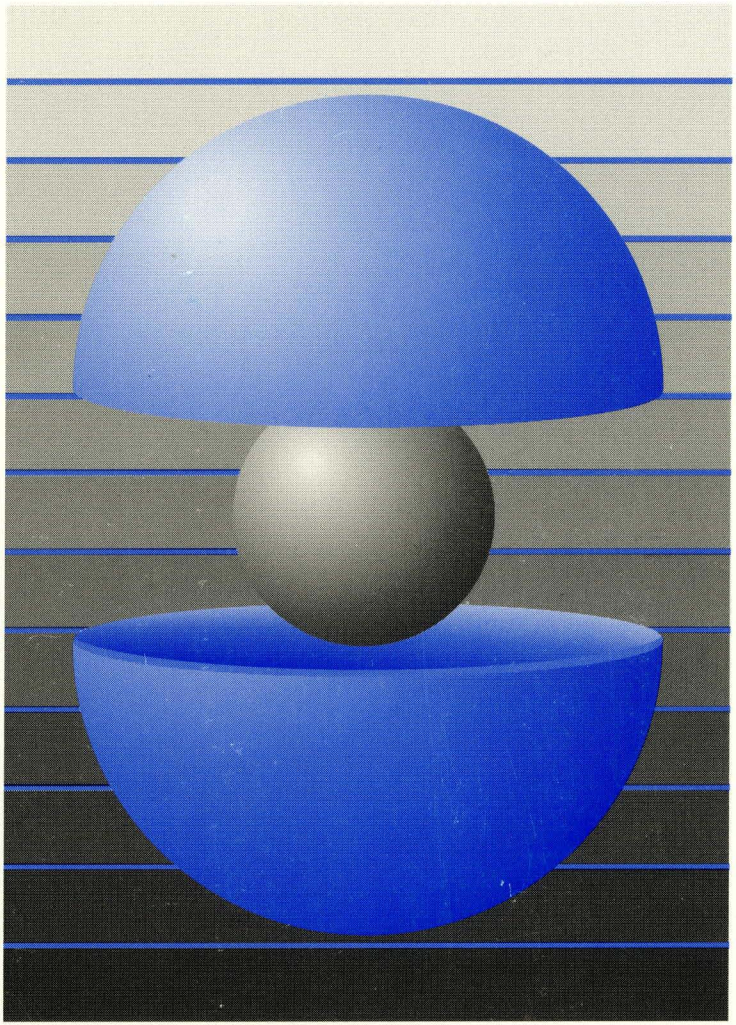
DEC OSF/1

digital

Writing EISA Bus Device Drivers



Part Number: AA-Q0R6A-TE

DEC OSF/1

Writing EISA Bus Device Drivers

Order Number: AA-Q0R6A-TE

February 1994

Product Version: DEC OSF/1 Version 2.0 or higher

This guide contains information systems engineers need to write device drivers that operate on the EISA bus. The guide describes EISA bus-specific topics, including EISA bus architecture and the data structures that EISA bus drivers use.

digital equipment corporation
Maynard, Massachusetts

Contents

About This Book

1 Review of Device Driver Concepts

2 EISA Bus Architectures

3 Structure of an EISA Bus Device Driver

4 Data Structures That EISA Bus Device Drivers Use

5 Device Autoconfiguration on the EISA Bus

6 EISA Bus Device Driver Example

7 EISA Bus Device Driver Configuration

A EISA Bus-Specific Reference Information

Figures

Tables

About This Book

This book discusses topics associated with writing device drivers that operate on the Extended Industry Standard Architecture (EISA) bus.

Audience

This book is intended for systems engineers who:

- Use standard library interfaces to develop programs in the C language
- Know the Bourne or some other UNIX-based shell
- Understand basic DEC OSF/1 concepts such as kernel, shell, process, configuration, and autoconfiguration
- Understand how to use the DEC OSF/1 programming tools, compilers, and debuggers
- Develop programs in an environment involving dynamic memory allocation, linked list data structures, and multitasking
- Understand the hardware device for which the driver is being written
- Understand the basics of the CPU hardware architecture, including interrupts, direct memory access (DMA) operations, and I/O

This book assumes you have a strong background in operating systems based on the UNIX operating system. It also assumes you have a strong background in systems and C programming, and that you are familiar with topics presented in *Writing Device Drivers, Volume 1: Tutorial*. In addition, the book assumes that you have no source code licenses.

Scope of the Book

The book is directed towards DEC OSF/1 on computer systems developed by Digital Equipment Corporation. The book builds on the concepts and topics presented in *Writing Device Drivers, Volume 1: Tutorial*, the core book for developing device drivers on DEC OSF/1. This book introduces driver writers to topics specific to the EISA bus and that are beyond the scope of the core tutorial. The book presents an example driver that operates on the EISA bus.

The book does not emphasize any specific types of device drivers. However, mastering the concepts and examples presented in this book is useful preparation for writing a variety of device drivers that operate on the EISA bus.

Organization

The book contains the following chapters and appendixes:

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	EISA Bus Architectures Discusses aspects of the EISA bus hardware and software architectures with which you must be familiar.
Chapter 3	Structure of an EISA Bus Device Driver Describes the sections that make up an EISA bus device driver.
Chapter 4	Data Structures That EISA Bus Device Drivers Use Discusses the data structures developed for use with EISA bus device drivers.
Chapter 5	Device Autoconfiguration on the EISA Bus Discusses the sequence of events that occurs during the autoconfiguration of devices that operate on the EISA bus.
Chapter 6	EISA Bus Device Driver Example Describes the `/dev/envram` device driver, which operates on the EISA bus.
Chapter 7	EISA Bus 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 EISA bus device drivers.
Appendix A	EISA Bus-Specific Reference Information Describes, in reference (man) page style, the header file, data structures, kernel support interfaces, and device driver interfaces that are specific to the EISA bus.
Appendix B	EISA Bus Device Driver Source Listing Contains the source code listing for the `/dev/envram` example driver.

Glossary	Glossary Provides definitions of terms related to the EISA 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; driver writers require knowledge in a variety of areas. One way to acquire this knowledge is to have at least the following categories of documentation available:

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

The following sections list the documentation associated with each of these categories. EISA bus device driver writers should also be familiar with the *Extended Industry Standard Architecture Revision 3.10* specification. This specification defines a high-performance, open-architecture bus available to personal computer (PC) manufacturers, expansion board vendors, software

developers, and semiconductor suppliers.

Hardware Documentation

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

Bus-Specific Device Driver Documentation

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

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

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

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

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

- *Writing TURBOchannel Device Drivers*

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

Programming Tools Documentation

To create your device drivers, you use a number of programming development tools and should have on hand the manuals that describe how to use these tools. The following manuals provide information related to

programming tools used in the DEC OSF/1 operating system environment:

- *Kernel Debugging*

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

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

- *Programming Support Tools*

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

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

- *Programmer's Guide*

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

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

System Management Documentation

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

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

Porting Documentation

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

This manual has three audiences, as follows:

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

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

Reference Pages

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

- *Reference Pages Section 2*

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

- *Reference Pages Section 3*

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

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

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

- Section 7 describes special files, related device driver functions, databases, and network support. This section is for programmers and system administrators.

- *Reference Pages Section 8*

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

Reader's Comments

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

- Internet electronic mail:
  `readers_comment@ravine.zk3.dec.com`
- Fax: 603-881-0120 Attn: USG Documentation, ZK03-3/Y32
- A completed Reader's Comments form (postage paid, if mailed in the United States). Two Reader's Comments forms are located at the back of each printed DEC OSF/1 manual.

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

Conventions

This book uses the following conventions:

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

	Vertical bars separating items that appear in the syntax definitions used in driver configuration indicate that you choose one item from among those listed.

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

Review of Device Driver Concepts 1

Before writing a driver for an Extended Industry Standard Architecture (EISA) device, you must be familiar with general driver concepts as well as specific tasks you need to perform to successfully code the driver. This book presumes you understand the following concepts:

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

The following sections review the tasks associated with writing device drivers. These tasks are discussed in detail in *Writing Device Drivers, Volume 1: Tutorial*.

1.1 Gathering Information

The first task in writing a device driver is to gather pertinent information about the host system and the device for which you are writing the driver. 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 Designing a Device Driver

After you gather information about the host system and the device, you are ready to design and develop 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.

Note

Loadable device drivers are not supported on the EISA bus. The `/dev/envram` example driver discussed in Chapter 6 does not implement the loadable device driver-specific code. If you are interested in learning about loadable device drivers in anticipation of future support, see *Writing Device Drivers, Volume 1: Tutorial.*

1.3 Allocating 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`	Specifies information such as pointers to the `probe`, `slave`, and `attach` interfaces used in the device driver. You initialize this data structure 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.

Chapter 4 describes the data structures that EISA bus device drivers use.

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.

The `/dev/envram` example driver uses the dynamic allocation technique discussed in *Writing Device Drivers, Volume 1: Tutorial.*

1.4 Writing Portable Device Drivers

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
- Memory mapping issues
- 64-bit versus 32-bit issues

1.5 Reviewing Device Driver Configuration Models

When you are ready to write your driver, you may 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. *Writing Device Drivers, Volume 1: Tutorial* describes the configuration models and provides example development environments. The tutorial discusses how to write device drivers for computer systems running the DEC OSF/1 operating system.

EISA Bus Architectures 2

The Extended Industry Standard Architecture (EISA) bus is an industry-standard high-performance bus that is a superset of the 8- and 16-bit Industry Standard Architecture (ISA). This chapter presents an overview of the EISA bus hardware and software architectures. Specifically, the chapter discusses the following:

- EISA bus hardware architecture
- EISA bus software architecture

For detailed information on EISA bus architectures, see the *Extended Industry Standard Architecture Revision 3.10.*

2.1 EISA Bus Hardware Architecture

The EISA bus is a 32-bit address and 32-bit data bus with support for 8-, 16-, 24, and 32-bit bus cycles. (A bus cycle refers to the action that occurs on the bus to transfer the data.) Support for different bus cycle sizes provides backward compatibility with the Industry Standard Architecture (ISA) bus options. The ISA bus is used in computer systems that adhere to the ISA. The ISA bus supplies the signals for performing the following basic functions of the computer system: memory, I/O, and direct memory access (DMA). There are two types of ISA options: one type supports only 8-bit bus cycles (PC and XT) and one type supports 8- and 16-bit bus cycles (AT).

The EISA bus also supports DMA and bus mastering. Although the EISA bus supports a maximum of 15 slots, most computer systems support a maximum of eight. The rest of this section describes the following EISA bus hardware architecture topics relevant to the device driver writer:

- Address spaces
- Data sizes
- Byte ordering

2.1.1 Address Spaces

The EISA bus defines two address spaces:

- I/O address space

The EISA bus supports a 64-kilobyte (KB) I/O address space. This I/O address space contains 16 slots, each of which is 4 KB in size. This I/O address space is referred to as slot-specific I/O space. The system board and the ISA-compatible I/O address space share slot zero (0).

- Memory address space

  The EISA bus supports a 4-gigabyte (GB) memory address space. This memory address space is available to any device on the bus. Assignment of the memory address space is done with the EISA configuration utility (ECU).

2.1.2 Data Sizes

The EISA bus supports 8-bit (D16), 16-bit (D16), 24-bit (D24), and 32-bit (D32) data sizes.

2.1.3 Byte Ordering

While the EISA bus does not specify any particular byte ordering, most devices use the Intel model, which is little endian. The Digital model is also little endian.

2.2 EISA Bus Software Architecture

The EISA bus supports a 4-gigabyte (GB) address space and a 64-kilobyte (KB) I/O space.

Structure of an EISA Bus 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 shows 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 shown in the figure, and more complex drivers can use additional sections. Both character and block device drivers can contain:

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

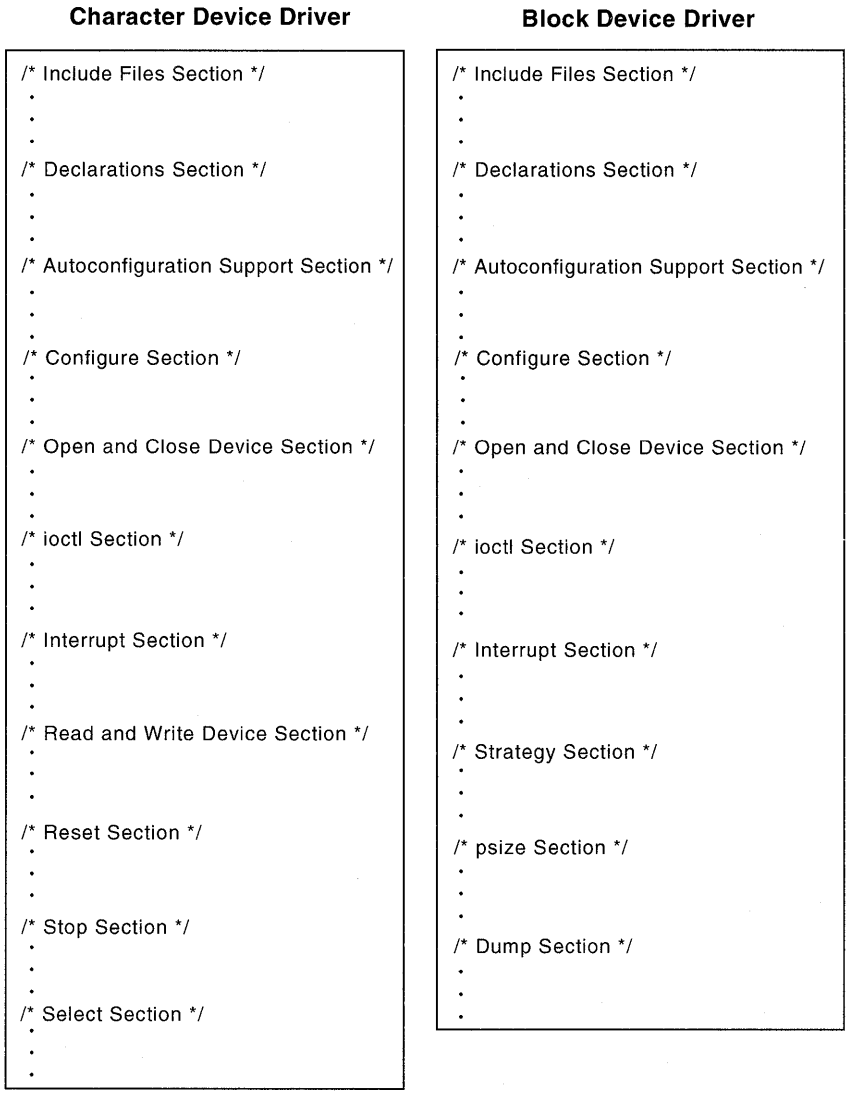
The block device driver can also contain a strategy section, a `psize` section, and a dump section.

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

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

Writing Device Drivers, Volume 1: Tutorial discusses each of the driver sections. The remainder of this chapter describes the differences in the following driver sections as they relate to EISA bus device drivers: include file and autoconfiguration support (specifically, the `xx`**probe** and `xx`**slave** interfaces).

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



ZK-0875U-R

3.1 Include Files Section

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

- The device driver header file
- The common driver header files
- The loadable driver header files
- The device register header file
- The *name*`_data.c` file

These files are described in *Writing Device Drivers, Volume 1: Tutorial.* EISA bus device drivers use the following file in addition to the previously mentioned files:

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

The `eisa.h` file contains definitions specific to the EISA bus.

Another file of interest to EISA bus device driver writers is `/usr/sys/data/eisa_option_data.c`. The `eisa_option_data.c` file contains the array (table) of `eisa_option` structures, which store information about a specific EISA option. Each structure contains such information as the option board's product ID and the name of the controlling device driver. Digital initilizes the array of `eisa_option` structures with its own EISA bus option product information. Third-party device driver writers can directly edit the `eisa_option` array with their EISA bus option product information; or, they can use the `mkeisadata` utility to automatically add product entries to a copy of the `eisa_option` array. Chapter 7 discusses the `mkeisadata` utility.

3.2 Autoconfiguration Support Section

When DEC OSF/1 boots, the kernel determines what EISA bus devices are connected to the computer. After finding an EISA bus device, the EISA bus configuration code initializes it based on the data supplied in the device's configuration file. The probe interface determines if a particular EISA bus device is present and the attach interface initializes the device. If the device is a disk controller, the slave interface determines if the device is present. The autoconfiguration support section of an EISA bus 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

Writing Device Drivers, Volume 1: Tutorial discusses each of these interfaces.

The following sections describe the `xxprobe` and `xxslave` interfaces as they apply to EISA bus device drivers. For convenience in referring to the names for the driver interfaces, the chapter uses the prefix *xx*. For example, `xxprobe` refers to a probe interface for some XX device.

3.2.1 Setting Up the Probe Interface

An EISA bus device driver's `xxprobe` interface performs the tasks necessary to determine if the device exists and is functional on a given system. At boot time, the EISA bus configuration code reads the configuration information saved by the EISA configuration utility (ECU) and performs checks to determine if the device is present before calling `xxprobe` for statically configured drivers. The EISA bus configuration code calls the `xxprobe` interface for each device that is defined according to specific rules discussed in Chapter 5. The `xxprobe` interface typically checks some device control status register (CSR) to determine whether the physical device is present. 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.

The `xxprobe` interface returns a nonzero value if the probe operation was successful. It returns the value zero (0) to indicate that the driver did not complete the probe operation.

The arguments you pass to the probe interface differ according to the bus on which the driver operates. The following code fragment shows you how to set up a probe interface for a driver that operates on an EISA bus:

```
xxprobe(bus_io_handle, ctlr)
io_handle_t bus_io_handle; 1
struct controller *ctlr; 2
{
/* Variable and structure declarations */
   •
   •
   •
```

```
/* Code to perform necessary checks */
    •
    •
    •
```

1 Specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space. The EISA bus configuration code passes this I/O handle to the driver's `xxprobe` interface during device autoconfiguration. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

Device drivers pass the I/O handle to the following categories of interfaces, which are discussed in *Writing Device Drivers, Volume 1: Tutorial*. These interfaces can process the I/O handle to access the desired bus address space.

- CSR I/O access interfaces

  The CSR I/O access interfaces are `read_io_port` and `write_io_port`. These are generic interfaces that allow device drivers to read from and write to device registers. Using these interfaces to read data from and write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

- I/O copy interfaces

  The I/O copy interfaces are `io_copyin`, `io_copyio`, `io_copyout`, and `io_zero`. These are generic interfaces that allow device drivers to perform I/O copy operations. Using these interfaces to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

Section 6.3 shows how the `/dev/envram` device driver uses the `read_io_port` and `write_io_port` interfaces to construct driver-specific interfaces to accomplish the tasks of reading from and writing to the device registers of an EISA bus nonvolatile random access memory (NVRAM) memory expansion board.

The `/dev/envram` device driver example described in Chapter 6 uses the `io_copyin`, `io_copyout`, and `io_zero` interfaces.

2 Specifies a pointer to the `controller` structure associated with this device. The bus configuration code passes this pointer to the driver's `xxprobe` interface. The device driver can reference hardware resource and other information contained in the `controller` structure pointer.

Section 6.6.1 shows the probe interface for the `/dev/envram` device driver.

3.2.2 Setting Up the Slave 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 (or the system manager) specify the attachments of these slave devices for static device drivers in the system configuration file or `stanza.static` file fragment. The following code fragment shows you how to set up a slave interface for a driver that operates on an EISA bus:

```
xxslave(device, bus_io_handle)
struct device *device; 1
io_handle_t bus_io_handle; 2
{
/* Variable and structure declarations */
   •
   •
   •
/* Code to check that the device is valid */
   •
   •
   •
```

1 Specifies a pointer to a `device` structure for this device. The bus configuration code passes this pointer to the driver's `xxslave` interface. The device driver can reference such information as the logical unit number of the device, whether the device is functional, and the bus number the device resides on.

2 Specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space. The EISA bus configuration code passes this I/O handle to the driver's `xxslave` interface during device autoconfiguration. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

Device drivers pass the I/O handle to the following categories of interfaces, which are discussed in *Writing Device Drivers, Volume 1: Tutorial*. These interfaces can process the I/O handle to access the desired bus address space.

- CSR I/O access interfaces

  The CSR I/O access interfaces are `read_io_port` and `write_io_port`. These are generic interfaces that allow device drivers to read from and write to device registers. Using these

interfaces to read data from and write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

- I/O copy interfaces

  The I/O copy interfaces are `io_copyin`, `io_copyio`, `io_copyout`, and `io_zero`. These are generic interfaces that allow device drivers to perform I/O copy operations. Using these interfaces to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

Data Structures That EISA Bus Device Drivers Use 4

Data structures are the mechanism used to pass information between the DEC OSF/1 kernel and device driver interfaces. EISA bus device driver writers must understand the data structures described in *Writing Device Drivers, Volume 1: Tutorial*. To write EISA bus device drivers, driver writers must also understand the following categories of data structures:

- `controller` structure members as specified on the EISA bus
- EISA bus structures

The following sections describe the structures associated with each category. For reference (man) page-style descriptions of these data structures, see Section A.2.2.

4.1 controller Structure Members as Specified on the EISA Bus

The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation.

Table 4-1 lists those members of the `controller` structure that have specific meanings on the EISA bus along with their associated data types. Following the table are descriptions of these members as they apply to the EISA bus.

Table 4-1: controller Structure

Member Name	Data Type
`slot`	`int`
`physaddr`	`caddr_t`

4.1.1 slot Member

The `slot` member specifies the bus slot or node number. As it applies to the EISA bus, the `slot` member specifies the slot number that the EISA bus board is plugged into. The slot number ranges from 1– *n*, where *n* represents the maximum number of slots. This value corresponds to the one that you (the device driver writer) specified in the system configuration file or the `stanza.static` file fragment. In the system configuration file, you precede the slot number for the slot that the EISA bus board is plugged into with the `slot` keyword, as follows:

```
slot 1
```

The autoconfiguration software uses the syntax following the `slot` keyword in the system configuration file or the `stanza.static` file fragment to obtain this value. The bus configuration code obtains this value from the autoconfiguration software and uses it to initialize the `slot` member of the `controller` structure associated with this EISA bus device.

The following code fragment shows that the `/dev/envram` device driver saves the slot number in a softc structure pointer:

```
   •
   •
   •
struct  envram_softc {
   •
   •
   •
 u_int slot; /* Slot this board is located in */
   •
   •
   •
};
   •
   •
   •
envram_probe(bus_io_handle, ctlr)
 io_handle_t bus_io_handle; /* I/O handle */
 struct controller *ctlr; /* Pointer to controller structure */
   •
   •
   •
{
 register struct envram_softc *sc; /* Pointer to softc structure */
   •
   •
   •
 sc->slot = ctlr->slot; /* Save the slot number that this */
                        /* board is located in */
   •
   •
   •
```

4.1.2 physaddr Member

The `physaddr` member specifies the physical address that corresponds to the virtual address set in the `addr` member. As it applies to the EISA bus, the `physaddr` member specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space.

