driver. You need to specify the driver type and whether the driver you write links into the kernel dynamically at run time (loadable) or requires a reboot (static). During the design of the driver, you also identify device driver entry points and describe the driver states.

1.3 Data Structures

Data structures are the mechanism used to pass information between the kernel and device driver interfaces. The following table summarizes data structures that are described in detail in *Writing Device Drivers, Volume 1: Tutorial*:

`buf` and `uio`	Used in I/O operations.
`controller`	Contains members that store information about hardware resources and store data for communication between the kernel and the device driver.
`device`	Contains information that identifies the device. There is one `device` data structure for each device connected to the controller.
`driver`	Is initialized by the driver writer in the device driver. This data structure specifies information such as the `probe`, `slave`, and `attach` interfaces used in the device driver.
`bus`	Represents an instance of a bus entity to which other buses or controllers are logically attached.
`cdevsw`	Defines a device driver's entry points in the character device switch table.
`bdevsw`	Defines a device driver's entry points in the block device switch table.

Appendix A describes data structures that are used only with the TURBOchannel bus: `tc_info`, `tc_intr_info`, `tc_memerr_status`, and `tc_option`.

Although loadable device drivers are not supported in this version of DEC OSF/1, you may want to implement loadable device drivers that operate on the TURBOchannel bus. If so, you also need to consider data structures specific to loadable device drivers: `ihandler_t`, `handler_key`, and `device_config_t`. The structure `tc_intr_info`, which is used only for the TURBOchannel bus, is also specific to loadable drivers.

When designing your device driver, you must decide on the technique you will use for allocating data structures. Generally, there are two techniques you can use: static allocation and dynamic allocation. Dynamic allocation is the recommended method for all new drivers; some existing drivers allocate data structures statically. If you do not plan to implement loadable drivers now or in the future, the static allocation method will suffice. Otherwise (or if you know that the maximum number of devices is greater than five or that the driver uses numerous data structures), plan to use the dynamic allocation method.

1.4 Portability

Whenever possible, design your device driver so that it can accommodate peripheral devices that operate on more than one CPU architecture and more than one CPU type within the same architecture. The following list notes some of the issues you need to consider to make your drivers portable:

- Control status register (CSR) access issues
- Input/Output (I/O) copy operation issues
- Direct memory access (DMA) operation issues
- 64-bit versus 32-bit issues

Note

See *Writing Device Drivers, Volume 1: Tutorial* for information on the CSR I/O access interfaces. These interfaces allow you to read from and write to a device's CSR addresses without directly accessing its device registers. Each of these interfaces takes an I/O handle that the bus configuration code passes to the driver's probe interface.

1.5 Configuration Models

When you are ready to write your driver, you probably want to study the device driver configuration models and create an appropriate device driver development environment. If you plan to ship a device driver product to customers, you also need to create a device driver kit development environment.

This is a brief summary of device driver concepts and the considerations you must make prior to writing a driver. If you are unfamiliar with any of these, consult *Writing Device Drivers, Volume 1: Tutorial*. The tutorial discusses how to write device drivers for computer systems running the DEC OSF/1 operating system.

TURBOchannel Architecture 2

The TURBOchannel is a synchronous, asymmetrical I/O channel that is supported on some Alpha AXP CPUs.

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

- Include files
- Writes to the hardware device register
- DMA-to-host memory transfers
- Device interrupt line to the processor

2.1 Include Files

TURBOchannel device drivers, in addition to the usual header files required by DEC OSF/1 device drivers, need this header file:

```
#include <io/dec/tc/tc.h>
```

See *Writing Device Drivers, Volume 1: Tutorial* for information on header files required by all device drivers.

2.2 Writes to the Hardware Device Register

Whenever a TURBOchannel device driver writes to a hardware device register, the write may be delayed by the system write buffer used to synchronize the CPU on the TURBOchannel. A subsequent read of that register is not guaranteed to wait for the write to complete. To ensure that a write to I/O space completes prior to a subsequent read operation, the driver calls the `mb` kernel support interface. To ensure that multiple writes to the same hardware device register result in the device receiving the correct number of write requests in the proper order, you must insert calls to `mb` between each write. Otherwise, write requests may be merged in the write buffer with the result that the device receives fewer write requests than you intend or receives them in the wrong order.

The `wbflush` interface is available on MIPS systems and is aliased to `mb` on Alpha AXP systems. Section 4.5 contains an example of using the `mb` interface to ensure that a write to I/O space completes.

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

There are several TURBOchannel-specific interfaces associated with DMA transfers. The `tc_map_alloc`, `tc_loadmap`, and `tc_map_free` interfaces let you allocate, load, and free (return to the free pool) entries for scatter-gather maps. The `tc_isolate_memerr` interface lets you isolate and log errors that occur on a DMA transfer.

2.4 Device Interrupt Line

If a device needs to have its interrupts enabled or disabled during configuration or during operation, a TURBOchannel device driver can call the `tc_enable_option` and `tc_disable_option` interfaces. See Section 4.1 and Appendix A for more information about these interfaces.

Structure of a TURBOchannel Device Driver 3

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 that a block device driver can contain. Device drivers do not have to use all of the sections illustrated in the figure, and more complex drivers can use additional sections. Both character and block device 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 can contain the following sections not contained in a block device driver:

- A read and write device section
- A reset section
- A stop section

Writing Device Drivers, Volume 1: Tutorial discusses each of the driver sections. The remainder of this chapter describes the include file and autoconfiguration support sections as they apply to TURBOchannel device drivers.

Throughout this chapter there are frequent references to device driver coding that applies only to a static or a loadable version of the driver. As shown by an example in this chapter, you can structure your driver so that the same source file is used to support both static and loadable versions of the driver. Differences between static and loadable versions of a driver are determined at driver runtime rather than driver compilation time. When your driver code supports both versions, the customer who installs the driver can choose which version is configured on the system.

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 */
 .
 .
 .
```

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 */
 .
 .
 .
```

ZK–0875U–R

3.1 Include Files Section

Data structures are defined in header files that you include in the driver source code. The number and types of header files you specify depends on the structures, constants, and kernel interfaces your device driver references. You need to be familiar with:

- The device driver header file
- Common driver header files
- Loadable driver header files
- The device register header file
- The *name*`_data.c` file

These files are described in *Writing Device Drivers, Volume 1: Tutorial.* Two files specific to TURBOchannel device drivers are `tc.h` and `tc_option_data.c`. The include file `tc.h` specifies definitions and declarations of interfaces that are used only in TURBOchannel device drivers. The `tc_option_data.c` file contains the declaration of the `tc_option` data table. This table maps the device name in the ROM (Read Only Memory) on the hardware device module to the controlling driver in the DEC OSF/1 kernel.

3.2 Autoconfiguration Support Section

As described in *Writing Device Drivers, Volume 1: Tutorial*, when the operating system boots, the kernel determines which 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. To configure loadable drivers, the kernel uses a procedure that is functionally equivalent to the one just described. A loadable driver, like the static driver, has `probe`, `attach`, and possibly `slave` interfaces.

From the driver writer's point of view, the `probe`, `attach`, and `slave` interfaces are the same for static and loadable versions of the driver. However, the functions performed by these interfaces can differ, depending on the driver version. For example, the code that supports the loadable version of the driver in the *xx*`probe` interface registers the driver's interrupt handlers. This section of code does not apply to the static version of the driver because interrupt handlers for static drivers are specified in the system configuration file and built into the kernel.

The autoconfiguration support section of a TURBOchannel device driver contains the code that implements these interfaces and the section applies to both character and block device drivers. The section 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. Refer to *Writing Device Drivers, Volume 1: Tutorial* for a description of the `driver` structure. The following sections show you how to set up each of these interfaces for the TURBOchannel bus.

