*Chapter 4*

# Writing an EISA Device Driver

This chapter provides in-depth information on writing device drivers for Silicon Graphics computer systems equipped with the Extended Industry Standard Architecture (EISA) expansion bus. It gives a brief overview of the EISA-bus interface, describes configuration for EISA device drivers, and introduces several EISA-specific routines.

It also explains the model drivers use to control device DMA operations. This model makes it easier to port existing drivers from other operating systems.

This chapter contains the following sections:


- "EISA-bus Interface Overview" on page 106
- "Choosing a Device Driver Model" on page 110
- "Writing a User-level EISA Device Driver" on page 111
- "Writing a Kernel-level EISA Device Driver" on page 113


**Note:** Currently, only the Indigo$^2$ workstation and the CHALLENGE M server support the EISA bus.

## EISA-bus Interface Overview

The Extended Industry Standard Architecture (EISA) bus standard is an enhancement of the ISA (Industry Standard Architecture) bus standard developed by IBM for the PC/AT. EISA is backward compatible with ISA. It expands the ISA data bus from 16 bits to 32 bits and adds 23 address lines and 16 indicator and control lines.

The EISA bus supports:

- All ISA transfers (except ISA-bus masters)
- EISA-bus master devices
- Burst-mode DMA transfers
- 32-bit memory data and address path
- Peer-to-peer card communication
- Dynamic bus sizing (that is, 32-bit bus master to 16-bit memory)

### Initialization

A central component of the DOS-EISA world is a configuration program that allocates and sets up the resources that the EISA card uses. Once this program runs, the configuration results are downloaded into non-volatile memory for future driver use. Storing the configuration results serves two purposes:

1. The stored results make it easy for drivers to operate as boot devices because the drivers can initialize in a standalone environment, in which the operating system is not up and able to allocate resources to the driver. But, because the Silicon Graphics EISA environment does not now support booting from EISA devices, saving the resource allocation information in nonvolatile RAM is superfluous.

2. DOS saves the configuration file to store a set of card register programmatic instructions that initialize various I/O ports. This is useful when writing a generic device driver that must control boards that have different initialization registers but are basically the same, even though they come from different manufacturers.

   **Note:** Although Silicon Graphics does not support this functionality, it should not cause any problems because you can write the driver initialization routine to check the product ID and execute the appropriate initialization code. Be sure to consider the byte order when checking the product ID.

In the DOS environment, the EISA configuration program allocates memory spaces, DMA channels, and interrupt request lines. In IRIX release 5.x, **lboot** handles the allocation of memory and I/O spaces. **lboot** passes the allocation results to the driver via the *edt_t* structure. Memory and I/O spaces are passed via *edt*, but dynamic resources are not. The driver requests DMA channels and interrupt request lines or queues (IRQs) as dynamic resources as required.

## EISA-Bus Locked Cycles

The EISA bus provides a locked cycle that allows users to read/write the contents of a device register or memory location as an atomic operation. This facility is normally implemented for semaphores' bit test-and-set operations.

On the Indigo$^2$, the hardware implementation of a locked cycle is limited. Indigo$^2$ allows an EISA card to issue a locked cycle, but only for the duration of two contiguous read/write operations. Access to this class of cycles is provided on the CPU side, much like the *read/modify/write* cycle on a VME bus, through the **pio_rmw*** kernel functions. See "Writing a Kernel-level EISA Device Driver."

In general, **LOCK*** is not considered to be a supported feature of the Silicon Graphics EISA implementation.

### EISA-Bus Request Arbitration

EISA provides server DMA channels arranged into two channel groups (channels 0-3 and channels 5-7) for priority resolution. Silicon Graphics uses the rotating scheme described in the EISA-bus specification. Although the channels rotate in this scheme, channels 5-7 receive more cycles, in general, than channels 0-3.

### EISA-Bus Interrupts

The EISA bus supports 11 edge-triggerable or level-triggerable interrupts. IRQ0–IRQ2, IRQ8, and IRQ13 are used for internal functions only and are not available externally. The remaining 11 interrupt lines (IRQ3–IRQ7, IRQ9–IRQ12, IRQ14, IRQ15) are available for external system interrupts.

On Indigo$^2$, all EISA interrupts are received at the same CPU level. When the CPU receives an EISA-bus interrupt, it responds to each device in IRQ priority order (lower number first). The operating system then determines which interrupt routine to use, based on the value specified by the device driver. Devices may share the same IRQ level.

### EISA-Bus Data Transfers

The EISA bus supports 8-bit, 16-bit, and 32-bit data transfers through direct CPU access as well as DMA initiated by a bus-master card or the on-board DMA hardware.

### EISA-Bus Address Space

The EISA-bus address space is divided into I/O address space and memory address space.

I/O address space provides access to the 4 KB page that references the registers on an EISA card on a slot-specific basis and the registers on an ISA card on an address-range basis.

The EISA memory space supports memory that is configured to respond to the address range starting at 0xa0000. On Indigo$^2$ systems, the EISA memory address range extends 112 MB up to address 0x06ffffff (see Figure 4-1). Once properly mapped by the operating system, this memory space can be directly accessed by the CPU through loads and stores.