The following code fragment shows that the `/dev/envram` device driver saves the base physical address of the device's slot-specific I/O address space in the driver's softc structure pointer. The code fragment also shows that `/dev/envram` uses this address in the construction of a read and write device register interface.

```
   •
   •
   •
/*****************************************************
 * The ENVRAM_READIO- and ENVRAM_WRITE_IO-related     *
 * macros are defined in envram_data.c.  They use     *
 * the read_io_port and write_io_port interfaces and  *
 * OR the offset with the sc->regbase value.  The     *
 * sc->regbase value is the peroption physical base  *
 * address of the EISA NVRAM I/O registers            *
 *****************************************************/

#define ENVRAM_READIO_D8(a)  read_io_port(sc->regbase | a, 1, BUS_IO)
   •
   •
   •
#define ENVRAM_WRITEIO_D8(a,d) write_io_port(sc->regbase | a, 1, BUS_IO, d)
   •
   •
   •
struct  envram_softc {
   •
   •
   •
 vm_offset_t regbase; /* Base address for device registers */
   •
   •
   •
};
   •
   •
   •
envram_probe(bus_io_handle, ctlr)
 io_handle_t bus_io_handle; /* I/O handle */
 struct controller *ctlr; /* Pointer to controller structure */
   •
   •
   •
{
 register struct envram_softc *sc; /* Pointer to softc structure */
   •
   •
   •
```

```
sc->regbase = (u_int)ctlr->physaddr; /* Save the controller's base */
                                     /* address. */
  .
  .
  .
```

4.2 EISA Bus Structures

In addition to the structures described in the *Writing Device Drivers, Volume 1: Tutorial*, device driver writers must understand the `eisa_option` data structure. The `eisa_option` structure contains EISA bus option information such as the bus option board's product ID and the name of the controlling device driver. An array of `eisa_option` structures is declared in the file `/usr/sys/data/eisa_option_data.c`. You must be familiar with:

- The members of the `eisa_option` structure
- The contents of the `eisa_option_data.c` file

4.2.1 Members of the eisa_option Structure

Table 4-2 lists the members of the `eisa_option` structure along with their associated data types.

Table 4-2: eisa_option Structure Members and Associated Data Types

Member Name	Data Type
`board_id`	`char [EISA_IDNAMELEN + 1]`
`function`	`char [EISA_FUNCLEN + 1]`
`driver_name`	`char [EISA_NAMELEN + 1]`
`intr_b4_probe`	`int`
`intr_aft_attach`	`int`
`type`	`char`
`adpt_config`	`int (*adpt_config) ()`

The `board_id` member specifies the EISA bus option expansion board's product ID name. You set `board_id` to the character string that represents the option board's product ID name. The ID name consists of a 3-character manufacturer code, a 3-character product identifier, and a 1-character revision number. For example, the string `ADP0002` identifies this as a second

revision of an Adaptec ADP1742A SCSI option board.

The `function` member specifies the text that identifies the product up to a maximum of 79 characters. This text can include part numbers. The default is the null string (""). You set `function` to the character string that represents the product. For example, the string `AHA1740` identifies this as the product name for a Digital ADP1742A SCSI option board.

The `driver_name` member specifies the name of the controlling device driver. You set `driver_name` to the character string that represents the name of the controlling device driver up to a maximum of 8 characters. This same character string also specifies the name of the controlling device driver in the system configuration file. For example, the string `aha` identifies this as the controlling device driver for the device connected to an ADP1742A SCSI option board.

The `intr_b4_probe` member specifies whether the device needs interrupts enabled before the bus configuration code calls the driver's *xx*`probe` interface. You can set `intr_b4_probe` to the value zero (0) or the value 1. 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.

The `intr_aft_attach` member specifies whether the device needs interrupts enabled after the driver's *xx*`probe` and *xx*`cattach` or *xx*`dattach` interfaces complete execution. You can set `intr_aft_attach` to the value zero (0) or the value 1. 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.

The `type` member specifies the type of device. You can set `type` to the string `C` (the device is a controller) or `A` (the device is a bus or an adapter). The default value is `C`.

The `adpt_config` member specifies the name of the bus (or adapter) configuration interface to call. You set `adpt_config` to the string that identifies the bus configuration interface, if `type` was set to `A` (the device is a bus or an adapter). Otherwise, do not set `adpt_config` to any value.

4.2.2 Contents of the eisa_option_data.c File

Digital initilizes the array of `eisa_option` structures with its own EISA bus option product information. Third-party device driver writers can directly edit the `eisa_option` array with their EISA bus option product information; or, they can use the `mkeisadata` utility to automatically add product entries to a copy of the `eisa_option` array. Chapter 7 discusses the `mkeisadata` utility.

The following shows the Digital-provided `eisa_option_data.c` file:

```
struct  eisa_option     eisa_option[] =
{
/* board       function     driver  intr_b4 itr_aft         adpt */
/* id            name        name    probe   attach type  config */
/* ------     ------------ ------  ------- ------- ----  ------ */

{ "CPQ3011", "",            "vga",     0,      1,    'C',    0}, /* QVISION */
{ "DEC4220", "NET,ETH",     "ln",      0,      1,    'C',    0}, /* DEC_LANCE */
{ "ADP0002", "AHA1740",     "aha",     0,      1,    'C',    0}, /* ADP1742A */
                                                                  /* SCSI */
{ "ADP0002", "MSD,FPYCTL", "fdi",      0,      1,    'C',    0}, /* ADP1742A */
                                                                  /* FLPY */
{ "DEC3001", "",            "fta",     0,      1,    'C',    0}, /* DEC_FDDI */
{ "DEC3002", "",            "fta",     0,      1,    'C',    0}, /* DEC_FDDI */
{ "DEC2500", "",            "envram", 0,       0,    'C',    0}, /* EISA NVRAM */
{ "ISA1010", "COM,1",       "ace",     0,      1,    'C',    0}, /* COMM POR TS */
{ "ISA1010", "COM,2",       "ace",     0,      1,    'C',    0}, /* COMM POR TS */
{ "ISA1010", "PAR",         "ace",     0,      1,    'C',    0}, /* COMM PORTS */


/*
* Do not delete any table entries above this line or your system
* will not configure properly.
*
* Add any new controllers or devices here.
* Remember, the module name must be blank padded to 7 bytes.
*/

    /*
%%%Used by mkeisadata as placemarker for automatic installation
   */

/*
* Do not delete this null entry, which terminates the table or your
* system will not configure properly.
*/
{   "",             ""      }       /* Null terminator in the table */
};
```

Device Autoconfiguration on the EISA Bus 5

Autoconfiguration is a process that determines what hardware actually exists during the current instance of the running kernel. *Writing Device Drivers, Volume 1: Tutorial* discusses the events that occur during device autoconfiguration, using the TURBOchannel bus as an example. This chapter describes device configuration on the EISA bus, which consists of the following topics:

- Audiences involved in delivering an EISA bus product
- Autoconfiguration for devices connected to the EISA bus

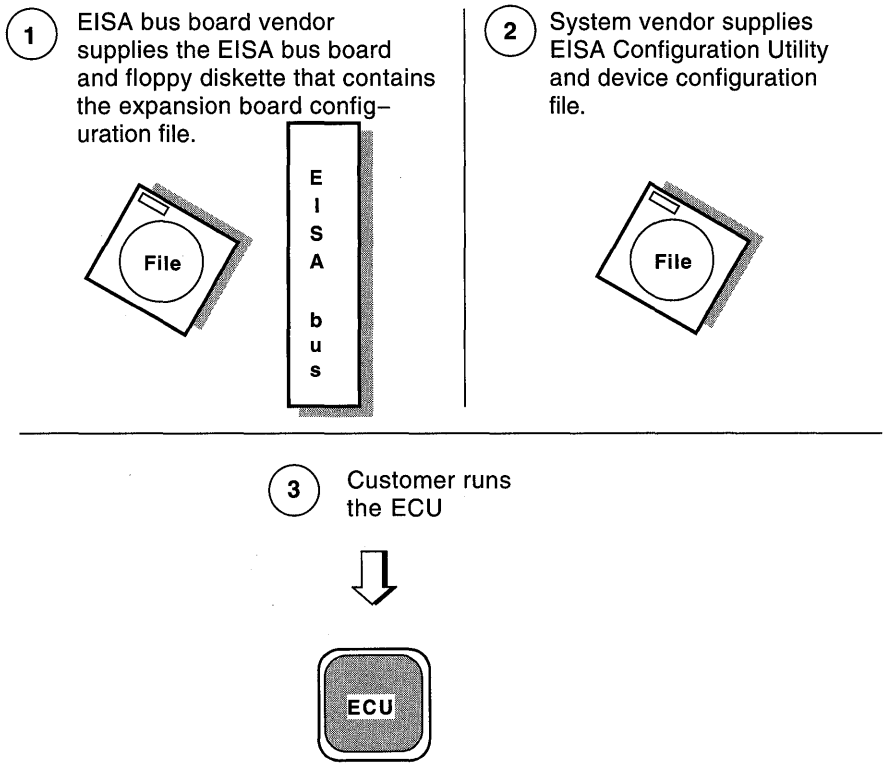
5.1 Audiences Involved in Delivering an EISA Bus Product

Figure 5-1 shows the audiences involved in delivering an EISA bus product. The figure shows at least three different audiences:

- An EISA bus expansion board manufacturer

  The manufacturer of the EISA bus expansion board supplies the board and its configuration requirements in a configuration file. An EISA bus expansion board does not provide for onboard configuration space. An expansion board configuration file provides information on the resources and register values required to program the EISA bus expansion board. Some of the resources supplied in this file are the board ID, manufacturer of the board, the interrupt line, DMA channel, and board memory. Do not confuse the expansion board configuration file with the DEC OSF/1 system configuration file, which is an ASCII text file that defines the hardware and software components of the system. Expansion board configuration files, unlike system configuration files, have `CFG` extensions. The figure shows that the manufacturer distributes the expansion board configuration file on a diskette.

Figure 5-1: Audiences Associated with an EISA Bus Product



ZK-0904U-R

- A system vendor

  A system vendor builds the CPU on which the EISA bus connects. For example, as a system vendor Digital builds the Alpha AXP CPUs and implements the software that allows an EISA bus device to connect to the CPU. The system vendor supplies the EISA configuration utility and device configuration file for its system. The EISA configuration utility (ECU) is a program that reads the expansion board and device configuration files and, for each option in the system, generates a conflict-free configuration. The ECU runs on top of the firmware, not the operating system. A device configuration file provides information on the resources required to program the EISA bus device. Some of the resources supplied in this file are dma channel, interrupt channel, and memory space information. Do not confuse the device configuration file with the DEC OSF/1 system configuration file, which is an ASCII text

file that defines the hardware and software components of the system. Device configuration files, unlike system configuration files, have `CFG` extensions.

After generating a conflict-free configuration, the ECU writes it to nonvolatile random-access memory (NVRAM) for future use by the firmware and the DEC OSF/1 operating system.

- A customer

  A customer purchases the EISA expansion board from the board manufacturer, the CPU from a system vendor like Digital, and the associated peripheral devices from a system vendor like Digital. The system vendor might also provide the device driver for the peripheral device. The customer loads the diskette that contains the expansion board configuration file, the device configuration file, and the ECU. The customer runs the ECU.

  The ECU powers up the CPU and the firmware compares the options present on the system with the configuration information stored in NVRAM. If the information matches, the powerup continues. If the information does not match, the powerup is discontinued and the user is instructed to run the ECU. As powerup continues, the firmware uses the configuration information in NVRAM to configure and initialize each option on the system. After initializing an option, the firmware enables the option through the EISA bus control register.

  A customer must run the ECU any time a new EISA expansion board is added to the system.

5.2 Configuration for Devices Connected to the EISA Bus

This section summarizes device autoconfiguration for EISA bus devices, focusing on the interaction of the expansion board configuration file, the device configuration file, the EISA configuration utility (ECU), and the DEC OSF/1 operating system. The following list summarizes the tasks associated with device autoconfiguration for EISA bus devices:

- Creating an expansion board configuration file
- Creating a device configuration file
- Running the EISA configuration utility (ECU)
- Reading the information generated by the ECU
- Guidelines for specifying entries in the device configuration file

The following sections describe these tasks.

5.2.1 Creating an Expansion Board Configuration File

As stated in Section 5.1, the EISA expansion board manufacturer creates an expansion board configuration file that contains such resources as the board ID, manufacturer identification, the interrupt line, DMA channel, and board memory. The *Extended Industry Standard Architecture Revision 3.10* specification describes the syntax for specifying the resources in the expansion board configuration file. The board manufacturer copies the expansion board configuration file to a diskette.

5.2.2 Creating a Device Configuration File

As stated in Section 5.1, the system vendor creates a device configuration file that contains such device resources as the DMA channel, interrupt channel, and memory space. The *Extended Industry Standard Architecture Revision 3.10* specification describes the syntax for specifying the device resources and indicating a selection of choices for each resource in the device configuration file. For example, a device configuration file might specify that a device requires an interrupt channel and that the device can use channels 9, 11, 14, or 15. The system vendor copies the device configuration file to a diskette (either the diskette that contains the expansion board configuration file or a separate diskette).

The system vendor also supplies the ECU. The *Extended Industry Standard Architecture Revision 3.10* specification provides guidelines on how to write the ECU.

5.2.3 Running the EISA Configuration Utility

The customer runs the ECU whenever the configuration changes. The ECU reads the board configuration file and device configuration file for each device in the system and assigns resources based on the information in the device configuration file. The ECU resolves any conflicts. The ECU saves the configuration information in nonvolatile random access memory (NVRAM).

5.2.4 Reading the Information Generated by the ECU

When the customer boots the CPU, the EISA bus configuration code reads the configuration information from NVRAM and does the following for each device in the configuration:

- Looks for a device entry in the `eisa_option` table. If there is no entry in the table, goes to the next device (and does not probe the device whose entry is missing from the table). After finding a valid entry in the table, continues.

- Looks for the device in the controller table. If found, continues. If not found, stops and goes to the next device (and does not probe the device whose entry is missing from the controller table).
- Configures the device based on the information saved by the ECU in NVRAM. Typically, this activity consists of writing a set of device registers to tell the device the resources assigned to it.
- Calls the device driver's *xx*`probe` interface.
- If the probe operation succeeds, calls the driver's *xx*`cattach` or *xx*`dattach` interface.

In summary, for the EISA bus configuration code to call the driver's *xx*`probe` interface the following must occur:

1. The system vendor creates a configuration file for the device and copies it to the system configuration diskette or a separate diskette that the ECU can read. The system configuration diskette, supplied by the board manufacturer, contains a configuration file for the expansion board. The diskette is in FAT format.
2. The device driver writer creates an entry for the device in the `eisa_option` table located in the `eisa_option_data.c` file.
3. The device driver writer creates an entry (or entries if there are multiple instances of the device) in the system configuration file.
4. The customer runs the ECU.

EISA Bus Device Driver Example 6

This chapter provides you with an opportunity to study an EISA bus device driver called `/dev/envram`. You can use the `/dev/envram` device driver as the basis for writing your own working EISA bus device drivers. The `/dev/envram` device driver operates on an EISA bus and implements many of the device driver interfaces shown in Chapter 3. It also implements other sections needed by the EISA bus NVRAM memory expansion board. The chapter begins with a section that gives you an overview of the tasks performed by the `/dev/envram` device driver. Following this overview are sections that describe each piece of the `/dev/envram` device driver. Table 6-1 lists the parts of the `/dev/envram` device driver and the sections of the chapter where each is described.

Table 6-1: Parts of the /dev/envram Device Driver

Part	Section
envram_reg.h Header File	Section 6.2
envram_data.c File	Section 6.3
Include Files Section for the /dev/envram Device Driver	Section 6.4
Declarations Section for the /dev/envram Device Driver	Section 6.5
Autoconfiguration Support Section for the /dev/envram Device Driver	Section 6.6
Status Section for the /dev/envram Device Driver	Section 6.7
Battery Status Section for the /dev/envram Device Driver	Section 6.8
Read and Write Device Section for the /dev/envram Device Driver	Section 6.9
Zero NVRAM Section	Section 6.10

The source code uses the following convention:

```
#define ENVRAM_MAPPED 1 [1]
```

[1] Numbers appear after some line or lines of code in the `/dev/envram` device driver example. Following the example, a corresponding number appears that contains an explanation for the associated line or lines. The source code does not contain any inline comments. If you prefer to read the `/dev/envram` driver source code in its entirety with the inline comments, see Appendix B.

6.1 Overview of the /dev/envram Device Driver

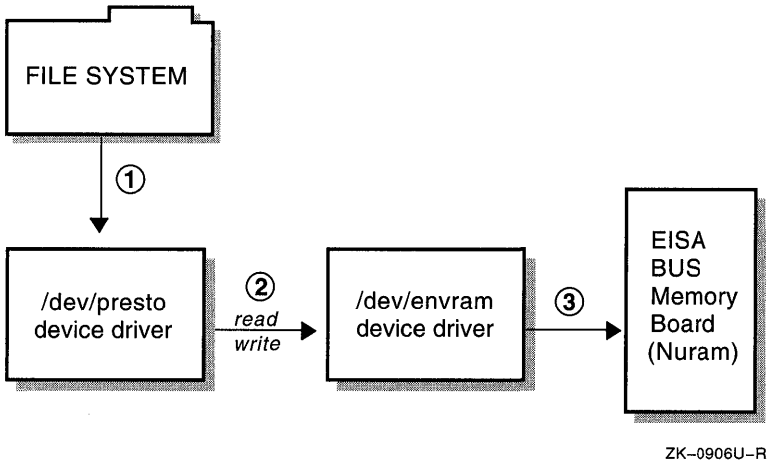
The `/dev/envram` device driver is a character device driver that provides read and write services to the `/dev/presto` device driver. The `/dev/presto` device driver is a disk driver that uses nonvolatile memory as a cache. It works as a layer between other drivers and the rest of the DEC OSF/1 kernel.

The `/dev/presto` device driver's interfaces appear as the entry points in the `cdevsw` and `bdevsw` switch tables, instead of the interfaces of the other drivers (including the `/dev/envram` driver) it works with. Whenever `/dev/presto` needs to perform actual I/O operations (for example, when the cache needs filling or draining), it calls the layered driver's entry points (strategy, close, read, and write).

Figure 6-1 shows the relationship between the `/dev/envram` and `/dev/presto` device drivers. The figure shows the flow of data between the different layers.

1. The file system makes a read or write request.
2. The `/dev/presto` device driver interprets the request as an actual I/O operation. It calls the `/dev/envram` device driver.
3. The `/dev/envram` device driver performs the actual work of reading from and writing data to the EISA bus NVRAM memory expansion board. In addition to providing services to the `/dev/presto` driver, the `/dev/envram` device driver also:
   - Manages the EISA bus NVRAM memory expansion board
   - Hides hardware idiosyncrasies from the `/dev/presto` driver
   - Allows the EISA bus NVRAM memory expansion board to communicate with some SCSI device

Figure 6-1: Relationship of the /dev/envram and /dev/presto Device Drivers



6.2 envram_reg.h Header File

The following `envram_reg.h` file is the device register header file for the `/dev/envram` device driver. It contains public declarations and the definitions that map to the device registers for the EISA bus NVRAM memory expansion board.

```
#define ENVRAM_CSR       0xc00  [1]
#define ENVRAM_BAT       0xc04
#define ENVRAM_HIBASE    0xc08
#define ENVRAM_CONFIG    0xc0c
#define ENVRAM_ID        0xc80
#define ENVRAM_CTRL      0xc84
#define ENVRAM_DMA0      0xc88
#define ENVRAM_DMA1      0xc8c

#define ENVRAM_DIAG_REGISTER   0x3f8 [2]
#define BOARD_FAILED           0x00000008
#define ENVRAM_DIAG_RESVED     0x400
#define ENVRAM_CACHE_OFFSET    0x400

#define SET_LED          0x0100  [3]
#define BAT_FAIL         0x0800
#define WRMEM            0x2000
#define SET_DREQ         0x4000
#define DMA_CHAN_7       0X80
#define DMA_CHAN_5       0x40

#define BAT_DISCON_BIT   0x0080  [4]

#define EISA_ENABLE_BOARD  0x1  [5]

#define ENVRAM_ID_MASK   0x0025a310 [6]

#define ENVRAM_MAPPED     1   [7]
#define ENVRAM_NOTMAPPED  0
#define ENVRAM_CACHED     1
#define ENVRAM_NOTCAHCED  0

#define ENVRAM_XFER_SIZE 1024 [8]
#define ENVRAM_ALLIGN    8192 [9]
```

[1] This `#define`, and the seven that follow, map to the device registers of the EISA bus NVRAM memory expansion board. The following list describes each device register definition:

- `ENVRAM_CSR`

  This is the control status register (CSR) for the EISA bus NVRAM memory expansion board. Section 6.6.1, Section 6.8.2, and Section 6.8.3 show that the `envram_probe`, `eisa_nvram_battery_enable`, and `eisa_nvram_battery_disable` interfaces pass `ENVRAM_CSR` as an argument to `ENVRAM_WRITEIO_D16`. This interface writes data to the CSR. Section 6.5 shows that the `/dev/envram` driver

uses write_io_port to construct ENVRAM_WRITEIO_D16.

Section 6.9.2 shows that the envram_write interface passes ENVRAM_CSR as an argument to ENVRAM_READIO_D16 and ENVRAM_WRITEIO_D16. The ENVRAM_READIO_D16 interface reads data from the CSR. Section 6.5 shows that the /dev/envram driver uses read_io_port to construct ENVRAM_READIO_D16.

- ENVRAM_BAT

  This is the battery disconnect device register for the EISA bus NVRAM memory expansion board. Section 6.8.2 and Section 6.8.3 show that the eisa_nvram_battery_enable and eisa_nvram_battery_disable interfaces pass ENVRAM_BAT as an argument to ENVRAM_WRITEIO_D8. This interface writes data to the battery disconnect device register. Section 6.5 shows that the /dev/envram driver uses write_io_port to construct ENVRAM_WRITEIO_D8.

- ENVRAM_HIBASE

  This is the extended memory configuration device register for the EISA bus NVRAM memory expansion board. The ECU sets up this device register.

- ENVRAM_CONFIG

  This is the configuration device register for the EISA bus NVRAM memory expansion board. The ECU sets up this device register.

- ENVRAM_ID

  This is the ID device register for the EISA bus NVRAM memory expansion board. Section 6.6.1 shows that the envram_probe interface passes ENVRAM_ID as an argument to ENVRAM_READIO_D32. The ENVRAM_READIO_D32 interfaces checks ENVRAM_ID to ensure that the module is correct for this driver. Section 6.5 shows that the /dev/envram driver uses read_io_port to construct ENVRAM_READIO_D32.