3.2.1 Setting Up the xxprobe Interface

A device driver's *xx*`probe` interface performs tasks necessary to determine if the device exists and is functional on a given system. Specific tasks performed by this interface vary, depending on whether the device driver is statically or dynamically configured:

- When drivers are configured statically, the kernel calls the *xx*`probe` interface at boot time to check for the existence of each device defined in the system configuration file. Typically, the *xx*`probe` interface determines whether the device is present by checking some device status register. Calling the `BADADDR` interface is one way to check device status registers. If the device is not present, the device is not initialized and not available for use. On success, the *xx*`probe` interface returns a nonzero value. On error, the *xx*`probe` interface returns zero (`0`).
- When device drivers are configured dynamically, the kernel indirectly calls the *xx*`probe` interface to register the device interrupt handlers. The interface registers the device interrupt handlers with calls to the `handler_add` and `handler_enable` interfaces.

  The *xx*`probe` interface is called for each stanza entry that was defined in the `stanza.loadable` file for the device driver. This file includes declarations of the driver's connectivity information as specified by the `Module_Config` fields. The kernel calls the `ldbl_stanza_resolver` interface to merge the driver's connectivity information into the system configuration of `bus`, `controller`, and `device` structures. Then, a call to the `ldbl_ctlr_configure` interface results in a call to the *xx*`probe` interface for each instance of the controller present on the TURBOchannel bus. On success, the *xx*`probe` interface returns a nonzero value. On error, the interface returns the value zero (`0`).

It is important to note the differences between the tasks performed by the static and loadable versions of the driver and to conditionalize your code accordingly:

- When a driver is dynamically configured, the `xxprobe` interface cannot call the `BADADDR` interface, which is available only in the early stages of system booting.
- When a driver is statically configured, the `xxprobe` interface does not register device interrupt handlers. Device interrupt handlers are defined in the system configuration file or the `stanza.static` file fragment and registered at system configuration time by the `config` program.

The following code fragment shows the `cbprobe` interface as defined for the `/dev/cb` device driver example in *Writing Device Drivers, Volume 1: Tutorial*. Note that tasks specific to loadable device drivers are identified by a conditional `if` statement that tests the value of the `cb_is_dynamic` variable:

```
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

        if (cb_is_dynamic) { 6

                handler.ih_bus = ctlr->bus_hd; 7
                info.configuration_st = (caddr_t)ctlr; 8
                info.config_type = TC_CTLR; 9

                info.intr = cbintr; 10

                info.param = (caddr_t)unit; 11
                handler.ih_bus_info = (char *)&info; 12

                cb_id_t[unit] = handler_add(&handler); 13
                if (cb_id_t[unit] == NULL) { 14

                        return(0);
                }
                if (handler_enable(cb_id_t[unit]) != 0) { 15

                        handler_del(cb_id_t [unit]); 16
                        return(0);
                }
        }

        num_cb++; 17

        return(1); 18
}
```

1 Declare 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 Declare 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 network interface or terminal controller. This line is applicable to the loadable or static version of the /dev/cb device driver.

3 Declare an ihandler_t data structure called handler to contain information associated with the /dev/cb device driver interrupt handling. 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 Declare a tc_intr_info data structure called info (used only by the loadable version of the driver).

5 Declare a *unit* variable and initialize it to the controller number. This controller number identifies the specific CB controller being retrieved by this call to cbprobe.

The controller number is contained in the ctlr_num member of the controller structure associated with this CB device. This member is used as an index into a variety of tables to retrieve information about this instance of the CB device.

6 Register 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 *Writing Device Drivers, Volume 1: Tutorial.*

The items from 7 – 17 set up the driver's interrupt handler and are applicable only if the driver is dynamically loaded.

7 Specify 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. This setting of the ih_bus member is necessary because the process of registering the interrupt handlers indirectly calls interrupt registration interfaces that are bus-specific.

8 Set the configuration_st member of the info data structure to the pointer to the controller structure associated with this CB device. This is the controller structure 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.

9 Set 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.

10 Set the intr member of the info data structure to cbintr, the /dev/cb device driver's interrupt service interface (ISI).

11 Set 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 this parameter to specify the instance of the controller with which the interrupt is associated.

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.

12 Set 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 the interrupt registration process indirectly calls bus-specific interfaces to register the interrupt handlers.

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.

13 Call the handler_add interface and save 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.

14 If the return value from handler_add equals NULL, return a failure status to indicate that registration of the interrupt handler failed.

15 If the handler_enable interface returns a nonzero value, return 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 *cb_id_t* array.

16 If the call to `handler_enable` fails, remove the previously registered interrupt handler by calling the `handler_del` interface before returning an error status.

17 Increment the number of instances of this controller found on the system.

18 The `cbprobe` interface simply returns the value `1` to indicate success status because the TURBOchannel initialization code already verified that the device was present.

3.2.2 Setting Up the xxslave Interface

A device driver's *xxslave* 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. You specify the attachments of these slave devices:

- For the static version of your driver, in the system configuration file or `stanza.static` file fragment
- For the loadable version of your driver, in the `stanza.loadable` file fragment

The following code fragment illustrates how to set up an *xxslave* interface:

```
xxslave(dev, addr)
struct device *device;  1
caddr_t addr;         2
{
/* declarations of variables and structures */
   .
   .
   .
/* code to check that the device is valid */
   .
   .
   .
}
```

1 Declare a pointer to the `device` structure for this device.

2 Declare an argument to specify the SVA (System Virtual Address) of the base of the TURBOchannel slot space for the controller to which this device is connected.

3.2.3 Setting Up the xxcattach and xxdattach Interfaces

The *xx*`cattach` and *xx*`dattach` interfaces perform controller- or device-specific initialization. These interfaces usually perform the tasks necessary to establish communication with the actual device. Such tasks may include, for a device attach interface, initializing a tape drive or putting a disk drive online. These interfaces initialize any global data structures that are used by the device driver.

At boot time, the autoconfiguration software calls these interfaces under the following conditions:

- If the device is connected to a controller, the *xx*`dattach` interface is called if the controller's `slave` interface returns a nonzero value, indicating that the device exists.
- If the device is not connected to a controller, the *xx*`cattach` interface is called if the `probe` interface returns a nonzero value, indicating that the device exists.

The following code fragment illustrates setup of the *xx*`cattach` interface:

```
xxcattach(ctlr)
struct controller *ctlr; [1]
{
        /* Attach interface goes here. */
        .
        .
        .
        return;
}
```

[1] Declare a a pointer to a `controller` structure for this controller.

The following code fragment illustrates setup of the *xx*`dattach` interface:

```
xxdattach(dev)
struct device *dev;  [1]
{
/* declarations of variables and structures */
   •
   •
   •
/* code to establish communication with the device */
   •
   •
   •
}
```

[1] Declare a pointer to the `device` structure for this device.

3.2.4 Setting Up the Controller Unattach Interface

Use the *xxctlr_unattach* interface to remove a `controller` structure from the list of controllers the device driver handles. This interface cleans up any in-memory data structures and removes any interrupt handlers that may have been established by the device driver. The *xxctlr_unattach* interface is applicable only to loadable drivers. The following code fragment illustrates setup of the *xxctrl_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] Declare a pointer to a `bus` structure and call 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. In this case, `bus` represents the bus that this controller is connected to. 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, a terminal or a network controller).

[2] Declare a pointer to a `controller` structure and call it `ctlr`. This is the `controller` structure you want to remove from the list of controllers handled by the `/dev/cb` device driver.

[3] Declare a *unit* variable and initialize 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.

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, return 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.

5. If *cb_is_dynamic* is equal to the value zero (0), return 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.

6. If the return value from the call to the handler_disable interface is not equal to the value zero (0), return 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.

7. If the return value from the call to the handler_del interface is not equal to the value zero (0), return 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.

   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. Return the value zero (0) to the bus code upon successful completion of the tasks performed by the cb_ctlr_unattach interface.

3.2.5 Setting Up the Device Unattach Interface

Use the *xx*dev_unattach interface to remove a device structure from the list of devices the device driver handles. This interface unloads a device and therefore applies only to loadable drivers.

The following code fragment illustrates setup of the *xx*`dev_unattach` interface:

```
xxdev_unattach (ctlr,dev)

struct controller *ctlr;  [1]
struct device *dev;  [2]
{
    •
    •
    •
}
```

[1] Declare a pointer to a `controller` structure for the controller to which this device is connected.

