IRIX™ Device Driver Programming Guide

Document Number 007-0911-050

CONTRIBUTORS

Revised and rewritten by Gary Sloane
Production by Gloria Ackley
Engineering contributions by Rich Altmaier, Rick Avila, Brian Berg, Brad Eacker, Lara Fabans, Bruce Johnson, Ben Mahjoor, Jay McCauley, Rick McLeod, Huy Nguyen, Neal Nuckols, Dave Olson, Alex Petruncola, Duncan Poole, Bhanu Prakash, Joyce Richards, Sarah Rosedahl, George Smith, Robert Stephens, Jean Tsai, and Steve Whitney.
Reviewed by David Cortesi, Dan Hachigian, Bill Fisher, Irene Kuffel, Tom Mitchell, Jim Putnam, Kathleen Roy, Mike Shannon, Bob Shaw, Jeff Smith, Kris Solem, Chris Wagner, Bill Warner, Ted Wilcox, Julie Yen, and Joe Yetter

IRIX™ Device Driver Programming Guide
Document Number 007-0911-050

Contents

Figures

Tables

Introduction

About This Guide

This manual, the *IRIX Device Driver Programming Guide*, provides information and procedures for developing, installing, and testing UNIX® device drivers for IRIX™ 5.2, 5.3, and 6.0.

Based on *Writing Device Drivers for Silicon Graphics Workstations* (007-0910-010), first published in 1989, the current version contains numerous corrections and updates as well as information for new platforms and operating systems.

The *IRIX Device Driver Reference Pages*, contain all the reference pages (man pages) relevant to writing user-level and kernel-level device drivers. (See "Related Documentation" and "Reference Material"for ordering information.)

Audience

This manual is a guide to writing device drivers for Silicon Graphics® workstations and servers. It is intended for experienced C programmers and C++ programmers who have a good working knowledge of the architecture of Silicon Graphics computer systems.

Further information and support are available through the Silicon Graphics Developer Program. For information on program membership, please contact the Developer Response Center at (800) 770-3033 or (415) 390-3033, or send email to `devprogram@sgi.com`.

Document Overview

This guide contains the following chapters and appendices:

Chapter 1, “Introduction to Device Drivers,” introduces basic concepts of devices and provides information on the system hardware / software.

Chapter 2, “Writing a Device Driver,” describes the general interface for both user-level and kernel-level device drivers and introduces the various device driver models.

Chapter 3, “Writing a VME Device Driver,” describes the VME-bus and explains how to write user-level and kernel-level VME device drivers.

Chapter 4, “Writing an EISA Device Driver,” describes the EISA-bus interface and explains how to write user-level and kernel-level EISA device drivers.

Chapter 5, “Writing a SCSI Device Driver,” describes the SCSI-bus interface and explains how to write user-level and kernel-level SCSI device drivers.

Chapter 6, “Writing Kernel-level GIO Device Drivers,” describes the GIO-bus interface and explains how to write kernel-level GIO device drivers.

Chapter 7, “Writing Kernel-level General Memory-mapping Device Drivers,” explains how to write kernel-level general memory-mapping device drivers.

Chapter 8, “Writing Multiprocessor Device Drivers,” addresses questions about device drivers that run on multiprocessor workstations.

Chapter 9, “Writing Network Device Drivers,” addresses questions particular to device drivers that run on networked workstations.

Chapter 10, “Driver Installation and Testing,” describes ***symmon***, the kernel debugger, and explains how to use it.

Chapter 11, “Kernel-level Dynamically Loadable Modules (DLMs),” describes how kernel modules can be loaded dynamically.

Appendix A, “System-specific Issues,” provides information on various CPUs and platforms. It addresses, among other topics, the differences in data cache invalidation, write buffer flushing, and VME addressing.

Appendix B, “SCSI Controller Error Messages,” lists common error messages.

Appendix C, “Device Driver Migration Notes,” gives the information required to make earlier IRIX device drivers compliant with releases 5.2, 5.3, and 6.0.

The Glossary contains definitions of some useful terms for device driver writers; the Index provides another set of entry points to the material in this manual.

Related Documentation

- *GIO Bus Specification*, Version 2.1, Silicon Graphics, Inc.
- *IRIX Device Driver Reference Pages*, Silicon Graphics, Inc., document number 007-2183-003
- *MIPSpro Porting and Transition Guide*, Silicon Graphics, Inc., document number 007-2391-001
- *STREAMS Programmer’s Guide*, Version 1.0, Silicon Graphics, Inc., document number 007-0833-020

Reference Material

- ANSI standards *X3.131-1986, 1014-1987*, and *X3T9.2/85-52 Rev 4B.*
- Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX® Device Driver.* John Wiley & Sons, 1992.
- Heinrich, Joseph. *MIPS R4000 User’s Manual*. PTR Prentice Hall, 1993.
- Hines, Robert M., and Spence Wilcox. *Device Driver Programming*, UNIX SVR4.2. Englewood Cliffs, New Jersey: UNIX Press, 1992.
- Kane, Gerry, and Joseph Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

- Leffler, Samuel J., et alia. *The Design and Implementation of the 4.3BSD UNIX® Operating System*. Palo Alto, California: Addison-Wesley Publishing Company, 1989.
- M. Maekawa, A. Oldehoeft, and R. Oldehoeft. *Operating Systems Advanced Concepts*. The Benjamin / Cummings Publishing Company, Inc., 1987.
- A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*, Third Edition. Addison Wesley Publishing Company, 1991.
- *STREAMS Modules and Drivers*, UNIX SVR4.2. UNIX Press, 1992.
- *UNIX System V Release 4 Programmer's Guide*, UNIX SVR4.2. UNIX Press, 1992.

Notation and Syntax Conventions

This guide uses the following notation and syntax conventions:

italics Indicates arguments in a command line that you must replace with a valid value. In commands and functions, it indicates a portion of the name that is a variable, for which you must make the appropriate substitution, as well as function arguments. In text, italics indicate document titles, filenames, glossary items, new terms, and variables.

bold Indicates commands, routines, and entry point names, as well as the names of UNIX and IRIX functions. A function with a man page in the *IRIX Device Driver Reference Pages*, manual is indicated as follows:

ioctl(D2)

where D2 is the number of the section that contains the man page.

Note: If a reference to a function is hyperlinked to the online version of the *IRIX Device Driver Reference Pages*, it looks like this instead:

ioctl(D2)

	The online version displays in red, and you can go directly to the online man page by clicking on the entry.
`courier`	Indicates computer output and program listings.
`courier bold`	Indicates user input to the computer and nonprinting keys.
[ ]	Enclose optional command arguments. (Do not enter the brackets.)
. . .	Indicates that the preceding optional items can appear more than once in succession.
|	Separates a list of items, of which you can choose one.

Chapter 1

Introduction to Device Drivers

This chapter introduces the basic concepts of hardware and software devices and provides information on the system hardware, including the MIPS processor and I/O bus architecture and the operating system software.

It contains the following sections:

Driver Overview

A *device driver* is a software module that enables communication between a *user process* and a *peripheral device*. It may perform some or all of the following functions:

- Take the device online and offline
- Set parameters in the device
- Transmit data from the kernel to the device
- Receive data from the device and pass it to the kernel
- Handle and report I/O errors
- Handle exclusion and other multiuser, multitasking arbitration

Device Types

There are two basic types of devices available on any UNIX system: software devices, such as RAM disks, and hardware devices, such as hard disks and printers. Most of the discussions in this book are about hardware devices.

Software Devices

In a UNIX system, the “device” driven by a software driver is usually a section of memory and is referred to as a *pseudo-device*. The function of a pseudo-device driver may be to provide access to system structures that are unavailable at the user level.

Hardware Devices

Some examples of hardware devices are CD ROMs, disk drives, tape drives, printers, scanners, and terminals.

Hardware devices are categorized as *block devices*, *character devices*, *mmapped devices*, or *networked devices*. A block device is a mass storage device (such as a disk) that can accept data, store it, and return data to the processor in fixed-length transfers. A block device driver uses the integrated page cache for all data transfers. Device drivers that support the block interface are complex and are not covered in this manual.

A character device (such as a terminal, network interface, or plotter) is a device that deals with arbitrary streams of data that typically have no particular structure. In addition, many character devices impose alignment restrictions (such as *quad-aligned*) and often require that you transfer data in multiples of the device's fundamental size. In particular, most IRIX devices doing DMA require the starting address at least to be aligned on a 32-bit boundary (the lowest two bits of the address are zeros). Unlike block devices, character devices do not use the integrated page cache.

Mmapped device drivers are those in which the hardware is memory mapped into a user's address space. No interrupt or DMA service routine is available to the user process.

Networked device drivers are covered in Chapter 9, "Writing Network Device Drivers".

It is possible, however, for some devices to fit both systems. Disk drivers often allow blocked, cached access as well as character, uncached access. Generally speaking, though, custom device drivers are most often written for character devices.

In some cases, a controller board may have more than one device connected to it. A SCSI-bus controller board, for example, normally has up to seven devices attached to it, and there may be multiple boards.

Levels of Device Drivers

There are two level of device drivers: *user-level* and *kernel-level*. For some devices, such as *GIO bus* cards, the device driver should be a kernel-level driver.[1] However, for devices that interface to a *SCSI bus*, *EISA bus*, or *VME bus*, it is possible to write a user-level device driver that controls the device by communicating directly to the bus.

[1] Although it is possible to write a user-level GIO bus driver, it is discouraged because the user-level interfaces are not publicly available; in any case, most GIO bus boards are designed to take advantage of DMA, which requires a kernel-level driver.

User-level Device Drivers

Users cannot always treat the user-level device as just another file to be opened, read, written, and closed with the standard IRIX system commands. If you write a user-level driver, you may have to provide your users with device-specific routines or encapsulate the functionality in an application. This is normally the case with printers and scanners, for example.

Kernel-level Device Drivers

Deciding whether you can write a user-level driver is not difficult. It is also fairly easy to decide whether to write a VME bus, EISA-bus, or SCSI-bus user-level driver. However, if you decide to write your own kernel-level device driver, it is a little more difficult to decide what sort of kernel-level device driver to write. This guide provides you with the criteria you need to determine the appropriate driver model for a given device.

Note: Because IRIX kernels cannot, as a rule, be preempted, any driver that sits in a loop waiting for some condition to be satisfied may tie a processor up for as long as it wants. Real-time processes, such as audio, are very sensitive to such delays.

System Hardware

The Silicon Graphics Indigo™, Indigo ^{2}™, Indy™, Crimson™, CHALLENGE™/Onyx™, and POWER CHALLENGE™/POWER Onyx™ families of workstations and servers may contain the following hardware components:

- One or more MIPS® RISC CPUs
- Local memory bus
- Zero or more VME-bus adapters
- Zero or more EISA-bus adapters
- One or more SCSI-bus adapters
- Zero or more GIO-bus adapters

Although each Silicon Graphics system provides a similar architectural interface, there are some hardware-specific differences that affect how you write a device driver. Most of this guide discusses only those features that are common to all systems. For a description of hardware-specific differences, see Appendix A, “System-specific Issues” (which also describes how to write drivers that work correctly across all Silicon Graphics systems).

Hardware Platforms

Each basic hardware design results in differences in the operating system kernel such that a driver must be compiled for each architecture. Because there is some duplication of CPUs across hardware architectures, Table 1-1 may be useful.

Table 1-1 Hardware Series and the CPUs They Use

Product Family	CPU	R2000	R3000	R4000	R8000
POWER CHALLENGE / POWER Onyx	IP21				X
POWER Indigo^2 Series	IP26				
CHALLENGE / Onyx Series	IP19			X	
Crimson Series	IP17			X	
Indigo Series	IP12		X		
	IP20			X	
	IP22			X	
Indigo^2 Series	IP22			X	
Indy Series	IP22			X	
IRIS-4D™ / 20 / 30 / 100 / 200 / 300 / 400 Series	IP4	X			
	IP5	X			
	IP6	X			
	IP7	X			
	IP9	X			
	IP11		X		
	IP12		X		
	IP15		X		

For purposes of writing device drivers, however, all R4000-series processors may be considered identical, although their clock speeds and performance characteristics may vary. That is, source code can be the same if interfaces are followed carefully. For further details, see “CPU Types” in Appendix AMIPS RISC Processors

All MIPS 32-bit and 64-bit RISC processors have an on-chip memory management unit (MMU) that supports demand-paged virtual memory. For detailed information on the MIPS architecture, see *MIPS RISC Architecture*.

MIPS RISC Processors Interrupt Masking

Each device interrupts the CPU at a specific *interrupt priority level*. While the CPU is serving an interrupt, it ignores any other interrupts at the same or lower interrupt level. To prevent device interrupts from occurring before your driver is ready for them, your driver can raise the processor interrupt level in the device driver at any time. After your driver executes the critical segment of code, it must restore the previous interrupt priority.

Note: Only kernel-level drivers can handle interrupts.

Raising the interrupt level is usually not sufficient to prevent your driver from being interrupted on multiprocessing systems. (See “Reliable Multiprocessor Spinlocks” in Appendix A, “System-specific Issues.”) Drivers on multiprocessing systems must use additional mechanisms, such as semaphores and spinlocks. (See psema(D3X) in the *IRIX Device Driver Reference Pages*.)

Understanding Driver Address Space

Drivers have different functional needs for addresses, including:

- Mapping to (usually cached) physical memory for the driver’s own code
- Static and stack data
- Dynamically allocated data
- Mapping to I/O control registers (called Programmed I/O or PIO)
- DMA address to map to physical memory for a controller to use

To describe kernel-resident driver address spaces, first recall the following points about the form of addresses in a *user process*:

- The virtual address is either 32 or 64 bits.
- The most significant bits are those in the virtual page number, which is translated to a physical page.
- An invalid address causes the user process to get a SIGSEGV, which typically results in a core dump.

With respect to addresses in a *kernel-resident driver*:

- The virtual address is either 32 or 64 bits.
- A range of values, varying by processor type, called *kseg0*, translates 1:1 to physical addresses.
- *kseg0* is often used for kernel code and data, as well as for some PIOs.
- A range of values, translated by the *translation look-aside buffer* (TLB), are often used for dynamically allocated kernel data.
- A driver should not assume that it knows that one type or the other is in use.