- ENVRAM_CTRL

  This is the controller device register for the EISA bus NVRAM memory expansion board. Section 6.6.1 shows that the envram_probe interface passes ENVRAM_CTRL as an argument to ENVRAM_WRITEIO_D8. This interface writes data to the controller device register. Section 6.5 shows that the /dev/envram driver uses write_io_port to construct ENVRAM_WRITEIO_D8.

- ENVRAM_DMA0

  This is the DMA address device register 0 for the EISA bus NVRAM memory expansion board. Section 6.9.2 shows that the

envram_write interface passes ENVRAM_DMA0 as an argument to ENVRAM_WRITEIO_D16. This interface writes data to DMA address device register 0. Section 6.5 shows that the /dev/envram driver uses write_io_port to construct ENVRAM_WRITEIO_D16.

- ENVRAM_DMA1

  This is the DMA address device register 1 for the EISA bus NVRAM memory expansion board. Section 6.9.2 shows that the envram_write interface passes ENVRAM_DMA1 as an argument to ENVRAM_WRITEIO_D16. This interface writes data to DMA address device register 1. Section 6.5 shows that the /dev/envram driver uses write_io_port to construct ENVRAM_WRITEIO_D16.

2 This #define, and the three that follow, map to the EISA NVRAM software diagnostic registers. The following list describes each EISA NVRAM software diagnostic register definition:

- ENVRAM_DIAG_REGISTER

  This is the software diagnostic register for the EISA bus NVRAM memory expansion board. The envram_probe interface uses the console diagnostics to set this device register.

- BOARD_FAILED

  This software diagnostic register is set if the EISA bus NVRAM memory expansion board failed software diagnostic tests. Section 6.6.1 shows that the envram_probe interface uses BOARD_FAILED to set the software diagnostic status in the softc structure.

- ENVRAM_DIAG_RESVED

  This software diagnostic register specifies the amount of space the software diagnostics require.

- ENVRAM_CACHE_OFFSET

  This software diagnostic register on the EISA bus NVRAM memory expansion board is the offset from the starting address of the NVRAM. This address identifies where the /dev/presto driver can use the NVRAM. Section 6.6.1 shows how the envram_probe interface uses ENVRAM_CACHE_OFFSET.

3 This #define, and the five that follow, are bit masks for the EISA bus NVRAM memory expansion board's CSR (the ENVRAM_CSR device register offset). The following list describes each bit mask:

- SET_LED

  This CSR bit mask turns on the Light Emitting Diode (LED) for the

EISA bus NVRAM memory expansion board. Section 6.8.2 shows that the `eisa_nvram_battery_enable` interface uses this CSR bit mask to turn on the LED.

- `ENBL_BAT_INT`

  This CSR bit mask enables the battery fail interrupt for the EISA bus NVRAM memory expansion board. The `/dev/envram` driver does not currently use this CSR bit mask.

- `ENBL_PFAIL_INT`

  This CSR bit mask enables the power failure interrupt for the EISA bus NVRAM memory expansion board. The `/dev/envram` driver does not currently use this CSR bit mask.

- `BAT_FAIL`

  Section 6.8.1 shows that the `eisa_nvram_battery_status` interface uses `BAT_FAIL` to check for a battery failure on the EISA bus NVRAM memory expansion board.

- `BAT_DISCON`

  This CSR bit mask indicates the status of the disconnect circuit for the EISA bus NVRAM memory expansion board. Section 6.8.2 and Section 6.8.3 show that the `eisa_nvram_battery_enable` and `eisa_nvram_battery_disable` interfaces use this CSR bit mask to provide the `/dev/presto` driver with the ability to enable and disable the battery.

- `WRMEM`

  Section 6.6.1 shows that the `envram_probe` interface uses `WRMEM` to enable writes to the EISA bus NVRAM memory expansion board. Section 6.8.2, Section 6.8.3, and Section 6.9.2 show that the `eisa_nvram_battery_enable`, `eisa_nvram_battery_disable`, and `envram_write` interfaces pass the `WRMEM` bit mask to the `ENVRAM_WRITEIO_D16` interface.

- `SET_DREQ`

  This CSR bit mask sets the device requirements for DMA operations on the EISA bus NVRAM memory expansion board. Section 6.9.2 shows how the `envram_write` interface uses this bit mask.

- `DMA_CHAN_7`

  This CSR bit mask identifies channel 7 for DMA operations on the EISA bus NVRAM memory expansion board. The map load code from the ECU sets this device register.

- DMA_CHAN_5

  This CSR bit mask identifies channel 5 for DMA operations on the EISA bus NVRAM memory expansion board. The map load code from the ECU sets this device register.

4 Defines a bit mask called BAT_DISCON_BIT for the EISA bus NVRAM memory expansion board's battery disconnect device register (the ENVRAM_BAT device register offset). Section 6.8.2 and Section 6.8.3 show how the eisa_nvram_battery_enable and eisa_nvram_battery_disable interfaces use the BAT_DISCON_BIT bit mask.

5 Defines a bit mask called EISA_ENABLE_BOARD for the EISA bus NVRAM memory expansion board's controller device register (the ENVRAM_CTRL device register offset). This bit mask makes memory on the EISA bus NVRAM memory expansion board available. Section 6.6.1 shows how the envram_probe interface uses this bit mask.

6 Defines a bit mask called ENVRAM_ID_MASK for the EISA bus NVRAM memory expansion board's ID device register (the ENVRAM_ID device register offset). Section 6.6.1 shows that the envram_probe interface uses the ENVRAM_ID_MASK bit mask to check the hardware ID in the EISA bus NVRAM memory expansion board's ID device register.

7 This #define, and the three that follow, are used to communicate with the /dev/presto device driver. The following list describes each definition:

- ENVRAM_MAPPED

  This constant indicates that the buffer was mapped. This constant is not currently used by the /dev/envram driver.

- ENVRAM_NOTMAPPED

  Section 6.6.2 shows that the envram_attach interface uses this constant to indicate to the /dev/presto device driver that the buffer was not mapped.

- ENVRAM_CACHED

  Section 6.6.2 shows that the envram_attach interface uses this constant to indicate to the /dev/presto device driver the use of kernel segment (kseg) space.

- ENVRAM_NOTCAHCED

  This constant indicates the use of a cached space. This constant is not currently used by the /dev/envram driver.

8 Defines a constant called ENVRAM_XFER_SIZE to indicate the maximum DMA transfer size to the NVRAM module. Section 6.6.2 and Section 6.9.2 show how the envram_attach and envram_write

interfaces use ENVRAM_XFER_SIZE.

9 Defines a constant called ENVRAM_ALLIGN to indicate the DMA alignment requirement. Section 6.9.2 shows how the envram_write interface uses ENVRAM_ALLIGN.

6.3 envram_data.c File

The /usr/sys/data/envram_data.c file is the *name*_data.c file for the /dev/envram device driver. It contains the softc structure, which is used to share data between the different /dev/envram device driver interfaces.

```
struct      envram_softc {
 io_handle_t regbase;
 io_handle_t cache_phys_start;
 io_handle_t cache_base;
 vm_offset_t cache_kseg_start;
 u_long saved_mem_sysmap;
 u_int cache_size;
 u_int cache_offset;
 io_handle_t diag_status;
 dma_handle_t sglp;
 struct controller *ctlr;
}; 1

struct  envram_softc *envram_softc; 2
struct  controller *envram_info[NENVRAM]; 3
```

1 Defines the softc data structure for the /dev/envram device driver and calls it envram_softc. Most of the sections of the /dev/envram device driver declare a pointer to the envram_softc structure.

The following list describes the members contained in this structure:

- regbase

  Stores the physical base address of the I/O device registers for the EISA bus NVRAM memory expansion board. Section 6.5 shows that the /dev/envram driver uses this member to construct the EISA NVRAM I/O device register interfaces (macros). Section 6.6.1 shows that the envram_probe interface sets this member to the controller's base address.

  The regbase member and other members of the envram_softc structure are I/O handles. An I/O handle is a data entity that is of type io_handle_t. Device drivers use the I/O handle to reference bus address space (either I/O space or memory space). The bus configuration code passes the I/O handle to the device driver's *xx*probe interface during device autoconfiguration. One purpose of the I/O handle is to hide CPU-specific architecture idiosyncracies that describe how to access a device's control status registers (CSRs) and how to perform I/O copy operations.

  Specifically, this member is the I/O handle base of the EISA NVRAM I/O registers associated with a specific EISA option.

- cache_phys_start

  Stores the starting physical address of the NVRAM cache. Section

6.6.1 shows that the envram_probe interface sets this address.

- cache_base

  Stores the base address of the NVRAM in EISA bus address space. Section 6.6.1 shows that the envram_probe interface sets this base address. Section 6.9.2 shows that the envram_write interface uses this member to set up the destination address passed from the /dev/presto driver.

- cache_kseg_start

  Stores the starting kseg address of the cache used by the /dev/presto driver. Section 6.6.1 shows that the envram_probe interface sets this address. Section 6.6.2 shows that the envram_attach interface passes this address to the /dev/presto driver's presto_init interface.

- saved_mem_sysmap

  Stores the sysmap portion of the I/O handle. Section 6.6.1 shows that the envram_probe interface initializes this member. Section 6.9.1 and Section 6.9.2 show that the envram_read and envram_write interfaces pass this member to the I/O copy interfaces io_copyin and io_copyout. Section 6.10 shows that the envram_zero interface passes this member to the I/O copy interface io_zero.

- cache_size

  Stores the size of the NVRAM cache. Section 6.6.1 shows that the envram_probe interface initializes this member. Section 6.6.2 shows that the envram_attach interface passes this size to the /dev/presto driver's presto_init interface.

- cache_offset

  Stores the offset to the first NVRAM location from the start of the EISA slot address. Section 6.6.1 shows that the envram_probe interface initializes this member.

- diag_status

  Stores the bit masks that indicate whether the EISA bus NVRAM memory expansion board passed the software diagnostic tests. Section 6.6.1 shows that the envram_probe interface initializes this member. Section 6.7 shows that this member is an implicit input to the eisa_nvram_status interface.

- sglp

  Specifies a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A

bus address/byte count pair is represented by the `ba` and `bc` members of an `sg_entry` structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

Section 6.6.2 and Section 6.9.2 show that the `envram_attach` and `envram_write` interfaces pass this DMA handle to the `dma_map_load` and `dma_map_alloc` interfaces.

- `ctlr`

  Declares a pointer to the `controller` structure associated with this EISA bus NVRAM memory expansion board. Section 6.6.1 and Section 6.6.2 show that several members of the `controller` structure pointer are implicit inputs to the `envram_probe` and `envram_attach` interfaces.

2 Declares a pointer to the `envram_softc` data structure.

3 Declares a pointer to an array of `controller` structures. The compile time variable is used as an index into the array. If the compile time variable *NENVRAM* is greater than zero (0), declares pointers to arrays of `controller` structures. Thus, there is one `controller` structure for each EISA bus NVRAM memory expansion board. The /dev/envram driver does not currently support this.

6.4 Include Files Section for the /dev/envram Device Driver

The following code shows the include files section for the `/dev/envram` device driver:

```
#include "envram.h" 1
#include <vm/vm_kern.h>
#include <sys/presto.h> 2
#include <io/common/devdriver.h>
#include <io/dec/eisa/eisa.h> 3
#include <data/envram_data.c> 4
#include <machine/rpb.h>
#include <io/dec/eisa/envram_reg.h> 5
```

1 The `envram.h` file is generated by the `config` program and it contains `#define` statements for the number of EISA bus devices on the system. This file is also included in `/usr/sys/io/common/conf.c`, which is where you define the entry points for the driver. However, the entry points for the `/dev/envram` driver are not defined in `/usr/sys/io/common/conf.c` because its entry points are called by the `/dev/presto` device driver. It is the `/dev/presto` device driver's entry points that are defined in `/usr/sys/io/common/conf.c`.

Section 6.6.2 shows how the `/dev/envram` driver initializes a data structure with the names of its entry points so that the `/dev/presto` driver can call them.

2 This header file contains the definitions and structure definitions for the `/dev/presto` device driver. Section 6.6.2 shows how the `/dev/envram` driver initializes the `presto_interface0` data structure with its entry points.

3 The `/usr/sys/include/io/dec/eisa/eisa.h` file contains data structures referenced by EISA bus device drivers and by the bus configuration code. For a summary description of this header file, see Section A.2.1.

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

4 This is the *name*`_data.c` file for the `/dev/envram` device driver. The `envram_data.c` file contains the softc structure used by the `/dev/envram` device driver. Section 6.3 describes the members of the softc structure.

5 This is the device register header file for the `/dev/envram` driver. It contains public declarations and `#define` statments used by the

`/dev/envram` device driver. The following lists some of the categories of information contained in this file:

- EISA bus NVRAM register definitions
- NVRAM device register offset definitions
- CSR register bit mask definitions
- Battery disconnect register bit mask definitions
- EISA bus control register bit masks

Section 6.2 describes the contents of the `envram_reg.h` device register header file.

6.5 Declarations Section for the /dev/envram Device Driver

The following code shows the declarations section for the `/dev/envram` device driver:

```
#define ENVRAM_READIO_D8(a) \
 read_io_port((io_handle_t)sc->regbase | a, 1, 0)) 1
#define ENVRAM_READIO_D16(a) \
 read_io_port((io_handle_t)sc->regbase | a, 2, 0))
#define ENVRAM_READIO_D32(a) \
 read_io_port((io_handle_t)sc->regbase | a, 4, 0))

#define ENVRAM_WRITEIO_D8(a,d) \
 write_io_port((io_handle_t)sc->regbase | a, 1, 0, d)) 2
#define ENVRAM_WRITEIO_D16(a,d) \
 write_io_port((io_handle_t)sc->regbase | a, 2, 0, d))
#define ENVRAM_WRITEIO_D32(a,d) \
write_io_port((io_handle_t)sc->regbase | a, 4, 0, d))

int envram_probe(), envram_attach(), eisa_nvram_status();
int eisa_nvram_battery_enable(), eisa_nvram_battery_disable();
void envram_read(), envram_write(), envram_zero(); 3
struct driver envramdriver = {
        envram_probe,
        0,
        envram_attach,
        0,
        0,
        0,
        0,
        0,
        "envram",
        envram_info,
        0,
        0,
        0,
        0,
        0,
        0,
        0
}; 4
```

1 The `/dev/envram` device driver constructs the `ENVRAM_READIO_D8`, `ENVRAM_READIO_D16`, and `ENVRAM_READIO_D32` interfaces from the `read_io_port` interface. These are convenience interfaces that call `read_io_port`, which is a generic interface that maps to a bus- and machine-specific interface that actually performs the task of reading the byte, word, longword, or quadword from a device register. Use of these interfaces to read data from a device register makes the device driver more portable across different buses, different CPU architectures, and different CPU types within the same architecture.

The `read_io_port` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the read operation originates. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  For this first argument, the `/dev/envram` driver performs a bitwise inclusive OR operation on the value stored in the `regbase` member of the pointer to the `envram_softc` structure and an address representing one of the device register offsets. For example: `ENVRAM_CSR`.

- The second argument specifies the width (in bytes) of the data to be read. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. For this second argument, the `/dev/envram` driver specifies 1, 2, and 4.

- The third argument specifies flags to indicate special processing requests. Currently, no flags are used.

  For this third argument, the `/dev/envram` driver specifies the value zero (0).

2 The `/dev/envram` device driver constructs the `ENVRAM_WRITEIO_D8`, `ENVRAM_WRITEIO_D16`, and `ENVRAM_WRITEIO_D32` interfaces from the `write_io_port` interface. These are convenience interfaces that call `write_io_port`, which is a generic interface that maps to a bus- and machine-specific interface that actually performs the task of writing the byte, word, longword, or quadword to a device register. Use of these interfaces to write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

The `write_io_port` interface takes four arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  For this first argument, the `/dev/envram` driver performs a bitwise inclusive OR operation on the value stored in the `regbase` member of the pointer to the `envram_softc` structure and an address

representing one of the device register offsets. For example: `ENVRAM_CSR`.

- The second argument specifies the width (in bytes) of the data to be written. Valid values are 1, 2, 3, 4, and 8. Not all CPU platforms support all of these values. For this second argument, the `/dev/envram` driver specifies 1, 2, and 4.
- The third argument specifies flags to indicate special processing requests. Currently, no flags are used.

  For this third argument, the `/dev/envram` driver specifies the value zero (0).
- The fourth argument specifies the data to be written to the specified device register in bus address space.

  For this fourth argument, the `/dev/envram` driver specifies a variable to be passed by the different device driver interfaces.

3 These are the forward declarations of the entry points for the `/dev/envram` device driver.

4 The `driver` structure for the `/dev/envram` driver is called `envramdriver`. The value zero (0) indicates that the `/dev/envram` driver does not make use of a specific member of the `driver` structure. The following list describes those members initialized to a nonzero value:

- The driver's probe interface, `envram_probe`.
- The driver's attach interface, `envram_attach`.
- The value `envram`, which is the name of the controller.
- The value `envram_info`, which references the array of pointers to the `controller` structures declared in the `/usr/sys/data/envram_data.c` file. You index this array with the controller number as specified in the `ctlr_num` member of the `controller` structure.

6.6 Autoconfiguration Support Section for the /dev/envram Device Driver

Table 6-2 lists the three interfaces implemented as part of the Autoconfiguration Support Section for the /dev/envram Device Driver along with the sections in the book where each is described.

Table 6-2: Autoconfiguration Support Section

Interface	Section
Implementing the envram_probe Interface	Section 6.6.1
Implementing the envram_attach Interface	Section 6.6.2
Implementing the envram_ssn Interface	Section 6.6.3

6.6.1 Implementing the envram_probe Interface

The `envram_probe` interface is called by the EISA bus configuration code at system boot time and performs the following tasks:

- Determines if the controller for the EISA bus NVRAM memory expansion board exists
- Allocates and fills in the `envram_softc` data structure associated with this controller
- If the controller exists on the system, enables the EISA bus NVRAM memory expansion board to handle reads and writes for the `/dev/presto` device driver

The `envram_probe` interface has the following implicit input:

```
   •
   •
   •
ctlr->physaddr
   •
   •
   •
```

This member stores the controller's base register physical address.

The following shows the code that implements `envram_probe`:

```
envram_probe(bus_io_handle, ctlr)
 io_handle_t bus_io_handle; 1
 struct controller *ctlr; 2

{
  register struct envram_softc *sc; 3
  u_int hw_id = 0; 4
  struct bus_mem mem; 5
  struct dma dma_p; 6
  u_long eisa_addr_mask = 0xffffffff; 7

    if (ctlr->ctlr_num > 0)
         return(0); 8

  sc = (struct envram_softc *)kalloc(sizeof(struct envram_softc)); 9

  if (!sc) 10
      return(0);
  bzero((char *)sc, sizeof(struct envram_softc)); 11
  envram_softc = sc; 12

  sc->ctlr = ctlr; 13

  sc->regbase = bus_io_handle; 14

  hw_id = ENVRAM_READIO_D32(ENVRAM_ID); 15

  if (hw_id != ENVRAM_ID_MASK) 16
     {
```

```
        printf("envram_probe: Failed to read ID register\n");

        kfree(sc, sizeof(struct envram_softc));
        return(0);
      }
    else
       printf("envram_probe: EISA NVRAM present\n");

  sc->cache_offset = ENVRAM_CACHE_OFFSET; 17

  if (get_config(ctlr, RES_MEM, "", &mem, 0)) { 18
      printf("envram probe error\n");
      return(0);
  }
  sc->cache_size = mem.size; 19
  sc->cache_base =  (u_long)mem.start_addr;
  sc->cache_phys_start = (u_long)(sc->cache_base + sc->cache_offset);
  sc->cache_kseg_start = (vm_offset_t)
                         (PHYS_TO_KSEG(sc->cache_phys_start&eisa_addr_mask));
  sc->saved_mem_sysmap = sc->cache_phys_start & ~eisa_addr_mask;

  sc->cache_size = sc->cache_size - EISA_DIAG_RESVED; 20

  if (get_config(ctlr, EISA_DMA, "", &dma_p, 0)) { 21
      printf("envram probe error dma channel\n");
      return(0);
  }

  if (dma_p.channel != 7 && dma_p.channel != 5) { 22
      printf("envram: invalid dma channel %d\n",dma_p.channel);
      return(0);
  }

  ENVRAM_WRITEIO_D8(ENVRAM_CTRL, EISA_ENABLE_BOARD); 23
  mb(); 24

  ENVRAM_WRITEIO_D16(ENVRAM_CSR,WRMEM); 25
  mb(); 26

  envram_read(sc->cache_phys_start-8,
              &sc->diag_status, 4); 27

  if (sc->diag_status & BOARD_FAILED) { 28
      printf("Envram diag reg 0x%x\n",sc->diag_status);
      sc->diag_status = 0;
  }
  else {
     sc->diag_status = 1;
  }
  return(1); 29
}
```

1 Specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space. The EISA bus configuration code passes this I/O handle to the driver's *xx*`probe` interface during device autoconfiguration. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or

subtract an offset from the I/O handle.

2 This is a pointer to the `controller` structure associated with the controller for this EISA bus NVRAM memory expansion board. The EISA bus configuration code makes the values stored in several members of the `ctlr` pointer available to the `/dev/envram` device driver. These members include `addr`, `physaddr`, and `conn_priv`.

3 The `envram_probe` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

4 The `envram_probe` interface initializes *`hw_id`* to the value zero (0) and later stores it in the EISA bus NVRAM memory expansion board's ID device register. This ID device register is defined in Section 6.2 as `ENVRAM_ID`.

5 The `envram_probe` interface declares a `bus_mem` data structure and calls it `mem`. The `bus_mem` structure describes memory characteristics for an EISA bus expansion board. The bus configuration code initializes the members of the `bus_mem` structure during device autoconfiguration. Device drivers call the `get_config` interface to obtain information stored in the members of the `bus_mem` data structure.

6 The `envram_probe` interface declares a `dma` data structure and calls it *`dma_p`*. The DMA channel information is encapsulated in the `dma` structure. Each member of this structure describes information related to the DMA channel. The `envram_probe` interface passes the address of this structure to the `get_config` interface. Section A.2.2 describes the members of the `dma` structure.

7 The `envram_probe` interface declares and initializes a variable to store the EISA bus address mask. It performs bit operations by using this address mask bit to determine the values stored in the `cache_kseg_start` and `saved_mem_sysmap` members.

8 The `envram_probe` interface determines if the controller number for the controller associated with this EISA bus NVRAM memory expansion board is greater than zero (0). If so, `envram_probe` returns the value zero (0) to the bus configuration code. This indicates that `envram_probe` could not complete the probe operation because there is support for only one EISA bus NVRAM memory expansion board. Changes must be made to the `/dev/presto` device driver interface before multiple units (memory boards) can be supported.