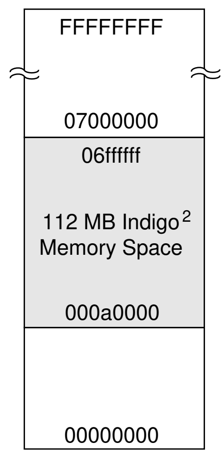


**Figure 4-1** Indigo$^2$ Memory Space

IRIX release 5.2 and later releases provide a set of procedural interfaces that a device driver must use to allocate and map the EISA I/O address space and memory address space.

The Indigo$^2$ has four peripheral card slots that accept EISA, GIO, or graphics cards. A server can support up to four EISA cards, but then a graphics card cannot be used. CHALLENGE M platforms have four EISA slots available. Graphics cards are available that use one, two, or three slots:

- With Extreme graphics installed, one slot is available for use by an EISA card.
- With XZ graphics installed, two slots are used by the graphics, and two are available for EISA cards.
- The XL graphics uses only one slot, so up to three EISA cards can be accommodated.

### Byte Swapping

An important implementation detail of the EISA bus you need to be aware of is that the EISA bus uses *little-endian* byte ordering; the CPU running IRIX uses *big-endian* byte ordering. Hence, a byte reference by the CPU to the low-order byte within a word results in a reference to the high-order byte on the EISA bus. This can sometimes become complex when referencing data structures across the bus.

## Choosing a Device Driver Model

You can control a device connected to the EISA bus through a *user-level* device driver, a *kernel-level* device driver, or a *kernel-level memory-mapping* device driver.

The simplest way to access an EISA device is through a user-level driver that controls the device by directly interfacing with the EISA-bus adapter.

### When to Write a User-level Device Driver

There are several reasons why you might want to write a user-level device driver instead of a kernel-level device driver:

- If your user does not need to use EISA device interrupts or DMA operations.

  **Tip:** On many EISA boards that generate interrupts or use DMA operations, these features can be disabled so that simple, user-level drivers can be used.

- It is more convenient to write a user-level EISA-bus device driver to determine whether a device is responding to the correct address or simple register tests.

  **Tip:** This can be very useful for prototyping: you can quickly integrate boards into a system with the interrupts turned off, then later write a kernel-level driver that uses interrupts or DMA operations for better performance.

- You can use a user-level device driver in real applications that require low-overhead access to on-board device registers or memory.

### When to Write a Kernel-level Device Driver

The main reasons for writing a kernel-level device driver are:

- You must use interrupts to access the EISA device.
- You must use DMA operations to transfer data from/to the EISA device.

### When to Write a Kernel-level Memory-mapping Device Driver

The main reasons for writing a kernel-level memory-mapping device driver are:

- You need a driver that allows the user to access the EISA device as memory in user space yet also supports DMA and interrupts.
- You need an efficient way to share main memory between a kernel driver and a user program.

See Chapter 7, "Writing Kernel-level General Memory-mapping Device Drivers," for a description of the memory-mapping facilities.

## Writing a User-level EISA Device Driver

A typical application for a user-level EISA-bus device driver is to handle data acquisition hardware (that is, hardware that reads large amounts of data into device memory). Because the device memory is available to the user program directly (it is "mmapped" into the address space of the user program), the user program can avoid copying the data into host memory and can process the data in the device memory instead. However, on some systems, this PIO access may have substantially lower performance than DMA-based kernel drivers.

## User-level EISA Special Files

IRIX Release 5.x contains *special* files in the */dev/eisa* directory that allow a user-level program to map arbitrary EISA or ISA devices into its address space. These files can be used to write a user-level memory-mapped device driver. The driver uses the **mmap**() system call to map the card's address space into user-level program address space. Then the driver can access the card through simple loads and stores of program variables.

The special device files found in the */dev/eisa* directory are:

| | |
|---|---|
| *eisaAio* | EISA-bus adapter's 64 KB I/O address space |
| *eisaAmem* | EISA-bus adapter's 112 MB memory address space |

Where *A* is the specific EISA-bus adapter.

An Indigo$^2$ system has a single EISA-bus slot, so its special device files are limited to *eisa0io* and *eisa0mem*.

## Using the mmap Operating System Function

A user-level driver may access a generic EISA device by opening the appropriate */dev/eisa* special file, followed by an **mmap**() call to map in the device. For example:

```
int fd, len, off;
char *addr;
fd=open("/dev/eisa/eisa0io", O_RDWR);
len=4096;
off=0x1000;
addr=mmap(0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, off);
 .
 .
 .
/* addr can be used to access any card register by adding the
 * appropriate offset and then dereferencing the pointer
 */
 .
 .
 .
```

Note that in the example above, the offset starts at 0X1000, which is the base address of the first EISA slot. On Indigo$^2$ systems, the slots are numbered in ascending order from top to bottom (that is, slot one is the top and slot four is on the bottom.). See the mmap(D2) man page for a complete description of this system call.