A driver also has to manipulate DMA addresses. These address values cannot be used for driver (processor) load / store instructions; rather, they are for controller usage in DMA operations.

Caution: If a driver executes a load / store to an address that is not valid, data corruption may result, or the kernel may panic.

Virtual to Physical Memory Mapping

The R2000 / 3000 uses 4096-byte pages for virtual address mapping in the format shown in Figure 1-1. The most significant 20 bits of a 32-bit virtual address (the virtual page number, or VPN) allow mapping of 4 KB pages. The least significant 12 bits (offset within a page) are passed along unchanged. The three most significant bits of VPN (bits 31-29) further define how the addresses are mapped, according to whether the R2000 / 3000 processor is in *user mode* or *kernel mode.*

Note: For all device drivers, the R2000 and the R3000 processors are considered identical.

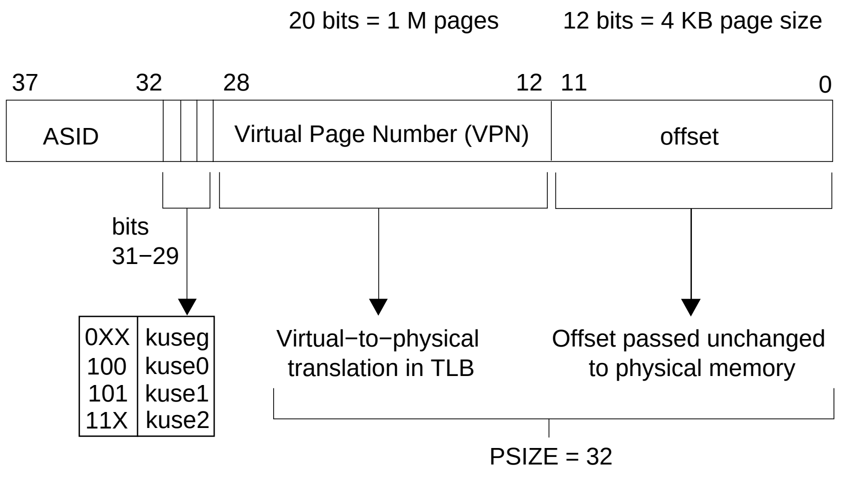


Figure 1-1 MIPS 32-bit Virtual Address Format (MIPS II Mode)

The Crimson, R4000 Indigo, Indigo^2, and Indy series workstations use a MIPS R4000 series microprocessor in MIPS II mode (see Figure 1-1). R4000 MIPS II mode implements the same address map as R2000/3000. (See the *MIPS R4000 User's Manual* for further details.)

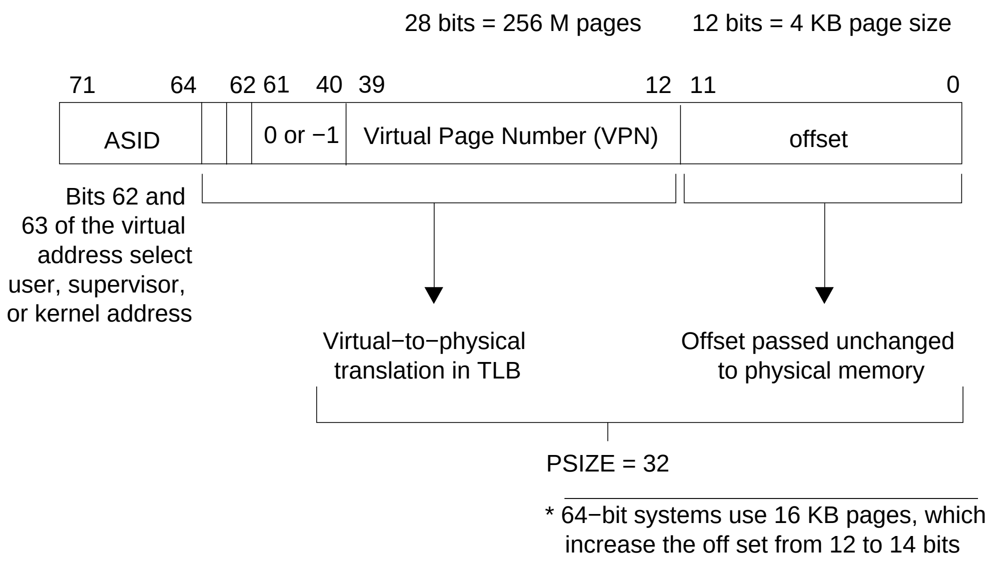

28 bits = 256 M pages
12 bits = 4 KB page size
71
64
62
61
40
39
12
11
0
ASID
0 or −1
Virtual Page Number (VPN)
offset
Bits 62 and 63 of the virtual address select user, supervisor, or kernel address
Virtual−to−physical translation in TLB
Offset passed unchanged to physical memory
PSIZE = 32
* 64−bit systems use 16 KB pages, which increase the off set from 12 to 14 bits

Figure 1-2 MIPS 64-bit Virtual Address Format (MIPS III Mode)

The CHALLENGE/Onyx series uses the R4400, which is functionally the same as the R4000 for driver purposes, and the POWER CHALLENGE/POWER Onyx series uses the R8000 processor. All MIPS processors use the same address mapping scheme in 32-bit mode; in 64-bit mode, they use R8000 (MIPS III) address mapping (see Figure 1-2).

Privilege States/Modes

The R2000/3000 provide two privilege modes:

Kernel	Analogous to the “supervisor” mode provided by other systems.
User	The mode in which the system executes non-supervisory programs.

The R4000/4200/4400/4600 provide three privilege modes:

Kernel	Full privilege state (same as the R2000/3000 kernel mode).

Supervisor	A state of lesser privilege than kernel mode. When executing in supervisor state, the system has access to the supervisor and user address space, but not the kernel address space. This mode is currently unused in IRIX.
User	The same as the R2000/3000 user mode.

The R8000 also provides three privilege modes:

Kernel	Full privilege state.
User 32-bit	The same as the R2000/3000/4000 user mode.
User 64-bit	R8000 64-bit user mode.

User Mode Virtual Addressing

In user mode, a 32-bit process has 2 GB of virtual address space, appearing to start at location zero. Therefore, all valid user-mode virtual addresses have the most significant bit cleared. If, when in user mode, your code tries to reference an address with the most significant bit set, it will generate an *Address Error Exception*. To help programmers detect a common error, page 0 is never mapped.

Kernel Mode Virtual Addressing

Because kernel virtual memory divides physical memory several different ways, you can control the use of data caches and *Translation Look-aside Buffers* (TLBs) by specifying ranges of virtual addresses with different attributes.

When the processor is operating in kernel mode, three distinct address spaces (in addition to *kuseg*) are simultaneously available:

kseg0 — This virtual address range is cached but not mapped by the TLBs. This is the type of memory you get when you declare a global in your driver code. This memory goes through the data cache during read / write operations but is not mapped by the TLBs. It is easy to convert the *kseg0* address to a physical address by using the masking address bits.

Note: Driver globals may not always reside in *kseg0*; with loadable drivers, they may live in *kseg2*.

kseg0 consists of 512 MB of cached, unmapped address space, starting at virtual address 0xa800000000000000.

When the most significant three bits of an address are "100," the virtual address space selected is the 512 MB of kernel physical space, *kseg0*. The R2000 / 3000 directly maps references within *kseg0* onto the first 512 MB of physical address space. These references use cache memory, but they do not use TLB entries. Typically, the operating system uses this segment of memory for kernel-executable code and static kernel data.

kseg1

This virtual address range is neither cached nor mapped. Memory does not go through the data cache during read / write operations, nor are the addresses translated by the TLBs. This space is used for *volatile memory.*

kseg1 consists of 512 MB of uncacheable, unmapped virtual address space, starting at 0x9000000000000000.

When the most significant three bits of a virtual address are "101," the virtual address space selected is the 512 MB of kernel physical space, *kseg1*. The processor directly maps *kseg1* onto the first 512 MB of physical space. The operating system typically uses *kseg1* for I / O registers and ROM code.

kseg2

Although this virtual address range can be both cached and mapped by the TLBs, it is not physically contiguous. This is where automatic variables declared in your driver come from, and it is where calls such as **kmem_alloc**() get their memory.

Note: Use the *PHYSCONTIG* flag to request physically contiguous pages.

kseg2 consists of and address range of 1024 MB of cacheable, mapped virtual address space starting at 0xc000000000000000.

When the most significant two bits of the virtual address are "11," the virtual address space selected is the 1 GB of kernel virtual address, *kseg2*, which uses TLB entries to map virtual addresses to arbitrary physical ones, with or without caching. The operating system uses *kseg2* for stacks and per-process data that it must trap on context switches, for user page tables (memory map), and for some dynamically allocated data areas.

kuseg	This user physical space has only physical pages, from the point of view of a device driver. In kernel mode, access generates a bus error.

Kernel virtual memory spaces *k0*, and *k1* remain mapped unless you specifically unmap them; consequently, you can read from and write to these spaces from the bottom half of your driver. This is not true for *kuseg*.

MIPS processors enter kernel mode whenever an interrupt, a system instruction, or an exception occurs, and return to user mode only with a "Return from Exception" instruction. In general, address mapping is different for user and kernel modes. However, the translation lookaside buffer (TLB) maps all references to user address space, *kuseg*, identically, whether those references are made from kernel or user mode. In addition, the TLB controls cache access. Figure 1-3 is a diagram of the address / data path flow corresponding to the preceding descriptions.

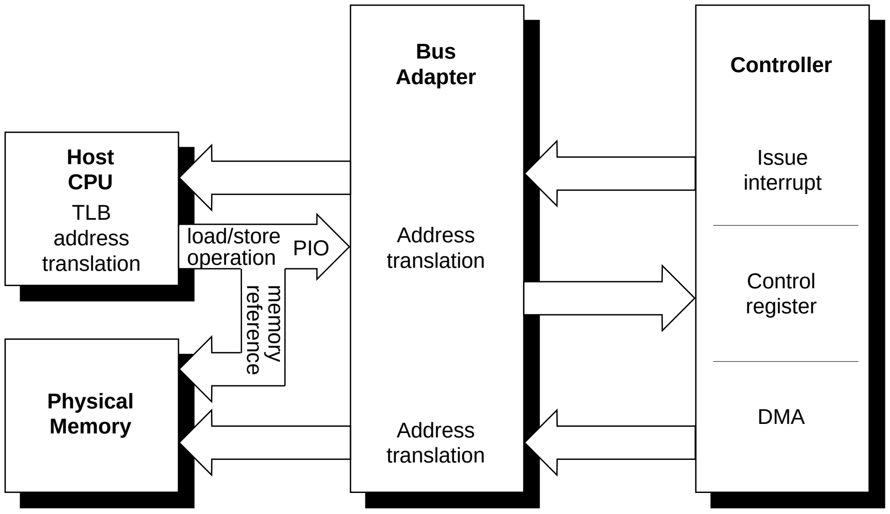


Figure 1-3 Architectural Block Diagram of Address / Data Flow

To simplify the management of user mode from within the kernel, the user-mode address space is a subset of the kernel-mode address space.

Figure 1-4 illustrates the virtual-to-physical memory mapping for both user and kernel modes, and Figure 1-5 contrasts 32-bit (MIPS II) with 64-bit (MIPS III) modes for R4000 and R8000 platforms. There is a description of address mapping in various modes after the figures.

Note: Not all systems have physical memory at location 0. Also, while the class of device determines the VME address range, each GIO device responds to the same address range.