9 The `envram_probe` interface allocates memory for the `envram_softc` data structure by calling the `kalloc` interface.

If kalloc successfully allocates the memory for the envram_softc structure, it returns the address of the allocated memory. The sc pointer now points to this memory. Otherwise, kalloc returns the value zero (0).

10 If the memory test fails, the kernel did not allocate the memory and the probe operation exits.

11 The envram_probe interface calls the bzero interface to zero the number of bytes associated with the previously allocated envram_softc structure.

The bzero interface takes two arguments. The first argument specifies a pointer to a string of at least the number of bytes to be zeroed. For this first argument, envram_probe passes the sc pointer that now points to the previously allocated memory for the envram_softc structure. This is the address at which bzero starts to zero the bytes.

The second argument specifies the number of bytes to zero. For this second argument, envram_probe passes the previously allocated envram_softc structure.

12 The envram_probe interface sets the envram_softc structure for this EISA bus NVRAM memory expansion board to the pointer to the allocated memory.

13 The envram_probe interface sets the ctlr member of the sc pointer to the ctlr pointer associated with this EISA bus NVRAM memory expansion board. This assignment allows the /dev/envram driver to access the members of this controller structure through the softc structure associated with this EISA bus NVRAM memory expansion board.

14 The envram_probe interface stores the I/O handle passed in by the bus configuration code in the regbase member of the sc pointer. Section 6.3 shows how the /dev/envram driver uses regbase to construct the read and write interfaces used to read from and write data to the device registers.

15 The envram_probe interface calls the ENVRAM_READIO_D32 interface to read the ID device register for the EISA bus NVRAM memory expansion board. In this call, envram_probe passes ENVRAM_ID, which represents the offset to the ID device register. The ENVRAM_READIO_D32 interface returns the requested data, which in this call is a 32-bit value that identifies a valid ID device register.

Section 6.3 shows how the /dev/envram driver uses read_io_port to construct ENVRAM_READIO_D32.

16 The envram_probe interface checks the value returned by ENVRAM_READIO_D32 to determine if there is a valid ID device

register for this EISA bus NVRAM memory expansion board. If the ID device register is not valid, `envram_probe` calls `printf` to display an appropriate error message on the console terminal. It deallocates the EISA bus NVRAM memory expansion board's `envram_softc` structure by calling `kfree`.

The `envram_probe` interface returns the value zero (0) to the bus configuration code to indicate an error and that the probe operation failed.

If the ID device register is valid, `envram_probe` calls `printf` to display a message on the console terminal.

17 The `envram_probe` interface initializes the `cache_offset` member of the `envram_softc` structure associated with this EISA bus NVRAM memory expansion board. Section 6.2 shows that the `ENVRAM_CACHE_OFFSET` device register is defined in the `envram_reg.h` file.

18 The `envram_probe` interface calls `get_config` to obtain memory-related configuration data associated with the EISA bus NVRAM memory expansion board. If `get_config` cannot obtain the requested memory-related information, it prints an appropriate error message on the console terminal and returns the value zero (0) to the bus configuration code that indicates the driver's probe failed. Otherwise, it initializes the `cache_size`, `cache_base`, `cache_phys_start`, `cache_kseg_start`, and `saved_mem_sysmap` members of the `envram_softc` structure associated with this EISA bus NVRAM memory expansion board.

The `get_config` interface takes five arguments:

- The first argument specifies a pointer to the `controller` structure associated with the controller that connects to this device. The `get_config` interface obtains the device whose assigned configuration data you want through its controller's associated `controller` structure pointer. In this call, `envram_probe` passes the `controller` structure pointer passed to it by the bus configuration code.
- The second argument specifies the configuration data item you want to obtain for the specified device. In this call, `envram_probe` requests bus memory-related information by passing the constant `RES_MEM`.
- The third argument specifies a bus-specific argument. Not all buses need to pass a value to this argument. For the EISA bus, this argument specifies the function type string that appears in the device's `eisa_option` structure. In this call, `envram_probe` passes the null string.

- The fourth argument specifies a pointer to a structure appropriate for storing the requested data. In this call, `envram_probe` passes the address of the `bus_mem` data structure called `mem`.
- The fifth argument specifies a handle returned by `get_config` if there is more configuration data of the type requested in the *`config_item`* (the second) argument. You must pass the value zero (0) on the first call to `get_config`. On subsequent calls to `get_config` for this configuration data type, pass the value returned in the previous call to `get_config`.

  In this call, `envram_probe` passes the value zero (0).

19 The `envram_probe` interface initializes the bus memory-related members of the `sc` pointer to the values returned by `get_config` to the `bus_mem` structure. It also initializes several other members. The following list describes the initialized members:

- `cache_size`

  This member stores the size of the cache that the `/dev/presto` driver can use.
- `cache_base`

  This member now stores an I/O handle that indicates where the bus configuration code maps the memory block. The EISA bus configuration code sets the I/O handle during device autoconfiguration.
- `cache_phys_start`

  This member now stores the starting physical address of the NVRAM cache. The `envram_probe` interface uses the physical starting address stored in *`cache_base`* and the offset to cache to calculate the starting physical address of the NVRAM cache.
- `cache_kseg_start`

  This member now stores the starting kseg address of the cache used by the `/dev/presto` driver. The `envram_probe` interface obtains the starting kseg address by calling `PHYS_TO_KSEG`. The `PHYS_TO_KSEG` interface takes one argument: the physical address to convert to a buffer virtual address. In this call, the physical address to convert is stored in the `cache_phys_start` member.
- `saved_mem_sysmap`

  This member now stores the sysmap portion of the I/O handle.

20 This member now stores the amount of space the software diagnostics require and assures 2K-bytes of alignment for DMA operations.

21 The envram_probe interface calls get_config to obtain DMA channel-related configuration data associated with the EISA bus NVRAM memory expansion board. If get_config cannot obtain the requested DMA channel-related information, it prints an appropriate error message on the console terminal and returns the value zero (0) to the bus configuration code that indicates the driver's probe failed. Otherwise, it performs checks on the channel member of the dma structure. The envram_probe interface passes the same values to the first, third, and fifth arguments as it passed in the previous call to get_config. To request DMA channel-related information, envram_probe passes the constant RES_DMA to the second argument. The envram_probe interface passes the address of the dma data structure called dma_p0 to the fourth argument so that get_config can store in it the DMA channel-related information.

22 Specifies the DMA channel number or numbers that this EISA bus device can use. The EISA bus configuration code sets the channel number to a number in the range of 0–7. In this case, if channel is any value except for 5 and 7, then envram_probe displays an appropriate error message on the console terminal and returns the value zero (0) to the bus configuration code to indicate the probe failed.

23 The envram_probe interface enables the EISA bus NVRAM memory expansion board by calling ENVRAM_WRITEIO_D8. The ENVRAM_WRITEIO_D8 interface writes a byte (8 bits) to a device register located in the bus adddress space. Section 6.5 shows how the /dev/envram driver constructs ENVRAM_WRITEIO_D8 by using the write_io_port interface.

The ENVRAM_WRITEIO_D8 interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In this call, ENVRAM_WRITEIO_D8 performs a bitwise inclusive OR operation on the value stored in the regbase member of the sc pointer and the controller device register, ENVRAM_CTRL.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, ENVRAM_WRITEIO_D8 passes the EISA_ENABLE_BOARD bit as the data to be written.

24 The envram_probe interface calls mb after the write to perform a memory barrier.

25 The envram_probe interface enables the EISA bus NVRAM memory expansion board for writing by calling ENVRAM_WRITEIO_D16. The ENVRAM_WRITEIO_D16 interface writes a word (16 bits) to a device register located in the bus adddress space. Section 6.5 shows how the /dev/envram driver constructs ENVRAM_WRITEIO_D16 by using the write_io_port interface.

The ENVRAM_WRITEIO_D16 interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In this call ENVRAM_WRITEIO_D16 performs a bitwise inclusive OR operation on the value stored in the regbase member of the sc pointer and the control status register, ENVRAM_CSR.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, ENVRAM_WRITEIO_D16 passes the WRMEM bit as the data to be written.

26 The envram_probe interface calls mb after the write to perform a memory barrier.

27 To check the console diagnostic results, envram_probe calls the envram_read interface. Section 6.9.1 describes envram_read.

28 If the BOARD_FAILED bit is set, then the software diagnostic tests failed and envram_probe prints an appropriate error message on the console terminal. It also sets the diag_status member of the sc pointer to the value zero (0) to indicate that the EISA bus NVRAM memory expansion board did not pass the software diagnostic tests. Otherwise, envram_probe sets the diag_status member to the value 1 to indicate that the EISA bus NVRAM memory expansion board passed the software diagnostic tests.

29 The envram_probe interface returns the value 1 to the bus configuration code to indicate that the driver's probe operation is successful.

6.6.2 Implementing the envram_attach Interface

The `envram_attach` interface is called by the EISA bus configuration code at system boot time and performs the following tasks:

- Allocates resources for DMA data transfers.
- Initializes the `presto_interface0` data structure with the entry points for the `/dev/envram` driver. These entry points allow the `/dev/presto` driver to access the NVRAM data cache and the EISA bus NVRAM memory expansion board.
- Adjusts the sizes of the data block for DMA data transfers to be accepted by the `/dev/presto` driver.
- Calls `presto_init` to initialize the `/dev/presto` device driver.
- Calls `envram_ssn` to perform several operations on the system serial number in the hwrbp.

The `envram_attach` interface has the following implicit input:

```
   •
   •
   •
ctlr->physaddr
   •
   •
   •
```

This member stores the controller's base register physical address.

The following shows the code that implements `envram_probe`:

```
envram_attach(ctlr)
  struct controller *ctlr; 1
{

   register struct envram_softc *sc = envram_softc; 2

   if (dma_map_alloc(ENVRAM_XFER_SIZE,
                     sc->ctlr, &sc->sglp, 0) == 0) 3
     panic("envram: dma_map_alloc error\n");

  presto_interface0.nvram_status = eisa_nvram_status; 4
  presto_interface0.nvram_battery_status= eisa_nvram_battery_status;
  presto_interface0.nvram_battery_disable= eisa_nvram_battery_disable;
  presto_interface0.nvram_battery_enable= eisa_nvram_battery_enable;

  presto_interface0.nvram_ioreg_read = envram_read; 5
  presto_interface0.nvram_ioreg_write = envram_write;
  presto_interface0.nvram_block_read = envram_read;
  presto_interface0.nvram_block_write = envram_write;
  presto_interface0.nvram_ioreg_zero = envram_zero;
  presto_interface0.nvram_block_zero = envram_zero;

  presto_interface0.nvram_min_ioreg = sizeof(int); 6
```

```
    presto_interface0.nvram_ioreg_align = sizeof(int);
    presto_interface0.nvram_min_block = PRFSIZE;
    presto_interface0.nvram_block_align = PRFSIZE;

      presto_init(sc->cache_kseg_start, sc->cache_size,
                  ENVRAM_NOTMAPPED, ENVRAM_CACHED,
                  envram_ssn()); 7
}
```

1 This is a pointer to the `controller` structure associated with the controller for this EISA bus NVRAM memory expansion board. The EISA bus configuration code makes the values stored in several members of the `ctlr` pointer available to the `/dev/envram` device driver. These members include `addr`, `physaddr`, and `conn_priv`.

2 The `envram_attach` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

3 The `envram_attach` interface attempts to allocate resources for DMA data transfers by calling `dma_map_alloc`. The `dma_map_alloc` interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the allocation of system resources associated with DMA data transfers. Using this interface in DMA read and write operations makes the device driver more portable across different bus and CPU architectures. If `dma_map_alloc` cannot allocate the resources, `envram_attach` calls `panic` to cause a system crash.

The `dma_map_alloc` interface takes four arguments:

- The first argument specifies the maximum size (in bytes) of the data to be transferred during the DMA transfer operation. The kernel uses this size to determine the resources (mapping registers, I/O channels, and other software resources) to allocate.

  In this call, `envram_attach` passes the constant `ENVRAM_XFER_SIZE`, which represents the maximum size of the data to be transferred. Section 6.2 shows that this constant is defined in the `envram_reg.h` file.

- The second argument specifies a pointer to the `controller` structure associated with this controller. The interface uses this pointer to obtain the bus-specific interfaces and data structures that it needs to allocate the necessary mapping resources.

  In this call, `envram_attach` passes the `ctlr` pointer accessed through the `sc` pointer. Section 6.6.1 shows that `envram_probe`

sets this `ctlr` pointer.

- The third argument specifies a pointer to a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the `ba` and `bc` members of an `sg_entry` structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

  Typically, the device driver passes an argument of type `dma_handle_t *`. The `dma_map_alloc` interface returns to this variable the address of the DMA handle. The device driver uses this address in a call to `dma_map_load`.

  In this call, `envram_attach` simply passes the address of `sglp`, the DMA handle accessed through the `sc` pointer. Section 6.3 shows that this pointer is defined as part of the `envram_softc` structure.

- The fourth argument specifies special conditions that the device driver wants the system to perform. In this call, `envram_attach` passes the value zero (0) to indicate no flag settings.

4 The `envram_attach` interface initializes the members of the `presto_interface0` structure to the device driver interfaces that allow the `/dev/presto` device access to the NVRAM data cache. As the code shows, these interfaces are `eisa_nvram_status`, `eisa_nvram_battery_status`, `eisa_nvram_battery_disable`, and `eisa_nvram_battery_enable`. These interfaces are discussed in Section 6.7, Section 6.8.1, Section 6.8.2, and Section 6.8.3.

The `/usr/sys/include/sys/presto.h` file defines a data structure called `presto_interface`. It also declares an instance of this structure called `presto_interface0`. This file provides additional information on the members of the `presto_interface` structure.

5 The `envram_attach` interface initializes the members of the `presto_interface0` structure to the device driver interfaces that allow the `/dev/presto` device access to the EISA bus NVRAM memory expansion board. As the code shows, these interfaces are `envram_read`, `envram_write`, and `envram_zero`. These interfaces are discussed in Section 6.9.1, Section 6.9.2, and Section 6.10. See the `/usr/sys/include/sys/presto.h` file for additional information on these members of the `presto_interface` structure.

6 In the next four lines, `envram_attach` sets the minimum size of a small and large I/O device register data block. It also sets the byte

alignment restriction for an I/O device register block. The PRFSIZE constant is defined in /usr/sys/include/sys/presto.h as the largest buffer size (in bytes) handled by the /dev/presto driver. See the /usr/sys/include/sys/presto.h file for additional information on these members of the presto_interface structure.

7 The envram_attach interface calls presto_init to perform initialization tasks for the /dev/presto driver.

The presto_init interface takes five arguments:

- The first argument specifies the NVRAM address. In this call, envram_attach passes the starting kseg address of the cache used by the /dev/presto driver. Section 6.6.1 shows how envram_probe stores this kseg address in the cache_kseg_start member of the sc pointer. This is the envram_softc structure associated with this EISA bus NVRAM memory expansion board.
- The second argument specifies the size of the NVRAM cache. In this call, envram_attach passes the size stored in the cache_size member. Section 6.6.1 shows how envram_probe stores this size in the cache_size member of the sc pointer. This is the envram_softc structure associated with this EISA bus NVRAM memory expansion board.
- The third argument specifies whether the NVRAM is mapped. This argument takes one of the following values: 0 (the NVRAM is not mapped) or 1 (the NVRAM is mapped).

  In this call, envram_attach passes the constant ENVRAM_NOTMAPPED to indicate that the NVRAM is not mapped. Section 6.2 shows that this constant (and the ENVRAM_MAPPED constant) are defined in the envram_reg.h file.
- The fourth argument specifies which kernel segment space the /dev/presto driver uses. This argument has no meaning on Alpha AXP CPUs.
- The fifth argument specifies a unique machine (CPU) ID. In this call, envram_attach passes the name of an interface called envram_ssn. The envram_ssn interface returns a unique machine ID to presto_init. Section 6.6.3 describes the envram_ssn interface.

6.6.3 Implementing the envram_ssn Interface

The envram_attach interface passes envram_ssn as an argument to presto_init. The presto_init interface calls envram_ssn to obtain the machine (CPU) ID.

The envram_ssn interface performs the following tasks:

- Determines an unsigned 32-bit unique number from the system serial number in the hwrpb
- Converts the serial number from ASCII to a hexadecimal number
- Converts to 0xf modulo any letter over F (or f)

The envram_ssn returns the machine (CPU) ID located in the hwrpb. The following shows the code that implements envram_ssn:

```
envram_ssn()
{
  extern struct rpb *rpb; [1]

  u_int ssn = 0; [2]
  int i;
  char *cp;
  cp = rpb->rpb_ssn + 9; [3]
  if (*cp == '\0') { [4]
      cp = "NO System Serial Number"+8;
      printf("envram_ssn:  %s\n",cp-8);
  }
  for (i = 0 ; i < 8 ; i++, cp--){ [5]
          if (*cp < '9')
                  ssn += (*cp - '0' ) << (i*4);
          else if (*cp < 'G')
                  ssn += (*cp - 'A' + 0xa ) << (i*4);
          else if (*cp < 'a')
                  ssn += ( *cp % 0xf ) << (i*4);
          else if (*cp < 'g')
                  ssn += (*cp - 'a' + 0xa ) << (i*4);
          else
                  ssn += ( *cp % 0xf ) << (i*4);
      }
      return(ssn); [6]
}
```

[1] The envram_ssn interface declares a pointer to an rpb (restart parameter block) data structure. This data structure is defined in /usr/sys/include/arch/alpha/rpb.h. The rpb_ssn member stores the system serial number (ssn) for this CPU. The ssn consists of 10 ASCII characters.

[2] The envram_ssn interface declares a variable to store the system serial number and initializes it to the value zero (0).

[3] The envram_ssn interface stores the system serial number in the *cp* variable.

4 The `envram_ssn` interface prints an appropriate message on the console terminal if the system serial number stored in `rpb_ssn` is the null character.

5 The `envram_ssn` interface uses a `for` loop to parse the ASCII serial number and convert it to hexadecimal.

6 The `envram_ssn` interface returns the system serial number to the `presto_init` interface. The `presto_init` interface was called by `envram_attach`.

6.7 Status Section for the /dev/envram Device Driver

The `eisa_nvram_status` interface provides the `/dev/presto` device driver with the status of diagnostics run on the NVRAM. Section 6.6.1 shows that the `envram_probe` interface sets the `diag_status` member to indicate whether the EISA bus NVRAM memory expansion board passed software diagnostic tests. Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_status` member of the `presto_interface0` structure to `eisa_nvram_status`. This is the mechanism the `/dev/presto` device driver uses to call the interface that returns the diagnostic status of the NVRAM.

The following code shows the implementation of the `eisa_nvram_status` interface.

```
int eisa_nvram_status()
{

  register struct envram_softc *sc = envram_softc; [1]

  if (sc->diag_status) [2]
        return(NVRAM_RDONLY);
    else
        return(NVRAM_BAD);
}
```

[1] The `eisa_nvram_status` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

[2] The `eisa_nvram_status` interface checks the `diag_status` member of the `sc` pointer to determine whether this EISA bus NVRAM memory expansion board passed the software diagnostic tests. If the board passed these tests, `eisa_nvram_status` returns the constant `NVRAM_RDONLY` to the `/dev/presto` device driver. If the board failed these tests, `eisa_nvram_status` returns the constant `NVRAM_BAD` to the `/dev/presto` device driver.

6.8 Battery Status Section for the /dev/envram Device Driver

Table 6-3 lists the three interfaces implemented as part of the Battery Status Section for the /dev/envram Device Driver along with the sections in the book where each is described.

Table 6-3: Interfaces Implemented as Part of the Battery Status Section for the /dev/envram Device Driver

Part	Section
Implementing the eisa_nvram_battery_status Interface	Section 6.8.1
Implementing the eisa_nvram_battery_enable Interface	Section 6.8.2
Implementing the eisa_nvram_battery_disable Interface	Section 6.8.3

6.8.1 Implementing the eisa_nvram_battery_status Interface

The `eisa_nvram_battery_status` interface provides the `/dev/presto` device driver with the status of the battery on the NVRAM. Specifically, `eisa_nvram_battery_status` performs the following tasks:

- Fills in the battery-related members of the `nvram_batteries0` structure.
- Reads the control status register (CSR) for the EISA bus NVRAM memory expansion board to determine if the `BAT_FAIL` bit is set
- Reports the battery status to the `/dev/presto` device driver

Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_battery_status` member of the `presto_interface0` data structure to `eisa_nvram_battery_status`. This is the mechanism the `/dev/presto` device driver uses to call the interface that returns the status of the battery for the NVRAM.

The following shows the code that implements `eisa_nvram_battery_status`:

```
int eisa_nvram_battery_status()
{
  register struct envram_softc *sc = envram_softc; [1]

  nvram_batteries0.nv_nbatteries = 1; [2]
  nvram_batteries0.nv_minimum_ok = 1;
  nvram_batteries0.nv_primary_mandatory = 1;
  nvram_batteries0.nv_test_retries = 1;

  if ((ENVRAM_READIO_D16(ENVRAM_CSR) & BAT_FAIL)) [3]
      {
          nvram_batteries0.nv_status[0] = BATT_OK;
          return(0);
      }
  else
      {

          return(1); [4]
        }
}
```

[1] The `eisa_nvram_battery_status` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

[2] The `/usr/sys/include/sys/presto.h` file defines a data structure called `nvram_battery_info`. It also declares an instance of this structure called `nvram_batteries0`. The

`eisa_nvram_battery_status` interface fills in the members of the `nvram_batteries0` data structure on this and the following three lines. The following list briefly describes these members:

- `nv_nbatteries`

  Stores the number of batteries supported by the hardware. The `eisa_nvram_battery_status` interface sets this member to the value 1, indicating that the EISA bus NVRAM memory expansion board supports one battery.

- `nv_minimum_ok`

  Stores the minimum number of batteries that are enabled and that have enough power for use by the `/dev/presto` driver. The `eisa_nvram_battery_status` interface sets this member to the value 1 because the EISA bus NVRAM memory expansion board supports only one battery.

- `nv_primary_mandatory`

  Stores the value indicating whether the primary battery is operational. The `eisa_nvram_battery_status` interface sets this member to the value 1, indicating that the primary (and only) battery must be operational.

- `nv_test_retries`

  Stores the number of successive calls to `eisa_nvram_battery_status` for each battery check made by the `/dev/presto` device driver. The `eisa_nvram_battery_status` sets this member to the value 1, indicating one retry.

3 The `eisa_nvram_battery_status` interface determines if the EISA bus NVRAM memory expansion board's battery is operational. If the battery is operational, `eisa_nvram_battery_status` sets the `nv_status` member to `BATT_OK`. The `nv_status` member is defined in `/usr/sys/include/sys/presto.h` as an array of size `BATTCNT`.