The **mmap**() routine maps *len* bytes starting at EISA address *off* to the user virtual address *addr*. Any loads or stores to the address range *addr* to *addr+len* will result in read or writes to the EISA I/O space.

**Note:** You are responsible for identifying the offset to reference each device on a slot-specific basis. The offset must be incremented by 4096 to reference the next EISA card's I/O space; the offset to an ISA card is its ISA address.

## Writing a Kernel-level EISA Device Driver

This section provides in-depth information about data structures and drivers that interface through the kernel to the EISA bus. It describes system configuration for EISA device drivers and introduces several driver initialization routines.

### Configuring a Kernel-level EISA Device Driver

To configure a kernel-level EISA device driver, you must add certain system files to the kernel, create various data structures, and write the driver.

#### File Requirements

The following files must be created or modified from existing models and added to the kernel:

1. A system file with a directive telling **lboot**, the configuration utility, how to include your driver and specify which memory space your device will allocate. This is installed into */var/sysgen/system* directory using the driver name appended by "*.sm*". See "Including EISA Drivers in the Kernel," for information on what to include in this file.
2. A master file under */var/sysgen/master.d*. The name of the master file is the same as the name of the object file for the driver, but the master file must not have the *.o* suffix.
3. Copy the driver object file (the *.o* file) to */var/sysgen/boot*.

## Determining ISA/EISA Device Addresses

The I/O address space on an EISA card plugged into a card slot responds to the range of bus addresses for that slot. All EISA cards are identified by a manufacturer-specific device ID that the operating system uses to register the existence of each card. ISA cards, in contrast, are jumpered to respond to a specific address range that corresponds to the device's I/O registers.

Your driver can map these I/O addresses into the host processor address space; it can then access these registers with simple reads and writes. For a card's memory space to be accessible, the card must be configured or jumpered to respond to the appropriate address range. The memory space can then be mapped to the host address space with the kernel mapping routines. The specified address range must be selected to avoid conflicts with other EISA/ISA devices.

## Including EISA Drivers in the Kernel

This section describes the configuration information needed to add an EISA driver to the kernel.

To add a new kernel-level device driver, you must create your own system file. This file should contain the appropriate directive that tells **lboot** how to include your driver, and it should specify which memory space will be allocated by your device. The filename must include the *.sm* suffix and reside in the directory */var/sysgen/system*. Because **lboot** can probe for EISA devices, it can conditionally include an EISA device driver into the kernel.

### VECTOR Directive

The system file must have a *VECTOR:* line in it that causes **lboot** to check whether a hardware device corresponding to the particular module is connected to the system. If the device is found, and a driver exists, it is included in the kernel.

#### Synopsis

`VECTOR: bustype=EISA module=`*driver_name* `[options]`

#### Arguments

| | |
|---|---|
| *bustype* | This must be set to "*EISA*". |
| *module* | The *drivername* declares the name of the object file for the driver and its *master.d* configuration file, respectively, in the */var/sysgen/boot* and */var/sysgen/master.d* directories. |

The optional fields that can be appended to the *VECTOR:* directive are:

| | |
|---|---|
| *adapter* | The adapter number identifying which EISA bus out of possibly several. Indigo$^2$ systems currently support only a single EISA bus, which is referenced as "*adapter=0*". You can also use "*adapter=**" to reference all possible EISA adapters (as potentially available in future hardware). The default is zero. |
| *ctlr* | The device number that differentiates between more than one device of the same type. This number is passed to the driver by the *edt* structure when multiple devices of the same type are connected. Generally, this number has some correspondence to the minor device number. |

A final set of *VECTOR:* line options probes for and configures address spaces used by the EISA or ISA card. EISA address space is divided into I/O and memory address space.

The *VECTOR:* line space declarations are a triple of the form:

(*space_name*, *address_offset*, *size*)

#### Arguments

| | |
|---|---|
| *space_name* | Must be either *EISAIO* or *EISAMEM* (even for ISA cards). |

*address_offset* and *size*

Specified in bytes. If the card is an EISA card (as opposed to an ISA card), each card's I/O register space starts at

`0x1000 (4K) bytes * slot number`

and extends the full 0x1000 (4 KB) to the next slot's space. For ISA cards, the I/O space addresses are not slot dependent, but rather card jumper dependent. They can be in the range starting at byte offset 0x100 (256) and extending up to 0x400 (1024). Similarly, the ISA address space size is card dependent and limited only by the ISA address space size. The *EISAMEM* space can begin at offset 0xa0000 (640 KB) and extend up to 112 MB.

The *VECTOR:* fields that use these space declarations are:

*iospace1*
*iospace2*
*iospace3*

These space declarations are defined so that the driver can map to the registers and memory on the card. Typically, *iospace1* is used to define the I/O register space to be referenced, and *iospace2* and *iospace3* are used to gain access to one or two different on-card memory spaces.