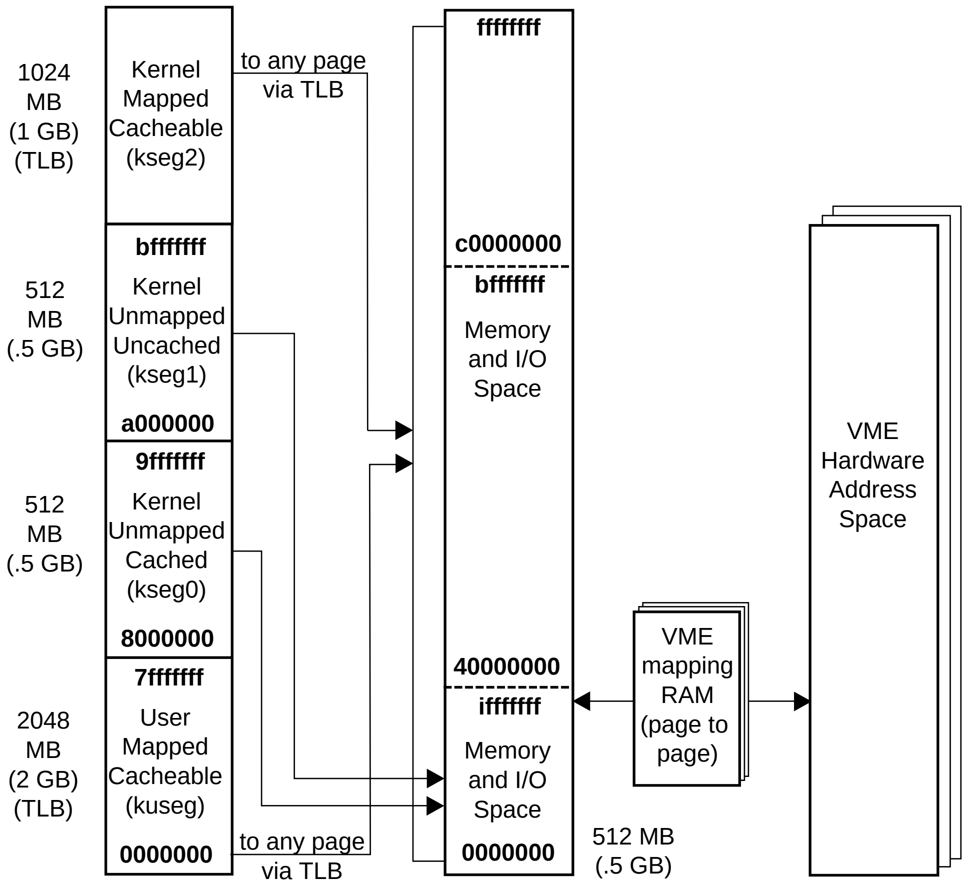


Figure 1-4 MIPS II Virtual Memory Map

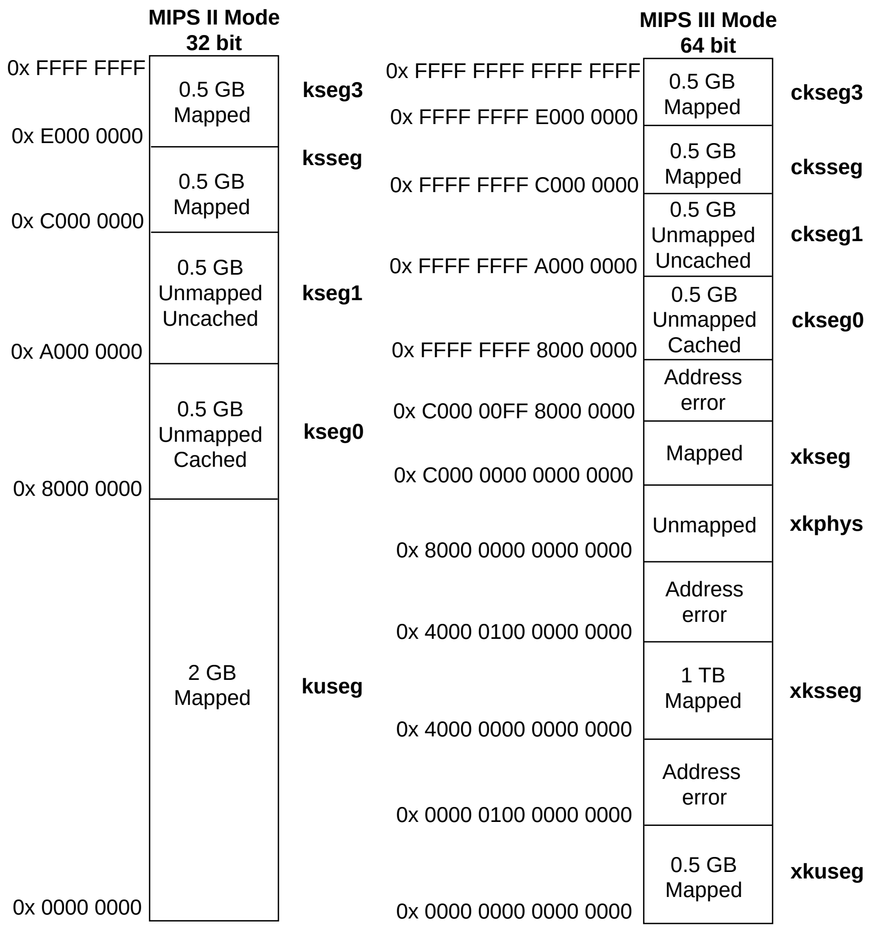


Figure 1-5 MIPS II 32-bit versus MIPS III 64-bit Kernel Mode Address Space

Bus Interfaces

There are several types of bus interfaces available for Silicon Graphics workstations and servers:

- VME-bus interface
- SCSI-bus interface
- EISA-bus interface
- GIO-bus interface

Not all bus interfaces are available on all systems. Table 1-2 lists the bus interfaces available for each Silicon Graphics platform. The individual bus interfaces are discussed briefly below.

Table 1-2 Bus Interfaces for Silicon Graphics Platforms

Product Family	**VME**	**SCSI**	**EISA**	**GIO**
POWER CHALLENGE / POWER Onyx Series Systems	X	X		X[a]
CHALLENGE / Onyx L and XL Series Systems	X	X		X[a]
Crimson Series Systems	X	X		X[b]
Indigo Series Systems		X		X
CHALLENGE M and Indigo^2 Series Systems		X	X	X
CHALLENGE S and Indy Series Systems		X		X
IRIS-4D / 20 / 30 / 100 / 200 / 300 / 400 Series Systems	X	X		

a. Requires an IBus to GIO adapter. Not available for custom devices.

b. Crimson systems with 4GI adapters support GIO-bus graphics. Not available for custom devices.

VME-bus Interface

The VME (VERSA Module Eurocard) bus is an industry-standard bus for interfacing devices. It supports the following features:

- Seven levels of prioritized processor interrupts
- 16-, 24-, 32-, and 64-bit address spaces
- 8-, 16,- 32-, and 64-bit data accesses
- DMA to and from main memory

The VME-bus does not distinguish between I/O and memory space, and it supports multiple address spaces. This feature allows you to put 16-bit devices in the 16-bit space, 24-bit devices in the 24-bit space, and 32-bit devices in the 32-bit space. So you must know which of the three address spaces that the board uses when designing a VME device driver. Most VME systems also support VME-SCSI adapters with two interfaces per board.

IRIX assumes that VME devices are I/O channel resources and that they will relinquish bus access promptly to the MIPS processor. IRIX has no model for multiprocessing on the VME bus. PIO access is much slower than DMA, so you may want to “Just say ‘No’ to PIO” for better performance.

Note: On some devices, you can use jumpers or switch settings to configure the device to use a particular address space. Some Silicon Graphics systems have DMA-mapping registers to make memory appear contiguous to the VME card.

For additional information on VME-bus operation, see the *ANSI/IEEE 1014-1987 Standard.*

EISA-bus Interface

The EISA (Extended Industry Standard Architecture) 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 and expands the ISA data bus from 16 bits to 32 bits and provides 23 more address lines and 16 more indicator and control lines.

The EISA bus supports the following features:

- all ISA transfers
- bus master devices
- burst-mode DMA transfers
- 32-bit memory data and address path
- peer-to-peer card communication
- dynamic bus sizing (i.e., 32-bit bus master to 16-bit memory)

For additional information on EISA-bus operation, see the *ANSI/IEEE 1014-1987 Standard.*

SCSI-bus Interface

The SCSI-bus is an industry standard I/O bus designed to provide host computers with device independence within a class of devices, such as disk drives, tape drives, and image scanners. SCSI is an acronym for *Small Computer System Interface.*

All Silicon Graphics systems that run IRIX 5.x or 6.0 provide an interface to at least a single SCSI-bus for peripherals that support the SCSI standard. Your device driver can place commands on the bus by using the SCSI host adapter driver. Systems with POWERchannel™ I/O processor boards (IO3) support two SCSI interfaces per POWERchannel board; CHALLENGE systems support up to 32 SCSI interfaces. POWERchannel-2™ (IO4) boards support many more SCSI interfaces per board.

Caution: All SCSI devices on a bus should support the connect/disconnect strategy while performing operations that take relatively long periods to perform. However, while the device driver can be configured not to time out, serious system throughput and reliability issues could occur.

Most VME systems also support VME-SCSI adapters with two interfaces per board.

For additional information on SCSI-bus operation, see the ANSI standards *X3.131-1986* and *X3T9.2/85-52 Rev 4B.*

GIO-bus Interface

The GIO-bus is a family of synchronous, multiplexed address-data buses for connecting high-speed devices to main memory and CPU for Silicon Graphics systems. The GIO-bus has three varieties: GIO32, GIO32-bis, and GIO64.

- The GIO32 is a 32-bit, synchronous, multiplexed address-data bus that runs at speeds from 25 to 33 MHz. This bus is found on R3000-based Indigos.
- The GIO32-bis is a 32-bit version of the non-pipelined GIO64 bus or a GIO32 bus with pipelined control signals. This bus is found on R4000-based Indigo and Indy workstations.
- The GIO64 bus is a 64-bit, synchronous, multiplexed address-data bus that can run at speeds up to 33 MHz. It supports both 32- and 64-bit GIO64 devices. GIO64 has two slightly different varieties: non-pipelined for internal system memory, and pipelined for graphics and pipelined GIO64 slot devices. This bus is implemented in the Indigo^2 platform.

For additional information on the operation of the GIO bus, see the *GIO Bus Specification*.

System Software

For kernel-level device drivers, all 4.x and later versions of the IRIX operating system provide a consistent, device-independent interface that allows the user to treat a device as a file to be opened, read, written, and closed. These calls serve as the interface between the user and the device (see Figure 1-6). This means that, for most I/O operations, you need not provide the user with device-specific system calls. Instead, the user can use the standard system call, **open**(), to get a file descriptor for the device, then read, write, and close the “file” pointed to by the file descriptor. Internally, the system calls use the driver module that you have written to handle the device.

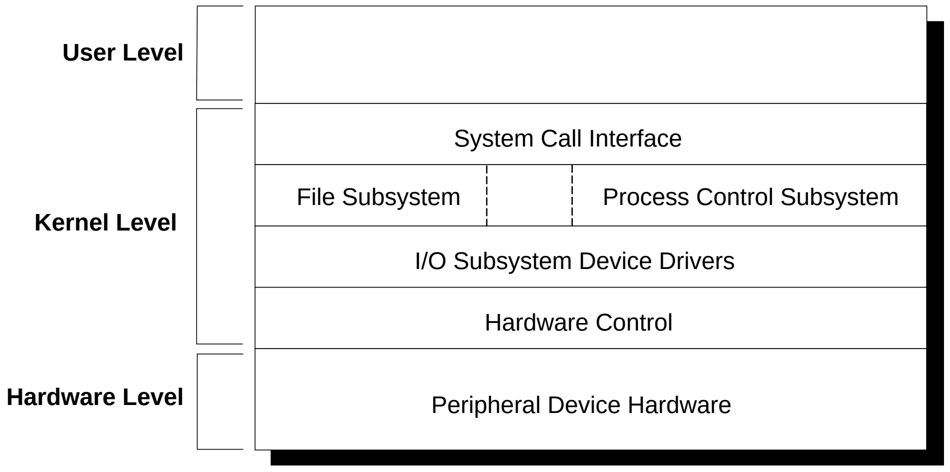


Figure 1-6 Device Driver Position in the Kernel

Chapter 2

Writing a Device Driver

This chapter describes the general interface for both user-level and kernel-level device drivers and introduces the various user-level and kernel-level device driver models.

It contains the following sections:

- "Creating Device Drivers" on page 21
- "Device-special File" on page 22
- "Including a Device Driver in the Kernel" on page 26
- "Driver Entry Points" on page 28
- "Writing Other Driver Routines" on page 42

Creating Device Drivers

There are two levels of device drivers: *user-level* and *kernel-level*. For some devices, such as GIO-bus cards, the device driver *must* be a kernel-level driver. You can write a user-level device driver, however, for devices that interface to a SCSI, EISA, or VME bus.

Creating User-level Device Drivers

User-level device drivers let you use system functions to map the device to user space and perform simple I/O operations. You do not have to understand how the software environment affects devices in the IRIX operating system. However, where specific versions of IRIX, such as 5.2 and 5.3 (both 32-bit) and 6.0 (64-bit), affect your decisions, or the performance of your driver, the differences are noted.

Creating Kernel-level Device Drivers

If you decide to write a kernel-level device driver, you need to become familiar with the software environment, conventions, and data structures that apply to device drivers running under the IRIX operating system. To create a kernel-level driver from scratch, you must:

1. Create a device-special file.
2. Create a master file.
3. Write and compile the driver code (*-coff*)[1].
4. Create a kernel that includes the driver object code.
5. Reboot using the new kernel.
6. Debug the driver.

Steps 4 and 5 may be omitted if the driver is loadable. See Chapter 11, "Kernel-level Dynamically Loadable Modules (DLMs)," on how to make a device driver loadable.

Except for step 3, all the steps in this procedure are simple and mechanical.

Device-special File

Once you write a kernel-level IRIX device driver, communication with a device is a matter of accessing a file called a *device-special* file. Each device has its own device-special file, conventionally kept in the */dev* directory. Because IRIX makes kernel-driven devices look like files, a user-level process can use the standard operating system calls to open the file / device, read from the file / device, write to the file / device, and so on. For most I / O operations, the user program needs no device-specific system call when it deals with a device driven by a kernel-level device driver. See the ioctl(D2) man page.

[1] Compile the object file with the *-coff* compiler flag for all IRIX 5.x drivers but not for IRIX 6.0 drivers. While Indigo and Indigo^2 platforms require this flag, IRIX 64-bit compilers do not support it. For the most appropriate flags for various system configurations, see the file */var/sysgen/Makefile.kernio*.

Creating Device-special Files

The device-special file is not an ordinary file. You need to use a special system administration command, **mknod**, to create a device-special file.

Synopsis

`mknod` *filename class major# minor#*

Arguments

filename	The pathname of the device-special filename. The directory the file commonly resides in */dev*.
class	Specifies the class type of the device—block or character—to which the device-special file refers. *b* specifies a block device. A block device, such as a magnetic tape or disk drive, transfers data in blocks through the *buf* structure. c specifies a character device. A character device, such as a terminal or printer, transfers data character-by-character, perhaps assembling the stream into blocks as needed by the underlying hardware.
major#	The major number of the device.
minor#	The minor number of the device.

Major and Minor Device Numbers

Internally, the kernel does not deal with filenames to differentiate among devices. Instead, the kernel uses major and minor device numbers. The major device number identifies the driver module to use for a given special device. This varies among operating systems:

- IRIX 5.2 defines 255 distinct major numbers (0 to 254).
- IRIX 6.0 uses the same numbering scheme as IRIX 5.2.
- IRIX 5.3, on the other hand, defines only 511 major device numbers (0 to 510).

While the change from IRIX 5.2 to IRIX 5.3 does not permit the use of all 14 bits of the SVR4 *major_t* value, it is a compromise between a demand for more major numbers and conserving kernel data space, since the number of major values defines the size of the *MAJOR* table and the [cb]*devsw* tables. This increases the size of the variable necessary to contain a major device number from an unsigned char to at least a short. The *master.d/README* files contain further information on this topic.

Most device drivers do not need to know what their major number(s) are; those that do should use the DDI **getmajor**() routine and *major_t* data type to manipulate them.

If you have been accessing the *MAJOR* array as an array of unsigned chars, it is now an array of unsigned shorts. The *DONTCARE* value has also changed, and the **lboot** program has been modified to accommodate these alterations.

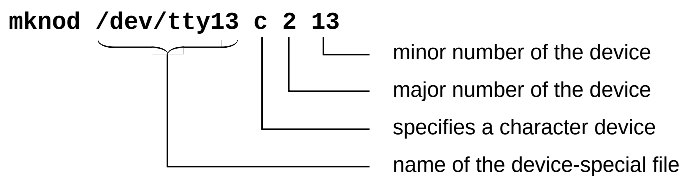
In any case, the number you choose as the major number for your device driver must not be assigned to any other device. See */usr/include/sys/major.h* on your system for a list of assigned major numbers.