Note the use of the `ENVRAM_READIO_D16` interface to read the CSR. In this call, `eisa_nvram_battery_status` passes `ENVRAM_CSR`, which represents the CSR. The `ENVRAM_READIO_D16` interface returns the requested data. Section 6.5 shows how the `/dev/envram` driver uses `read_io_port` to construct `ENVRAM_READIO_D16`.

4 The `eisa_nvram_battery_status` returns the value 1 to the `/dev/presto` device driver if the battery is not operational.

6.8.2 Implementing the eisa_nvram_battery_enable Interface

The `eisa_nvram_battery_enable` interface is called by the `/dev/presto` device driver and performs the following tasks:

- Enables writes to NVRAM memory and turns on the light emitting diode (LED)
- Disconnects the battery disconnect circuit
- Returns success or failure to the `/dev/presto` driver

Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_battery_enable` member of the `presto_interface0` data structure to `eisa_nvram_battery_enable`. This is the mechanism the `/dev/presto` device driver uses to call the interface that enables the battery on the EISA bus NVRAM memory expansion board.

The following shows the code that implements `eisa_nvram_battery_enable`:

```
int eisa_nvram_battery_enable()
{

  register struct envram_softc *sc = envram_softc; [1]

  ENVRAM_WRITEIO_D16(ENVRAM_CSR, WRMEM|SET_LED); [2]
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT); [3]
  mb(); [4]

  return(0); [5]
}
```

[1] The `eisa_nvram_battery_enable` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

[2] The `eisa_nvram_battery_enable` interface enables writes to NVRAM memory and turns on the light emitting diode (LED) by calling `ENVRAM_WRITEIO_D16`. The `ENVRAM_WRITEIO_D16` interface writes a word (the `WRMEM` and `SET_LED` CSR bit masks) to the CSR device register (represented by the `ENVRAM_CSR` offset) located in the bus address space. Section 6.5 shows how the `/dev/envram` driver constructs `ENVRAM_WRITEIO_D16` by using the `write_io_port` interface.

The `ENVRAM_WRITEIO_D16` interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O

space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

In this call, `ENVRAM_WRITEIO_D16` performs a bitwise inclusive OR operation on the value stored in the `regbase` member of the `sc` pointer and the CSR device register offset called `ENVRAM_CSR`.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, `ENVRAM_WRITEIO_D16` performs a bitwise inclusive OR operation on the CSR device register bit masks that enables writes to NVRAM memory and turns on the LED. This ORed value becomes the data to be written to the CSR device register offset.

3 The `eisa_nvram_battery_enable` interface disables the battery disconnect control bit by calling `ENVRAM_WRITEIO_D8`. The `ENVRAM_WRITEIO_D8` interface writes a byte (the `BAT_DISCON_BIT` battery disconnect bit mask) to the battery disconnect device register (represented by the `ENVRAM_BAT` offset) located in the bus address space. Section 6.5 shows how the `/dev/envram` driver constructs `ENVRAM_WRITEIO_D8` by using the `write_io_port` interface.

The `ENVRAM_WRITEIO_D8` interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In this call, `ENVRAM_WRITEIO_D8` performs a bitwise inclusive OR operation on the value stored in the `regbase` member of the `sc` pointer and the battery disconnect device register called `ENVRAM_BAT`.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, `ENVRAM_WRITEIO_D8` takes the battery disconnect device register bit mask that indicates the status of the battery disconnect circuit.

4 The `eisa_nvram_battery_enable` interface calls `mb` after the writes to perform a memory barrier.

5 The `eisa_nvram_battery_enable` interface returns to the `/dev/presto` driver the value zero (0) to indicate that it successfully enabled the battery for the EISA bus NVRAM memory expansion board.

6.8.3 Implementing the eisa_nvram_battery_disable Interface

The `eisa_nvram_battery_disable` interface performs the following tasks:

- Enables writes to NVRAM memory
- Performs a sequence of write operations to send sequence 11001 to the battery disconnect device register
- Returns success or failure to the `/dev/presto` driver

The `eisa_nvram_battery_disable` interface is called by the `/dev/presto` device driver when it needs to disable the battery on the EISA bus NVRAM memory expansion board. Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_battery_disable` member of the `presto_interface0` data structure to `eisa_nvram_battery_disable`. This is the mechanism the `/dev/presto` device driver uses to call the interface that enables the battery on the EISA bus NVRAM memory expansion board.

The following shows the code that implements `eisa_nvram_battery_disable`:

```
int eisa_nvram_battery_disable()
{

  register struct envram_softc *sc = envram_softc; [1]

  ENVRAM_WRITEIO_D16(ENVRAM_CSR,WRMEM); [2]
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT); [3]
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT);
  mb();

  return(0); [4]
}
```

[1] The `eisa_nvram_battery_disable` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

[2] The `eisa_nvram_battery_disable` interface enables writes to NVRAM memory by calling `ENVRAM_WRITEIO_D16`. The `ENVRAM_WRITEIO_D16` interface writes a word (the `WRMEM` CSR bit

mask) to the CSR device register (represented by the ENVRAM_CSR offset) located in the bus address space. Section 6.5 shows how the /dev/envram driver constructs ENVRAM_WRITEIO_D16 by using the write_io_port interface.

The ENVRAM_WRITEIO_D16 interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In this call ENVRAM_WRITEIO_D16 performs a bitwise inclusive OR operation on the value stored in the regbase member of the sc pointer and the CSR device register offset called ENVRAM_CSR.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, ENVRAM_WRITEIO_D16 takes the CSR device register bit mask that enables writes to NVRAM memory. This value becomes the data to be written to the CSR device register offset.

3 The eisa_nvram_battery_disable interface sends a sequence of 11001 to the battery disconnect device register by making five calls to ENVRAM_WRITEIO_D8. Note that eisa_nvram_battery_disable also performs a memory barrier after each write by calling mb.

The ENVRAM_WRITEIO_D8 interface writes a byte (the BAT_DISCON_BIT battery disconnect bit mask) to the battery disconnect device register (represented by the ENVRAM_BAT offset) located in the bus I/O space. Section 6.5 shows how the /dev/envram driver constructs ENVRAM_WRITEIO_D8 by using the write_io_port interface.

The ENVRAM_WRITEIO_D8 interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). This I/O handle references a device register in the bus address space where the write operation occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

  In this call ENVRAM_WRITEIO_D8 performs a bitwise inclusive OR operation on the value stored in the regbase member of the sc pointer and the battery disconnect device register offset called

ENVRAM_BAT.

- The second argument specifies the data to be written to the specified device register in bus address space. In this call, ENVRAM_WRITEIO_D8 takes the battery disconnect device register bit mask that indicates the status of the battery disconnect circuit. This value becomes the data to be written to the battery disconnect device register offset. Note that to send the zeroes in the 11001 sequence, eisa_nvram_battery_disable logically negates the bit.

4 The eisa_nvram_battery_disable interface returns to the /dev/presto driver the value zero (0) to indicate that it successfully disabled the battery for the EISA bus NVRAM memory expansion board.

6.9 Read and Write Device Section for the /dev/envram Device Driver

Table 6-4 lists the two interfaces implemented as part of the Read and Write Device Section for the /dev/envram Device Driver along with the sections in the book where each is described.

Table 6-4: Interfaces Implemented as Part of the Read and Write Device Section for the /dev/envram Device Driver

Part	Section
Implementing the envram_read Interface	Section 6.9.1
Implementing the envram_write Interface	Section 6.9.2

6.9.1 Implementing the envram_read Interface

The `envram_read` interface is called by `envram_probe` and the `/dev/presto` device driver and performs the following tasks:

- Converts the source address passed in by `envram_probe` and the `/dev/presto` driver from the NVRAM address into a physical address
- Copies data from the NVRAM bus address space to system memory

Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_ioreg_read` (read small pieces of NVRAM) and `nvram_block_read` (read large pieces of NVRAM) members of the `presto_interface0` data structure to `envram_read`. These members both point to the same interface, which means `envram_read` handles both small and large reads from the EISA bus NVRAM memory expansion board.

Note

An *xx*`read` interface implemented on DEC OSF/1 typically has three arguments: *`dev`*, *`uio`*, and *`flag`*. The reason that `envram_read` has different arguments is that it is not called directly from the I/O system as the result of a `read` system call. The read request from the I/O system is made to the `/dev/presto` driver's read entry point, which then calls `envram_read` to perform the actual read operation.

The following shows the code that implements `envram_read`:

```
void envram_read(source, dest, len)
  caddr_t source;   1
  caddr_t dest;     2
  u_int   len;      3
{

  register struct envram_softc *sc = envram_softc; 4

  io_copyin((io_handle_t)
          KSEG_TO_PHYS((u_long)source|sc->saved_mem_sysmap),
                       (vm_offset_t)dest,len); 5
}
```

1 Specifies the source address of the data to be written. Because this source address is passed in to `envram_read` by `envram_probe` and the `/dev/presto` device driver, the address format is a kernel segment (kseg) logical physical address.

2 Specifies the destination address of where to write the data. Because this destination address is passed in by `envram_probe` and the `/dev/presto` device driver, the format is a kernel segment (kseg) logical physical address.

3 Specifies the length of the block of data to be written. This length is passed in by `envram_probe` and the `/dev/presto` device driver.

4 The `envram_read` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

5 The `envram_read` interface calls `io_copyin` to copy data from bus address space to system memory. The `io_copyin` interface is a generic interface that maps to a machine-specific interface that actually performs the copy from bus address space to system memory. Using `io_copyin` to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The `io_copyin` interface takes three arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). For `io_copyin`, the I/O handle identifies the location in bus address space where the copy originates.

  In this call, the I/O handle is actually the physical address returned by `KSEG_TO_PHYS`. The `KSEG_TO_PHYS` interface takes one argument that specifies the buffer virtual address to convert to a physical address. In this call, the buffer virtual address is the result of ORing the source address and the value stored in the sysmap portion of the I/O handle. The `envram_probe` interface and the `/dev/presto` driver pass this source address to `envram_read`.

- The second argument specifies the kernel virtual address where `io_copyin` copies the data to in system memory. In this call, the *`dest`* `dest` argument contains this address in system memory, which is passed in by `envram_probe` and the `/dev/presto` driver.

- The third argument specifies the number of bytes in the data block to be copied. The interface assumes that the buffer associated with the data block is physically contiguous. In this call, the *`len`* argument contains this address in system memory, which is passed in by `envram_probe` and the `/dev/presto` driver.

6.9.2 Implementing the envram_write Interface

The `envram_write` interface is called by the `/dev/presto` device driver and performs the following tasks:

- Sets up the pointer to the `eisa_info` structure
- Converts the destination address passed in by the `/dev/presto` driver from a main memory virtual address into a physical address
- Performs byte alignment operations
- Allocates the resources needed for DMA operations
- Writes the data

Section 6.6.2 shows that the `envram_attach` interface sets the `nvram_ioreg_write` (write small pieces of NVRAM) and `nvram_block_write` (write large pieces of NVRAM) members of the `presto_interface0` data structure to `envram_write`. These members both point to the same interface. This means `envram_write` handles both small and large reads from the EISA bus NVRAM memory expansion board.

Note

An *xx*`write` interface implemented on DEC OSF/1 typically has three arguments: *`dev`*, *`uio`*, and *`flag`*. The reason that `envram_write` has different arguments is that it is not called directly from the I/O system as the result of a `write` system call. The read request from the I/O system is made to the `/dev/presto` driver's write entry point, which then calls `envram_write` to perform the actual write operation.

The `envram_write` interface is called by the `/dev/presto` device driver.

The following shows the code that implements `envram_write`:

```
void envram_write(source, dest, len)
  caddr_t source;     1
  caddr_t dest;       2
  u_int   len;        3
{

  register struct envram_softc *sc = envram_softc; 4
  vm_offset_t destptr; 5
  register int xfer; 6
  int retry; 7
  char *ddest = dest; 8

    if (len > 32) { 9

      destptr = KSEG_TO_PHYS(dest) - sc->cache_base; 10
```

```
        ddest = (char *)destptr; 11

        if (!(xfer = ENVRAM_XFER_SIZE - ((int)ddest & (ENVRAM_XFER_SIZE-1)))) 12
          xfer = ENVRAM_XFER_SIZE;

        if (xfer > len)
            xfer = len;

        if ((u_int)source/ENVRAM_ALLIGN !=
             ((u_int)source+xfer)/ENVRAM_ALLIGN) 13
              xfer = xfer - (((u_int)source+xfer) & (ENVRAM_ALLIGN-1));

        while (1) {

            if (!(dma_map_load(xfer, source, (struct proc *)0, 14
                               sc->ctlr, &sc->sglp, 0, DMA_OUT)))
                panic("envram: dma_map_load failure\n");

            ENVRAM_WRITEIO_D16(ENVRAM_DMA0,((u_int)(ddest-4) << 6)); 15
            ENVRAM_WRITEIO_D16(ENVRAM_DMA1,((u_int)ddest >> 5));

            ENVRAM_WRITEIO_D16(ENVRAM_CSR,SET_DREQ|WRMEM|SET_LED); 16
            mb(); 17
            len -= xfer; 18
            source += xfer;
            ddest += xfer;

            if (!(xfer = ENVRAM_XFER_SIZE - (((int)ddest 19
                  & (ENVRAM_XFER_SIZE-1)))))
                  xfer = ENVRAM_XFER_SIZE;

            if (xfer > len)
                xfer = len;
            if ((u_int)source/ENVRAM_ALLIGN != 20
               ((u_int)source+xfer)/ENVRAM_ALLIGN)
                xfer = xfer - (((u_int)source+xfer) & (ENVRAM_ALLIGN-1));

            retry = 10; 21
            while (--retry)
                if (!(ENVRAM_READIO_D16(ENVRAM_CSR) & SET_DREQ))
                    break;

            if (!length)
                break;

            if (!retry) 22
                panic("envram: DMA retry expired\n");
        }
        return;
    }

    io_copyout((vm_offset_t)source, (io_handle_t)
              (KSEG_TO_PHYS((u_long)dest)|sc->saved_mem_sysmap),
                          len); 23
}
```

1 Specifies the source address of the data to be written. Because this source address is passed in to `envram_write` by the `/dev/presto` device driver, the address format is a kernel segment (kseg) logical

physical address.

2 Specifies the destination address of where to write the data. Because this destination address is passed in by the `/dev/presto` device driver, the format is a kernel segment (kseg) logical physical address.

3 Specifies the length of the block of data to be written. This length is passed in by the `/dev/presto` device driver.

4 The `envram_write` interface declares a pointer to an `envram_softc` data structure and calls it `sc`. The `envram_softc` data structure allows the `/dev/envram` device driver's associated interfaces to share data. This data structure is defined in the `/usr/sys/data/envram_data.c` file. Section 6.3 describes the contents of this file, including the members of `envram_softc`.

5 This variable stores the physical address returned by `KSEG_TO_PHYS`.

6 This variable stores the size of each partial data transfer.

7 This variable stores the retry counter.

8 This variable stores the destination pointer.

9 The `envram_write` interface performs a DMA operation if the length of the data passed in by the `/dev/presto` device driver is greater than 32 bytes.

10 The `envram_write` interface calls `KSEG_TO_PHYS` to convert a kernel unmapped virtual address to a physical address. The `KSEG_TO_PHYS` interface takes one argument that specifies the buffer virtual address to convert to a physical address. In this call, the buffer virtual address is the result of subtracting the physical starting address where the memory block was mapped to from the location of the write. The `/dev/presto` driver passes this location to `envram_write`.

11 The `envram_write` interface stores the destination address in an internal variable.

12 This sequence of code aligns the destination to 1K.

13 This sequence of code aligns the source to 8K.

14 The `envram_write` interface loads and sets the allocated system resources for DMA data transfers by calling `dma_map_load`. If `dma_map_load` cannot load and set the allocated system resources, `envram_write` calls `panic` to cause a system crash. Section 6.6.2 shows that these resources were previously allocated by calling `dma_map_alloc`.

The `dma_map_load` interface takes seven arguments:

- The first argument specifies the maximum size (in bytes) of the data to be transferred during the DMA transfer operation. The kernel uses

this size to determine the resources (mapping registers, I/O channels, and other software resources) to allocate, load, and set.

In this call, `envram_write` passes the size stored in the *xfer* variable.

- The second argument specifies the virtual address where the DMA transfer occurs. The interface uses this address with the pointer to the `proc` structure to obtain the physical addresses of the system memory pages to load into DMA mapping resources.

  In this call, `envram_write` passes the address contained in *source*. The `/dev/presto` driver passed in this address.

- The third argument specifies a pointer to the `proc` structure associated with the valid context for the virtual address specified in *virt_addr*. The interface uses this pointer to retrieve the `pmap` that is needed to translate this virtual address to a physical address. If *proc_p* is equal to zero (0), the address is a kernel address.

  In this call, `envram_write` passes the value zero (0) to indicate that the address is a kernel address.

- The fourth argument specifies a pointer to the `controller` structure associated with this controller. The `dma_map_load` interface uses the pointer to get the bus-specific interfaces and data structures that it needs to load and set the necessary mapping resources.

  In this call, `envram_write` passes the `controller` structure pointer associated with this EISA bus NVRAM memory expansion board. Section 6.6.1 shows that `envram_probe` set this `controller` structure pointer in the softc structure.

- The fifth argument specifies a pointer to a handle to DMA resources associated with the mapping of an in-memory I/O buffer onto a controller's I/O bus. This handle provides the information to access bus address/byte count pairs. A bus address/byte count pair is represented by the `ba` and `bc` members of an `sg_entry` structure pointer. Device drivers can view this handle as the tag to the allocated system resources needed to perform a direct memory access (DMA) operation.

  In this call, `envram_write` passes the address of the DMA handle defined in the softc structure. Section 6.3 shows the declaration of the `sglp` member in the softc structure.

- The sixth argument specifies the maximum-size byte-count value that should be stored in the `bc` members of the `sg_entry` structures. In this call, `envram_write` passes the value zero (0).

- The seventh argument specifies special conditions that the device driver wants the system to perform. In this call, `envram_write`

passes the constant DMA_OUT. This constant indicates that dma_map_load should perform a DMA read operation from main core memory.

15 The envram_write interface calls ENVRAM_WRITEIO_D16 to set up the NVRAM address.

16 The envram_write interface calls ENVRAM_WRITEIO_D16 to start the transfer of data to the NVRAM.

17 The envram_write interface calls mb after the write to perform a memory barrier.

18 The envram_write interface performs several mathematical operations on the transfer size and the length, source, and destination pointer.

19 The envram_write interface sets up for the next DMA operation by aligning the destination to 1K. The NVRAM handles DMA operations only inside of a 1K aligned address range.

20 The envram_write interface aligns the source to 8K. The source is the system memory, thus there are 8K for DEC OSF/1 pages.

21 The envram_write interface sets up a while loop that causes the driver to spin on the SET_DREQ bit. If the hardware works, this bit should never be set.

22 If the retry expires, the hardware is broken and envram_write calls panic to cause a system crash.

23 The envram_write interface calls io_copyout to copy data from system memory to bus address space. The io_copyout interface is a generic interface that maps to a bus- and machine-specific interface that actually performs the copy to bus address space. Using io_copyout to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The io_copyout interface takes three arguments:

- The first argument specifies the kernel virtual address where the copy originates from in system memory.

  In this call, envram_write passes the address contained in *source*. The /dev/presto driver passed in this address.

- The second argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O space or memory space). For io_copyout, the I/O handle identifies the location in bus address space where the copy occurs. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

In this call, the I/O handle is actually the physical address returned by KSEG_TO_PHYS. The KSEG_TO_PHYS interface takes one argument that specifies the buffer virtual address to convert to a physical address. In this call, the buffer virtual address is the result of ORing the destination address and the value stored in the sysmap portion of the I/O handle. The /dev/presto driver passes this destination address to envram_write.

- The third argument specifies the number of bytes in the data block to be copied. The interface assumes that the buffer associated with the data block is physically contiguous. In this call, the *len* argument contains this address in system memory, which is passed in by the /dev/presto driver.

6.10 Zero NVRAM Section

The envram_zero interface is called by the /dev/presto device driver to zero (clear) a specified length of NVRAM starting at a specified address. Section 6.6.2 shows that the envram_attach interface sets the nvram_ioreg_zero (zero small pieces of NVRAM) and nvram_block_zero (zero large pieces of NVRAM) members of the presto_interface0 data structure to envram_zero. These members both point to the same interface. This means envram_zero zeros (clears) both small and large lengths from the EISA bus NVRAM memory expansion board.

The following code shows the implementation of the envram_zero interface:

```
void envram_zero(addr, len)
  caddr_t addr;   1
  u_int   len; 2
{

  register struct envram_softc *sc = envram_softc; 3

  io_zero((io_handle_t)
        KSEG_TO_PHYS((u_long)addr|sc->saved_mem_sysmap), len); 4
}
```

1 Specifies the starting address of the NVRAM for this EISA bus NVRAM memory expansion board to zero. Because this address is passed in by the /dev/presto, the format is a kernel segment (kseg) logical physical address.

2 Specifies the length of the block of data to be zeroed. This length is passed in by the /dev/presto device driver.

3 The envram_zero interface declares a pointer to an envram_softc data structure and calls it sc. The envram_softc data structure allows the /dev/envram device driver's associated interfaces to share data. This data structure is defined in the /usr/sys/data/envram_data.c file. Section 6.3 describes the contents of this file, including the members of envram_softc.

4 The envram_zero interface calls io_zero to zero a block of memory in bus address space. The io_zero interface is a generic interface that maps to a machine-specific interface that actually writes zeros to some location in bus address space. Using io_zero to perform the zero operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

The io_zero interface takes two arguments:

- The first argument specifies an I/O handle that you can use to reference a device register located in bus address space (either I/O

space or memory space). For `io_zero`, this I/O handle identifies the location in bus address space where the zero operation occurs.

In this call, the I/O handle is actually the physical address returned by `KSEG_TO_PHYS`. The `KSEG_TO_PHYS` interface takes one argument that specifies the buffer virtual address to convert to a physical address. In this call, the buffer virtual address is the result of ORing the source address and the value stored in the sysmap portion of the I/O handle. The `envram_probe` interface and the `/dev/presto` driver pass this source address to `envram_zero`.

- The second argument specifies the number of bytes in the data block to be zeroed. The interface assumes that the buffer associated with the data block is physically contiguous. In this call, the *`len`* argument contains the number of bytes, which is passed in by the `/dev/presto` driver.

EISA Bus Device Driver Configuration 7

Writing Device Drivers, Volume 1: Tutorial describes the device driver configuration models provided by the DEC OSF/1 operating system. Third-party device driver writers who want to ship loadable and static drivers to customers whose systems run DEC OSF/1 should use the third-party device driver configuration model. 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.