*exprobe_space*

There are two different techniques used to probe for the existence of an EISA or ISA card. For an EISA card, a read is compared to an expected value to determine the existence of a particular EISA card in a slot. Thus, probing for an EISA card is done by specifying a *VECTOR:* line with an *exprobe_space* read if the card's unique manufacturer product identifier at each EISA slot's product IS I/O address. The manufacturer's product ID is a 4-bit quantity that is encoded in a format specified by the EISA bus specification.

The general form of *exprobe_space* allows for a more generalized extensive read/write sequence, which can be used to search for an ISA card at a particular address. The probing is accomplished by writing a value to a register located on the card and then reading it back to verify that a board responds to that location. Do not use a comparison value of 0xff because the bus floats to that value, which would cause the probe command always to believe that it had found a new board.

*exprobe_space* is specified as a six-tuple consisting of: *(rwseg, bus_space, address, size, value, mask)*

*rwseg* – A sequence of one or more *r*'s or *w*'s indicating a read or write. To test for an EISA card, use a simple *r*. To test for an ISA card, use *wr*, which causes the value to be written then reread.

*bus_space* – Either *EISAIO* or *EISAMEM*. The manufacturer ID is located in *EISAIO* space.

*address* – The address offset to read. The manufacturer ID is located at 0x*z*C80, where *z* is the slot number for EISA cards. ISA cards must specify the address of a register that can be read from and written to.

*size* – The manufacturer ID is four bytes long, as described below. ISA cards must use the appropriate register width.

*value* – Compressed 4-byte value to look for (EISA). Non-0xff value for ISA.

*mask* – A bit mask to specify which bytes of the value to compare against.

### EISA Product Identifier (ID)

EISA expansion boards, embedded devices, and system boards have a four-byte product identifier (ID) that can be read from I/O port addresses 0x*z*C80 through 0x*z*C83. For example, the slot 1 product ID can be read from I/O port addresses 0x1C80-0x1C83.

The first two bytes (0x*z*C80 and 0x*z*C81) contain a compressed representation of the manufacturer code. The manufacturer code is a three-character code (uppercase, ASCII characters in the range of A to Z) chosen by the manufacturer and registered with the firm that distributes the EISA specification.

The manufacturer code "ISA" is used to indicate a generic ISA adapter. The three-character manufacturer code is compressed into three 5-bit values so that it can be incorporated into the two I/O bytes at 0x*z*C80 and 0x*z*C81, where *z* represents the slot number probed.

The compression procedure is:

1. Find the hexadecimal ASCII value for each letter:

   ASCII for "A"-"Z" : "A" = 0x41, "Z" = 0x5a

2. Subtract 0x40 from each ASCII value:

   Compressed "A" = 0x41-0x40 = 0x01 = 0000 0001
   Compressed "Z" = 0x5a-0x40 = 0x1A = 0001 1010

3. Retain five least significant bits for each letter:

   Discard three most significant bits (they are always zero)
   Compressed "A" = 0001. Compressed "Z" = 1010

4. Compressed code = concatenate "0" and the three 5-bit values:

   "AZA" = 0 00001 11010 00001 (16-bit value)
   0x*z*C80 = 00000111, 0x*z*C81h = 01000001

Figure 4-2 shows the format of the product ID (addresses 0x*z*C80-0x*z*C83).