[2] Declare a pointer to the `device` structure you want to remove from the list of devices handled by the device driver.

3.3 The Configure Section

Loadable drivers contain an *xx*`_configure` interface that is called from the `cfgmgr` interface in response to system manager commands. The *xx*`_configure` interface returns data necessary to load and unload the device driver and responds to requests for configuration information. The following code fragment illustrates setup of the *xx*`_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]
```

[1] Declare 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 (not shown here) and evaluates to one of these valid constants: `SYSCONFIG_CONFIGURE`, `SYSCONFIG_UNCONFIGURE`, and `SYSCONFIG_QUERY`.

[2] Declare 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 Declare an argument called `indatalen` to store the size of this input data structure, in bytes.

4 Declare 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 Declare an argument called `outdatalen` to store the size of this output data structure, in bytes.

Kernel I/O Support Interfaces Used by TURBOchannel Device Drivers 4

This chapter describes when and why you would use the kernel interfaces developed for use with TURBOchannel device drivers. The chapter provides brief examples to illustrate how to use these interfaces in device drivers. For complete descriptions of the definitions and arguments for these kernel interfaces, see Appendix A.

When writing device drivers for the TURBOchannel bus, you need to be familiar with kernel interfaces that:

- Enable or disable a device's interrupt line to the processor
- Determine the base address of a device
- Determine the name of a specific option module
- Isolate and determine the status of a memory error
- Ensure that a write to I/O space completes
- Obtain the page frame number
- Manage memory map registers

Additionally, the I/O subsystems for the DEC 3000 series processors allow a TURBOchannel device driver to transfer data to and from non-contiguous physical memory using a hardware scatter-gather map.

4.1 Enabling or Disabling a Device's Interrupt Line

The kernel automatically enables the device's interrupts after autoconfiguration, depending on what you specified in the `tc_option` data table. However, some devices need interrupts alternately enabled and disabled during autoconfiguration, and the `tc_enable_option` and `tc_disable_option` interfaces are available for this purpose. Note that calling the `tc_enable_option` interface when the device does not require interrupts to be alternately enabled and disabled during autoconfiguration can result in generation of interrupts before the device is prepared to receive them.

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

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

   •
   •
   •
```

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

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

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

4.2 Determining the Name of an Option Module

The `tc_module_name` interface returns the name of a specific TURBOchannel option module. You pass a pointer to a `controller` structure and a character array to be filled in by `tc_module_name`.

The following code fragment illustrates a call to the `tc_module_name` interface:

```
   •
   •
   •
struct controller *ctlr;  1
char cp[TC_ROMNAMLEN + 1];  2
   •
   •
   •
        if (tc_module_name(ctlr, cp) == -1) {  3
```

```
                printf("Module name conversion failed\n");  4
        }
        else {
                printf("Module name is %s\n", cp);  5
        }
```

1 Declare a pointer to the `controller` structure for this controller.

2 Declare a character array to hold the module name. This character array is of size `TC_ROMNAMLEN + 1`, which is large enough to accommodate the module's name.

3 If the call returns `-1`, then name lookup failed.

4 Print an error message if the conversion failed.

5 If the call does not return `-1`, then name lookup was successful; print the module name.

4.3 Determining a Device's Base Address

In the `tc_module_name` interface, you pass a pointer to a `controller` structure to determine the TURBOchannel option's module name. However, the pointer to the `controller` structure is not valid in the driver's *xx*`probe` interface. Therefore, if you need to determine the base address of a device in the *xx*`probe` interface, use the `tc_addr_to_name` interface. The `tc_addr_to_name` interface returns the TURBOchannel option's module name referred to by the base address of the device (the base address of the device is the address passed to the driver's *xx*`probe` interface).

The following code fragment illustrates a call to the `tc_addr_to_name` interface:

```
    •
    •
    •
struct controller *ctlr;  1
char cp[10];  2

        if (tc_addr_to_name(ctlr->addr, cp) == -1) {  3
                printf("Address to Name lookup failed\n");  4
        }
        else {
                printf("Module name is %s\n", cp);  5
        }
    •
    •
    •
```

1 Declare a pointer to the `controller` structure for this controller; the `addr` member of the `controller` structure contains the address of the controller.

[2] Declare a character array to hold the module name.

[3] If the conversion fails, the interface returns -1.

[4] Print an error message if the conversion fails.

[5] On success, print the module name.

4.4 Isolating and Handling Memory Errors

Use the `tc_isolate_memerr` interface to obtain and log information about errors that occur on a direct memory access. Support for this interface is platform specific. The call to `tc_isolate_memerr` returns `-1` if the interface is not supported by the platform. The `tc_isolate_memerr` interface takes a pointer to a structure that contains the following information:

- The physical address of the error
- The virtual address of the error
- A flag to indicate whether to log the error
- The size of the DMA block being transferred
- The error type

  Error types are returned to indicate if a parity (transient, hard, or soft) error occurred or if no error occurred.

The following code fragment for the `/dev/none` device driver illustrates a call to the `tc_isolate_memerr` interface:

```
   •
   •
   •
#include <io/dec/tc/tc.h> [1]
   •
   •
   •

struct tc_memerr_status none_memerr;  [2]
none_memerr_status = &none_memerr;  [3]

none_memerr_status->pa = err_addr;  [4]
none_memerr_status->log = TC_LOG_MEMERR;  [5]

tc_isolate_memerr(none_memerr_status)  [6]
   •
   •
   •
```

[1] Include the header file that defines the `tc_memerr_status` data structure and associated flags.

2 Declare a pointer to the tc_memerr_status structure.

3 Initialize the pointer to the tc_memerr_status structure.

4 Store the error address.

5 Request that the error be logged.

6 Call the tc_isolate_memerr interface, passing to it the pointer to the tc_memerr_status structure.

4.5 Ensuring a Write to I/O Space Completes

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

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

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

The following code fragment illustrates a call to mb:

```
.
.
.
if (reg->csr & ERROR)
{
    return(0);
}
reg->csr=0;
mb(); 1
.
.
.
```

[1] This code fragment shows that if the result of the bitwise AND operation produces a nonzero value (that is, the error bit is set), then the value zero (0) is returned. If the result of the bitwise AND operation is a zero value (that is, the error bit is not set), then the device's control status register is set to zero (0) and the mb interface is called to ensure that a write to I/O space completes. Note that mb takes no arguments.

Note that on Alpha AXP systems, the wbflush interface is aliased to mb.

4.6 Using Scatter-Gather Maps

The I/O subsystems of some CPUs allow a TURBOchannel device driver to transfer data to and from noncontiguous physical memory using a hardware scatter-gather map. Once the scatter-gather map is set up, transfers from noncontiguous physical memory appear to the DMA engine as physically contiguous memory for the duration of the transfer.

Using a scatter-gather map consists of three operations:

- Allocation (reservation) of portions of the scatter-gather map
- Filling the map with the physical address of the data to be transferred
- Deallocation of the map

The interfaces to the scatter-gather map provide a consistent means for device drivers to reserve, use, and release scatter-gather maps and relieves the driver of the burden of finding physical addresses for each page of the transfer. The size or granularity of the transfer is not restricted.

Resource maps allocate and deallocate the map entries. The resource maps also manage fragmentation and compaction of the list of free map ram entries as they are allocated and deallocted by drivers; the drivers are not responsible for maintaining this information.

As an aid in debugging device drivers, the resource map code provides a warning if the scatter-gather map becomes overly fragmented and panics the system if the allocation and deallocation interfaces are called with inconsistent byte counts.

Before calling interfaces to a scatter-gather map, you may want to test the cpu variable to determine if the CPU type is one with map registers, for example:

```
if cpu != <type-value-list>

       <perform task that does not use map registers>

else

      <call tc_map_alloc or tc_loadmap interface>
```

For Alpha AXP CPUs, the `cpu` variable and associated CPU type values are defined in the file `/usr/sys/include/arch/alpha/hal/cpuconf.h`. Check your hardware documentation to determine which Alpha AXP CPUs have map registers.