This chapter assumes that you are familiar with the device driver configuration models and the syntaxes used to populate configuration-related files. Discussion in this chapter is limited to the following topic related to configuring device drivers on the EISA bus: using an `eisa_data` file with the `mkeisadata` utility.

7.1 Using an eisa_data File with the mkeisadata Utility

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

```
[ #Entry= ]
BOARD_ID=board_id
[ FUNCTION=function ]
DRV_NAME=driver_name
```

[INTR_B4=*intr_b4_probe*]
[INTR_AFT=*intr_aft_attach*]
[TYPE=*type*]
[ADPT_CNFG=*adpt_config*]
[COMMENT=*comment*]

#Entry
Specifies the start of an entry in the eisa_data file.

board_id
Specifies the EISA bus option expansion board's product ID name. You set board_id to the character string that represents the option board's product ID name. The ID name consists of a 3-character manufacturer code, a 3-character product identifier, and a 1-character revision number. For example, the string ADP0002 identifies this as a second revision of an Adaptec ADP1742A SCSI option board.

function
Specifies the text that identifies the product up to a maximum of 79 characters. This text can include part numbers. The default is the null string (""). You set function to the character string that represents the product. For example, the string AHA1740 identifies this as the product name for a Digital ADP1742A SCSI option board.

driver_name
Specifies the name of the controlling device driver. You set driver_name to the character string that represents the name of the controlling device driver up to a maximum of 8 characters. This same character string also specifies the name of the controlling device driver in the system configuration file. For example, the string aha identifies this as the controlling device driver for the device connected to an ADP1742A SCSI option board.

intr_b4_probe
Specifies whether the device needs interrupts enabled before the bus configuration code calls the driver's *xx*probe interface. You can set intr_b4_probe to the value zero (0) or the value 1. 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 *xx*probe and *xx*cattach or *xx*dattach interfaces complete execution. You can set intr_aft_attach to the value zero (0) or the value 1. 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.

type
: Specifies the type of device. You can set `type` to the string `C` (the device is a controller) or `A` (the device is a bus or an adapter). The default value is `C`.

adpt_config
: Specifies the name of the bus (or adapter) configuration interface to call. You set `adpt_config` to the string that identifies the bus configuration interface, if `type` was set to `A` (the device is a bus or an adapter). Otherwise, do not set `adpt_config` to any value.

comment
: Specifies a C comment to appear at the end of an entry in the `eisa_option` data table. Your comment can be a maximum of 24 characters, including the C comment delimiters. The following example shows such a comment that you can pass to comment: `/* ADP1740A SCSI */`.

You can specify lines in the `eisa_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 shows a section of the `eisa_data` file with an entry for the `/dev/envram` driver:

```
#Entry
BOARD_ID=DEC2500
DRV_NAME=envram
INTR_B4=0
INTR_AFT=0
TYPE=C
COMMENT=/* EISA ENVRAM */
```

Each element in the entry initializes the corresponding member of the `eisa_option` data structure. The `config` program automatically invokes the `mkeisadata` utility through a `callout` keyword contained in the system configuration file. The `mkeisadata` utility adds the information contained in the driver's `eisa_data` file to a copy of the `eisa_option` table. The `config` program resumes when the `mkeisadata` subprocess completes.

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

EISA Bus-Specific Reference Information A

This appendix describes:

- Conventions for device driver-related reference (man) pages
- Reference pages for the EISA bus

A.1 Conventions for Device Driver-Related Reference (Man) Pages

The following are the categories of device driver-related reference pages included in this appendix:

- Header files
- Data structures
- Device driver interfaces

The following sections describe the conventions for each category.

A.1.1 Conventions for Header Files

The header file reference (man) pages can include the following sections:

Name

This section lists the name of the header file along with a summary description of its contents.

Location

This section presents the pathname for the header file. The pathname makes it easier for you to locate specific header files.

Description

This section briefly describes the contents of the header file.

When to Include

This section explains when to include a header file for block and character drivers.

Of Special Interest

This section lists specific structures, macros, constant values, and so forth that are of interest to device driver writers.

Related Information

This section lists related kernel interfaces, structures, system calls, and so forth.

A.1.2 Conventions for Data Structures

The data structure reference (man) pages can include the following sections:

Name

This section lists the name of the structure along with a summary description of its purpose.

Include File

This section lists the header file, including the path, where the structure is defined.

Synopsis

This section considers the following when describing structures:

- The C structure declaration is shown

  This occurs when you need to understand or need to initialize all of the members of a structure.

- The structure members are shown in a table

  This occurs when you need to understand or need to reference some of the members of a particular structure.

- No structure declaration or structure table is shown

  This occurs when the structure is opaque. This means the members of the structure are manipulated by the DEC OSF/1 operating system. You do not manipulate or reference any of the members.

Members

This section provides a short description of each member of the structure.

Description

This section gives more details about the purpose of the structure.

Related Information

This section lists related kernel interfaces, structures, system calls, and so forth.

A.1.3 Conventions for Device Driver Interfaces

The device driver interface reference (man) pages can include the following sections:

Name

This section lists the name of the driver interface along with a summary description of its purpose. In general, there is one interface described for each reference page. However, in some cases it makes sense to describe more than one interface on the same page if the interfaces are related. When this occurs, this section lists the names of all the interfaces it describes.

Entry Point

This section lists the structure or file where you specify the entry for the device driver interface.

Synopsis

This section shows the device driver interface function definition. The style used is that of the function definition, not the function call. This book assumes that you understand how to interpret the function definition and write an appropriate call for a specific interface. The presentation shown in the example is of the function definition:

```
int xxprobe(ctlr)
struct controller *ctlr;
```

The previous interface function definition gives you this information:

- Return type

  Gives the data type of the return value, if the interface returns data. If the

interface does not return a value, the function definition shows a `void` data type.

- Interface name

  Gives the driver interface name. In the example, *xx*`probe` is the name of the driver interface. The prefix *xx* indicates that this part of the name is variable. Replace it with the character prefix that represents the name of the device for which the driver is being written.

- Argument names

  Gives the name of each driver interface argument. In the example, the argument name is *ctlr*.

- Argument types

  Gives the types for each of the arguments. If you must define the type of the argument, a `void *` is used. In the example, this type is `struct controller *`. The term *driver interface* is used instead of *driver routine* to establish consistent terminology with that used for the kernel interfaces.

Arguments

This section provides descriptions for the arguments associated with a given driver interface. In most cases, argument descriptions begin with the word *specifies* to indicate that the driver writer passes the argument (with some specified value) to the driver interface.

Description

This section contains explanations of the tasks performed by the driver interface.

Notes

This section contains information about the driver interface pertinent to the device driver writer.

Return Values

This section describes the values that a given driver interface can return.

Related Information

This section lists related kernel interfaces, structures, system calls, and so forth.

A.2 Reference (Man) Pages for the EISA Bus

The following are the EISA bus-specific reference (man) pages that appear in this appendix:

- Header file specific to EISA bus device drivers
- Data structures specific to EISA bus device drivers
- Device driver interfaces specific to EISA device drivers

A.2.1 Header File Specific to EISA Bus Device Drivers

Table A-1 lists the header file related to EISA bus device drivers, along with a short description of its contents. Device drivers should include header files that use the relative pathname instead of the explicit pathname. For example, although `buf.h` resides in `/usr/sys/include/sys/buf.h`, device drivers should include it as:

```
<sys/buf.h>
```

Table A-1: Summary Description of Header File Specific to EISA Bus Device Drivers

Header File	Contents
`eisa.h`	Contains definitions specific to the EISA bus.

Name

`eisa.h` – Contains definitions specific to the EISA bus

Location

`/usr/sys/include/io/dec/eisa/eisa.h`

Description

The `eisa.h` file contains definitions specific to the EISA bus.

When to Include

You include this file in block and character device drivers that connect to the EISA bus.

Of Special Interest

This file defines data structures referenced by EISA bus device drivers and by the bus configuration code.

Related Information

Section A.2.2, Data Structures Specific to EISA Bus Device Drivers: `bus_mem`, `dma`, `eisa_option`, `e_port`, `irq`

A.2.2 Data Structures Specific to EISA Bus Device Drivers

Table A-2 lists the data structures specific to EISA bus device drivers, along with short descriptions of their contents.

Table A-2: Summary Description of Data Structures Specific to EISA Bus Device Drivers

Data Structure	Contents
`bus_mem`	Describes EISA bus memory characteristics.
`controller`	Represents an instance of a controller entity.
`dma`	Contains DMA channel information.
`eisa_option`	Contains EISA bus options.
`e_port`	Contains I/O port information.
`irq`	Contains EISA bus interrupt channel characteristics.

Name

bus_mem – Describes EISA bus memory characteristics

Include File

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

Synopsis

Member Name	Data Type
isram	int
decode	int
unit_size	int
size	long
start_addr	io_handle_t

Members

isram

Specifies whether the memory associated with the EISA bus expansion board is read-only memory (ROM) or random-access memory (RAM). This member is initialized by the bus configuration code during device autoconfiguration. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The memory associated with the EISA bus expansion board is ROM (that is, read only).
1	The memory associated with the EISA bus expansion board is RAM (that is, read/write).

`decode`

Specifies the number of address lines decoded by an EISA bus expansion board. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The number of address lines decoded by an EISA bus expansion board equals 20.
1	The number of address lines decoded by an EISA bus expansion board equals 24.
2	The number of address lines decoded by an EISA bus expansion board equals 32.

`unit_size`

Specifies the number of bits in the I/O port associated with this EISA bus expansion board. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	There is one byte (8 bits) in the I/O port associated with this EISA bus expansion board.
1	There is one word (16 bits) in the I/O port associated with this EISA bus expansion board.
2	There is one longword (32 bits) in the I/O port associated with this EISA bus expansion board.

`size`

Specifies the size of the memory block for this EISA bus expansion board. The EISA bus configuration code sets the size of the memory block up to a maximum of 64 megabytes during device autoconfiguration.

`start_addr`

Specifies an I/O handle that indicates where the bus configuration code maps the memory block. The EISA bus configuration code sets the I/O handle during device autoconfiguration.

Description

The `bus_mem` structure describes memory characteristics for an EISA bus expansion board. The bus configuration code initializes the members of the `bus_mem` structure during device autoconfiguration. Device drivers call the `get_config` interface to obtain information stored in the members of the `bus_mem` data structure.

Related Information

Section A.2.1, Header File Specific to EISA Bus Device Drivers: `eisa.h`

Name

`controller` – Represents an instance of a controller entity

Include File

`/usr/sys/include/io/common/devdriver.h`

Synopsis

Member Name	Data Type
`slot`	`int`
`physaddr`	`caddr_t`

Members

`slot`

Specifies the slot number that the EISA bus board is plugged into. The slot number ranges from 1– *n*, where *n* represents the maximum number of slots. This value corresponds to the one that you (the device driver writer) specified in the system configuration file or the `stanza.static` file fragment. In the system configuration file, you precede the slot number for the slot that the EISA bus board is plugged into with the `slot` keyword, as follows:

```
slot 1
```

The autoconfiguration software uses the syntax following the `slot` keyword in the system configuration file or the `stanza.static` file fragment to obtain this value. The bus configuration code obtains this value from the autoconfiguration software and uses it to initialize the `slot` member of the `controller` structure associated with this EISA bus device.

`physaddr`

Specifies the base physical address of the controller's I/O space.

Description

The `controller` structure represents an instance of a controller entity, one that connects logically to a bus. A controller can control devices that are directly connected or can perform some other controlling operation, such as a network interface or terminal controller operation.

Name

dma – Contains DMA channel information

Include File

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

Synopsis

Member Name	Data Type
channel	uint_t
is_shared	uint_t
xfer_size	uint_t
timing	uint_t
mode	uint_t

Members

channel
: Specifies the DMA channel number or numbers that this EISA bus device can use. The EISA bus configuration code sets the channel number to a number in the range of 0–7.

is_shared
: Specifies whether two devices can share the same DMA channel number. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The DMA channel number cannot be shared.
1	The DMA channel number can be shared.

`xfer_size`

Specifies the DMA device data transfer width. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
00	The DMA device data transfer size is 8 bits (byte).
01	The DMA device data transfer size is 16 bits (word).
10	The DMA device data transfer size is 32 bits (longword).

`timing`

Specifies the bus cycle type executed by the DMA controller during the data transfer. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
00	The bus cycle type is Default. This bus cycle type is compatible with ISA DMA devices.
01	The bus cycle type is Type A. This bus cycle type is used with higher performance ISA DMA devices to achieve faster data transfers.
10	The bus cycle type is Type B. Like bus cycle Type A, this type is used with higher performance ISA DMA devices to achieve faster data transfers.
11	The bus cycle type is Type C (burst). This bus cycle type is used with DMA devices that support EISA bus cycles to achieve the fastest data transfers.

`mode`

Specifies the transfer mode.

Description

The `dma` structure describes information related to the DMA channel. The bus configuration code initializes the members of the `dma` structure during device autoconfiguration. Device drivers call the `get_config` interface to obtain information stored in the members of the `dma` data structure.

Related Information

Section A.2.1, Header File Specific to EISA Bus Device Drivers: `eisa.h`

Name

eisa_option – Contains EISA bus options

Include File

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

Synopsis

Member Name	Data Type
board_id	char [EISA_IDNAMELEN + 1]
function	char [EISA_FUNCLEN + 1]
driver_name	char [EISA_NAMELEN + 1]
intr_b4_probe	int
intr_aft_attach	int
type	char
adpt_config	int (*adpt_config) ()

Members

board_id
: Specifies the EISA bus option expansion board's product ID name. You set board_id to the character string that represents the option board's product ID name. The ID name consists of a 3-character manufacturer code, a 3-character product identifier, and a 1-character revision number. For example, the string ADP0002 identifies this as a second revision of an Adaptec ADP1742A SCSI option board.

function
: Specifies the text that identifies the product up to a maximum of 79 characters. This text can include part numbers. The default is the null string (""). You set function to the character string that represents the product. For example, the string AHA1740 identifies this as the product name for a Digital ADP1742A SCSI option board.

driver_name
: Specifies the name of the controlling device driver. You set driver_name to the character string that represents the name of the controlling device driver up to a maximum of 8 characters. This same

character string also specifies the name of the controlling device driver in the system configuration file. For example, the string `aha` identifies this as the controlling device driver for the device connected to an ADP1742A SCSI option board.

`intr_b4_probe`
Specifies whether the device needs interrupts enabled before the bus configuration code calls the driver's *xx*`probe` interface. You can set `intr_b4_probe` to the value zero (0) or the value 1. 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 *xx*`probe` and *xx*`cattach` or *xx*`dattach` interfaces complete execution. You can set `intr_aft_attach` to the value zero (0) or the value 1. 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.

`type`
Specifies the type of device. You can set `type` to the string `C` (the device is a controller) or `A` (the device is a bus or an adapter). The default value is `C`.

`adpt_config`
Specifies the name of the bus (or adapter) configuration interface to call. You set `adpt_config` to the string that identifies the bus configuration interface, if `type` was set to `A` (the device is a bus or an adapter). Otherwise, do not set `adpt_config` to any value.

Description

The `eisa_option` structure contains EISA bus option information such as the bus option board's product ID and the name of the controlling device driver. An array of `eisa_option` structures is declared in the file `/usr/sys/data/eisa_option_data.c`.

Related Information

Section A.2.1, Header File Specific to EISA Bus Device Drivers: `eisa.h`

Name

`e_port` – Contains I/O port information

Include File

`/usr/sys/include/io/dec/eisa/eisa.h`

Synopsis

Member Name	Data Type
`base_address`	`vm_offset_t`
`numb_of_ports`	`uint_t`
`is_shared`	`uint_t`

Members

`base_address`
Specifies the base address of the I/O ports.

`numb_of_ports`
Specifies the number of ports in the range identified by the I/O handle.

`is_shared`
Specifies whether the device can share this device's I/O port. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The device cannot share the I/O port.
1	The device can share the I/O port.

Description

The `e_port` structure describes bus I/O port information. The bus configuration code initializes the members of the `e_port` structure during device autoconfiguration. Device drivers call the `get_config` interface to obtain information stored in the members of the `e_port` data structure.

Related Information

Section A.2.1, Header File Specific to EISA Bus Device Drivers: `eisa.h`

Name

`irq` – Contains EISA bus interrupt channel characteristics

Include File

`/usr/sys/include/io/dec/eisa/eisa.h`

Synopsis

Member Name	Data Type
`channel`	`uint_t`
`trigger`	`uint_t`
`is_shared`	`uint_t`

Members

`channel`
Specifies the interrupt channel number. The bus configuration code sets the interrupt channel number to a number in the range zero (0) to F.

`trigger`
Specifies whether the read-only memory (ROM) of the EISA bus expansion board initializes the interrupt controller to edge or level triggered. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The interrupt controller is initialized to edge triggered.
1	The interrupt controller is initialized to level triggered.

`is_shared`
Specifies whether the EISA bus expansion board can share this interrupt with other EISA bus expansion boards. The EISA bus configuration code sets this member to one of the following values during device autoconfiguration:

Value	Meaning
0	The EISA bus expansion board cannot share this interrupt.
1	The EISA bus expansion board can share this interrupt.

Description

The `irq` data structure specifies EISA bus interrupt channel characteristics assigned to a device. The bus configuration code initializes the members of the `irq` structure during device autoconfiguration. Device drivers call the `get_config` interface to obtain information stored in the members of the `irq` data structure.

Related Information

Section A.2.1, Header File Specific to EISA Bus Device Drivers: `eisa.h`

A.2.3 Device Driver Interfaces That EISA Device Drivers Use

Table A-3 lists the device driver interfaces specific to EISA bus device drivers, along with short descriptions of their contents.

Table A-3: Summary Description of Driver Interfaces that EISA Device Drivers Use

Driver Interface	Contents
xxprobe	Determines whether the device exists
xxslave	Checks that the device is valid for this controller

Name

xx`probe` – Determines whether the device exists

Entry Point

The `driver` structure

Synopsis

```
int xxprobe(bus_io_handle, ctlr)
io_handle_t bus_io_handle;
struct controller *ctlr;
```

Arguments

`bus_io_handle`

Specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space. The EISA bus configuration code passes this I/O handle to the driver's *`xx`*`probe` interface during device autoconfiguration. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

`ctlr`

Specifies a pointer to the `controller` structure associated with this device. The bus configuration code passes this pointer to the driver's *`xx`*`probe` interface. The device driver can reference hardware resource and other information contained in the `controller` structure pointer.

Description

An EISA bus device driver's *`xx`*`probe` interface performs the tasks necessary to determine if the device exists and is functional on a given system. At boot time, the EISA bus configuration code reads the configuration information saved by the EISA configuration utility (ECU) and performs checks to determine if the device is present before calling *`xx`*`probe` for statically configured drivers. The EISA bus configuration code calls the *`xx`*`probe` interface for each device that is defined according to specific rules discussed in Chapter 5. The *`xx`*`probe` interface typically checks some device control status register (CSR) to determine whether the physical device is present. 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. The Synopsis section shows the arguments associated with a probe interface for device drivers that operate on the EISA bus.

Notes

Device drivers pass the I/O handle to the following categories of interfaces, which are discussed in *Writing Device Drivers, Volume 1: Tutorial*. These interfaces can process the I/O handle to access the desired bus address space.

- CSR I/O access interfaces

  The CSR I/O access interfaces are `read_io_port` and `write_io_port`. These are generic interfaces that allow device drivers to read from and write to device registers. Using these interfaces to read data from and write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

- I/O copy interfaces

  The I/O copy interfaces are `io_copyin`, `io_copyio`, `io_copyout`, and `io_zero`. These are generic interfaces that allow device drivers to perform I/O copy operations. Using these interfaces to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

Return Values

The *xx*`probe` interface returns a nonzero value if the probe operation was successful. It returns the value zero (0) to indicate that the driver did not complete the probe operation.

Related Information

Section A.2.3, Device Driver Interfaces That EISA Device Drivers Use: *xx*`slave`

Name

xx`slave` – Checks that the device is valid for this controller

Entry Point

The `driver` structure

Synopsis

```
int xxslave(device, bus_io_handle)
struct device *device;
io_handle_t bus_io_handle;
```

Arguments

device
: Specifies a pointer to a `device` structure for this device. The bus configuration code passes this pointer to the driver's *xx*`slave` interface. The device driver can reference such information as the logical unit number of the device, whether the device is functional, and the bus number the device resides on.

bus_io_handle
: Specifies an I/O handle that you can use to reference a device register located in the EISA bus address space. This I/O handle is for the base of the device's slot-specific I/O address space. The EISA bus configuration code passes this I/O handle to the driver's *xx*`slave` interface during device autoconfiguration. You can perform standard C mathematical operations on the I/O handle. For example, you can add an offset to or subtract an offset from the I/O handle.

Description

A device driver's *xx*`slave` interface is called only for a controller that has slave devices connected to it. This interface is called once for each slave attached to the controller. You (or the system manager) specify the attachments of these slave devices for static device drivers in the system configuration file or `stanza.static` file fragment.

The arguments you pass to the slave interface differ according to the bus on which the driver operates. The Synopsis section shows the arguments associated with a slave interface for an EISA bus.

Notes

Device drivers pass the I/O handle to the following categories of interfaces, which are discussed in *Writing Device Drivers, Volume 1: Tutorial*. These interfaces can process the I/O handle to access the desired bus address space.

- CSR I/O access interfaces

  The CSR I/O access interfaces are `read_io_port` and `write_io_port`. These are generic interfaces that allow device drivers to read from and write to device registers. Using these interfaces to read data from and write data to a device register makes the device driver more portable across different bus architectures, different CPU architectures, and different CPU types within the same CPU architecture.

- I/O copy interfaces

  The I/O copy interfaces are `io_copyin`, `io_copyio`, `io_copyout`, and `io_zero`. These are generic interfaces that allow device drivers to perform I/O copy operations. Using these interfaces to perform the copy operation makes the device driver more portable across different CPU architectures and different CPU types within the same architecture.

Return Values

The *xx*`slave` interface returns a nonzero value if the device is present.