The minor number is 18 bits long and can contain values from 0 to 0x3FFFF. The minor device number has no predetermined use, so your device driver can use the minor device number as you see fit. For example, the driver can use the minor device number to differentiate multiple devices on the same controller.

See the man pages for the *MAKEDEV(1M)*, **master**(4), **mknod**(1M) commands for additional information.

Device-special File Example

To create a device-special file, use the **mknod** command. For example:



```
mknod /dev/tty13 c 2 13
```

minor number of the device
major number of the device
specifies a character device
name of the device-special file

Configuration Files

Device controls are an extensible way to change or query things about devices. They fall into two categories: those intercepted by the X server and those used by the device drivers. The server uses the **x_init** controls, which change the way the X server views devices. The device drivers use **device_init** controls, which change device characteristics.

You can issue X server device controls on the fly by using the **devctrl** program (in *4Dgifts*[1]) or by calling **XSGIDeviceControl** from within a program, or by storing them in configuration files, which reside in the */usr/lib/X11/input/config* directory.

There are (potentially) two configuration files per device in the directory */usr/lib/X11/input/config*. The **device_init** options live in a file with the same name as the STREAMS module that implements the device (this is also the name of the link created in */dev/input*). The **x_init** options live in a file with the X name of the device (as supplied by the STREAMS modules). Some devices use the same name for the STREAMS module and for the X device (**tablet**, **mouse**), but some use different names for the two:

STREAMS Name	X Device Name
sball	spaceball
calcomp	tablet

When the X server finds a new device (or when it starts up), it:

- opens the device and finds a STREAMS module
- issues **device_init** controls
- asks the device to describe itself
- issues **x_init** controls
- closes the device (unless **autostart** is on for it).

[1] While some of the files in */usr/poeple/4Dgifts* are in the IRIX 6.0 release, *4Dgifts* itself is not included.

When a program opens a device that is not **autostart**ed or opened by another program, the X server:

- opens the device and finds the STREAMS module
- issues **device_init** controls
- issues **x_init** controls
- starts reporting events from the device.

The X server intercepts about a dozen **x_init** controls. For a list of the **x_init** controls and some of the more common **device_init** controls, see the *README* file in */usr/lib/X11/input/config*.

Including a Device Driver in the Kernel

The **lboot** utility allows you to link device drivers to the kernel. It requires the following files, all of which must reside under the */var/sysgen* directory:

boot This file is a symbolic link to the directory */usr/cpu/sysgen/IPxxboot*, where *xx* represents the CPU type. This directory contains all the device driver object files and archives. When your driver is successfully compiled, you must copy it to the */usr/cpu/sysgen/IPxxboot* directory. The name of your driver must end with an "*.o*" suffix (or with "***.a***" if it is a library). See "CPU Types" on page 320 for a listing of MIPS CPUs and their IP numbers.

Note: For successful compilation, IRIX 5.x drivers require the -*coff* option; IRIX 6.0 drivers cannot use the -*coff* option.

master This file contains information that **lboot** uses to create the *device switch table*, as well as to indicate dependencies among other kernel modules. Each driver must have a *master* file stored in the */var/sysgen/master.d* directory. The name of the *master* file must be the same as the software module. Among other things, the *master* file contains the major device number for the device-special file. It also contains a prefix used to build the driver entry points. For more information, see the **master**(4) man page.

mtune	This directory contains information on the *external system tunable parameters* of the driver module, including default values and valid value ranges. For more information, see the **mtune**(4) man page.
system	This directory contains files with directives that tell **lboot** whether to: 1. Include a driver module. 2. Conditionally include a driver module. 3. Exclude a driver module.

For each driver, you must create a system file in the directory */var/sysgen/system*. The restriction on filenames is that they must end in *.sm* in order for **lboot** to recognize and process them. See the **system**(4) man page for more information.

Chapter 3, "Writing a VME Device Driver," Chapter 4, "Writing an EISA Device Driver," Chapter 5, "Writing a SCSI Device Driver," and Chapter 6, "Writing Kernel-level GIO Device Drivers," provide details on the syntax of these files.

When these files are present under */var/sysgen*, you can create a kernel that includes the new driver. To create a new kernel:

1. Become root.
2. Copy the current kernel to a safe place before rebooting.[1]
   ```
   # cp /unix /unix.orig
   ```
3. Create the new kernel, */unix.install*, by running:
   ```
   # /etc/autoconfig -f
   ```
   (Use the *-v* option during debugging.)

[1] You can save disk space by using the **ln** command instead of **cp**; However, when you reboot, *unix.install* gets copied to *unix*, thus wiping out the old kernel if it is linked. Use **ln** to save space, use **cp** for reliability.

4. Reboot the system. When you issue the **reboot** command, the system removes the current kernel and renames *unix.install*, the kernel you have just created, to */unix*:

   ```
   # reboot
   ```

Note: If you include a just-written and undebugged device driver, create a debuggable kernel. See “Making a Debuggable Kernel” in Chapter 10 for more information. It is also useful in this case to examine the generated file */var/sysgen/master.c* to confirm that the entries for your new driver are correct.

Driver Entry Points

A set of *driver entry point* routines define what the system must do when a user-level program executes a system call, such as **open**(), that accesses the device. Because the user expects to treat the device as a file, you must write a driver entry point routine for each operation normally performed on a file, such as *open*, *read*, *write*, and *close*. You will probably also have to write additional driver routines to handle initialization at *system power-up*.

When you successfully configure a driver into the kernel, **lboot** automatically adds members (one for each entry point in the driver) to the *cdevsw* structure, the character device switch table.

Note: The *cdevsw* structure is used for character device drivers; a block device driver structure would be named *bdevsw*. STREAMS drivers, which have user-accessible device nodes, such as */dev/llc2*, also belong in the *cdevsw* structure; STREAMS modules, which have no device nodes, belong in *fmodsw*.

The section of the *cdevsw* structure that maintains the pointers to the device entry points for a device called *drv* would look like this:

```
struct cdevsw cdevsw[] = {
    { nodevflag, 0, drvopen, drvclose, drvread, drvwrite,
    drvioctl, drvmmap, drvmap, drvunmap, drvpoll, 0, 0 },
};
```

When the kernel handles a system call, it can find a specific entry point for a device if it constructs the name of the appropriate *cdevsw* member. For example, if the kernel must handle an **open**() for a device, *drv*, the kernel knows that *drv***open** is the member of *csdevsw* that contains a pointer to the open routine for the *drv* device.

Missing Driver Entry Points

If your driver is missing a definition for an entry point, **lboot** generates a stub that points to **nulldev**(). If the user makes the corresponding system call on that device, the system call returns an error. Your driver must always include definitions for some driver entry points, such as the device **open**() and **close**() entry points. However, many devices do not perform memory mapping and, therefore, do not need the **map**() and **unmap**() entry points. You may omit such entry points from the driver object module.

Character and Block Entry Point Driver Routines

Currently, the standard names for entry points are as shown in Table 2-1:

Table 2-1 Standard Entry Points

*drv***open**()	*drv***close**()	*drv***read**()	*drv***write**()
*drv***init**()	*drv***edinit**()	*drv***mmap**()	*drv***map**()
*drv***unload**()	*drv***unmap**()	*drv***poll**()	*drv***ioctl**()
*drv***halt**()			

Your driver normally contains an entry point named for at least *drv***open**(), *drv***close**(), *drv***read**(), and *drv***write**(). See Table 2-2 for a somewhat fuller description of these entry points.

Table 2-2 Entry Point Driver Routines

Routine	Description
open	The kernel calls *drv***open**() when the user process issues an **open**() system call.
close	The user process invokes the **close**() system call when it is finished with a device, but the system does not necessarily execute your *drv***close**() entry point for that device.
read or **write**	The kernel executes the *drv***read**() or *drv***write**() entry point whenever a user process calls the **read**() or **write**() system calls
ioctl	Character devices may include a "special function" entry point, *drv***ioctl**().
poll	A character device driver may include a *drv***poll**() entry point so that users can use **select**() or **poll**() to poll the file descriptors opened on such devices.
mmap, **map**, and **unmap**	The System VR4.x **mmap**() function establishes a mapping between a process's virtual address space and a memory object. The IRIX device *drv***mmap**(), *drv***map**(), and *drv***unmap**() entry points are used in device drivers for memory-mapped devices. See the respective man pages for details.
devflag	This sets the bitmask of flags that specify the driver's characteristics to the system.

The arguments and expected return values of each driver entry point are described below. The examples use a generic driver prefix *drv* where appropriate.

Note: The names of the procedures in your driver must start with the letter prefix of up to 14 letters for the device as given in the *master.d* file. For instance, if you write a driver for a device called *cdr*, the names of the entry points (and all the other routines defined in the driver) must start with *cdr*—*cdr***open**, *cdr***close**, *cdr***read**, and so on. Procedures in this manual use the prefix *drv*.

open – Gain Access to a Device

The kernel calls the *drv***open**() routine when the user process issues an **open**() system call. You must write your *drv***open**() entry point so that it prepares the device for I/O operations.

Your code for the *drv***open**() routine must be able to handle requests from multiple processes and to make appropriate responses, depending on the current state of the device. For example, an exclusive user device may be in a busy or not busy state; or a multiuser device may be not in use and in need of initialization; or the same device may be in use, initialized, and able to handle more users or not.

Also, drivers need a way to determine the *ABI* (Application Binary Interface) of the current user process so they can properly interpret structures passed in for **ioctl**s. By using the following defines, which give the driver the size of various entities in bytes, a function in *usrabi* returns an error if no user process is running or else copies the type size information into a structure provided by the caller. (See *ddi.h* for a definition of *usrabi*.) A good driver will handle all possibilities or, at least, **assert**() that 64-bit longs and pointers go togther.

```
typedef struct __userabi {
        short uabi_szint;
        short uabi_szlong;
        short uabi_szptr;
        short uabi_szlonglong;
} __userabi_t;
```

Synopsis

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/vmereg.h>    /* For VME drivers */

int drvopen   (dev_t *devp, int flag, int otyp, cred_t *crp)
{
    /* <your code> */
   return value;  /* 0 or value from errno.h */
}
```

Arguments

devp	Device major and minor numbers. Use **getemajor**() and **geteminor**() to extract the major and minor device numbers from this parameter. The minor number helps you identify which device of a multidevice controller is being opened. **Note:** This is a *pointer* to a device.
flag	Mode argument from the **open**() system call. Your code must check *flag* for *FREAD* and *FWRITE* bits. Typically, *flag* tells your code why the user wants to open the device.
otyp	A flag that tells your code the class of the device that it must open. This is useful if your driver must handle both character and block devices. For character devices, this flag is usually *OTYP_CHR*, but *OTYP_LYR* is also possible. **Note:** For each *OTYP_LYR* **open**, you will always get an *OTYP_LYR* **close**. If your **close** routine actually frees memory or clears driver data structures, you must track *OTYP_LYR* **open**s and **close**s separately. Ensure that all outstanding DMA operations have cleared prior to a **free**.
crp	A pointer to the user credential structure.

Returns

If the device cannot be opened in the way requested, your code for this entry point must return an appropriate error code from *sys/errno.h.*

Notes

If you want the driver to enforce mutual exclusion on a device, enforce it by having the *drv***open**() routine test to see whether the device is busy. This requires adding reference counting between your **open**() and **close**() routines, which must be protected. If the device is busy, it can sleep until completion of the current activity, then awaken.

close – Relinquish Access to a Device

The user program invokes the **close**() system call when it is finished with a device, but the system does not necessarily execute your *drv***close**() entry point for that device. The system executes the *drv***close**() entry point only after all processes that have opened the device have also called **close**().

If the device is opened frequently, you may not actually want *drv***close**() to free all the memory and other resources allocated to the open device.

Synopsis

```
#include  <sys/types.h>
#include  <sys/file.h>
#include  <sys/errno.h>
#include  <sys/open.h>
#include  <sys/cred.h>
#include  <sys/ddi.h>
#include  <sys/vmereg.h>

drvclose (dev_t dev, int flag, int otyp, cred_t *crp)
{
    <your code>
   return value;  /* 0 or value from errno.h */
}
```

dev	Device major and minor numbers. Use **getemajor**() and **geteminor**() to get the major and minor device numbers from this parameter. The minor number helps you identify which device of a multidevice driver is being closed.
flag	A mode argument from the **close**() system call. Your code must check *flag* for *FREAD* and *FWRITE* bits. Typically, *flag* tells your code why the user wants to close the device.
otyp	A flag that tells your code the class of the device that it must close. This is useful if your driver must handle both character and block devices. For character devices, this flag is usually *OTYP_CHR*, but *OTYP_LYR* is also possible.
crp	A pointer to the user credential structure.

Returns

If your code for *drv***close** encounters an error, it must return an appropriate error code from *sys/errno.h*. Even if it returns an error, your *drv***close** routine must really close the device—it won't be called again.

read or write – Read/Write Data from/to a Device

The kernel executes the *drv***read**() or *drv***write**() entry point whenever a user process calls the **read**() or **write**() system call. The following is an outline of what your driver entry points do:

1. Validate the addresses.
2. Protect the data from being paged out.
3. Start up the data transfer.
4. Set protection timeout.
5. Sleep while the data transfers.
6. Wake up when data transfer is complete.
7. Check the status of the data transfer.
8. Clear timers.
9. Report the status of the data transfer.
10. Return to user.

Because IRIX provides you with a rich set of powerful kernel functions, you can implement the above procedure in a number of ways, each sensitive to the particular strengths and limitations of the device you are controlling. However, not all methods of implementing the above procedure work for all devices. (For example, what works for non-DMA type devices does not always work for DMA-type devices if the user's virtual addresses are not currently mapped.)

Using the kernel functions **physiock**() and **biodone**() and your own *drv***strategy**() and *drv***intr**() routines, you can write *drv***write**() and *drv***read**() points that are appropriate for all types of character devices (more on *drv***strategy**() later in this chapter).

Synopsis

```
#include  <sys/types.h>
#include  <sys/errno.h>
#include  <sys/uio.h>
#include  <sys/cred.h>
#include  <sys/vmereg.h>
#include  <sys/ddi.h>
```