4.6.1 Preallocating a Scatter-Gather Map

To use a scatter-gather map, a TURBOchannel device driver must allocate one map entry per page of noncontiguous physical memory to be transferred. You need to decide whether your device driver allocates map entries each time the memory map is loaded for a transfer operation or preallocates map entries for use during multiple transfers. Preallocation of map entries is most appropriate for drivers that handle small transfers that occur continuously. When a device driver preallocates map entries, the corresponding CPU map registers are not available to other device drivers.

Use the `tc_map_alloc` interface to preallocate the map entries. The following code fragment illustrates a call to this interface:

```
struct buf *bp;
unsigned long addr;

     if( (addr = tc_map_alloc(bp->b_bcount, TC_MAP_SLEEP)) == -1) [1]
             {
             printf("tc_map_alloc failed\n");
             return;
             }
    •
    •
    •
```

[1] This line shows that the `tc_map_alloc` interface takes two arguments. The first argument specifies the number of bytes to be transferred. In this example, the number of bytes to be transferred is contained in the `b_bcount` member of the pointer to the `buf` structure. This member is initialized by the kernel as the result of an I/O request. The example references this member to determine the size of the I/O request.

The second argument specifies one or more flags. In this example, the `TC_MAP_SLEEP` flag is specified to force the `tc_map_alloc` interface to sleep if the requested map entries cannot be allocated immediately.

The code fragment sets up a condition statement to test if the call to `tc_map_alloc` fails (returns the value `-1`); on failure, the actions taken are to:

- Print an error message
- Return to the calling interface

4.6.2 Mapping the Transfer

Prior to loading an address and starting the DMA engine, a TURBOchannel driver calls the `tc_loadmap` interface. If the device driver does not pass this interface the value returned by a preceding call to `tc_map_alloc`, the `tc_loadmap` interface first calls `tc_map_alloc` to allocate map entries. After map entries are allocated, `tc_loadmap` walks through each page of a region of virtual memory to be transferred and fills scatter-gather map entries with the physical addresses of those pages. The `tc_loadmap` interface returns a physical address that is loaded into a DMA engine. This address appears to the DMA engine as a contiguous region of physical memory. The starting virtual address has no alignment restrictions and the physical addresses need not be in contiguous pages. There is no restriction on the size of the transfer.

The process that calls `tc_loadmap` must ensure that the data to be transferred is locked in memory at the time of the call and for the duration of the DMA. The calling driver must also manipulate the physical address returned by `tc_loadmap` so that it conforms with the DMA hardware being used.

The following code fragment illustrates a call to the `tc_loadmap` interface:

```
struct buf *bp;
unsigned long addr;
long base = 0;
   •
   •
   •
      if ( (addr = tc_loadmap(bp->b_proc, &base, bp->b_un.b_addr,
              bp->b_bcount, (TC_MAP_SLEEP | TC_MAP_FAILSOFT)) ) == -1) [1]
              {
              printf("tc_loadmap failed\n");
              return;
              }
   •
   •
   •
```

[1] This line shows that the `tc_loadmap` interface takes five arguments:

- The first argument specifies a pointer to the `proc` structure that represents the process on whose behalf the data is being transferred. In the example, the `proc` pointer is obtained by referencing the `b_proc` member of the pointer to the `buf` structure. The `proc` pointer is required to get the proper physical addresses into the map registers.
- The second argument specifies either a pointer to the address of the variable that contains the value returned in a previous call to `tc_map_alloc` or the value zero to indicate that map entries are to be dynamically allocated. In this example, the value contained in the `base` variable is zero (`0`), indicating that map registers need to be

allocated for this transfer.

- The third argument specifies the starting virtual address of the data to be transferred. In this example, the address is obtained by referencing the b_un.b_addr member of the pointer to the buf structure. This member is set by the kernel and is the main memory address where the I/O operation occurs.
- The fourth argument specifies the number of bytes to be transferred. In this example, the number of bytes to be transferred is contained in the b_bcount member of the pointer to the buf structure.
- The fifth argument specifies one or more flags. In this example, the TC_MAP_SLEEP flag forces the tc_loadmap interface to sleep until contiguous map entries are available for the transfer and the TC_MAP_FAILSOFT flag prevents a system panic if the call cannot obtain map entries for the transfer.

If the tc_loadmap interface fails (returns the value -1), the actions taken are to:

- Print an error message
- Return to the calling interface

4.6.3 Freeing Map Entries

When a DMA completes, a TURBOchannel device driver can return the map entries to the pool using the tc_map_free interface. The following code fragment illustrates a call to this interface:

```
struct buf *bp;
unsigned long addr;
   •
   •
   •
addr = tc_loadmap(...);
   •
   •
   •
tc_map_free(addr, bp->b_bcount, TC_MAP_INVAL); [1]
   •
   •
   •
```

[1] This line shows that the tc_map_free interface takes three arguments:

- The first argument specifies the value returned by the tc_loadmap interface. This value is the physical address that was passed to the DMA hardware.
- The second argument specifies the amount of transferred data, in bytes, as specified in a previous call to tc_loadmap. In this example, the number of bytes is contained in the b_bcount member of the pointer to the buf structure.

- The third argument specifies one or more flags. In this example, the call specifies the TC_MAP_INVAL flag so that the map entries used for the transfer are zeroed and marked as invalid before being returned to the pool of free map entries. This action prevents memory corruption in the event the DMA engine attempts to use one of the map entries.

Device Autoconfiguration 5

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

- The name of the I/O module as it appears in read-only memory (ROM) on the device
- The name of the controller or device attached to the TURBOchannel bus
- The TURBOchannel I/O slot number
- The number of slots occupied by the I/O module
- A pointer to the `interrupt` interface
- The unit number of the device
- The base physical address of the device
- The class of the I/O module (device or controller)
- Parameters to determine when interrupts are enabled
- A pointer to the adapter configuration interface
- Additional parameters and flags

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

- Probe TURBOchannel option slots
- Obtain the I/O module's name
- Map TURBOchannel slot numbers
- Register the driver's interrupt handler
- Determine when interrupts are enabled

Following the discussion of these tasks, there is a brief discussion of the `tc_option` table in the file `/usr/sys/data/tc_option_data.c`.

5.1 Probing TURBOchannel Option Slots

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

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

5.1.1 Obtaining the I/O Module's Name

The way in which the operating system obtains the I/O module name differs, depending on whether driver configuration is static or dynamic.

5.1.1.1 Static Configuration

After probing the TURBOchannel I/O slots, the operating system looks up the module name for the device's controller in the `tc_option` data table. This table maps TURBOchannel module names to names as they are specified in the system configuration file. The `tc_option` table contains a structure entry for each of the TURBOchannel I/O options on the system. The following example shows the entry for the `/dev/cb` device driver in the system configuration file:

```
controller          cb0          at tc0        vector cbintr
```

The following example shows the corresponding entry for the driver in the `tc_option` data table in the `/usr/sys/data/tc_option_data.c` file:

```
struct tc_option tc_option [] =
{
   /*  module           driver  intr_b4 itr_aft       adpt    */
   /*  name             name    probe   attach  type  config  */
       ------           ------  ------- ------- ----  ------
   .
   .
   .
   {   "CB      ",      "cb",      0,      1,    'C',   0},     */ CB */
   .
   .
   .
   /*
    * Do not delete this null entry, which terminates the table or your
    * system will not configure properly.
    */
   {   "",              ""      }      /* Null terminator in the table */
};
```

The operating system compares the device names found in the I/O slots and the `tc_option` table (optional as well as fixed devices) with the names in the system configuration file. These device names appear in the `ctlr_list` table (an array of `controller` structures). Each entry in the system configuration file specifies the interrupt interface for the device. In the previous example, the interrupt interface is called `cbintr`.

The name of the interrupt interface is placed in the `ctlr_list` table by the `config` program.