Related Information

Section A.2.3, Device Driver Interfaces That EISA Device Drivers Use: *xx*`probe`

EISA Bus Device Driver Source Listing B

This appendix contains the source listing for the `/dev/envram` device driver.

```
/*****************************************************
 *                                                   *
 *           Copyright (c) 1993 by                   *
 *   Digital Equipment Corporation, Maynard, MA      *
 *            All rights reserved.                   *
 *                                                   *
 * This software is furnished under the terms and    *
 * conditions of the TURBOchannel Technology         *
 * license and may be used and copied only in        *
 * accordance with the terms of such license and     *
 * with the inclusion of the above copyright         *
 * notice.  No title to and ownership of the         *
 * software is hereby transferred.                   *
 *                                                   *
 * The information in this software is subject to    *
 * change without notice and should not be           *
 * construed as a commitment by Digital Equipment    *
 * Corporation.                                      *
 *                                                   *
 * Digital assumes no responsibility for the use     *
 * or reliability of its software on equipment       *
 * which is not supplied by Digital.                 *
 *****************************************************/

/*****************************************************
 * envram_reg.h   Device Register Header File for    *
 * envram.c 13-April-1993                            *
 *****************************************************/


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

/*****************************************************
 * psgfix: wired up and ignored for power on.        *
 * Diagnostic soft register tells us if              *
```

```
 * diagnostics passed and the size of the board.   *
 ************************************************/

#define ENVRAM_DIAG_REGISTER   0x3f8 /* 1k - 1 */
#define BOARD_FAILED          0x00000008 /* Bit is set if board
                                          passed diags */
#define ENVRAM_DIAG_RESVED    0x400      /* The amount of space
                                          diagnostics require
                                          and assure 2K alignment
                                          for DMA */

/************************************************
 * Where firmware puts offset to cache last 32     *
 * bits in nvram 4mb space                         *
 ************************************************/

#define ENVRAM_CACHE_OFFSET  0x400 /* PSGFIX - this is my cookie */
                                   /* location */

/************************************************
 * CSR register bit mask definitions               *
 ************************************************/

#define SET_LED          0x0100  /* Turn LED on */
#define BAT_FAIL         0x0800  /* Indicated Battery failure */
#define WRMEM            0x2000  /* Enable writes to ENVRAM memory */
#define SET_DREQ         0x4000  /* Set DREQ for DMA */
#define DMA_CHAN_7       0X80    /* Channel 7 for DMA */
#define DMA_CHAN_5       0x40    /* Channel 5 for DMA */

/*******************************************************
 * Battery disconnect register bit mask defs            *
 *                                                      *
 *******************************************************/
#define BAT_DISCON_BIT  0x0080  /* Bit to hit with connect sequence */

/************************************************
 * EISA Control Register bit masks                 *
 ************************************************/

#define EISA_ENABLE_BOARD  0x1  /* EISA config enable - makes
                                   memory visible */

/************************************************
 * EISA ID register bit mask                       *
 ************************************************/

#define ENVRAM_ID_MASK  0x0025a310 /* EISA ID register bit mask */

/************************************************
 * Define constants used for communication with    *
 * the /dev/presto driver.                         *
 ************************************************/

#define ENVRAM_MAPPED    1   /* Buffer is mapped */
#define ENVRAM_NOTMAPPED 0   /* Buffer is not mapped */
#define ENVRAM_CACHED    1   /* Use kseg space */
#define ENVRAM_NOTCAHCED 0   /* Use a cached space */

/*****************************************************
```

```
 * Define allignment boundaries                                *
 *                                                             *
 ***************************************************************/

#define ENVRAM_XFER_SIZE 1024 /* Maximum DMA transfer size
                                 to NVRAM module */
#define ENVRAM_ALLIGN    8192 /* DMA allignment required */

/*****************************************************
 *                                                   *
 *           Copyright (c) 1993 by                   *
 *   Digital Equipment Corporation, Maynard, MA      *
 *            All rights reserved.                   *
 *                                                   *
 * This software is furnished under the terms and    *
 * conditions of the TURBOchannel Technology         *
 * license and may be used and copied only in        *
 * accordance with the terms of such license and     *
 * with the inclusion of the above copyright         *
 * notice.  No title to and ownership of the         *
 * software is hereby transferred.                   *
 *                                                   *
 * The information in this software is subject to    *
 * change without notice and should not be           *
 * construed as a commitment by Digital Equipment    *
 * Corporation.                                      *
 *                                                   *
 * Digital assumes no responsibility for the use     *
 * or reliability of its software on equipment       *
 * which is not supplied by Digital.                 *
 *****************************************************/
/*************************************************************
 * envram_data.c   data.c file for envram.c 13-April-1994    *
 *                                                           *
 *************************************************************/

/*****************************************************
 * Digital EISA non-voltile RAM driver  (DEC2500)    *
 *****************************************************/


/*****************************************************
 * Define the softc structure for the EISA NVRAM     *
 * driver                                            *
 *****************************************************/
struct      envram_softc {
 io_handle_t regbase;           /* base address for registers */
 io_handle_t cache_phys_start; /* Physical start address of NVRAM cache
                                  of NVRAM cache */
 io_handle_t cache_base;        /* base address of NVRAM in */
                                  EISA address space */
 vm_offset_t cache_kseg_start; /* KSEG start addr of
                                  the presto cache */
 u_long saved_mem_sysmap;       /* sysmap portion of mem io_handle_t */
 u_int cache_size;              /* Size of NVRAM cache */
 u_int cache_offset;            /* Offset to the first nvram location
                                   from start of EISA slot address */
 io_handle_t diag_status;       /* If the board passed diags or not */
 dma_handle_t sglp;             /* Pointer to byte address/byte count pair */
 struct controller *ctlr;       /* Pointer to nvram controller */
```

```
};

struct  envram_softc *envram_softc;
struct  controller *envram_info[NENVRAM];


/*************************************************
 * envram.c  Driver for presto device  13-Apr-1994 *
 *                                                 *
 * The /dev/envram device driver is an example     *
 * driver that performs read/write services for    *
 * the /dev/presto device driver.                  *
 *                                                 *
 *************************************************/
/*************************************************
 * The /dev/envram device driver written by        *
 * Randy Arnott, Paul Grist, and Randall Brown.    *
 *************************************************/

/*************************************************
 *            Include Files Section                *
 *************************************************/

#include "envram.h" /* Driver header file created by config */
#include <vm/vm_kern.h>
#include <sys/presto.h> /* Definitions for the /dev/presto
                                device driver */
#include <io/common/devdriver.h>
#include <io/dec/eisa/eisa.h> /* Header file specific to the
                                 EISA bus */
#include <data/envram_data.c> /* data.c file specific to the
                                 /dev/envram driver */
#include <machine/rpb.h>
#include <io/dec/eisa/envram_reg.h> /* Device register header file */


/*************************************************
 * EISA NVRAM I/O register Read/Write Macros       *
 *                                                 *
 * These macros are built using the standard I/O   *
 * bus interfaces read_io_port and write_io_port.  *
 * The base address is referenced from             *
 * sc->regbase.  Simply || the register offset.    *
 *************************************************/

#define ENVRAM_READIO_D8(a) \
 read_io_port((io_handle_t)sc->regbase | a, 1, 0))
#define ENVRAM_READIO_D16(a) \
 read_io_port((io_handle_t)sc->regbase | a, 2, 0))
#define ENVRAM_READIO_D32(a) \
 read_io_port((io_handle_t)sc->regbase | a, 4, 0))

#define ENVRAM_WRITEIO_D8(a,d) \
 write_io_port((io_handle_t)sc->regbase | a, 1, 0, d))
#define ENVRAM_WRITEIO_D16(a,d) \
 write_io_port((io_handle_t)sc->regbase | a, 2, 0, d))
#define ENVRAM_WRITEIO_D32(a,d) \
write_io_port((io_handle_t)sc->regbase | a, 4, 0, d))

/*************************************************
```

```
 * Declarations Section                                    *
 ************************************************/

/************************************************
 * Do forward declaration of driver entry points   *
 * and define information structures for driver    *
 * structure definition and initialization below.  *
 ************************************************/

int envram_probe(), envram_attach(), eisa_nvram_status();
int eisa_nvram_battery_enable(), eisa_nvram_battery_disable();
void envram_read(), envram_write(), envram_zero();

struct driver envramdriver = {
        envram_probe,     /* probe */
        0,                /* slave */
        envram_attach,    /* cattach */
        0,                /* dattach */
        0,                /* go */
        0,                /* addr_list */
        0,                /* dev_name */
        0,                /* dev_list */
        "envram",         /* ctlr_name */
        envram_info,      /* ctlr_list */
        0,                /* xclu */
        0,                /* addr1_size */
        0,                /* addr1_atype */
        0,                /* addr2_size */
        0,                /* addr2_atype */
        0,                /* ctlr_unattach */
        0                 /* dev_unattach */
};
/************************************************
 * External references                             *
 ************************************************/


/*********************************************************************
 * Autoconfiguration Support Section                                 *
 *********************************************************************/

/*********************************************************************
 * The autoconfiguration support section for the /dev/envram device *
 * device driver contains these interfaces:                          *
 *                                                                   *
 *   o envram_probe                                                  *
 *   o envram_attach                                                 *
 *********************************************************************/

/*********************************************************************
 * INTERFACE NAME: envram_probe                                      *
 *                                                                   *
 * FUNCTIONAL DESCRIPTION:                                           *
 *                                                                   *
 * The envram_probe interface:                                       *
 *                                                                   *
 *   o Determines if the controller for the EISA bus NVRAM memory   *
 *     board exists                                                  *
 *   o Allocates and fills in the driver's softc data structure     *
 *   o Enables the EISA bus NVRAM memory board to handle reads and  *
 *     writes, if the controller exists on the system                *
```

```
 *                                                                    *
 * CALLED BY: Bus configuration code at boot time                     *
 *                                                                    *
 * FORMAL PARAMETERS:                                                 *
 *                                                                    *
 *   o addr -- Base physical address of the EISA bus NVRAM memory     *
 *             board registers                                        *
 *   o ctlr -- Pointer to the controller structure for this device    *
 *                                                                    *
 * IMPLICIT INPUTS:                                                   *
 *                                                                    *
 *   o ctlr->slot -- EISA slot number of this controller              *
 *   o ctlr->conn_priv[0] -- Pointer to eisainfo structure            *
 *   o eisainfo->irq.intr.intr_num -- EISA interrupt level for this   *
 *                                    controller                      *
 *   o eisainfo->irq.intr.trigger -- EISA interrupt high/low flag,    *
 *                                   0 = edge (high),                 *
 *                                   1 = level (low)                  *
 *   o ctlr->addr -- KSEG address of controller's base register       *
 *   o ctlr->physaddr -- Controller's base register physical          *
 *                       address                                      *
 *                                                                    *
 * IMPLICIT OUTPUTS: None                                             *
 *                                                                    *
 * RETURN VALUE:                                                      *
 *                                                                    *
 *      Success:        Size of the softc structure.                  *
 *      Failure:        NULL.                                         *
 **********************************************************************/

envram_probe(bus_io_handle, ctlr)
 io_handle_t bus_io_handle; /* Base physical address of the EISA
                            bus NVRAM registers*/
 struct controller *ctlr; /* Pointer to controller structure */

{
/* Pointer to softc structure */
  register struct envram_softc *sc;
  u_int hw_id = 0; /* Stores EISA bus ID register */
  struct bus_mem mem; /* bus_mem structure describes EISA */
                      bus memory characteristics */
  struct dma dma_p; /* dma structure */
  u_long eisa_addr_mask = 0xffffffff; /* EISA address mask */

/**********************************************************************
 * Determine if this is unit 0.  Currently, there is support for      *
 * only one EISA bus NVRAM memory expansion board.   *
 * Changes must be made to the /dev/presto device driver      *
 * interface before multiple units (memory boards) can be supported. *
 **********************************************************************/

    if (ctlr->ctlr_num > 0)
         return(0);
/**********************************************************************
 * Allocate memory for softc structure.                               *
 **********************************************************************/
  sc = (struct envram_softc *)kalloc(sizeof(struct envram_softc));

  if (!sc)
      return(0);
```

```
  bzero((char *)sc, sizeof(struct envram_softc));
  envram_softc = sc;

/*****************************************************************
 * Save the ctrl struct pointer in the driver's softc structure.    *
 *****************************************************************/
  sc->ctlr = ctlr;

/*****************************************************************
 ******** I/O Register Access Scheme *****************************
 *                                                               *
 * The /dev/envram device driver uses a logical addressing scheme     *
 * for I/O register access.  This logical addressing scheme:          *
 *                                                               *
 *   o Uses the envram_softc data structure to store the io_handle_t  *
 *     for the physical base address                             *
 *   o Passes the offset of the target register to the ENVRAM_READIO- *
 *     and ENVRAM_WRITE_IO-related macros, which perform the I/O      *
 *     access.                                                   *
 *                                                               *
 * The ENVRAM_READIO- and ENVRAM_WRITE_IO-related macros are defined  *
 * in envram_data.c.  They use the read_io_port and write_io_port     *
 * interfaces and OR the offset with the sc->regbase value.  The      *
 * sc->regbase value is the per-option physical base address of the   *
 * EISA NVRAM I/O registers.                                     *
 *****************************************************************/

/*****************************************************************
 * This next sequence of code gets the controller's base address and  *
 * saves the slot number.  For EISA bus controllers, the physical     *
 * address of the adapter base register can be calculated by shifting *
 * the EISA slot number by 12 bits.  It is also passed in the         *
 * controller structure.                                         *
 *****************************************************************/
  sc->regbase = bus_io_handle;

/*****************************************************************
 * Read the controller's ID register to ensure that it is actually  *
 * a DEC 2500.                                                   *
 *****************************************************************/

  hw_id = ENVRAM_READIO_D32(ENVRAM_ID);

  if (hw_id != ENVRAM_ID_MASK)
     {
       printf("envram_probe: Failed to read ID register\n");
       /* deallocate sc resources */
       kfree(sc, sizeof(struct envram_softc));
       return(0);
     }
   else
      printf("envram_probe: EISA NVRAM present\n");

/*****************************************************************
 ********* EISA Configuration ************************************
 *****************************************************************/

/*****************************************************************
 * Set up the members of the softc structure for the location and   *
 * offset of NVRAM cache for the /dev/presto device driver.  The    *
```

```
 * starting io_handle_t of the NVRAM bus memory is available from   *
 * the bus support information.                                     *
 ********************************************************************/

  sc->cache_offset = ENVRAM_CACHE_OFFSET; /* 1MB and 0x400 offset */

/*********************************************************************
 * Get nvram size and io_handle_t of starting address by calling    *
 * the get_config interface.                                        *
 ********************************************************************/

  if (get_config(ctlr, RES_MEM, "", &mem, 0)) {
      printf("envram probe error\n");
      return(0);
  }
  sc->cache_size = mem.size;
  sc->cache_base =  (u_long)mem.start_addr;
  sc->cache_phys_start = (u_long)(sc->cache_base + sc->cache_offset);
  sc->cache_kseg_start = (vm_offset_t)
                         (PHYS_TO_KSEG(sc->cache_phys_start&eisa_addr_mask));
  sc->saved_mem_sysmap = sc->cache_phys_start & ~eisa_addr_mask;

/*********************************************************************
 * Account for the diagnostic space.                                *
 ********************************************************************/
  sc->cache_size = sc->cache_size - EISA_DIAG_RESVED;

/*********************************************************************
 * Get nvram dma channel information                                *
 ********************************************************************/
  if (get_config(ctlr, EISA_DMA, "", &dma_p, 0)) {
      printf("envram probe error dma channel\n");
      return(0);
  }

/*********************************************************************
 * Fail the probe if invalid dma channel.                           *
 ********************************************************************/
  if (dma_p.channel != 7 && dma_p.channel != 5) {
      printf("envram: invalid dma channel %d\n",dma_p.channel);
      return(0);
  }

/*********************************************************************
 * Enable the module by calling the ENVRAM_WRITEIO_D8 interface.    *
 * Call the mb interface after the write completes.                 *
 ********************************************************************/

  ENVRAM_WRITEIO_D8(ENVRAM_CTRL, EISA_ENABLE_BOARD);
  mb();

/*********************************************************************
 * Initialize the CSR and enable the NVRAM memory for writes        *
 ********************************************************************/

  ENVRAM_WRITEIO_D16(ENVRAM_CSR,WRMEM);
  mb();

/*********************************************************************
 * Check the console diagnostic results                             *
```

```
 ************************************************************************/

  envram_read(sc->cache_phys_start-8,
              &sc->diag_status, 4);

  if (sc->diag_status & BOARD_FAILED) {
      printf("Envram diag reg 0x%x\n",sc->diag_status);
      sc->diag_status = 0;
  }
  else {
     sc->diag_status = 1;
  }
  return(1);
}


/************************************************************************
 * INTERFACE NAME:  envram_attach                                       *
 *                                                                      *
 * FUNCTIONAL DESCRIPTION:                                              *
 *                                                                      *
 * The envram_attach interface defines the /dev/envram driver           *
 * interfaces to the /dev/presto device driver.  It also calls          *
 * presto_init to initialize the /dev/presto device driver.             *
 *                                                                      *
 * CALLED BY: Bus configuration code at boot time                       *
 *                                                                      *
 * FORMAL PARAMETERS:                                                   *
 *                                                                      *
 *   o ctlr -- Pointer to the controller structure for this device      *
 *   o dev -- Pointer to the device structure for this device           *
 *                                                                      *
 * IMPLICIT INPUTS:                                                     *
 *                                                                      *
 *   o ctlr->slot -- EISA slot number of this controller                *
 *   o ctlr->conn_priv[0] -- Pointer to eisainfo structure              *
 *   o eisainfo->irq.intr.intr_num -- EISA interrupt level for this     *
 *                                    controller                        *
 *   o eisainfo->irq.intr.trigger -- EISA interrupt high/low flag,      *
 *                                   0 = edge (high),                   *
 *                                   1 = level (low)                    *
 *   o ctlr->addr -- KSEG address of controller's base register         *
 *   o ctlr->physaddr -- Controller's base register physical            *
 *                       address                                        *
 *   o softc structure is available with all EISA NVRAM values          *
 *                                                                      *
 * IMPLICIT OUTPUTS: None                                               *
 *                                                                      *
 * RETURN VALUE: None                                                   *
 ************************************************************************/

envram_attach(ctlr)
  struct controller *ctlr; /* Pointer to controller structure */
{
   /* Pointer to softc structure */
   register struct envram_softc *sc = envram_softc;

/************************************************************************
 * Allocate resources for DMA data transfers by calling the             *
 * dma_map_alloc interface.                                             *
```

```
 ************************************************************************/

   if (dma_map_alloc(ENVRAM_XFER_SIZE,
                     sc->ctlr, &sc->sglp, 0) == 0)
     panic("envram: dma_map_alloc error\n");

/************************************************************************
 * The following code initializes the presto_interface0 data        *
 * structure to the /dev/envram device driver interfaces that allow *
 * the /dev/presto device driver to access the NVRAM data cache.    *
 ************************************************************************/

  presto_interface0.nvram_status = eisa_nvram_status;
  presto_interface0.nvram_battery_status= eisa_nvram_battery_status;
  presto_interface0.nvram_battery_disable= eisa_nvram_battery_disable;
  presto_interface0.nvram_battery_enable= eisa_nvram_battery_enable;

/************************************************************************
 * The following code initializes the presto_interface0 data        *
 * structure to the /dev/envram device driver interfaces that allow *
 * the /dev/presto device driver to access the EISA NVRAM.  Note    *
 * that the ioreg and block interfaces are all expected to have     *
 * these formal parameters: src, dest, and count.  However, the     *
 * envram_zero interface has these formal parameters: addr and      *
 * length.                                                          *
 ************************************************************************/

  presto_interface0.nvram_ioreg_read = envram_read;
  presto_interface0.nvram_ioreg_write = envram_write;
  presto_interface0.nvram_block_read = envram_read;
  presto_interface0.nvram_block_write = envram_write;
  presto_interface0.nvram_ioreg_zero = envram_zero;
  presto_interface0.nvram_block_zero = envram_zero;

/************************************************************************
 * The EISA granularity is a byte, but force the use of 32-bit      *
 * quantities for performance reasons.                              *
 ************************************************************************/


/************************************************************************
 * Minimum size of a "small" ioreg data block                       *
 ************************************************************************/
  presto_interface0.nvram_min_ioreg = sizeof(int);

/************************************************************************
 * Byte alignment restriction for ioreg block                       *
 ************************************************************************/
  presto_interface0.nvram_ioreg_align = sizeof(int);
/************************************************************************
 * Minimum size of a "large" block data transfer in bytes.          *
 ************************************************************************/
  presto_interface0.nvram_min_block = PRFSIZE;

/************************************************************************
 * Byte alignment restriction for block data transfers              *
 ************************************************************************/
  presto_interface0.nvram_block_align = PRFSIZE;

/************************************************************************
```

```
 * PRFSIZE = smallest fragment size for buffer (1K)                    *
 ************************************************************************/


/************************************************************************
 * Call the presto_init interface to initialize the /dec/presto        *
 * device driver interfaces.                                           *
 ************************************************************************/


/************************************************************************
 * RMA - fix Need unique sysid without etherrom!!                      *
 * Call the presto_init interface to perform initialization tasks      *
 * the /dev/presto device driver.                                      *
 ************************************************************************/
    presto_init(sc->cache_kseg_start, sc->cache_size,
                ENVRAM_NOTMAPPED, ENVRAM_CACHED,
                envram_ssn());
}


/************************************************************************
 * INTERFACE NAME:  envram_ssn                                         *
 *                                                                     *
 * FUNCTIONAL DESCRIPTION:                                             *
 *                                                                     *
 * The envram_ssn interface determines an unsigned 32-bit unique       *
 * number from the system serial number in the hwrbp.  It converts     *
 * the serial number from ASCII to a hexadecimal number.  It also      *
 * converts to 0xf modulo any letter over 'F' (or f).                  *
 *                                                                     *
 * CALLED BY:                                                          *
 *                                                                     *
 * The envram_attach interface passes envram_ssn as an argument to     *
 * presto_init.  The presto_init interface calls envram_ssn to         *
 * obtain the machine (CPU) ID.                                        *
 *                                                                     *
 * FORMAL PARAMETERS: None                                             *
 *                                                                     *
 * IMPLICIT INPUTS: None                                               *
 *                                                                     *
 * IMPLICIT OUTPUTS: None                                              *
 *                                                                     *
 * RETURN VALUE: The machine (CPU) ID.                                 *
 ************************************************************************/

envram_ssn()
{
  extern struct rpb *rpb; /* Pointer to restart parameter */
                          /* block (rpb) structure */
  u_int ssn = 0; /* Stores the system serial number */
  int i;
  char *cp;

/* Grab the system serial number */
  cp = rpb->rpb_ssn + 9;

/************************************************************************
 * Display an appropriate message on the console terminal if the       *
 * system serial number equals the null character.                     *
```

```
 ************************************************************************/

  if (*cp == '\0') {
      cp = "NO System Serial Number"+8;
      printf("envram_ssn:  %s\n",cp-8);
  }
/************************************************************************
 * Use a for loop to parse the serial number and convert it to          *
 * hexadecimal                                                          *
 ************************************************************************/
  for (i = 0 ; i < 8 ; i++, cp--){
          if (*cp < '9')
                  ssn += (*cp - '0' ) << (i*4);
          else if (*cp < 'G')
                  ssn += (*cp - 'A' + 0xa ) << (i*4);
          else if (*cp < 'a')
                  ssn += ( *cp % 0xf ) << (i*4);
          else if (*cp < 'g')
                  ssn += (*cp - 'a' + 0xa ) << (i*4);
          else
                  ssn += ( *cp % 0xf ) << (i*4);
      }
      return(ssn); /* Return the system serial number */
}


/************************************************************************
 * INTERFACE NAME:  eisa_nvram_status                                   *
 *                                                                      *
 * FUNCTIONAL DESCRIPTION:                                              *
 *                                                                      *
 * The eisa_envram_status interface provides the /dev/presto device     *
 * driver with the status of diagnostics run on the NVRAM.              *
 *                                                                      *
 * CALLED BY: The /dev/presto device driver.  The nvram_status          *
 *            member of the presto_interface0 structure is set to       *
 *            eisa_nvram_status by envram_attach.  The /dev/presto      *
 *            driver accesses the NVRAM cache interfaces through        *
 *            presto_interface0 structure.                              *
 *                                                                      *
 * FORMAL PARAMETERS: None                                              *
 *                                                                      *
 * IMPLICIT INPUTS:                                                     *
 *                                                                      *
 *   o sc->diag_status -- diagnostic flag set in envram_probe           *
 *                                                                      *
 * IMPLICIT OUTPUTS: None                                               *
 *                                                                      *
 * RETURN VALUE: (sys/presto.h defined status values)                   *
 *                                                                      *
 *         NVRAM_RDWR -- Passed R/W diags                               *
 *         NVRAM_RDONLY   -- Passed RO diags                            *
 *         NVRAM_BAD  -- Failed diags                                   *
 ************************************************************************/

int eisa_nvram_status()
{
  /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;
```

```
  if (sc->diag_status)
        return(NVRAM_RDONLY);     /* Passed RO diags */
    else
        return(NVRAM_BAD);        /* Failed diags */
}


/*****************************************************************
 * INTERFACE NAME:  eisa_nvram_battery_status                    *
 *                                                               *
 * FUNCTIONAL DESCRIPTION:                                       *
 *                                                               *
 * The eisa_nvram_battery_status interface provides the /dev/presto *
 * device driver with the status of the battery on the EISA bus  *
 * NVRAM memory expansion board.                                 *
 *                                                               *
 * CALLED BY:                                                    *
 *                                                               *
 * The /dev/presto device driver calls this interface through the  *
 * nvram_battery_status member of the presto_interface0 structure:  *
 *                                                               *
 * presto_interface0.nvram_battery_status=eisa_nvram_battery_status;*
 *                                                               *
 * The envram_attach interface performs the initialization of    *
 * nvram_battery_status.                                         *
 *                                                               *
 * FORMAL PARAMETERS: None                                       *
 *                                                               *
 * IMPLICIT INPUTS: None                                         *
 *                                                               *
 * IMPLICIT OUTPUTS:                                             *
 *                                                               *
 * The eisa_nvram_battery_status interface fills in the battery-    *
 * related members of the nvram_battery_info data structure.  Note  *
 * that presto.h defines an external data structure called       *
 * nvram_batteries0, which is an instance of nvram_battery_info.    *
 *                                                               *
 * RETURN VALUE: (sys/presto.h defined status values)            *
 *                                                               *
 *       BATT_NONE -- No battery                                 *
 *       BATT_ENABLED -- Battery enabled                         *
 *       BATT_HIGH -- Battery has minimal energy stored          *
 *       BATT_OK -- Battery ok                                   *
 *       BATT_SELFTEST -- Battery exists, but charge state unknown  *
 *       BATT_CHARGING -- Battery does not have enough power     *
 *****************************************************************/

int eisa_nvram_battery_status()
{
/* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;

  nvram_batteries0.nv_nbatteries = 1; /* always one battery */
  nvram_batteries0.nv_minimum_ok = 1; /* Battery must be good */
  nvram_batteries0.nv_primary_mandatory = 1; /* Primary battery must be OK */
  nvram_batteries0.nv_test_retries = 1; /* Call this interface one time */

/*****************************************************************
 * Check the battery status by reading the CSR.  If the battery is  *
 * okay, set the battery's nv_status and return zero (0) to the     *
```

```
 * /dev/presto device driver.  Otherwise, return 1 to indicate the  *
 * the battery is not okay.                                         *
 ********************************************************************/

  if ((ENVRAM_READIO_D16(ENVRAM_CSR) & BAT_FAIL))
      {
          nvram_batteries0.nv_status[0] = BATT_OK;
          return(0);
      }
  else
      {

          return(1);
        }
}


/*********************************************************************
 * INTERFACE NAME:  eisa_nvram_battery_enable                        *
 *                                                                   *
 * FUNCTIONAL DESCRIPTION:                                           *
 *                                                                   *
 * The eisa_nvram_battery_enable provides the /dev/presto device     *
 * driver with the ability to enable the battery on the EISA bus     *
 * NVRAM memory expansion board.                                     *
 *                                                                   *
 * CALLED BY:                                                        *
 *                                                                   *
 * The /dev/presto device driver calls this interface through the    *
 * nvram_battery_enable member of the presto_interface0 structure:   *
 *                                                                   *
 * presto_interface0.nvram_battery_enable=eisa_nvram_battery_enable;*
 *                                                                   *
 * The envram_attach interface performs the initialization of        *
 * nvram_battery_enable.                                             *
 *                                                                   *
 * FORMAL PARAMETERS: None                                           *
 *                                                                   *
 * IMPLICIT INPUTS: None                                             *
 *                                                                   *
 * IMPLICIT OUTPUTS: None                                            *
 *                                                                   *
 * RETURN VALUE:                                                     *
 *                                                                   *
 *       0 -- Battery enabled successfully                           *
 *       1 -- Battery not enabled                                    *
 *********************************************************************/

int eisa_nvram_battery_enable()
{
  /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;
/*********************************************************************
 * The required action is to zero the BDISC control bit.  This       *
 * disables the battery disconnect circuit, thus enabling the        *
 * battery.                                                          *
 *********************************************************************/
  ENVRAM_WRITEIO_D16(ENVRAM_CSR, WRMEM|SET_LED);
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT);
  mb(); 7* Perform a memory barrier after the writes. */
```

```
  return(0); /* Battery successfully enabled */
}


/*****************************************************************
 * INTERFACE NAME:  eisa_nvram_battery_disable                   *
 *                                                               *
 * FUNCTIONAL DESCRIPTION:                                       *
 *                                                               *
 * The eisa_nvram_battery_disable provides the /dev/presto device *
 * driver with the ability to disable the battery on the EISA bus *
 * NVRAM memory expansion board.                                 *
 *                                                               *
 * The /dev/presto device driver calls this interface through the *
 * nvram_battery_disable member of the presto_interface0 structure: *
 *                                                               *
 * presto_interface0.nvram_battery_disable=eisa_nvram_battery_disable;*
 *                                                               *
 * The envram_attach interface performs the initialization of    *
 * nvram_battery_disable.                                        *
 *                                                               *
 * FORMAL PARAMETERS: None                                       *
 *                                                               *
 * IMPLICIT INPUTS: None                                         *
 *                                                               *
 * IMPLICIT OUTPUTS: None                                        *
 *                                                               *
 * RETURN VALUE:                                                 *
 *                                                               *
 *       0 -- Battery disabled successfully                      *
 *       1 -- Battery not disabled                               *
 *****************************************************************/

int eisa_nvram_battery_disable()
{
  /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;

/*****************************************************************
 * The required action is to send sequence "11001" to the battery *
 * disconnect device register.  This enables the battery disconnect *
 * circuit.                                                      *
 *****************************************************************/
  ENVRAM_WRITEIO_D16(ENVRAM_CSR,WRMEM);
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,!BAT_DISCON_BIT);
  mb();
  ENVRAM_WRITEIO_D8(ENVRAM_BAT,BAT_DISCON_BIT);
  mb();

  return(0); /* Battery successfully disabled */
}
```

```
/*****************************************************************
 * INTERFACE NAME:  envram_read                                  *
 *                                                               *
 * FUNCTIONAL DESCRIPTION:                                       *
 *                                                               *
 * The envram_read interface provides the DMA slave capability to:  *
 *                                                               *
 *   o Convert the source address passed in by envram_probe and the *
 *     /dev/presto driver from the NVRAM address into a      *
 *     physical address                                          *
 *   o Copies data from the NVRAM bus address space to       *
 *     system memory                                             *
 *                                                               *
 * Specifically, the envram_read interface reads the length block   *
 * of data pointed to by the source address parameter to the EISA   *
 * NVRAM destination parameter.  This assumes:                   *
 *                                                               *
 *   1. The source is *always* from the NVRAM                    *
 *   2. The destination is to Host (Main) memory                 *
 *                                                               *
 * CALLED BY:                                                    *
 *                                                               *
 * The /dev/presto device driver calls this interface through the   *
 * nvram_ioreg_read and nvram_block_read members of the          *
 * presto_interface0 structure:                                  *
 *                                                               *
 * presto_interface0.nvram_ioreg_read=envram_read;               *
 * presto_interface0.nvram_block_read=envram_read                *
 *                                                               *
 * The envram_attach interface performs the initialization of    *
 * nvram_ioreg_read and nvram_block_read.                        *
 *                                                               *
 * The envram_probe interface also calls envram_read.            *
 *                                                               *
 * FORMAL PARAMETERS:                                            *
 *                                                               *
 *   o srcaddr -- Specifies the source address of the data to be    *
 *                written.  Because this source address is passed   *
 *                in to envram_read by envram_probe and the         *
 *                /dev/presto device driver, the address format is  *
 *                a kernel segment(KSEG) logical physical address.  *
 *                                                               *
 *   o destaddr -- Specifies the destination address of where to    *
 *                 write the data.  Because this destination        *
 *                 address is passed in by the envram_probe and the *
 *                 /dev/presto device device driver, the format is  *
 *                 a kernel segment (KSEG) logical physical         *
 *                 address.                                         *
 *                                                               *
 *   o length -- Specifies the length of the block of data to be    *
 *               written.  This length is passed in by envram_probe *
 *               and the /dev/presto device driver.                  *
 *                                                               *
 * IMPLICIT INPUTS:                                              *
 *                                                               *
 *   o srcaddr (See above description)                           *
 *   o destaddr (See above description)                          *
 *   o length (See above description)                            *
 *                                                               *
 * IMPLICIT OUTPUTS: None                                        *
```

```
 *                                                                    *
 * RETURN VALUE: None                                                 *
 **********************************************************************/

void envram_read(source, dest, len)
  caddr_t source;    /* Address of the source data */
  caddr_t dest;      /* Destination for the source data */
  u_int   len;       /* Length of the block */
{
  /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;

/**********************************************************************
 * Copy the data from bus address space to system memory by calling *
 * the io_copyin interface.  This is a generic interface that maps  *
 * to a machine-specific interface that actually performs the copy  *
 * from bus address space to system memory.  Using io_copyin to     *
 * perform the copy operation makes the device driver more portable *
 * across different CPU architectures and different CPU types       *
 * within the same architecture.                                    *
 **********************************************************************/

  io_copyin((io_handle_t)
          KSEG_TO_PHYS((u_long)source|sc->saved_mem_sysmap),
                      (vm_offset_t)dest,len);
}


/**********************************************************************
 * INTERFACE NAME:  envram_write                                      *
 *                                                                    *
 * FUNCTIONAL DESCRIPTION:                                            *
 *                                                                    *
 * The envram_write interface provides the DMA slave capability to:  *
 *                                                                    *
 *   o Write to the NVRAM                                             *
 *   o Perform programmed I/O                                         *
 *   o Copy to the NVRAM                                              *
 *                                                                    *
 * Specifically, the envram_write interface writes                    *
 * the length block of data pointed to by the source                  *
 * address parameter to the EISA NVRAM destination                    *
 * parameter.  This assumes:                                          *
 *                                                                    *
 *   1. The destination is *always* the NVRAM                         *
 *   2. The source is from Host (Main) memory                         *
 *                                                                    *
 * CALLED BY:                                                         *
 *                                                                    *
 * The /dev/presto device driver calls this interface through the     *
 * nvram_ioreg_write and nvram_block_write members of the             *
 * presto_interface0 structure:                                       *
 *                                                                    *
 * presto_interface0.nvram_ioreg_write=envram_write;                  *
 * presto_interface0.nvram_block_write=envram_write                   *
 *                                                                    *
 * The envram_attach interface performs the initialization of         *
 * nvram_ioreg_write and nvram_block_write.                           *
 *                                                                    *
 * FORMAL PARAMETERS:                                                 *
```

```
 *                                                                    *
 *   o srcaddr -- Specifies the source address of the data to be      *
 *                written.  Because this source address is passed     *
 *                in to envram_read by the /dev/presto device         *
 *                driver, the address format is a kernel              *
 *                segment(KSEG) logical physical address.             *
 *                                                                    *
 *   o destaddr -- Specifies the destination address of where to      *
 *                 write the data.  Because this destination          *
 *                 address is passed in by the /dev/presto device     *
 *                 device driver, the format is a kernel              *
 *                 segment (KSEG) logical physical address.           *
 *                                                                    *
 *   o length -- Specifies the length of the block of data to be      *
 *               written.  This length is passed in by the            *
 *               /dev/presto device driver.                           *
 *                                                                    *
 * IMPLICIT INPUTS:                                                   *
 *                                                                    *
 *   o srcaddr (See above description)                                *
 *   o destaddr (See above description)                               *
 *   o length (See above description)                                 *
 *                                                                    *
 * IMPLICIT OUTPUTS: None                                             *
 *                                                                    *
 * RETURN VALUE: None                                                 *
 **********************************************************************/

void envram_write(source, dest, len)
  caddr_t source;     /* Address of the source data */
  caddr_t dest;       /* Destination for the source data */
  u_int   len;        /* Length of the block */
{
  /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;
  vm_offset_t destptr; /* Stores the destination address */
  register int xfer; /* size of each partial transfer */
  int retry; /* retry counter */
  char *ddest = dest; /* destination pointer */

/*********************************************************************
 * Presto WRITE operation:  Write to NVRAM from Main Memory          *
 *********************************************************************/

/*********************************************************************
 * Use DMA if size is larger than 32 bytes                           *
 *********************************************************************/
    if (len > 32) {

      /*
       * Set up destination address passed from Presto,
       * the dest is a main memory virtual address
       */
      destptr = KSEG_TO_PHYS(dest) - sc->cache_base;
      ddest = (char *)destptr;

        /*
         *  Allign destination to 1K
         */
      if (!(xfer = ENVRAM_XFER_SIZE - ((int)ddest & (ENVRAM_XFER_SIZE-1))))
```

```
  xfer = ENVRAM_XFER_SIZE;

if (xfer > len)
    xfer = len;

  /*
   * Allign source to 8K
   */
if ((u_int)source/ENVRAM_ALLIGN !=
     ((u_int)source+xfer)/ENVRAM_ALLIGN)
      xfer = xfer - (((u_int)source+xfer) & (ENVRAM_ALLIGN-1));

while (1) {

    /*
     * Set up the 82357 dma controller
     */
    if (!(dma_map_load(xfer, source, (struct proc *)0,
                       sc->ctlr, &sc->sglp, 0, DMA_OUT)))
        panic("envram: dma_map_load failure\n");

    /*
     * Set up NVRAM source address
     */
    ENVRAM_WRITEIO_D16(ENVRAM_DMA0,((u_int)(ddest-4) << 6));
    ENVRAM_WRITEIO_D16(ENVRAM_DMA1,((u_int)ddest >> 5));

    /*
     * Start NVRAM transfer
     */
    ENVRAM_WRITEIO_D16(ENVRAM_CSR,SET_DREQ|WRMEM|SET_LED);
    mb();

    /*
     * Bookeeping, bury behind DMA
     */
    len -= xfer;
    source += xfer;
    ddest += xfer;

    /*
     *  Set up for next, align destination to 1K,
     *  NVRAM only handles DMAs inside of a 1k
     *  alligned address range.
     */
    if (!(xfer = ENVRAM_XFER_SIZE - (((int)ddest
          & (ENVRAM_XFER_SIZE-1)))))
          xfer = ENVRAM_XFER_SIZE;

    if (xfer > len)
        xfer = len;

    /*
     * Align source to 8K, source will be memory
     * hence 8K for DEC OSF/1 pages.
     */
    if ((u_int)source/ENVRAM_ALLIGN !=
       ((u_int)source+xfer)/ENVRAM_ALLIGN)
        xfer = xfer - (((u_int)source+xfer) & (ENVRAM_ALLIGN-1));
```

```
            /*
             * Spin on SET_DREQ bit. If the hardware
             * works, this bit should never be set.
             */
            retry = 10;
            while (--retry)
                if (!(ENVRAM_READIO_D16(ENVRAM_CSR) & SET_DREQ))
                    break;

            if (!length)
                break;

            /*
             * If retry expires the hardware is broken.
             */
            if (!retry)
                panic("envram: DMA retry expired\n");
        }
        return;
    }

    /*
     * The envram_write interface copies the data from system
     * memory to bus address space by calling io_copyout.
     * The io_copyout interface is a generic interface that
     * maps to a bus- and machine-specific interface that
     * actually performs the copy to bus address space.
     * Using io_copyout to perform the copy operation
     * makes the device driver more portable across different
     * CPU architectures and different CPU types within
     * the same architecture.
     */
    io_copyout((vm_offset_t)source, (io_handle_t)
              (KSEG_TO_PHYS((u_long)dest)|sc->saved_mem_sysmap),
                        len);
}


/*****************************************************************
 * INTERFACE NAME:  envram_zero                                  *
 *                                                               *
 * FUNCTIONAL DESCRIPTION:                                       *
 *                                                               *
 * The envram_zero interface zeroes the "len" bytes of EISA NVRAM *
 * memory starting at "addr".                                    *
 *                                                               *
 * FORMAL PARAMETERS:                                            *
 *                                                               *
 *   o addr - Specifies the starting address of the              *
 *            NVRAM for this EISA bus memory expansion board to  *
 *            zero. Because this address is passed in by the     *
 *            /dev/presto device driver, the format is a kernel  *
 *            segment (KSEG) logical physical address.           *
 *                                                               *
 *   o length - Specifies the number of bytes to zero.  This length *
 *              is passed in by the /dev/presto device driver.   *
 *                                                               *
 * CALLED BY:                                                    *
 *                                                               *
 * The /dev/presto device driver calls this interface through the *
```

```
 * nvram_ioreg_zero and nvram_block_zero members of the             *
 * presto_interface0 structure:                                     *
 *                                                                  *
 * presto_interface0.nvram_ioreg_zero=envram_zero;                  *
 * presto_interface0.nvram_block_zero=envram_zero;                  *
 *                                                                  *
 * The envram_attach interface performs the initialization of       *
 * nvram_ioreg_zero and nvram_block_zero.                           *
 *                                                                  *
 * IMPLICIT INPUTS:                                                 *
 *                                                                  *
 *   o addr (See above description)                                 *
 *   o length (See above description)                               *
 *                                                                  *
 * IMPLICIT OUTPUTS: None                                           *
 *                                                                  *
 * RETURN VALUE: None.                                              *
 ********************************************************************/

void envram_zero(addr, len)
  caddr_t addr;   /* Starting address of EISA NVRAM to zero */
  u_int   len; /* Number of bytes to zero */
{
 /* Pointer to softc structure */
  register struct envram_softc *sc = envram_softc;

/*********************************************************************
 * Zero a block of memory in bus address space by calling the       *
 * io_zero interface.   This is a generic interface that maps to a  *
 * machine-specific interface that actually writes zeros to some    *
 * location in bus address space. Using io_zero to perform the      *
 * zero operation makes the device driver more portable across      *
 * different CPU architectures and different CPU types within the   *
 * same architecture.                                               *
 *********************************************************************/

  io_zero((io_handle_t)
        KSEG_TO_PHYS((u_long)addr|sc->saved_mem_sysmap), len);
}
```