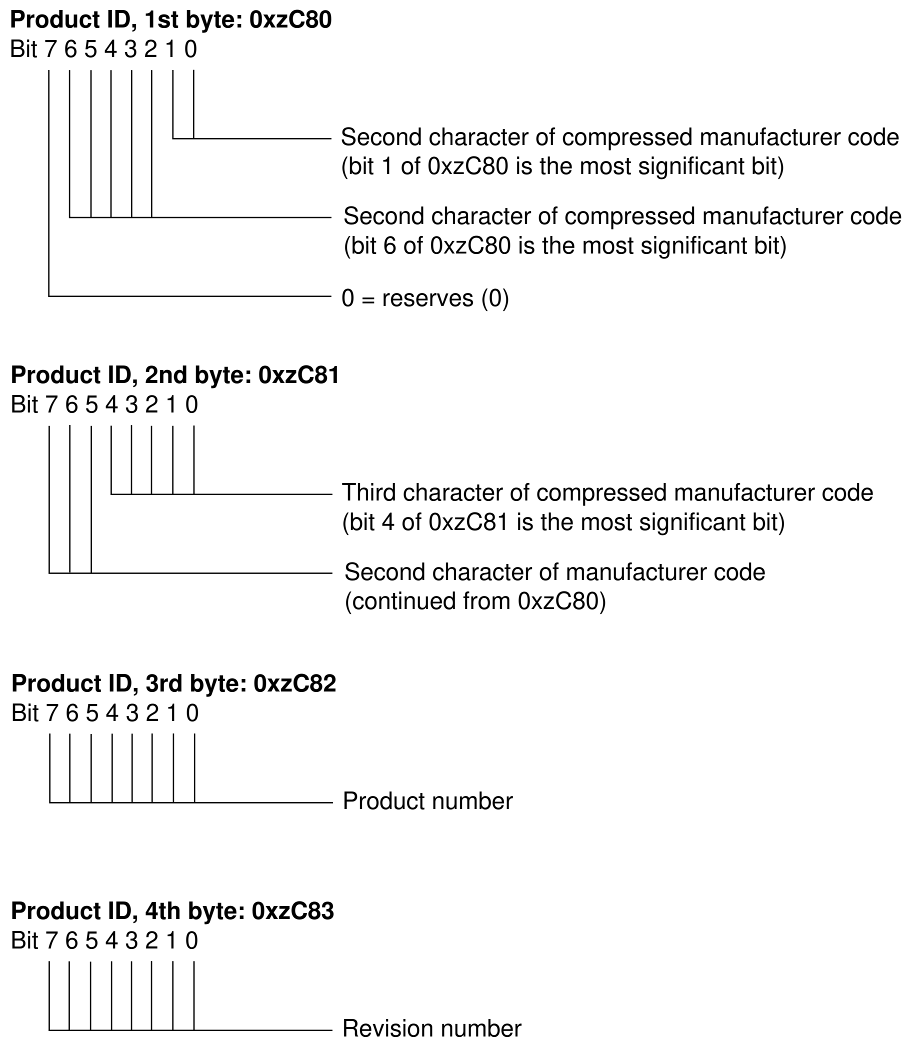


**Figure 4-2** Product ID Format

The manufacturer ID is located at bytes offset 0x*z*C80-0x*z*C83, where *z* represents the slot number probed. *exprobe_space* usage when searching for an EISA card is:

```
exprobe_space = (r, EISAIO, 0xzC80, 4, mfg_id, 0xffffffff)
```

The first argument (*r*) specifies that this probe consists of doing a single read. The second argument (*EISAIO*) declares that the address space is in the EISA register range. *0xzC80* is the offset of the manufacturer's unique ID. The size of this read is four bytes. *mfg_id* is the compressed 4-byte value that is returned by the particular card being searched for. The last argument is a mask which the *mfg_id* is ANDed against.

In addition to the system file described above, you must create a master file under */var/sysgen/master.d. (*The name of the master file is the same as the name of the object file for the driver, but the master file must not have the "*.o*" suffix.) The FLAG field of the master file must at least include the character device flag *c.* (You do not need the *s* flag for EISA device drivers because **lboot** can probe for EISA devices.)

A more detailed description of the master file can be found in Chapter 2, "Writing a Device Driver," and in the **master**(1M) man page.

The following example explores adding an ISA card device driver to the kernel. Assuming that the driver is called *isacard.o*, the following steps are necessary:

1. Copy the driver object file *isacard.o* to */var/sysgen/boot*.
2. Create an *isacard.sm* system file in */var/sysgen/system*.

   The file might contain the following lines:

   ```
   VECTOR: bustype=EISA module=isacard ctlr=0 adapter=0
   iospace=(EISAIO,0x300,48) probe_space=(wr,EISAIO,0x300,1,
   0x11,0xff)
   ```

   ```
   VECTOR: bustype=EISA module=isacard ctlr=1 adapter=0
   iospace=(EISAIO,0x330,48) probe_space=(wr,EISAIO,0x330,1,
   0x11,0xff)
   ```

   This would allow a kernel to support two of the same ISA cards located at different addresses. A one-byte test for card existence will be made at ISA I/O addresses 0x300 and 0x330. If either card returns the value written at one of those locations, the driver will be included in the kernel. Also a 48-byte ISA register address space will be allocated to the card without a memory address space being allocated in the example above.

3. Create a master file for the driver in the */var/sysgen/master.d* directory. Check */usr/include/sys/major.h* for available major device numbers or **lboot** can assign one. See Chapter 11, “Kernel-level Dynamically Loadable Modules (DLMs),” for more information on loadable drivers and master.d configuration:

   | *Flag | Prefix | Soft | #Dev | Dependencies |
   |---|---|---|---|---|
   | c | isa | 60 | – | |

   Note that even though the module name is referred to as *isacard*, the prefix for each of the driver routines can be independently declared, in this case as simply “*isa*.”

### System File Example

To add an EISA driver to the kernel, the system file might look like this:

```
VECTOR: bustype=EISA module=eisacard ctlr=0 adapter=0
        iospace=(EISAIO,0x1000,0x1000)
        iospace2=(EISAMEM,0xC8000,0x10000)
        exprobe_space=(r,EISAIO,0x1C80,4,
        0x00008107, 0xffffffff)
VECTOR: bustype=EISA module=eisacard ctlr=1 adapter=0
        iospace=(EISAIO,0x2000,0x1000)
        iospace2=(EISAMEM,0xD8000,0x10000)
        exprobe_space=(r,EISAIO,0x2C80,4,
        0x00008107, 0xffffffff)
VECTOR: bustype=EISA module=eisacard ctlr=2 adapter=0
        iospace=(EISAIO,0x3000,0x1000)
        iospace2=(EISMEM,0xE8000,0x10000)
        exprobe_space=(r,EISAIO,0x3C80,4,
        0x00008107,0xffffffff)
VECTOR: bustype=EISA module=eisacard ctlr=3 adapter=0
        iospace=(EISAIO,0x3000,0x1000)
        iospace2=(EISMEM,0xF8000,0x10000)
        exprobe_space=(r,EISAIO,0x3C80,4,
        0x00008107,0xffffffff)
```

This will support up to four cards. In this example, the manufacturer ID is AZA with a product number and revision number of zero. In addition to specifying the I/O register space on a per-slot basis, the *VECTOR*: lines in the above example also reserve 0x10000 bytes of memory space for each card. Each card placed in the system will be uniquely identified to the system by its *ctlr* number, which also happens to correspond directly to the slot number (although this is not a requirement).