It is recommended that you do not directly edit the `tc_option_data.c` file to add entries for third-party device drivers. Instead, you can provide data for entries in a `config.file` file fragment that you include in your device driver installation kit. Refer to *Writing Device Drivers, Volume 1: Tutorial* for more information about the `config.file` file fragment.

5.1.1.2 Dynamic Configuration

When device drivers are dynamically loaded, the operating system retrieves TURBOchannel option data from the `/etc/sysconfigtab` database rather than the system configuration file. The following example shows the entry for the `/dev/cb` device driver in the `/etc/sysconfigtab` database:

```
Module_Config2 = controller   cb0     at       tc0
```

The name of the interrupt handler is not part of a device driver entry in the `/etc/sysconfigtab` database; for loadable drivers, the interrupt interface is registered through the device driver's `xxprobe` interface.

For your device driver to be loadable, you must supply configuration data in the declarations and definitions section of your driver. The following example illustrates the code in the declarations and definitions section for the loadable implementation of the `/dev/cb` device driver:

```
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 */
};
```

The configuration information contained in `cb_option_snippet` does not have to be added directly in the `tc_option` data table. For loadable drivers, the operating system configures the device using the information in the `tc_option` table along with the information in `cb_option_snippet`. The `cb_option_snippet` structure is parsed as an argument to the `ldbl_stanza_resolver` interface to configure the device.

5.1.2 Mapping TURBOchannel Slot Numbers

If the operating system matches a device name in the `tc_option` table with a device name in the system configuration file, the system puts an entry in the `tc_slot` table. During the configuration of a loadable driver, the operating system uses the device name specified in the bus-specific parameter provided to the `ldbl_stanza_resolver` interface. For the TURBOchannel bus, the bus-specific parameter is the `tc_option` snippet table (the `cb_option_snippet` structure in the example in Section 5.1.1.2).

If the operating system matches a module name in a module ROM that is not in the `tc_option` data table (static configuration) or the `tc_option` snippet table (dynamic configuration), then the system warns that the device is unknown.

At system configuration time, the operating system does not configure any device whose name is not in the system configuration file. That is, the operating system does not call the driver's `probe`, `slave`, or `attach` interface and disables the device's interrupt line if the device is specified only in the `tc_option` table. When drivers are configured dynamically, the operating system calls the driver's `probe`, `slave`, or `attach` interface and enables device interrupt lines when the driver is loaded.

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

The operating system handles adapters in a way similiar to the way it handles devices and controllers. Adapters have an adapter line in the system configuration file, with no `interrupt` interface. The operating system configuration code looks up the adapter module name in the `tc_option` data table and obtains the name of the adapter configuration interface to call. One of the arguments passed to the adapter configuration interface is an address where that configuration interface places the address of the interrupt handling interface.

TURBOchannel Device Driver Configuration 6

Writing Device Drivers, Volume 1: Tutorial describes the device driver configuration models provided by the DEC OSF/1 operating system. The third-party device driver configuration model is recommended for third-party device driver writers who want to ship loadable and static drivers to customers whose systems run DEC OSF/1. In the third-party configuration model, you supply information in the form of file fragments that are included in the device driver product kit. These file fragments are input to automated configuration tools, which combine the information supplied by a file fragment with information supplied by other files of the same type. The traditional device driver configuration model, in which you directly edit system files to add driver information, is suitable for driver writers during the initial stages of driver development. See *Writing Device Drivers, Volume 1: Tutorial* for:

- A detailed comparison of the third-party and traditional device driver configuration models
- Descriptions of the syntaxes and mechanisms used to populate device driver configuration-related files

This chapter assumes that you are familiar with the device driver configuration models, the syntaxes used to populate configuration-related files, and the steps performed to configure loadable and static device drivers on the TURBOchannel bus. Discussion in this chapter is limited to the following topics related to configuring device drivers on the TURBOchannel bus:

- Using a `tc_data` file with the `mktcdata` utility

  This topic applies to drivers that are statically configured in accordance with the third-party configuration model.

- Creating a `tc_option` snippet table

  This topic applies to drivers that are dynamically configured in accordance with either the traditional or the third-party configuration model.

6.1 Using a tc_data File with the mktcdata Utility

When drivers are statically configured, driver information must exist not only in the system configuration file or associated `config.file` fragment, but also in the `tc_option` data table. This table is defined in the `tc_option_data.c` file. When using the third-party configuration model, you do not directly edit the `tc_option_data.c` file to add the information for your device. You provide such information in a `tc_data` file that resides in the driver product kit area. During autoconfiguration, the `mktcdata` utility copies the `tc_option_data.c` file from the `/usr/sys/data` directory and adds entries for all registered third-party drivers that include `tc_data` files in their driver kit areas. The format of the tc_data file is as follows:

```
#Entry
ROM_ID=module_name
DRV_NAME=driver_name
[ INTR_B4=intr_b4_probe ]
[ INTR_AFT=intr_aft_attach ]
[ TYPE=device_type ]
[ ADPT_CNFG=config_interface]
```

`#Entry`
: Specifies the start of an entry in the tc_data file.

module_name
: Specifies the device name in the ROM on the hardware device. This name can be one to eight characters, and must be unique among names of devices that are configured for the TURBOchannel bus.

driver_name
: Specifies the name of the driver as it appears in the system configuration file. This name must be unique among names of drivers that are configured for the TURBOchannel bus.

intr_b4_probe
: Specifies whether the device needs interrupts enabled before execution of the driver's *xxprobe* interface. A zero (`0`) value indicates that the device does not need interrupts enabled; this is the default value. A value of `1` indicates that the device needs interrupts enabled.

intr_aft_attach
: Specifies whether the device needs interrupts enabled after the driver's *xxprobe* and *attach* interfaces complete execution. A zero (`0`) value indicates that the device does not need interrupts enabled; this is the default value. A value of `1` indicates that the device needs interrupts enabled.

device_type
: Specifies the type of device, either: `C` (controller) or `A` (adapter). The default value is `C`.

config_interface
: Specifies the name of the interface to configure the adapter if *device_type* is `A` (adapter). If *device_type* is `C` (controller) do not specify a *config_interface* entry in the `tc_data` file.

You can specify lines in the `tc_data` file entry in any order; however, do not include space characters within the line. Make sure each entry in the file begins with the line `#Entry` and is separated from other entries with a blank line.

The following example illustrates a section of the `tc_data` file with an entry for the `/dev/cb` driver:

```
#Entry
ROM_ID=CB
DRV_NAME=cb
INTR_B4=0
INTR_AFT=1
TYPE=C
ADPT_CNFG=0

#Entry
    •
    •
    •
```

Each element in the entry initializes the corresponding member of the `tc_option` data structure. The `mktcdata` utility is automatically invoked by the `config` program through a `callout` entry contained in the system configuration file. The `mktcdata` utility adds information to the `tc_option` table from the driver's `tc_data` file. The `config` program resumes when the `mktcdata` subprocess completes.

Refer to *Writing Device Drivers, Volume 1: Tutorial* for more detailed discussion of `callout` entries.

6.2 Creating a tc_option Table Snippet

When drivers are configured statically, data for the `tc_option` table is provided through the `tc_data` file. For loadable drivers, data to be added to the `tc_option` table is provided by a snippet table in the configuration support declarations and definitions section of the device driver. *Writing Device Drivers, Volume 1: Tutorial* illustrates how to create a snippet table; however, an example is repeated here for easy comparison with the `tc_data` file described in Section 6.1:

```
   •
   •
   •
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 */ };
   •
   •
   •
```

When the driver is loaded, the address of the `tc_option` snippet table is passed to the `ldbl_stanza_resolver` interface.

TURBOchannel-Specific Reference Information A

This appendix provides reference information for the header file, data structures, and kernel I/O support interfaces that are specific to the TURBOchannel bus. Refer to *Writing Device Drivers, Volume 2: Reference* for reference information on header files, data structures, and interfaces that are not bus-specific.

A.1 Header File Used Only by TURBOchannel Device Drivers

The `/usr/sys/include/io/dec/tc/tc.h` file is the only header file used exclusively by TURBOchannel device drivers.