```
drvread (dev_t dev, uio_t *uiop, cred_t *crp)
{
    <your code>
   return physiock(drvstrategy, 0, dev, B_READ,
    nblocks_uiocp);
}
drvwrite (dev_t dev, uio_t *uiop, cred_t *crp)
{
    <your code> (see above)
   return physiock(drvstrategy, 0, dev, B_WRITE,
    nblocks_uiocp);
}
```

Arguments

dev	Major and minor device numbers of the device involved in the read or write operation. Use **getemajor**() and **geteminor**() to extract this information from *dev*.
uiop	On entry, the *uiop* parameter contains a pointer to a *uiop* structure that contains, among other things, the location *(uiop->uio_iov->iov_base*) and size *(uiop->uio_iov->iov_len*) of the buffer in user space from which to read or to which to write information.
crp	A pointer to the user credential structure.

Returns

As with the *drv***open**() and *drv***close**() entry points, your code for the *drv***read**() and the *drv***write**() entry points must (when necessary) return appropriate error codes.

ioctl – Control a Character Device

Character devices may include a “special routine” entry point, *drv***ioctl**(). You can use this entry point to perform a number of device-dependent functions other than the standard operations (such as read and write). The kernel executes the *drv***ioctl**() entry point when a user program issues the **ioctl**() system call.

Synopsis

```
#include  <sys/types.h>
#include  <sys/file.h>
#include  <sys/cred.h>
#include  <sys/errno.h>
#include  <sys/ddi.h
#include  <sys/vme.h

drvioctl (dev_t dev, int cmd, void *arg, int mode,
           cred_t *crp, int *rvalp)
{
    <your code>
   return value;  /* 0 or value from errno.h */
}
```

Arguments

dev	Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev*.
cmd	This parameter is useful when you have more than one "special routine." The user cannot call these special routines directly. However, the user can call **ioctl**() with the appropriate value as its second parameter, and thus specify which special routine it wants. Within your code for the *drv***ioctl**() entry point, you must test the *cmd* parameter and take the appropriate action.
arg	This parameter can be used or ignored by your code as needed. Its type depends on the *cmd* argument. It can be either an integer value or a pointer to a device-specific data structure. (If it is a pointer, do not reference that address directly; instead, use **copyin**() or **copyout**() to retrieve the contents.) **Note:** The size of int and pointer passed in can vary depending on the ABI outside a 64-bit kernel. See **userabi** and **userabi_t**. in "Device-special File" on page 22.
mode	The file modes set when the device was opened. Your driver can use this information to determine whether the device was opened for reading or writing.
crp	A pointer to the user credential structure.

rvalp	Is a pointer to the return value for the calling process. The driver may elect to set the value if **ioctl**() succeeds. This is distinct from the *errno* return value of the *drv***ioctl**() function itself.

Returns

As with the other driver entry points, your code for the *drv***ioctl**() entry point must return an appropriate error code from *sys/errno.h* in case of an error.

poll – Poll Entry Point for a Non-STREAMS Character Driver

A character device driver may include a *drv***poll**() entry point so that users can use **select**() or **poll**() to poll the file descriptors opened on such devices. These system calls tell the user whether input from the device is available or whether output to the device is possible.

Synopsis

```
#include <sys/poll.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/types.h>

struct drvinfo {
    . . .
    struct pollhead *phead;         /* output poll queue */
} drvinfo[MAXUNITS];

drvpoll(dev, events, anyyet, reventsp, phpp)
        dev_t  dev;
        short  events;
        int    anyyet;
        short  *reventsp;
        struct pollhead **phpp
{
    *reventsp = events;

    if ((events & (POLLIN|POLLRDNORM)) && no input available ) {
            *reventsp &= ~(POLLIN|POLLRDNORM);
    }

    if ((events & (POLLOUT) && output not possible ) {
            *reventsp &= ~POLLOUT;
    }

    if ((events & (POLLPRI|POLLRDBAND) && no out of band data ) {
            *reventsp &= ~(POLLPRI|POLLRDBAND);
    }

    if (device error) {
        *reventsp = POLLERR;
        return 0;
    }
    if (!*reventsp)
        return 0;

    if (!anyyet) {
        *phpp = drvinfo[getminor(dev)].phead;
        return 0;
    }
}
```

Arguments

dev	Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev*.
events	A mask that indicates the events being polled. The significance of the bits of this value is defined in *sys/poll.h*. When the driver's **poll**() entry point is called, the driver must verify whether any of the events requested in *events* have occurred.
anyyet	A flag that indicates whether the driver must return a pointer to its *pollhead* structure to the caller.[1] If none of the events is pending, the driver must check the *anyyet* flag and, if it is zero, store the address of the device's *pollhead* structure in the pointer pointed to by *phpp*.

[1] Routines that return a pointer to the caller must verify the caller's ABI and return data of the correct type without inadvertent conversions.

reventsp	A pointer to a bitmask of the returned events satisfied. The driver must store the mask consisting of the subset of events that are pending in the short pointed to by *reventsp*. Note that this mask may be zero if none of the events is pending.
phpp	A pointer to a pointer to a *pollhead* structure (defined in *sys/poll.h*).

A driver that supports polling must provide a *pollhead* structure for each minor device supported by the driver. Use **phalloc**() to allocate the *pollhead* structure. Use **phfree**() to free the structure.

When an event occurs, the driver must issue a call to **pollwakeup**(), passing it the event that occurred and the address of the *pollhead* structure associated with the device. For example, in the device interrupt routine, *drv***intr**():

```
drvintr()
{
...
  if (input available)
    pollwakeup (drvinfo[getminor(dev)].phead, POLLIN, POLLRDNORM);
  if (output possible)
    pollwakeup (drvinfo[getminor(dev)].phead, POLLOUT);
...
```

Returns

*drv***poll** can return an error and “hang up” by returning *POLLERR* and *POLLHUP*. You cannot specify these events in **events* on entry to *drv***poll**. If your code for *drv***poll**() encounters an error, it must return an appropriate error code from *sys/errno.h*.

map or unmap – Check Virtual Mapping for a Memory-mapped Device

Use the *drv***map**() and *drv***unmap**() entry points in device drivers for memory-mapped devices. They are described in Chapter 3, “Writing a VME Device Driver,” in greater detail.

Synopsis

Note: These routines are nonstandard to System VR4.x.

```
#include "sys/types.h"
#include "sys/region.h"
#include "sys/mman.h"

drvmap (dev_t dev,vhandl_t *vt,off_t off,
        int length,int prot)
{
    <your code>
   return value;  /* 0 or value from errno.h */
}
drvunmap (dev,vt)
          dev_t    dev;
          vhandl_t *vt;
{
    <your code>
   return value;  /* 0 or value from errno.h */
}
```

Arguments

dev	Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev*.
vt	A handle to the virtual space in the calling process to which the device is mapped. (The structure for the handle is subject to change, so do not attempt to reference the members of the structure pointed to by the handle directly.)
off	An offset to an address within the device memory. This address is the start of the device memory that the user wants your code to map into user space. (The user may not want to map in all of the device memory.)
length	The number of bytes to map.
prot	A description of the protection to apply to the region it maps in. The values for this parameter can be found in *sys/man.h*.

devflag – driver flags

Synopsis

```
#include <sys/conf.h>
#include <sys/ddi.h>
int drvdevflag = 0;
```

Every driver must define a global integer variable called *drv***devflag**. This variable contains a bitmask of flags used to specify the driver's characteristics to the system. (When *drv***devflag** is defined, UNIX SVR4 conventions apply; if it is not defined, UNIX SVR3 conventions apply.)

The valid flags that may be set in *drv***devflag** are:

D_MP	The driver is multithreaded (it handles its own locking and serialization).
D_WBACK	The driver writes back cache before calling its *drv***strategy** routine.
D_OLD	The driver uses the old-style interface (pre-5.0 release). This flag is not recommended for new work.

If no flags are set for the driver, then *drv***devflag** must be set to 0. If this is not done, then **lboot** will assume that this is an old-style driver, and it will set *D_OLD* flag as a default.

Writing Other Driver Routines

In addition to entry points, your device driver may include other routines to handle interrupts from the device and to handle device initialization at boot time (see Table 2-3). You may also want your driver to include routines (such as *drv***strategy**) that are not strictly necessary but that simplify writing the standard entry point routines.

Table 2-3 Interrupt and Initialization Handling Routines

Routine	Description
intr	Processes a device interrupt after a transfer terminates, whether normally (upon completion) or abnormally (due to some error).
strategy	Performs block I/O.
edtinit	Initializes the device at boot time. Same as **init**().
init	Initializes the device at boot time. Same as **edtinit**().
halt	Shuts down the driver when the system shuts down.
start	Initializes a device at system startup.
unload	Cleans up a loadable kernel module.

intr – Process a Device Interrupt

When your device driver does a read or write, the driver usually puts itself to sleep while it waits for the transfer to complete. When the transfer terminates, whether normally (upon completion) or abnormally (due to some error), the device sends an interrupt to the CPU. When the system receives the interrupt from the device, it looks in your device driver for the *drv***intr**() routine and executes that routine. Some devices can interrupt when the open count is zero. The interrupt still must be handled.

When the device I/O completes., the *drv***intr**() routine awakens the sleeping process. Within the *drv***intr**() routine, you can use the kernel function **biodone**() to awaken the sleeping process and report the status of the transfer (whether normal or error).

For a SCSI device, there must not be a *drv***intr**() routine because the driver is a “soft” driver that does not interact directly with the hardware. Instead, a callback routine is often provided. This routine may be given any name, but it is often of the form *drv***_intr**():

```
drv_intr(scsi_request_t *sp);
```

Arguments

sp	A pointer to a *scsi_request_t* type structure. (See the sample code in Chapter 5, “Writing a SCSI Device Driver,” for an example of a *drv***_intr**() routine written for a SCSI type device.) You must explicitly pass *drv***_intr**() in the *sr_notify* member of the *scsi_request_t* structure allocated for the device.

strategy – Perform Block I/O

The *drv***strategy**() routine is not a character device driver entry point in the strictest sense (the user does not call it). However, when writing a device driver, you will probably want to write a *drv***strategy**() routine. Typically, you call the *drv***strategy**() routine from the *drv***read**() and *drv***write**() routines, through the **physiock**() kernel routine:

```
drvread (dev_t dev, uio_t *uiop, cred_t *crp)
{
    return physiock(drvstrategy, 0, dev, B_READ,
           nblocks, uiop);
}
drvwrite (dev_t dev, uio_t *uiop, cred_t *crp)
{
    return physiock(drvstrategy, 0, dev, B_WRITE,
           nblocks, uiop);
}
```

physiock() is a kernel routine that:

- Verifies whether the requested transfer is valid by checking whether the offset is at or past the end of the device and verifying that the offset is a multiple of the block size (512).
- Sets up a buffer header that describes the transfer.
- Faults pages in and locks the pages involved in the I/O transfer so they cannot be swapped out.
- Calls the strategy routine passed by the first parameter.
- Sleeps until the transfer is complete and awakens when the driver's I/O completion handler calls **biodone**().
- Performs the necessary cleanup and updates, then returns to the routine that called it.

physiock() reports a data transfer as valid if the following conditions are met:

- the specified data location exists on the device
- the user has specified a storage area that exists in user memory space
- the user-space storage area is large enough.

For more information, see the physiock(D3) man page.

Note: In IRIX 5.x and earlier, pages are 4 KB, and the default maximum DMA size is 4 MB; in IRIX 6.0, pages are 16 KB, and the default maximum DMA size is 16 MB. You can change the DMA size by modifying *maxdmasz*, in */var/sysgen/mtune/kernel*, using *page* as the basic unit. For other ways to modify this parameter, see the **systune**(1M) man page. I/O larger than what is allowed by *maxdmasz* produces the UNIX error ENOMEM. See Appendix B, "SCSI Controller Error Messages".

If the second argument is 0, **physiock**() then allocates an IRIX buffer header (a kernel-level structure of type *buf*) and primes it with appropriate transfer information; otherwise, **physiock**() uses the argument as a pointer to a *buf_t*. This structure encapsulates all the information of a single I/O transfer.

physiock() assigns the values of the following *buf* type structure members:

b_un.b_addr	Contains the kernel virtual address from which information is read or to which information is written.
b_flags	Contains a bit mask of flags that describe the transfer. *B_BUSY* is set to indicate that the buffer is in use for an I/O transfer. *B_READ* is set if the transfer is a read.
b_bcount	Contains the number of bytes to be transferred.
b_edev	Contains the major and minor device numbers.
b_blkno	Contains the device block number to be transferred.
b_resid	On completion, before calling **biodone**(), the driver must set this member to the number of bytes that were not transferred.
b_biodone	If nonzero, this is taken as a function pointer, and the routine in question is called from **biodone**(); all normal **biodone**() processing is skipped. *b_biodone* may also be set by the user.

Finally, **physiock**() calls *drv***strategy**() and hands it a pointer to this *buf* structure. (See a description of physiock (D3) in the *IRIX Device Driver Reference Pages* for more details on this kernel procedure.)

Synopsis