## Writing edtinit()

If you use the *VECTOR*: directive to configure a driver into the kernel, your driver can use a routine of the form *drv***edtinit**(), where *drv* is the driver prefix. If your device driver object module includes a *drv***edtinit**() routine, the system executes the *drv***edtinit**() routine when the system boots. In general, you can use your *drv***edtinit**() routine to perform any device driver initialization you want. Typically, the **edtinit**() routine is used to allocate and map in the resources that are needed for the driver to initialize the hardware and execute. The resources that might need to be initialized are:

- semaphores or other locks used by the driver
- I/O and memory space
- interrupt (IRQ) inputs
- DMA channels.

When the system calls your driver's **edtinit**() routine, it hands the routine a pointer to a structure of type *edt*.

#### Synopsis

```
void drvedtinit(struct edt *e)
{
   your code here
}
```

The *edt_t* structure is filled in with the information taken from the system file when the operating system probes for the existence of the card. The *edt_t* structure is defined below.

#### edt_t Structure

The *edt_t* structure, defined in *system/edt.h*, contains resources specification information derived from the system file through the **lboot** operation.

### Synopsis

```
#define NBASE 3

typedef unsigned long iopaddr_t;
typedef struct iospace {
    unchar      ios_type;       /* io space type on the adapter */
    iopaddr_t   ios_iopaddr;    /* io space base address */
    ulong       ios_size;
    caddr_t     ios_vaddr;      /* kernel virtual address */
} iospace_t;

typedef struct edt {
    uint_t      e_bus_type;    /* vme, scsi, eisa... */
    unchar      v_cpuintr;     /* cpu to send intr to */
    unchar      v_setcpuintr;  /* cpu field is valid */
    uint_t      e_adap;        /* adapter */
    uint_t      e_ctlr;        /* controller identifier */
    void*       e_bus_info;    /* bus dependent info */
    int         (*e_init)(struct edt *); /* device init/run-time probe */
    iospace_t   e_space[NBASE];
} edt_t;
```

### Arguments

| | |
|---|---|
| *e_bus_type* | Defined as *EISA_ADAP* as specified by *bustype=EISA*. This parameter is used in several other places, such as **pio_mapalloc**(). |
| *v_cpuintr* | Currently unused. Reserved for future hardware use. |
| *v_setcpuintr* | Currently unused. Reserved for future hardware use. |
| *e_adap* | The Indigo$^2$ is the only EISA-capable Silicon Graphics system, and it has only one EISA-bus slot, so *e_adap* is 0. |
| *e_ctlr* | This software specified device number is taken from the *ctlr* field of the system file *VECTOR*: line to differentiate more than one instance of the same device. |
| *e_bus_info* | This bus-dependent field is presently unused by EISA drivers. For bus type *EISA_ADAP*, it points to *NULL*. |
| *e_init* | Device initialization routine. |
| *e_space* | This array of *iospace_t* declarations tells the driver which I/O and memory spaces the hardware is programmed to respond to. |

## I/O and Memory Space Initialization

In the example below, *e_space* array contains the *iospace* structures that are used to map in the card's address space. There are two steps in this process: allocating a PIO map and then mapping the requested bus space address into a kernel address.

On the Indigo$^2$ the mapping is a fixed mapping, which permanently maps the entire EISA I/O and memory address space into kernel space, but using the **piomap**() call will allow your driver to remain fully compatible with potential future hardware that uses programmable I/O space map registers like those used for VME on the CHALLENGE and Onyx architectures. (Simplifies porting of the driver to future hardware.)

**Note:** The device will only respond to the requested memory address range if it has been programmed through software (in the case of an EISA card) or jumpered on the card (in the case of an ISA card).

Before actually beginning to program the card after mapping in the I/O space, it is wise to probe for the card's existence by writing and rereading a register located on the card. This allows the driver to recover if the device has been removed from the system since the kernel was built or if the kernel is copied to another system with a different hardware configuration.

**Caution:** The probe during **edtinit**() has a bus error handler in place. After this, a bus error ordinarily causes an operating system panic.

In the **edtinit**() for your driver, you must allocate the necessary mapping resources before you actually map the memory address. The mapping resource allocation and mapping calls apply across all of Silicon Graphics bus adapters (i.e., VME). Their structures are defined in <*sys/pio.h*.>.

```
piomap_t *pio_mapalloc(uint_t bus, uint_t adap, iospace_t
          iospace, int flag, char *name)
```

This routine allocates a handle that specifies a mapping from kernel virtual space to I/O and memory address space. The returned *piomap_t* handle is used to provide the mapping to the address space through additional PIO access functions.