Name

tc.h — Defines TURBOchannel I/O and option slots, data structures, and kernel I/O support interfaces

Location

/usr/sys/include/io/dec/tc/tc.h

Description

The tc.h file contains definitions for TURBOchannel-specific data structures and interfaces. It also contains definitions for flags and other parameters used in those interfaces.

When to Include

All TURBOchannel device drivers contain the tc.h file in the Include Files section of the driver.

Of Special Interest

Items of interest to device driver writers are:

- The tc_info, tc_intr_info, tc_memerr_status, and tc_option structures
- Flag definitions for enabling and disabling option modules and error logging
- Status definitions for memory error types
- Definitions associated with TURBOchannel map registers
- Interfaces used by drivers that operate on the TURBOchannel bus

Related Information

Section A.3: tc_addr_to_name, tc_disable_option, tc_enable_option, tc_isolate_memerr, tc_module_name, and tc_option_control

A.2 Data Structures Used Only by TURBOchannel Device Drivers

Table A-1 summarizes TURBOchannel-specific data structures.

Table A-1: Summary of Data Structures for TURBOchannel Device Drivers

Data Structure	Summary Description
`tc_info`	Contains information passed by the `confl1` and `confl2` interfaces to the configuration interface of any bus connected to the TURBOchannel bus.
`tc_intr_info`	Contains information associated with interrupt handlers for loadable device drivers.
`tc_memerr_status`	Contains information associated with errors that occur on a direct memory access.
`tc_option`	Contains the driver's module name, configuration name, type, and interrupt handling attributes that are used in the `tc_option` slot table.

Name

tc_info — Contains bus information that is passed to the configuration interface of any bus connected to the TURBOchannel bus

Include File

/usr/sys/include/io/dec/tc/tc.h

Synopsis

Member Name	Data Type
addr	caddr_t
physaddr	caddr_t
slot	int
unit	int
intr	int (**intr)()
bus_hd	struct bus *

Members

addr
: Specifies the virtual address of the slot containing the bus adapter.

physaddr
: Specifies the physical address of the slot containing the bus adapter.

slot
: Specifies the unit number of the bus adapter.

unit
: Specifies the logical unit number of the controller as specified in the system configuration file or the `config.file` file fragment (for drivers that are statically configured) or the `stanza.loadable` file fragment (for drivers that are dynamically configured).

intr
: Specifies an array that contains the interrupt interfaces for the bus adapter.

bus_hd
: Specifies a pointer to the bus structure for this TURBOchannel bus.

Description

The `tc_info` data structure contains bus information that the `confl1` and `confl2` interfaces pass to the configuration interface of any bus connected to the TURBOchannel bus.

Related Information

Writing Device Drivers, Volume 2: Reference: `bus` data structure

Name

tc_intr_info — Contains information associated with interrupt handlers for loadable device drivers

Include File

/usr/sys/include/io/dec/tc/tc.h

Synopsis

Member Name	Data Type
configuration_st	caddr_t
intr	int (*intr)()
param	caddr_t
config_type	unsigned int

Members

configuration_st
: Specifies a pointer to the bus or controller structure for which an associated interrupt handler is written.

intr
: Specifies a pointer to the interrupt handler for the specified bus or controller.

param
: Specifies a member whose content is passed to the interrupt service interface.

config_type
: Specifies the driver type.

  You can set this member to one of these constants defined in /usr/sys/include/io/dec/tc/tc.h: TC_CTLR (controller), TC_ADPT (bus), TC_DEV (device).

Description

The `tc_intr_info` data structure contains interrupt handler information for device controllers that are connected to the TURBOchannel bus. Loadable drivers initialize the members of the `tc_intr_info` structure, usually in the driver's *`xx`*`probe` interface.

Related Information

Section A.4: *`xx`*`probe`

Name

tc_memerr_status — Contains information associated with logging memory errors on the TURBOchannel bus

Include File

/usr/sys/include/io/dec/tc/tc.h

Synopsis

Member Name	Data Type
pa	caddr_t
va	caddr_t
log	int
blocksize	int
errtype	u_int

Members

pa
: Specifies the physical address of the error.

va
: Specifies the virtual address of the error.

log
: Specifies whether to log the error, using one of the following flags:

Flag	Meaning
TC_NOLOG_MEMERR	Do not log the error.
TC_LOG_MEMERR	Log the error.

blocksize
: Specifies the size of the DMA block.

errtype
: Specifies one of the following symbolic values:

Error	Meaning
TC_MEMERR_NOERROR	No error occurred.
TC_MEMERR_TRANS	A transient memory error occurred.
TC_MEMERR_SOFT	A soft parity error occurred.
TC_MEMERR_HARD	A hard parity error occurred.

Description

The `tc_memerr_status` data structure is used by the `tc_isolate_memerr` interface to obtain error information associated with devices operating on the TURBOchannel bus.

Related Information

Section A.3: `tc_isolate_memerr`

Name

tc_option — Contains information used with option data files for device drivers written for the TURBOchannel bus

Include File

/usr/sys/include/io/dec/tc/tc.h

Synopsis

Member Name	Data Type
modname	char [TC_ROMNAMLEN + 1]
confname	char [TC_ROMNAMLEN + 1]
intr_b4_probe	int
intr_aft_attach	int
type	char
adpt_config	int(*adpt_config)()

Members

`modname`
: Specifies the name of the option module.

`confname`
: Specifies the device or controller name as specified in the system configuration file or the `config.file` fragment (for statically configured drivers) or the `stanza.loadable` file (for dynamically configured drivers).

`intr_b4_probe`
: Specifies whether interrupts are enabled before the *xx*`probe` interface is called.

`intr_aft_attach`
: Specifies whether interrupts are enabled after the `attach` interface is called.

type

Specifies the module type using one of the following values:

Value	Meaning
A	Adapter
C	Controller

adpt_config

Specifies the adapter configuration interface to be called.

Description

The tc_option data structure contains information used with the tc_option_data.c file and associated tc_data file fragments that are included in device driver product kits.

Related Information

Chapter 5

A.3 Kernel Support Interfaces Used Only by TURBOchannel Device Drivers

Table A-2 summarizes the kernel interfaces for use in writing TURBOchannel device drivers.

Table A-2: Summary of Kernel Support Interfaces for TURBOchannel Device Drivers

Kernel Interface	Summary Description
`tc_addr_to_name`	Returns the option module name based on the address passed to the `xxprobe` interface.
`tc_disable_option`	Disables a device's interrupt line to the processor.
`tc_enable_option`	Enables a device's interrupt line to the processor.
`tc_isolate_memerr`	Returns information about memory errors.
`tc_loadmap`	Returns a physical address for scatter-gather map entries to be loaded into a DMA engine.
`tc_map_alloc`	Allocates TURBOchannel map registers.
`tc_map_free`	Returns map entries to the pool after completion of a DMA.
`tc_module_name`	Determines the name of a specific option module.
`tc_option_control`	Enables and disables various bus options.

Name

`tc_addr_to_name` — Determines the base address of a device

Synopsis

```
int tc_addr_to_name(addr, cp)
vm_offset_t addr;
char *cp;
```

Arguments

addr

Specifies the address (the base address of the device) that is passed to the device driver's *xx*`probe` interface.

cp

Specifies a pointer to a character array to be filled in with the option module name. You must declare the *cp* array to be of the size *cp*`[TC_ROMNAMLEN + 1]` to ensure that the character array is large enough to store the module name.

Description

The `tc_addr_to_name` interface fills in the character array *cp* with the ASCII string of the TURBOchannel option's module name referred to by the base address *addr*. In the `tc_module_name` interface, you pass a pointer to a `controller` structure, which is not valid in the driver's *xx*`probe` interface. Therefore, use `tc_addr_to_name` rather than `tc_module_name` in a driver's *xx*`probe` interface.

Return Value

This interface returns a value of `-1` if it is unable to use the *cp* pointer you passed. The interface returns zero (`0`) on success.