```
drvstrategy(struct buf_t *bp)
{
    <your code>
}
```

Your *drv***strategy**() routine programs the device for the transfer. The information it needs to do this is contained in the structure pointed to by *bp*. Typically, your *drv***strategy**() routine starts the I/O by programming appropriate registers. When *drv***strategy**() is done, control returns to **physiock**(). **physiock**() then calls **biowait**(), and the process sleeps until the transfer is complete.

Normally, your interrupt handler will call **biodone**(*bp*) on completion. But before calling **biodone**(), your driver must have saved the *bp* value passed to the strategy routine. (You must awaken the sleeping process even if there is some initial error condition.) In addition, your *drv***intr**() routine must indicate the success of the transaction by updating the *b_resid* member of the *buf_t* type structure to contain the number of bytes that were **not** transferred, then move to the next page.

To handle any I/O errors that occur, use **bioerror** (*bp, errno*), where *bp* is a pointer to the *buf_t* type structure passed in as the first parameter of your *drv***strategy**(), and *errno* is the appropriate error number. **bioerror**() sets the members of the buffer header so that higher level code can detect the error and call **geterror**() to retrieve the error number from the structure.

Caution: Your *drv***intr**() routine and the routines it calls must not try to access the *uiop* structure directly. The structure it gets might not belong to the process that made the I/O request.

edtinit and init – Initialize a Device

Most devices need some initialization at boot time. The system looks in the object module for the driver for either of two routines, *drv***init**() or *drv***edtinit**(), then executes the appropriate routine to initialize the device. If you use the *INCLUDE* directive (in the */var/sysgen/system/irix.sm* file) to add a device to the kernel, your driver must use the *drv***init**() routine to initialize the device at boot time. If you use the *VECTOR* directive, your routine must use the *drv***edtinit**() routine to initialize the device at boot time.

Because you use the *INCLUDE* directive to include SCSI device drivers in the kernel, your drivers for SCSI devices must include a *drv***init**() routine if you want to initialize the device at boot time (in which case, no **edtinit**() call will be generated). See Chapter 5, “Writing a SCSI Device Driver,” for a synopsis of the *drv***init**() routine.

Because you use the *VECTOR* directive to include VME device drivers in the kernel, your device drivers for VME devices must include a *drv***edtinit**() routine to initialize the device at boot time. See Chapter 3, “Writing a VME Device Driver,” for a synopsis of the *drv***edtinit**() routine and a discussion of VME-bus address space and PIO mapping.

Most device drivers of the general memory-mapping model are for VME type devices. (See Chapter 7, “Writing Kernel-level General Memory-mapping Device Drivers.”) Therefore, most device drivers of the general memory-mapping model are included in the kernel using the *VECTOR* directive. Your object module for this type of device driver usually contains a *drv***edtinit**() routine.

Synopsis

```
void drvedtinit(struct edt *e);
```

halt – Shut Down the Driver When the System Shuts Down

The *drv***halt**() routine, if present, is called to shut the driver down when the system is shut down. After the *drv***halt**() routine is called, no more calls are made to the driver entry points.

This entry point is optional. The device driver can not assume that the interrupts are enabled. The driver must make sure that no interrupts are pending from its device and must inform the device that no more interrupts are to be generated.

Synopsis

```
void drvhalt(void);
```

Return Values

None

start – Initialize a Device at System Startup

The *drv***start**() routine is called at system boot time (after system services are available and interrupts have been enabled) to initialize drivers and the devices they control.

This entry point is optional. The start routine can perform the following types of activity:

- initialize data structures
- allocate buffers for private buffering schemes
- map the device into virtual address space
- initialize hardware
- initialize time-outs

A driver that needs to perform setup and initialization tasks that must take place before system services are available and interrupts are enabled must use the *drv***init**() routine to perform such tasks. The *drv***start**() routine must be used for all other initialization tasks.

Synopsis

```
void drvstart(void);
```

Return Values

None

unload – Clean Up a Loadable Kernel Module

The *drv***unload**() routine handles any cleanup a loadable kernel module must perform before it can be unloaded dynamically from a running system.

This entry point is only required if a module is to be unloaded from the system. A loadable module's unload routine is defined in module-specific initialization code called wrapper code. The *drv***unload**() routine can perform activities such as:

- Deallocate memory acquired for private data
- Cancel any outstanding **itimeout**() or **bufcall**() requests made by the module

Synopsis

```
int drvunload(void);
```

Return Values

The *drv***unload**() routine returns 0 for success or the appropriate error number.

Synchronization Constraints

The *drv***unload**() routine must not sleep or call any functions that sleep, such as **biowait**(), **delay**(), **psema**(), or **sleep**().

STREAMS Driver Entry Points

The STREAMS driver entry points are listed in Table 2-4.

Table 2-4 STREAMS Driver Entry Points

Driver Entry Points			
put	**srv**	**open**	**close**

put – Coordinate Message Passing Between Queues in a Stream

The primary task of the **put** routine is to coordinate the passing of messages from one queue to the next in a stream. The **put** routine is called by the preceding component (module, driver, or stream head) in the stream. **put** routines are designated **write** or **read** depending on the direction of message flow.

This entry point is required in all STREAMS drivers and modules.

Synopsis

```
drvput(register queue_t *q, register inblk_t *mp);
```

Usage

Both modules and drivers must have **put** routines for writing. Modules must have **put** routines for reading, but drivers do not really need them because their interrupt handlers can do the work intended for the read **put** routine. If immediate processing is desired when a message is passed to the **put** routine, it can either process the message or queue it so that the service routine can process it later. See srv(D2).

Note: The majority of STREAMS drivers are software drivers, however, and do not have interrupt handlers.

The **put** routine must do at least one of the following when it receives a message:

- pass the message to the next component in the stream by calling the putnext(D3) function

- process the message, if immediate processing is required (for example, high-priority messages)
- queue the message with the putq(D3) function for deferred processing by the service routine

Typically, the **put** routine switches on the message type, which is contained in *mp->b_datap->db_type*, taking different actions depending on the message type. For example, a **put** routine might process high-priority messages and queue normal messages.

The **putq** function can be used as a module's **put** routine when no special processing is required and all messages are to be queued for the service routine.

Notes

Although queue flushing can be done in the service routine, drivers and modules usually handle it in their **put** routines.

- Drivers and modules should not call **put** routines directly.
- Drivers should free any messages they do not recognize.
- Modules should pass on any messages they do not recognize.
- Drivers should fail any unrecognized **M_IOCTL** messages by converting them into M_IOCNAK messages and sending them upstream.
- Modules should pass on any unrecognized **M_IOCTL** messages.

Return Values

Ignored

Synchronization Constraints

put routines do not have a user context and so may not call any function that sleeps.

srv – Service Routine

The **srv** (service) routine may be included in a STREAMS module or driver for a number of reasons. It provides greater control over the flow of messages in a stream by allowing the module or driver to reorder messages, defer the processing of some messages, or fragment and reassemble messages. The service routine also provides a way to recover from resource allocation failures.

Synopsis

```
drvsrv(register queue queue_t *q);
```

Usage

This entry point is optional and is valid for STREAMS drivers and modules only.

A message is first passed to a module's or driver's put(D2) routine, which may or may not process it. The **put** routine can place the message on the queue for processing by the service routine.

Once a message has been queued, the STREAMS scheduler calls the service routine at some later time. Drivers and modules should not depend on the order in which service procedures are run. This is an implementation-dependent characteristic. In particular, applications should not rely on service procedures running before returning to user-level processing.

Every STREAMS queue has limit values it uses to implement flow control (see queue(D4). High and low water marks are checked to stop and restart the flow of message processing. Flow control limits apply only between two adjacent queues with service routines. Flow control occurs by service routines following certain rules before passing messages along. By convention, high-priority messages are not affected by flow control.

STREAMS messages can be defined to have up to 256 different priorities to support some networking protocol requirements for multiple bands of data flow. At a minimum, a stream must distinguish between normal (priority band zero) messages and high-priority messages (such as M_IOCACK). High-priority messages are always placed at the head of the queue, after any other high-priority messages already queued. Next are messages from all included priority bands, which are queued in decreasing order of priority.

Each priority band has its own flow control limits. By convention, if a band is stopped, all lower priority bands are also stopped.

Once a service routine is called by the STREAMS scheduler, it must provide for processing all messages on its queue, restarting itself if necessary. Message processing must continue until either the queue is empty, the stream is flow-controlled, or an allocation error occurs. Typically, the service routine switches on the message type contained in *mp->b_datap->db_type*, taking different actions depending on the message type.

Each STREAMS module and driver can have a read and write service routine. If a service routine is not needed (because the **put** routine processes all messages), a NULL pointer should be placed in the module's qinit(D4) structure.

If the service routine finishes running for any reason other than flow control or an empty queue, then it must explicitly arrange for its rescheduling. For example, if an allocation error occurs during the processing of a message, the service routine can put the message back on the queue with **putbq** and, before returning, arrange to have itself rescheduled at a later time. See qenable(D3), bufcall(D3), and itimeout(D3).

Notes

Service routines can be interrupted by **put** routines unless the processor interrupt level is raised.

- Only one copy of a queue's service routine runs at a time.
- Drivers and modules should not call service routines directly. Use qenable(D3) to schedule service routines to run.
- Drivers (except multiplexors) should free any messages they do not recognize.
- Modules should pass on any messages they do not recognize.
- Drivers should fail any unrecognized **M_IOCTL** messages by converting them into **M_IOCNAK** messages and sending them upstream.
- Modules should pass on any unrecognized **M_IOCTL** messages.
- Service routines should never put high-priority messages back on their queues.

Return Values

Ignored

Synchronization Constraints

Service routines do not have a user context and so may not call any function that sleeps.

Chapter 3

Writing a VME Device Driver

This chapter provides in-depth information about drivers that interface to the VME bus. It gives a brief overview of the VME-bus interface, describes system configuration for VME device drivers, and introduces several VME-specific routines you must include in your device driver. Which of the several models for performing DMA *(direct memory access)* operations you choose for your device driver depends on the capability of the device (whether the device has scatter / gather registers, for example), the address space of the device (VME A24, A32, or A64), and whether the device provides address mapping capability.

It contains the following sections:

- “VME-bus Interface Overview” on page 58
- “Choosing a Driver Model” on page 63
- “Writing User-level VME Device Drivers” on page 64
- “Writing Kernel-level VME Device Drivers” on page 78

VME-bus Interface Overview

All high-end Silicon Graphics systems—Crimson, CHALLENGE / Onyx, POWER CHALLENGE / POWER Onyx, and the POWER series— support the VME bus with a VME-bus adapter. Old mid-range systems—IRIS 4D / 20, 4D / 25, 4D / 30, and 4D / 35—also supported VME. Silicon Graphics desktop systems—Indigo, Indigo^2, and Indy—do not currently support the VME bus.

The VME bus is an industry-standard bus for interfacing devices. It supports the following features:

- Seven levels of prioritized processor interrupts
- 16-bit, 24-bit, and 32-bit data addresses and 64-bit memory addresses
- 16-bit and 32-bit accesses (and 64-bit accesses in MIPS III mode)
- 8-bit, 16-bit, 32-bit, and 64-bit data transfer
- DMA to / from main memory

The VME bus does not distinguish between I / O and memory space, and it supports multiple address spaces. This feature allows you to put 16-bit devices in the 16-bit space, 24-bit devices in the 24-bit space, 32-bit devices in the 32-bit space, and 64-bit devices in 64-bit space.[1] So you must know which of the four address spaces the board uses when you design a VME device driver.

Note: On some devices, you can use jumpers or switch settings to configure the device to use a particular address space. Some systems have DMA-mapping registers to make memory appear contiguous to the VME card.

For additional information on VME-bus operation, see the ANSI standards specification for the VME bus.

[1] 64-bit data transfers, accesses, and memory addresses do not depend on a 64-bit kernel, so they can be mapped to all MIPS R4000 series platforms.

VME-bus Adapter

The term VME-bus adapter (see Figure 3-1) refers to a hardware *conduit* that translates host CPU operations to VME-bus operations and decodes some VME-bus operations (as though the conduit were a memory board) to translate them to the host side.

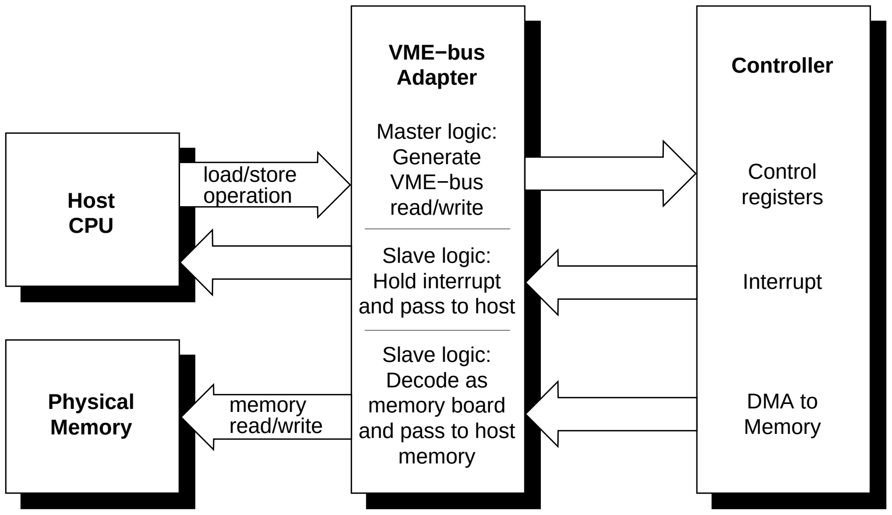

VME−bus Adapter
Master logic: Generate VME−bus read/write
Slave logic: Hold interrupt and pass to host
Slave logic: Decode as memory board and pass to host memory
Controller
Control registers
Interrupt
DMA to Memory
Host CPU
load/store operation
Physical Memory
memory read/write

Figure 3-1 VME-bus Adapter

VME-bus Address Space

The VME bus provides 32 address bits and six address-modifier bits. It supports four address sizes: 16-bit, 24-bit, 32-bit, and 64-bits (A16, A24, A32, and, on CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx series systems, A64). The VME bus allows the master to broadcast addresses at any of these sizes. The VME bus supports data transfer sizes of 8, 16, 32, or 64 bits. To best understand the VME-bus addressing and address space, think of the device as consisting of two halves: the *master* and the *slave*. When the CPU accesses the address space of the device, the device acts as a VME slave. When the VME device accesses main memory through direct memory access (DMA) operations, the VME device acts as a VME master.

Addressing behavior for a driver depends on whether the CPU or the device is the master. For example, a VME device can be a 16-bit slave and a 32-bit master. Silicon Graphics systems support 16-, 24-, and 32-bit slaves, but only 24- and 32-bit masters.

Some Silicon Graphics systems provide additional hardware mapping registers that map a VME-bus address to an arbitrary location in physical memory. Device drivers can take advantage of this mapping hardware to provide scatter / gather capabilities (and to support DMA operations to all of memory for A24 devices). The IRIX operating system provides a procedural interface by which your device driver can allocate and use these maps. This interface also has a provision to handle multiple VME-bus systems.

For other systems, 24-bit VME masters can access only the lowest 8 MB of physical memory,[1] so device drivers may need to allocate buffers in low memory and then copy data to its final destination. See */usr/include/sys/vmereg.h* for macro #devices to facilitate VME access.

VME-bus Read-Modify-Write Cycle

The VME bus provides a read-modify-write (or RMW) cycle that allows users to read and change the contents of a device register or memory location in a single atomic operation. Although this feature is typically used to implement synchronization primitives on VME memory, you may occasionally find this feature useful for certain devices. The VME-bus adapter provides access to VME read-modify-write cycles through a set of kernel functions, such as **pio_andh_rmw**() and **pio_orw_rmw**().

Caution: The VME RMW cycle is needed *only* when a controller allows both itself and a user to write a register. If a disk controller uses a single register for the status and command information for several disk drives, for example, you could be writing a command into the register from the driver while the disk controller is updating the status. The VME RMW cycle enforces exclusive use of an address. Since this operation is expensive, in terms of resources, it should rarely be used.

[1] The highest bit is used to distinguish between user and supervisor access.

Note: Silicon Graphics products do not support VME read-modify-write operations initiated by a VME master to host memory.

VME-bus Adapter Requests

The VME-bus adapter provides four levels of bus request, 0-3, (3 has the highest priority) for DMA arbitration. Do not confuse these *bus request levels* with the interrupt priorities described below. Bus requests prioritize the use of the physical lines representing the bus and are normally set by means of jumpers on the interface board. The IRIS-4D / 20, 4D / 25, 4D / 30, and 4D / 35 support only Bus Request level 3 and Bus Grant level 3.

VME-bus Interrupts

The VME bus supports seven levels of prioritized interrupts, 1 through 7 (where 7 has the highest priority). The VME-bus adapter has a register associated with each level. On Silicon Graphics systems, all VME interrupts come in at the same CPU interrupt level. When the system responds to the VME-bus interrupt, it services all devices identified in the interrupt vector register in order of their VME-bus priority (highest number first). The operating system then determines which interrupt routine to use, based on the interrupt level and the interrupt vector value.

Note: On systems equipped with multiple VME buses, adapter 0 has the highest priority; other adapters are prioritized in ascending order (after 0).

No device can interrupt the VME bus at the same level as an interrupt currently being serviced by the CPU because the register associated with that level is busy. A device that tries to post a VME-bus interrupt at the same VME-bus priority level as the interrupt being serviced must wait until the current interrupt is processed.

Therefore, when choosing VME-bus priority levels for devices, be sure that the priority levels are well distributed. If you must double up on VME-bus priority levels, double up on those devices not likely to need the CPU at the same time.

Note: All VME interrupt levels map into one CPU interrupt level.

Distribution of VME Interrupts on Multiprocessors

On CHALLENGE / Onyx, POWER CHALLENGE / POWER Onyx, and multiprocessor POWER Series systems, VME interrupt levels can be individually locked onto any processor in the system through the IPL directive. This prevents a processor running a real-time process or a process that needs a guaranteed response from being interrupted inconveniently, and it makes system load balancing easier. To lock a particular VME interrupt level to a processor, edit the */var/sysgen/system/irix.sm* file, then run **lboot** to implement the changes. The format is:

```
IPL: level cpu#
```

where *level* is the priority level (1-7, with 7 being the highest), and *cpu#* is the number of the CPU on which you want the VME interrupts of that level to occur. For example:

```
IPL: 4 1
```

designates VME interrupt priority level 4 on CPU number 1.

CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx and multiprocessor POWER Series™ systems take advantage of multiple CPUs by distributing interrupts across all processors. These distributed interrupts are called *sprayed* interrupts. To declare a CPU that is not suitable for sprayed interrupts (usually because they will be used for real-time activities), use the *NOINTR* directive.

Example: to declare that CPU 3 will not accept sprayed interrupts, use:

```
NOINTR: 3
```

You *can* tie a VME interrupt to a processor that accepts no sprayed interrupts using the IPL directives described above. You may *not* restrict CPU 0 from receiving interrupts. You can specify multiple CPUs on the NOINTR line.

After editing the *irix.sm* file, you must run **lboot** to reconfigure the system before the changes can take effect. **autoconfig** is a script in */etc/init.d* that runs **lboot**. See the **autoconfig**(1M), **lboot**(1M), and **system**(4) man pages for details.

Choosing a Driver Model

Choosing between a user-level device driver and a kernel-level device driver model usually depends on the method used to transfer data to and from the device. However, the two driver models are not necessarily mutually exclusive. It is possible, for example, for a single VME driver to use both direct memory access (DMA) and memory mapping to transfer data.

User-level VME-bus Device Driver

The easiest way to handle a VME device is to write a user-level program that controls the device by dealing directly with the special */dev/vme* driver. You can write a user-level device driver when your users need to access a VME-bus device that is not interrupt driven and does not require DMA operations. In fact, many boards that use DMA or generate interrupts can have these features turned off for simple, user-level device drivers.

User-level VME-bus device drivers are convenient for determining whether a device responds to the correct address or simple register tests. They can also be useful for prototyping: you can quickly integrate boards whose interrupts can be turned off into a system, then later write a kernel-level driver that turns the interrupts back on for higher performance. In addition, you can use a user-level VME-bus device driver in real applications that require low-overhead access to on-board device registers or memory.

A user-level VME-bus device driver might typically handle data acquisition hardware—hardware that reads large amounts of data into device memory. Because the device memory is memory-mapped into the address space of the user program, it is available to the user program directly; the user program can avoid copying the data into host memory, processing the data in the device memory instead. However, these PIO accesses may have substantially lower performance than DMA-based kernel drivers. Refer to “Programmed I/O (PIO)” on page 88.

Kernel-level VME Device Driver

You must write a kernel-level device driver for a VME device that is interrupt-driven or that requires DMA. See Chapter 2, “Writing a Device Driver,” for a description of the IRIX device driver interface.

Kernel-level General Memory-mapping Device Driver

If you want to write a driver that lets users access the VME device as memory in user space and also supports DMA and interrupts, you cannot use the general-purpose VME device driver. Instead, you must write a kernel-level device driver of the general memory-mapping model. Likewise, if you need an efficient way to share main memory between a kernel driver and a user program, you must write a device driver of the general memory-mapping model.

The general memory-mapping model is a kernel-level device driver similar to the user-level memory-mapped device driver described above. See Chapter 2, “Writing a Device Driver,” for a general description of kernel-level device drivers. See Chapter 7, “Writing Kernel-level General Memory-mapping Device Drivers,” for a description of the memory mapping facilities.

Writing User-level VME Device Drivers

The IRIX operating system contains special files in */dev/vme/vme** that provide access to the various address spaces on the system’s VME-bus adapters. These special files allow a user-level program to map arbitrary VME devices into its address space. You can take advantage of them to write a user-level memory-mapped device driver.

Byte addresses in */dev/vme/vme** are interpreted as VME-bus addresses. Not all addresses can be read from or written to because of read-only or write-only registers and unequipped addresses. Reads or writes to invalid VME-bus addresses normally result in a SIGBUS signal being sent to the offending process.

If multiple processes have the mapping for the VME address that got an error, a SIGBUS signal is sent to each of them. On multiprocessor systems, a write to an invalid VME-bus address behaves differently from one on a single-processor system. In these cases, since writes are asynchronous, processors do not wait for the completion of the write operation. If a write operation fails, it may take up to 10 milliseconds for the user VME process to be signaled about a failed write. (VME-bus time-out is about 80 microseconds.) So, if the user VME process has to confirm the successful completion of a write, it should wait for about 10 milliseconds. If the user VME process has already terminated by the time the kernel gets the VME write error interrupt, it sends a message to SYSLOG indicating the VME adapter number and failed VME-bus address.

When your driver maps a device into the address space of a user-level program (through the **mmap**() system call), the user-level program can use simple loads and stores to and from program variables to read or modify device registers or to read or set on-board device memory. If you use memory mapping, you do not need to modify any *irix.sm* files.

Recall that mmapped device drivers are slave devices in which the hardware is memory mapped into a user's address space. No interrupt or DMA service routine is available to the user process.

The special files found in */dev/vme*/* are named in the format:

`/dev/vme/`*vme<adapter-#><address-space><address-mode>*

Arguments

adapter-#	specifies which VME-bus adapter
address-space	specifies which address space, such as 16, 24, or 32 (see “VME-bus Space Reserved for Customer Drivers” on page 329.)
address-mode	identifies the addressing mode, which is *n* for non-privileged or *s* for supervisor

Use the **hinv**(1M) (*hardware inventory*) command to produce a list of valid VME-bus adapters present on the system. Adapter numbers range upwards from 0. These adapters can be used only for memory mapping VME-bus address space into the address space of a user's program. The address space can be 16, 24 or 32. The address mode is either *n* for non-privileged or *s* for supervisory. Thus, adapter 0, address space 16 in non-privileged mode is referred to as */dev/vme/vme0a16n*.

Use the technical specification for the device to determine the slave addressing mode.

The kernel driver for user-level VME is referred to as **usrvme**. If VME buses are added to an existing system, it may be necessary to run **MAKEDEV**(1M), specifying a target of **usrvme**, to have the additional */dev/vme* devices created.

Example VME Device Driver

The following code sample uses the user-level VME-bus interface to perform bus probes:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <stdio.h>
#include <fcntl.h>
#include <getopt.h>
#include <errno.h>
#include <limits.h>
#include <signal.h>

int state = 0;

#define S_DIR 0x01
#define S_ADAP 0x02
#define S_SPACE 0x04
#define S_ADDR 0x08
#define S_SIZE 0x10
#define S_VAL 0x20

#define D_READ 0x1
#define D_WRITE 0x2

#define READSTATE (S_DIR|S_ADAP|S_SPACE|S_ADDR|S_SIZE)
#define WRITESTATE (READSTATE|S_VAL)

char *progname;

char *spaces[] = {
    “a16n”,
    “a16s”,
    “a24n”,
    “a24s”,
    “a32n”,
    “a32s”
};

#define MAXSPACE (sizeof(spaces)/sizeof(spaces[0]))

void usage(void);
long ntol(char *);
long chkspc(char *);

char devnm[PATH_MAX];

static void probe_fail(int);

int
main(int ac, char *av[])
{
    int    c, errflg = 0;
    int    dir_f;
    long    adap_f;
    long    addr_f;
    long    size_f;
    long    val_f;
    char *space_f;
    int    fd;
    char *mapaddr;
    int    pgaddr, pgoff;
    int    pgsz;
    int    rtval;

    progname = av[0];

    while( (c = getopt(ac,av,”rws:a:b:p:v:”)) != -1 )
        switch( c ) {
        case ’r’:
            if( state & S_DIR ) {
                usage();
                return 1;
            }
            dir_f = D_READ;
            state |= S_DIR;
```

```
                break;
            case ’w’:
                if( state & S_DIR ) {
                    usage();
                    return 1;
                }
                dir_f = D_WRITE;
                state |= S_DIR;
                break;
            case ’s’:
                if( state & S_SPACE ) {
                    usage();
                    return 1;
                }
                if( chkspc(optarg) ) {
                    usage();
                    return 1;
                }
                state |= S_SPACE;
                space_f = optarg;
                break;
            case ’a’:
                if( ((adap_f = ntol(optarg)) < 0) ||
                    (state & S_ADAP) ) {
                    usage();
                    return 1;
                }
                state |= S_ADAP;
                break;
            case ’b’:
                if( ((addr_f = ntol(optarg)) < 0) ||
                    (state & S_ADDR) ) {
                    usage();
                    return 1;
                }
                state |= S_ADDR;
                break;
            case ’p’:
                if( ((size_f = ntol(optarg)) < 0) ||
                    (state & S_SIZE) ) {
                    usage();
                    return 1;
                }
                state |= S_SIZE;
                break;
```

```
case ’v’:
    if( ((val_f = ntol(optarg)) < 0) ||
        (state & S_VAL) ) {
        usage();
        return 1;
    }
    state |= S_VAL;
    break;
case ’?’:
    errflg++;
    break;
}

if( errflg || !(state & S_DIR) ) {
    usage();
    return 1;
}

if( (dir_f == D_READ) && (state != READSTATE) ) {
    usage();
    return 1;
}
if( (dir_f == D_WRITE) && (state != WRITESTATE) ) {
    usage();
    return 1;
}


/* check the size */
switch( size_f ) {
case 1:
case 2:
case 4:
    break;
default:
    (void)fprintf(stderr,”invalid size %d\n”,size_f);
    usage();
    return 1;
}

/* create name of device */
sprintf(devnm,”/dev/vme/vme%d%s”,adap_f,space_f);

/* open the usrvme device */
if( (fd = open(devnm,O_RDWR)) < 0 ) {
    perror(“open”);
    return 1;
}

/* we map in memory on page boundaries so figure out
 * the page and page offset
 */

pgsz = getpagesize();
pgaddr = (addr_f / pgsz) * pgsz;
pgoff = addr_f % pgsz;

/* map in the vme space surrounding the address */
if( (mapaddr = mmap(
        NULL,pgsz,PROT_READ|PROT_WRITE,MAP_PRIVATE,
        fd,pgaddr)) == (void*)-1 ) {
    perror(“mmap”);
    return 1;
}

/* catch bus errors */
signal(SIGBUS,probe_fail);

/* do the probe */
if( dir_f & D_READ ) {
    switch( size_f ) {
    case 1:
        rtval = *(char *)&mapaddr[pgoff];
        break;
    case 2:
        rtval = *(short *)&mapaddr[pgoff];
        break;
    case 4:
        rtval = *(int *)&mapaddr[pgoff];
        break;
    }
    printf(“read probe of 0x%x\n”,rtval);
}
else {
    switch( size_f ) {
    case 1:
        *(char *)&mapaddr[pgoff] = val_f;
        break;
    case 2:
        *(short *)&mapaddr[pgoff] = val_f;
```

```
                break;
            case 4:
                *(int *)&mapaddr[pgoff] = val_f;
                break;
            }
            printf(“write probe of 0x%x\n”,val_f);
            /* wait here to catch any bus errors... */
            sginap(CLK_TCK/50+1);
        }

        return 0;
}

long
ntol(str)
    char *str;
{
    char *strp;
    ulong ret;

    if( *str == ’”’ ) {
        str++;
        return (*str)?*str:-1;
    }

    ret = strtoul(str,&strp,0);

    if( ((ret == 0) && (strp == str)) ||
        ((errno == ERANGE) && (ret = -1)) )
        return (long)-1;

    return (long)ret;
}

long
chkspc(char *nm)
{
    int i;

    for( i = 0 ; i < MAXSPACE ; i++ )
        if( strcmp(nm,spaces[i]) == 0 )
            return 0;

    return 1;
}
```

```
void
usage()
{
 (void)fprintf(stderr,
    “usage: %s -r -a adap -s space -b busaddr -p
probesize\n”,
    progname);
 (void)fprintf(stderr,
    “usage: %s -w -a adap -s space -b busaddr -p probesize -
v val\n”,
    progname);
 (void)fprintf(stderr,
    “    space is one of a16n, a16s, a24n, a24s, a32n,
a32s\n”);
 (void)fprintf(stderr,
    “    probesize is one of 1 2 or 4\n”);
}

static void
probe_fail(int signo)
{
    fprintf(stderr,”*** probe failed\n”);
    exit(1);
}
```

Using mmap

After you have reconfigured the system correctly, the user-level driver can open the special file for a generic VME device */dev/vme/vme**. To map in the device, the user program must use the **mmap**() system call. For example:

```
#include "fcntl.h"
#include "sys/mman.h"
fd = open("/dev/vme/vme0a16s", O_RDWR);
addr = mmap(
         0, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, off);
```

The **mmap**() routine maps *len* bytes starting at (VME-bus) address *off* to the user virtual address *addr*. The *prot* argument is a bit mask that indicates the protection that the operating system enforces on access to the device memory. Thus, *PROT_WRITE* allows writing; *PROT_READ* allows reading. The *flags* argument can be either *MAP_PRIVATE* or *MAP_SHARED* when used with hardware devices (currently, */dev/vme/vme** makes no distinction between the two). These symbolic constants are defined in *sys/mman.h*. See the mmap(D2) man page for further information on the use of this system call.