Currently, the *bus* and *adap* arguments must be *ADAP_EISA* and *0* respectively, because there is only one EISA bus on Indigo$^2$ platforms. The name argument is a string used to identify which device has a particular space mapped.

The actual mapping is returned by:

```
caddr_t pio_mapaddr(piomap_t *piomap, iopaddr_t io)
```

This function returns the kernel virtual address that maps to the PIO-mapped I/O space address.

Silicon Graphics also supports a set of routines that allow a driver to operate on an I/O address space that has an allocated *piomap*, but does not have a kernel virtual address mapped to the I/O space. For example:

```
void pio_bcopyin(piomap_t *pmap, iopaddr_t io, caddr_t a,
    int len)
void pio_bcopyout(piomap_t *pmap, iopaddr_t io, caddr_t a,
    int len)
```

For a complete specification of the **piomap*** functions, see <*sys/pio.h*>.

**Note:** There is not currently any advantage to using the routines listed above, although they may be useful on future Silicon Graphics platforms.

## Dynamic Resource Allocation

Rather than bind the interrupt IRQ input and a specific channel for DMA to a specific *VECTOR:* entry at **lboot** time, the driver allocates these resources dynamically. In the case of interrupts, there are three parts to the process:

1. A specific IRQ number is obtained from the system on a "first come first served" basis through a call to **eisa_ivec_alloc**(). The mask parameter passed in its 16-bit bitmap vector specifies which IRQ choices are acceptable for this device hardware. This is necessary because most EISA cards can only be programmed to respond to a limited number of IRQ possibilities. In the case of an ISA card, which is jumpered to only respond to a single IRQ input, the vector only has a single bit "on" to specify this particular setting. Multiple cards can be assigned the same IRQ, although, by default, the allocation routine tries to assign an unused interrupt input. The allocation routine only returns failure if the

driver requests a vector for a mask, and all mask choices are already programmed to respond to an incompatible choice for edge- or level-sensitive interrupts. Each IRQ can respond to either edge- or level-sensitive interrupts, but not to both.

The EISA interrupt priority level is used to specify the interrupt IRQ number for the device. Its range is IRQ0-IRQ15, excluding IRQ2, which is used to cascade the interrupt controllers. EISA directly associates the interrupt priority to the IRQ input line. See the *Intel 82357 Specification* or the *EISA Technical Reference Guide* for a description of the priority ordering scheme.

There are two system resources that need to be reserved by some EISA/ISA drivers: interrupt request inputs (IRQ's) and DMA channels. Instead of pre-allocating these in the system file, a dynamic allocation scheme hands them out to the driver at initialization time. The driver is responsible for specifying which choices for IRQs and DMA channels are acceptable for the particular hardware. The operating system uses the appropriate allocation routine to assign them to the driver, which then uses the returned value to configure the hardware to respond appropriately.

2. The card is programmed to generate its interrupt on the assigned IRQ. This is a device-specific procedure. ISA cards typically cannot change their IRQ through software; they usually have to be jumpered.

   The value must be selected from IRQ 3-7, 9-12, 14, and 15. All EISA interrupts are channeled into one CPU interrupt level. The priority of this CPU interrupt is below that of the clock and at the same level as on-board devices. Although the EISA interrupt is generated at the same level, it is serviced when the CPU services EISA interrupts. It services the EISA-bus interrupts in order of their EISA-bus priority, at least in that order when multiple devices interrupt at the same time.

3. A specific routine and argument input is associated with the assigned IRQ. This is done through the call to **eisa_ivec_set**().

Once **eisa_ivec_set**() has been used to bind the selected IRQ to an interrupt handling routine, all interrupts generated to that IRQ cause the related handling routine to be called. Since multiple cards may be responsible for the interrupt, the driver is required to check its own hardware registers to test for the cause of the pending interrupt. The call is specified as:

```
eisa_ivec_set (vint_t adapter, int irq, void (func*)(),
     int arg);
```

*func* passes *arg* as an argument when called. Typically, *arg* corresponds to the *ctlr* number of the device.

The process for allocating a DMA channel is similar to that for obtaining an interrupt number. The call to the **eisa_dmachan_alloc**() routine uses an 8-bit mask to represent DMA channels zero through seven. Note that bit four must always be masked because that channel is unavailable for a card's case. The value returned by the channel allocation routine is then saved to program the device to transfer data on that particular channel, as specified in the section describing the EISA DMA utility functions.