Related Information

`tc_module_name`

Name

`tc_disable_option` — Disables a device's interrupt line to the processor

Synopsis

```
void tc_disable_option(ctlr)
struct controller *ctlr;
```

Arguments

ctlr
Specifies a pointer to a `controller` structure.

Description

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

Return Value

None.

Related Information

`tc_enable_option`

Name

`tc_enable_option` — Enables a device's interrupt line to the processor

Synopsis

```
void tc_enable_option(ctlr)
struct controller *ctlr;
```

Arguments

ctlr
: Specifies a pointer to a `controller` structure.

Description

The `tc_enable_option` interface enables a device's interrupt line to the processor. Use this interface only if the device must have its interrupts alternately enabled and disabled during autoconfiguration. Otherwise, interrupts may be generated before the device is ready to receive them.

The kernel automatically enables the device's interrupts after autoconfiguration, depending on what you specified in the `tc_option` data table.

Return Value

None.

Related Information

`tc_disable_option`

Name

`tc_isolate_memerr` — Logs memory errors associated with devices operating on the TURBOchannel bus

Synopsis

```
int tc_isolate_memerr(memerr_status)
struct tc_memerr_status *memerr_status;
```

Arguments

memerr_status
: Specifies a pointer to a `tc_memerr_status` data structure. This data structure contains such information as the physical address of the error, a flag to indicate whether to log the error, and so forth.

Description

The `tc_isolate_memerr` interface logs memory-related errors associated with devices that operate on the TURBOchannel bus. The mechanism for logging these errors is the `tc_memerr_status` data structure.

The `tc_isolate_memerr` interface checks the virtual address of the error (the `va` member of the `tc_memerr_status` structure). If the address equals the value zero, the interface uses the physical address (the `pa` member of the `tc_memerr_status` structure) to form a virtual address.

The `tc_isolate_memerr` interface calls a system-specific interface, which determines the exact error based on the virtual address and the physical address. If the `log` member of the `tc_memerr_status` structure is set to `TC_LOG_MEMERR`, the system-specific interface logs the error in the same manner a memory error is logged when it comes directly into the CPU.

Return Value

The `tc_isolate_memerr` interface returns the value `-1` if the supplied physical address is invalid or if the system-specific interface does not exist.

Related Information

Section A.4: `tc_memerr_status`

Name

`tc_loadmap` — Fills scatter-gather map entries with the physical addresses of pages to be transferred in a DMA operation

Synopsis

```
unsigned long tc_loadmap(proc, base, addr, count,
flags)
struct proc *proc;
long *base;
vm_offset_t addr;
int count;
int flags;
```

Arguments

proc
: Specifies a pointer to the `proc` structure that represents the process on whose behalf the DMA is being done. The *proc* pointer is required to get the proper physical addresses into the map registers.

base
: Specifies either a pointer to the address of the variable containing the value returned by a prior call to the `tc_map_alloc` interface or zero to indicate that map entries are to be dynamically allocated. If *base* is zero, then the `tc_loadmap` interface calls `tc_map_alloc` to allocate map entries. The `tc_map_alloc` interface fills in *base* with the starting index of the allocated map ram. Set either the `TC_MAP_FAILSOFT` or `TC_MAP_SLEEP` flag to prevent a system panic in the event that the call to `tc_map_alloc` fails.

addr
: Specifies the starting virtual address of the data to be transferred. This address may be any user or kernel virtual address. Alignment of the address is not important. However, any offset from the start of a page is reflected in the value returned by the `tc_loadmap` interface.

count
: Specifies the number of bytes to be transferred. The number you supply does not have to be an integral multiple of anything, but the hardware cannot map any quantity smaller than one page.

flags
: Specifies one or more of the following flags:

Flag	Meaning
TC_MAP_SCRATCH	Allocates (on the call to tc_map_alloc) an extra map entry to appear between the entry for the last map for the actual transfer and the guard map entry. The extra map entry points to an unused (scratch) page of physical memory and is necessary if the DMA hardware might overrun the actual transfer on a write operation to a device.
TC_MAP_SLEEP	Forces the tc_map_alloc interface to sleep until sufficient contiguous map entries are available for the transfer. This flag is ignored if the *base* parameter is a nonzero value.
TC_MAP_FAILSOFT	Prevents a system panic in the event the tc_loadmap interface calls tc_map_alloc and tc_map_alloc cannot get map entries for the transfer. This flag is ignored if the *base* parameter is a nonzero value or the TC_MAP_SLEEP flag is set.

Description

The tc_loadmap interface fills scatter-gather map entries with the physical addresses of the pages to be transferred during direct memory access (DMA). Before providing a device an address for DMA transfers, a TURBOchannel device driver calls the tc_loadmap interface, which returns a physical address to be passed directly to DMA hardware.

Return Value

The tc_loadmap interface returns -1 (failure) under the following conditions:

- The CPU type does not have map registers to support memory mapping
- The *base* parameter is zero, neither the TC_MAP_SLEEP nor the TC_MAP_FAILSOFT flag is set, and a call to tc_map_alloc fails.

All other conditions return a physical address.

Related Information

`tc_map_alloc` and `tc_map_free`

Name

`tc_map_alloc` — Allocates TURBOchannel map registers

Synopsis

```
unsigned long tc_map_alloc(count, flags)
int count;
int flags;
```

Arguments

count

Specifies the number of bytes to be transferred. The number of bytes you specify is divided by the page size to obtain the number of map entries required for the transfer. The `tc_map_alloc` interface allocates map entries using the formula *count* + 2. The two additional entries provide for a possible nonpage aligned transfer and a guard entry. The guard entry ensures that a DMA overrun is caught and flagged by the hardware rather than causing data corruption.

flags

Specifies one or more of the following flags:

Flag	Meaning
TC_MAP_SCRATCH	Allocates one more map entry in addition to the ones specified by the basic formula (*count* + 2). Use this flag if the DMA hardware might overrun the actual transfer on a write operation to a device. The `tc_map_alloc` interface points the extra map entry to a scratch page of memory.
TC_MAP_SLEEP	Forces the `tc_map_alloc` interface to sleep if the requested map entries cannot be allocated immediately. The interface wakes and attempts the allocation again, following a call to the `tc_map_free` interface. If you do not specify the TC_MAP_SLEEP flag, `tc_map_free` immediately returns an error (value -1) if the requested map entries cannot be allocated.

Description

The tc_map_alloc interface is called by device drivers that use a scatter-gather map to transfer data to and from noncontiguous pages in memory. Use the interface to allocate one scatter-gather map entry per page of noncontiguous physical memory to be transferred. A call to the tc_map_alloc interface returns a nonzero value that represents the starting index of the allocated map ram, including the guard and scratch entries. A later call to the tc_loadmap interface uses the value from tc_map_alloc to fill in scatter-gather maps.

Return Value

The tc_map_alloc interface returns -1 (failure) under the following conditions:

- *count* is a negative value
- The CPU does not support memory mapping (does not have map registers)
- The map entries cannot be allocated immediately and TC_MAP_SLEEP is not set

Otherwise, the interface returns a nonzero value that represents the starting index of the map ram.

Related Information

tc_loadmap and tc_map_free

Name

`tc_map_free` — Returns map entries to the memory pool after completion of a DMA

Synopsis

```
int tc_map_free(base, count, flags)
long base;
int count;
int flags;
```

Arguments

base
: Specifies the value returned by the `tc_map_alloc` or `tc_loadmap` interface.

count
: Specifies the number of transferred bytes of data as specified in the call to the `tc_map_alloc` or `tc_loadmap` interface.

flags
: Specifies one or more of the following flags:

Flag	Meaning
`TC_MAP_INVAL`	Zeroes and marks each map entry used for the transfer as invalid before returning the map entry to the pool of free map entries. Thereafter, any attempt by a DMA engine to use one of these map entries causes a hardware exception rather than possibly corrupting physical memory. If this flag is not set, map entries are returned to the free pool without further action.

Flag	Meaning
TC_MAP_SCRATCH	Zeroes and marks as invalid the map entry that was allocated in addition to those specified by the basic map entry allocation formula (count + 2). Use this flag if you specified the TC_MAP_SCRATCH flag on the call to the tc_map_alloc or tc_loadmap interface.