Glossary

device configuration file
A device configuration file provides information on the resources required to program the EISA bus device. Some of the resources supplied in this file are dma channel, interrupt channel, and memory space information. The system vendor supplies the device configuration file on a floppy diskette. Do not confuse the device configuration file with the DEC OSF/1 system configuration file, which is an ASCII text file that defines the hardware and software components of the system. Device configuration files, unlike system configuration files, have `CFG` extensions.

See also **expansion board configuration file.**

/dev/envram device driver
The `/dev/envram` device driver is a character device driver that provides read and write services to the `/dev/presto` device driver.

/dev/presto device driver
The `/dev/presto` device driver is a disk driver that uses nonvolatile memory as a cache. It works as a layer between other drivers and the rest of the DEC OSF/1 kernel.

EISA configuration utility (ECU)
The EISA configuration utility (ECU) is a program that reads the expansion board and device configuration files and, for each option in the system, generates a conflict-free configuration. The ECU runs on top of the firmware, not the operating system.

EISA bus expansion board
An EISA bus expansion board is a hardware entity that plugs into EISA bus connectors. An EISA bus expansion board must contain a readable product ID.

expansion board configuration file
An expansion board configuration file provides information on the resources and register values required to program the EISA bus expansion board. Some of the resources supplied in this file are the board ID, manufacturer of the board, the interrupt line, DMA channel, and board memory. The manufacturer of the EISA bus expansion board supplies the board's configuration file on a floppy diskette. Do not

confuse the expansion board configuration file with the DEC OSF/1 system configuration file, which is an ASCII text file that defines the hardware and software components of the system. Expansion board configuration files, unlike system configuration files, have `CFG` extensions.

See also **device configuration file**.

Extended Industry Standard Architecture (EISA)
The Extended Industry Standard Architecture (EISA) bus is an industry-standard high-performance bus that is a superset of the 8- and 16-bit Industry Standard Architecture (ISA).

ISA bus
The Industry Standard Architecture (ISA) bus is used in computer systems that adhere to the ISA. The ISA bus supplies the signals for performing the following basic functions of the computer system: memory, I/O, and direct memory access (DMA).

nonvolatile random-access memory (NVRAM)
Nonvolatile random-access memory contains the EISA bus board information specified in the board's configuration file. It is the ECU that writes this information to the NVRAM.

Index

D

E

F

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 EISA Bus Device Drivers
AA-Q0R6A-TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

Please rate this manual:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

What would you like to see more/less of? _____

_____

What do you like best about this manual? _____

_____

What do you like least about this manual? _____

_____

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

What version of the software described by this manual are you using? _____

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Email _____ Phone _____

---- Do Not Tear - Fold Here and Tape ----

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
OPEN SOFTWARE PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK ROAD
NASHUA NH 03062-9987

--- Do Not Tear - Fold Here ---

Cut Along Dotted Line

Reader's Comments

DEC OSF/1
Writing EISA Bus Device Drivers
AA-Q0R6A-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