The following example is a possible outline initialization routine for the EISA device module declared in the previous section about the system files:

```
#include <sys/types.h>
#include <sys/edt.h>
#include <sys/pio.h>
#include <sys/eisa.h>
#include <sys/cmn_err.h>

struct edrv_info {
   caddr_t e_addr[NBASE];
   int     e_dmachan;
} einfo[4];

#define CARD_ID        0x0163b30a
#define IRQ_MASK       0x0018
#define DMACHAN_MASK   0x7a

edrv_edtinit(edt_t *e)
{
   int iospace, eirq, edma_chan;
   struct edrv_info *einf;
   piomap_t *pmap;
   einf = &einfo[e->e_ctlr];
```

```
    /* map address spaces */
    for (iospace = 0; iospace < NBASE; iospace++) {
       if (!e->e_space[iospace].ios_iopaddr)
          break;

       pmap = pio_mapalloc(e->e_bus_type, e->e_adap,
          &e->e_space[iospace], PIOMAP_FIXED, "edrv");

       einf->e_addr[iospace] = pio_mapaddr(pmap,
          e->e_space[iospace].ios_iopaddr);
    }

       /* should mark not-present somewhere */
       return;
    }

    /* Initialize Interrupt Vector */
    eirq = eisa_ivec_alloc(e->e_adap, IRQ_MASK,
           EISA_EDGE_IRQ);
    if (eirq < 0) {
       cmn_err(CE_WARN,
       "edrv: ctlr %d could not allocate IRQ\n",
          e->e_ctlr);

       /* should mark unavailable */
       return;
    }

    eisa_ivec_set(e->e_adap, eirq, edrv_intr, e->e_ctlr);

    /* Allocate DMA Channel */
    edma_chan = eisa_dmachan_alloc(e->e_adap, DMACHAN_MASK);
    if (edma_chan < 0) {
       cmn_err(CE_WARN,
       "edrv: ctlr %d could not allocate DMA Chan\n",
          e->e_ctlr);

       /* should mark unavailable */
       return;
    }
    einf->e_dmachan = edma_chan;

}
```

## EISA Locked Cycles

The EISA specification provides that you can lock the bus and hold it for exclusive access if you assert **LOCK***. This allows atomic operations such as a test-and-set, and thus sets the basis for implementing semaphores between the CPU and bus masters. Because the actual system bus (a GIO bus) does not support a **LOCK*** operation, Silicon Graphics provides an interface that the CPU can use for read-modify-write style instructions.

On Silicon Graphics systems that support EISA, if a bus master asserts **LOCK***, some additional constraints apply to how long the bus master can hold the lock. Your locks should work correctly as long as your usage conforms to the read-modify-write paradigm used by the CPU programmatic interface. The following routines, which are also supported for VME device drivers, provide atomic *and/or* operations: they find the appropriate-sized mask by reading the PIO address, then they rewrite, modified by the mask value *m*.

```
void pio_orb_rmw(piomap_t* pio, iopaddr_t io, uchar m)
void pio_orh_rmw(piomap_t* pio, iopaddr_t io, ushort m)
void pio_orw_rmw(piomap_t* pio, iopaddr_t io, uint m)
void pio_andb_rmw(piomap_t* pio, iopaddr_t io, uchar m)
void pio_andh_rmw(piomap_t* pio, iopaddr_t io, ushort m)
void pio_andw_rmw(piomap_t* pio, iopaddr_t io, uint m)
```

## DMA Address Mapping

DMA devices use a logical address. If your driver passes addresses to bus masters, it must use a special set of DMA mapping routines to map the DMA target into physical addresses. The system uses these special mapping routines to support physical address spaces larger than 32 bits. On Silicon Graphics systems that support EISA, the physical address is used. The map structure returned, *dmamap_t*, is defined in *sys/dmamap.h*.

```
dmamap_t  *dma_mapalloc(int bus, int adap, int npages, int flag)
void       dma_mapfree(dmmap_t *dmamap)
int        dma_map(dmamap_t *dmamap, caddr_t addr, int len)
uint       dma_mapaddr(dmamap_t *dmamap, caddr_t addr)
```

Indigo[2] systems do not support bus address mapping, so calls to **dma_map** on an Indigo[2], as opposed to a CHALLENGE, system are limited to a single page. Multiple virtual pages cannot be mapped into physically contiguous pages with **dma_map** unless the virtual pages have been allocated to be contiguous.

For a complete description of these routines, refer to "Using DMA Maps" in Chapter 3, "Writing a VME Device Driver."

The mapped address returned by **dma_mapaddr**() must be used when specifying the bus address used by bus master cards or the following DMA routines. The map type used must be *DMA_EISA*.

## EISA DMA Utility Functions and Structures

Table 4-1 lists the DMA utility routines.

**Table 4-1** DMA Utility Routines

| Routine | Description |
|---|---|
| **eisa_dma_free_buf** | Free a previously allocated DMA buffer descriptor. |
| **eisa_dma_free_cb** | Free a previously allocated DMA command block. |
| **eisa_dma_get_buf** | Allocated DMA a buffer descriptor. |
| **eisa_dma_get_cb** | Allocated DMA a command block. |
| **eisa_dma_disable** | Disable recognition of hardware requests on a DMA channel. |
| **eisa_dma_enbable** | Enable recognition of hardware requests on a DMA channel. |
| **eisa_dma_stop** | Stop a software-initiated DMA operation on a channel and release it. |
| **eisa_dma_fswstart** | Initiate a DMA operation via software request. |
| **eisa_dma_prog** | Program a DMA operation for a subsequent software request. |