Description

The tc_map_free interface returns allocated map entries to the free pool after a direct memory access (DMA) completes. The map entries were allocated by the tc_map_alloc interface, called either directly by the driver writer or indirectly through a call to the tc_loadmap interface.

Return Value

The tc_map_free interface returns zero for success and -1 (failure) for the following conditions:

- The *count* argument is a negative value.
- The CPU type does not have map registers.

Related Information

tc_map_alloc and tc_loadmap

Name

`tc_module_name` — Determines the name of a specific option module

Synopsis

```
int tc_module_name(ctlr, cp)
struct controller *ctlr;
char *cp;
```

Arguments

ctlr
Specifies a pointer to a `controller` structure.

cp
Specifies a pointer to a character array to be filled in by `tc_module_name`. You must declare the *cp* array to be of the size *cp*`[TC_ROMNAMLEN + 1]`.

Description

The `tc_module_name` interface fills in the character array *cp* with the ASCII string of the TURBOchannel option's module name referred to by the pointer to the `controller` structure.

Return Value

The `tc_module_name` interface returns a value of `-1` if it was unable to use the *cp* pointer you passed.

Related Information

`tc_addr_to_name`

Name

`tc_option_control` — Enables and disables various bus options

Synopsis

```
int tc_option_control(ctrl, flags)
struct controller *ctlr;
int flags;
```

Arguments

ctlr

Specifies a pointer to a `controller` structure.

flags

Specifies the options to be enabled or requests option status. The following table lists and describes the flags:

Flag	Meaning
SLOT_PARITY	Enable or disable parity for the specified option.
SLOT_BLOCKMODE	Enable or disable block mode transfers for the specified option.
SLOT_MAPREGS	Enable or disable use of map registers for address translation for the specified module.
SLOT_STATUS	Request current state of preceding flags; do not change current settings.

Description

The `tc_option_control` interface enables or disables various slot-specific options. Use the `SLOT_PARITY`, `SLOT_BLOCKMODE`, and `SLOT_MAPREGS` flags to change option settings. A call to the `tc_option_control` interface returns the state of the option settings after execution. To find out what the option settings are without changing any of the settings, use the `SLOT_STATUS` flag.

Return Value

The return value of this interface is the current state of *`flags`* after completion of the operations requested by the call to `tc_option_control`.

A.4 Device Driver Interfaces Used Only by TURBOchannel Device Drivers

Table A-3 summarizes the device driver interfaces that have formal parameters specifically defined for the TURBOchannel bus. Refer to *Writing Device Drivers, Volume 2: Reference* for information on other driver interfaces.

Table A-3: Summary of Device Driver Interfaces for TURBOchannel Device Drivers

Interface	Summary Description
xx`cattach` or *xx*`dattach`	Performs controller- or device-specific initialization.
xx`probe`	Determines whether the device is functional on a given system.
xx`slave`	Checks to ensure that the device is valid for the controller.

Name

*xx*cattach , *xx*dattach — Performs controller- or device-specific initialization

Entry Point

The driver structure

Synopsis

```
void xxcattach(ctlr)
struct controller *ctlr;

void xxdattach(device)
struct device *device;
```

Arguments

ctlr
: Specifies a pointer to the controller structure for this controller. This structure contains such information as the controller type, the controller name, and the current status of the controller.

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

Description

The *xx*cattach and *xx*dattach interfaces usually perform the tasks necessary in establishing communication with the actual device. These tasks might include, for a device attach interface, initializing a tape drive, putting a disk drive online, or some similar task. In addition, *xx*cattach and *xx*dattach initialize any global data structures used by the device driver.

For statically configured drivers, these interfaces are called at system boot time by the autoconfiguration software under the following conditions:

- If the device is connected to a controller, the *xx*dattach interface is called if the controller's slave interface returns a nonzero value, indicating that the device exists.

- If the device is not connected to a controller, the *xx*cattach interface is called if the *xx*probe interface returns a nonzero value, indicating that the device exists.

If you set the cattach or dattach member of the driver structure to NULL, no call is made to the *xx*cattach or *xx*dattach interface. The *xx*cattach interface is passed a controller structure and the *xx*dattach interface is passed a device structure for this device.

For loadable drivers, the *xx*cattach or *xx*dattach interface is called indirectly when the driver is loaded.

Return Value

None.

Related Information

Writing Device Drivers, Volume 2: Reference: controller and device structures

Name

`xxprobe` — Determines whether the device exists

Entry Point

The `driver` structure

Synopsis

```
int xxprobe(addr, ctlr)
caddr_t addr;
struct controller *ctlr;
```

Arguments

addr

Specifies the System Virtual Address (SVA) of the base of the TURBOchannel slot space for this controller.

ctlr

Specifies a pointer to the `controller` structure for this controller. This structure contains such information as the controller type, the controller name, and the current status of the controller.

Description

A device driver's `xxprobe` interface performs tasks necessary to determine if the device exists and is functional on a given system. At boot time, the kernel performs checks to determine if the device is present before calling the `xxprobe` interface for statically configured drivers. The kernel calls the `xxprobe` interface for each device that was defined in the system configuration file or `config.file` file fragment (for statically configured drivers) or the `stanza.loadable` file fragment (for dynamically configured drivers). Likewise, the kernel calls the `xxprobe` interface for each stanza entry that was defined in the `stanza.loadable` file fragment for loadable drivers.

For loadable drivers, the `xxprobe` interface is called indirectly during the driver loading process. The driver writer specifies loadable driver configuration information in the `stanza.loadable` file fragment. This information includes the driver's name, location of the loadable object, device connectivity information, device special file information, and so forth.

When the system manager requests that the driver be dynamically loaded, the system accesses the information in the `stanza.loadable` file fragment.

The *xxprobe* interface calls the `ldbl_stanza_resolver` interface to merge the driver's connectivity information into the hardware topology tree, which consists of `bus`, `controller`, and `device` structures. Next, the `ldbl_ctlr_configure` interface is called, which results in the system calling *xxprobe* for each instance of the controller present on the TURBOchannel bus.

Some tasks performed by the *xxprobe* interface vary, depending on whether the device driver is configured as static or loadable:

- For static drivers

  The *xxprobe* interface typically checks some device status register to determine whether the physical device is present. Calling the BADADDR interface is one way to check device status registers. If the device is not present, the device is not initialized and not available for use.

  For static device drivers, the device interrupt handlers are defined in the system configuration file or the `stanza.static` file fragment and registered by the `config` program at system configuration time.

- For loadable drivers

  When device drivers are dynamically loaded, the loadable subsystem checks for the existence of the device before calling *xxprobe* . Note that loadable device drivers cannot call the BADADDR interface, which is available only in the early stages of system booting.

  For loadable device drivers, the *xxprobe* interface registers the device interrupt handlers by calling `handler_add` and `handler_enable`.

Return Value

The *xxprobe* interface returns the size of the control/status register address space for the autoconfiguration interfaces to use. A value of zero (failure) is returned to indicate that the driver did not complete the probe operation.

Related Information

Writing Device Drivers, Volume 2: Reference: `controller` structure

Name

`xxslave` — Checks that the device is valid for this controller

Entry Point

The `driver` structure

Synopsis

```
int xxslave(device, addr)
struct device *device;
caddr_t addr;
```

Arguments

device

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

addr

Specifies the System Virtual Address (SVA) of the base of the TURBOchannel slot space for the controller that this device is connected to.

Description

A device driver's `xxslave` 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. You (or the system manager) specify the attachments of these slave devices in the system configuration file or the `stanza.static` file fragment (for statically configured drivers) or in the `stanza.loadable` file fragment (for dynamically configured drivers).

Return Value

The `xxslave` interface returns a nonzero value if the device is present.

Related Information

Writing Device Drivers, Volume 2: Reference: `device` structure

Index

P

R

S

T

U

W

X

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 TURBOchannel Device Drivers
AA-PS3HB-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