Once the **mmap** call succeeds, reads and writes from the user virtual address *addr* for a length of *len* bytes result in the appropriate reads and writes for the VME device pointed to by the file descriptor *fd*.

Note: There is protection on a page boundary only. Even if the user-level program maps in less than a page, the entire page of device registers remains accessible to the user program. Use **getpagesize**(2) to determine the page size of the system.

The amount of VME address space that can be mapped into user address space depends on two factors:

- VME address space type (A16, A24, A32)
- hardware platform

For VME A16 address space, the entire 64 KB of A16-VME address space is mappable to user virtual address space.

For A24 address space, only 8 MB of the 16 MB of address space, starting at VME address 0x80000000, is mappable to the user virtual address space. The kernel reserves the remaining 8 MB of address space to support DMA transfers from A24 masters.

For A32 address space, there is some variation according to hardware platform.

CHALLENGE / Onyx systems support up to five VME buses. Users can map in a maximum of 96 MB of A32 address space on each VME bus.

Note: Mapping should be performed in 8 MB increments: Each **mmap**() system call can map a maximum of 8 MB of VME address space into user virtual address space, so eight such **mmap**() calls are needed to map the entire 96 MB of VME address space available.

This 96 MB of VME address space is shared between the kernel and user-level device drivers. Any installed kernel-level VME device drivers that use

VME address space reduce the amount of VME address space available for mapping by the user-level drivers.

On other IRIS platforms (IP5/7/9/17), a maximum of 256 MB can be mapped into user virtual address space. On these platforms, **mmap**() can support mapping in the entire 256 MB of VME address space in a single system call. On systems with dual VME buses, however, the amount of VME address space available for mapping is reduced by half.

See Appendix A for further details of what A32 address space is available for customer boards.

Accessing Mapped Space

VME accesses are sensitive to the access size, so extra caution is called for once VME address space is mapped into a user's virtual space. For example, A16D8 boards may support only 8-bit accesses, while A16D16 boards may support both 8-bit and 16-bit accesses. Similarly, A32D32 may support only 32-bit accesses or may support 8-bit, 16-bit, 32-bit accesses. User-level device drivers should ensure that the data structures onto which the VME address space is mapped generate the proper size of transaction on the VME bus.

VME-bus Error Handling for User-level Device Drivers

Bus errors can occur when a read or write on the VME times out. This can be triggered by user-level drivers accessing a mapped VME space for which no controller exists, or if the controller fails to respond.

Caution: If the kernel cannot determine whether the bus error is harmless to the system, the system panics.

Read Errors

If a VME-bus read error is triggered by a user-level VME driver, the driver process receives a SIGBUS signal. If the driver needs to be aware of the error, then it can catch the signal and take appropriate action. Read errors are synchronous, so the user process can get a definite idea of what PC or routine caused the error.

Write Errors

VME-bus write errors are asynchronous: when a user-level driver writes to the mapped VME address space, the CPU does not stall at that address, but continues executing further instructions. Since the write error time-out bus takes up to 80 microseconds on a VME bus and another finite amount of time for handling the error interrupt from the VME bus, handling write errors can be complicated.

A bad VME write error results in a SIGBUS signal being sent to the offending process, if that process is still running on the system. Since it takes a finite amount of time to send the signal to the user-level driver process that triggers a bad VME write, it is essential for the user-level driver to wait for up to 10 milliseconds before exiting. However, this is only necessary before exiting the system (to allow for the handling of the last few bad writes) or when the user process wants to assure that the write completed. It is not necessary to wait for 10 milliseconds after every VME write.

User-level DMA Library (*udmalib*)

A user-level interface for VME drivers provides access to DMA engines on CHALLENGE / Onyx (IP19) and POWER CHALLENGE / POWER Onyx (IP21) hardware platforms. This interface is meant to be used when performance is critical and the VME-bus board itself does not support DMA.

Users can move data between a buffer and a VME-bus board faster with a DMA engine than with normal PIO operations. However, because there is only one DMA engine per VME bus on the CHALLENGE series, the DMA engine is a scarce resource. The user-level DMA support library *udmalib* allocates the DMA engine exclusively to the first user to request it, and no other user can access it until the current user frees it up. See the **usrdma**(7M) and **udmalib**(3K) man pages for further detail and usage of user-level DMA library calls.

The following functions are supported by the user-level DMA library:

dma_open	Get exclusive use of a DMA engine
dma_close	Free up the DMA engine
dma_allocbuf	Allocate a buffer suitable for DMA

dma_freebuf	Free up a DMA buffer
dma_mkparms	Define a DMA operation
dma_freeparms	Free up DMA parms resources
dma_start	Perform DMA operation between a buffer and the VME bus

Here is a sample code fragment using the user DMA library:

```
#include <udmalib.h>
#include <stdio.h>

do_dma(int adap, void *vmeaddr, int size)
{
   udmaparm_t    *parms;
   udmaid_t     *dp;
   vmeparms_t    vparm;
   void        *iobuf;
   int          err = 0;

   if( (dp = dma_open(DMA_VMEBUS,adap)) == NULL ){
      (void)fprintf(stderr,
                  "unable to start adapter %d\n",adap);
      return 1;
   }

   /* get a buffer and phys address */
   if( (iobuf = dma_allocbuf(dp,size)) == NULL ) {
      (void)fprintf(stderr,"iobuf alloc failed\n");
      (void)dma_close(dp);
      return 1;
   }

   vparm.vp_block = 0;
   vparm.vp_datumsz = VME_DS_HALFWORD;
   vparm.vp_dir = VME_READ;
   vparm.vp_throt = VME_THROT_256;
   vparm.vp_release = VME_REL_RWD;
   vparm.vp_addrmod = 0xd;

   /* create DMA parms */
   if( (parms = dma_mkparms(dp,&vparm,iobuf,size)) == NULL
)  {
      (void)fprintf(stderr,"dma failed\n");
      (void)dma_freebuf(dp,iobuf);
```

```
         (void)dma_close(dp);
         return 1;
      }

      if( err = dma_start(dp,vmeaddr,parms) )
         (void)fprintf(stderr,"dma failed\n");

      if( dma_freebuf(dp,iobuf) ||
          dma_freeparms(dp,parms) ||
          dma_close(dp) )
         (void)fprintf(stderr,"dma release failed\n");

      return err;
   }
```

Writing Kernel-level VME Device Drivers

Determining VME Device Addresses

Each VME device has a set of VME-bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the VME device. Your driver can map these VME addresses into the host processor address space: your users can access the device with simple reads and writes. VME devices can be classified as A16, A24, A32, or A64 VME slaves. Each class specifies a range of addresses to which the device responds. You determine the slave addressing mode from the technical specification for the device.

Once you determine the addressing mode, choose VME addresses that do not conflict with existing VME device drivers. For each slave addressing class, Silicon Graphics has reserved a range of addresses for use by user-written drivers. These ranges are listed in */var/sysgen/system/irix.sm* and tabulated in Appendix A, “System-specific Issues”.

You must choose a VME-bus interrupt priority level for the device. The VME-bus interrupt priority level must be a value from one to seven. Later, all VME interrupts are channeled into one CPU interrupt level. The priority of this CPU interrupt is below that of the clock and any on-CPU devices.

In the past, it was necessary to reserve a VME interrupt vector. Since most VME devices can program the interrupt vector through software, a dynamic allocation scheme for vectors now hands VME vectors out to drivers at initialization time. However, if your VME device has a hard-wired or jumpered VME vector, it is still possible to reserve the VME vector that the device requires.

When CPU interrupts are assigned to the VME bus, the CPU services the VME-bus interrupts in order of their VME-bus priority. For each CPU interrupt, the system services only one device per VME-bus priority level. If more than one VME-bus interrupt occurs at the same VME-bus interrupt priority level, all but one device must wait until the next time the CPU services the VME-bus CPU interrupt.

After picking an appropriate set of addresses and an interrupt priority level, you must program the VME device to respond accordingly. Usually, you do this with jumpers or switches. Some VME devices allow you to program the VME vector and interrupt priority level at boot time (from your driver's *drv***edtinit**() routine).

If the device performs DMA, you need to know the addressing mode by which the device accesses main memory. This addressing mode is called its *master* addressing mode, as opposed to the *slave* addressing mode described above. Silicon Graphics supports A24, A32, and A64[1] VME master addressing. (POWERchannel-2 has master DMA capability.) The master addressing mode determines the driver structure to some degree.

Including VME Device Drivers in the Kernel

Chapter 2, "Writing a Device Driver," provides general information on adding a driver to the kernel. This section describes specifics concerning VME drivers.

To add a new kernel-level device driver, you must create your own *irix.sm* file that contains the appropriate directive telling **lboot** how to include your driver. The filename, which must end with "*.sm*", belongs in the directory */var/sysgen/system*. Because **lboot** can probe for VME devices, **lboot** can conditionally include a VME device driver into the kernel.

If the current system contains the VME device, **lboot** includes the driver; otherwise, it saves memory by leaving it out. Use the *VECTOR* directive to include a VME device conditionally. In addition to the module name, the *VECTOR* directive requires that you fill out these fields:

adapter	The adapter number identifying which VME bus out of possibly several.
bustype	This must be set to VME.
ctlr	The device number that differentiates between more than one device of the same type.

[1] R4000 – R4400 use 64-bit MIPS III mode in the CHALLENGE/Onyx chassis.

exprobe_space	This is an extended probe that can do reads and writes with compares against expected values to search for a VME device at an address. The first *arg* defines a sequence of reads and writes. The second *arg* specifies the address space. The third *arg* specifies the probe address. The fourth *arg* is the size of the probe, 1-4 bytes. The fifth *arg* is the expected read or write value. The last argument is a mask that the fifth *arg* is ANDed against.
iospace, iospace2, iospace3	This is a triple identifying a VME-bus space, address, and size. The bus space is one of *A16NP, A16S, A24NP, A24S, A32NP, A32S*.
ipl	The VME interrupt priority level. This must be a value from 1 to 7, as described above.
probe_space	This is a triple identifying a VME-bus space, address, and read byte count. The address specified is read by **lboot** to determine the existence of device. If you do not specify a probe address, the module is automatically included in the kernel.
vector	The VME interrupt vector value. This must not be used unless the VME device has hard-wired or jumpered vector values.

You must also create a master file under */var/sysgen/master.d.* A master file has four sections:

- a tabulated ordering of flags
- phrases and values interpreted by the configuration program and used to build device tables
- a list of stub routines
- a section of C code.

The first, non-blank, non-comment line is interpreted for flags, phrases, and values. Other non-comment lines that follow, up to a line that begins with a dollar sign, specify stubs. Anything that follows the line beginning with a dollar sign is processed to interpret special characters, then compiled into the kernel.

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 include at least the character device flag *c*. (You do not need the *s* flag for VME device drivers because **lboot** can probe for VME devices.) See */var/sysgen/master.d/README* and the **master**(4) man page.

Note: For network drivers, the FLAG field would blank with a "**-**" (hyphen).

As an example, suppose you want to add a mythical VME device driver to the kernel. You must copy the driver object file *vdk.o* to */var/sysgen/boot*, and you must add a line similar to the following to a file called *vdk.sm* in */var/sysgen/system*:

```
VECTOR: bustype=VME module=vdk ipl=1 ctlr=0 adapter=0
iospace=(A16S,0x400,0x200) iospace2=(A16S,0x800,0x100)
probe_space=(A16S,0x404,2)
```

Note: The above lines must all be on one line in the *irix.sm* file.

Note that the bus addresses and sizes are specified in hexadecimal format. The *ctlr=* value helps identify a device when more than one uses the same driver. If there is more than one device, give each a unique number, starting from zero. In the above example, **lboot** reads two bytes at probe address 0x404 to determine whether the device is present.

After examining */usr/include/sys/major.h*, you determine that major device number 51 is available and can be used for this device. You then create a master file *vdk* in */var/sysgen/master.d*, and enter:

```
*FLAG     PREFIX    SOFT      #DEV       DEPENDENCIES
 c        vdk       51        -
```

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.

Synopsis

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

edt Type Structure

When the system calls your *drv***edtinit**() routine, it hands the routine a pointer to a structure of type *edt*. (This structure type is defined in the *sys/edt.h* header file.)

Structure Definition

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

#define NBASE 3

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)( );   /* device init/run-time probe */
   iospace_t e_iospace[NBASE];
};
```

Based on *e_bus_type*, **lboot** will set up *e_bus_info* to point to the corresponding bus-dependent data structure (that is, *vme_intrs*). With this two-layer structure, it is easier to extend *edt* to support EISA, GIO, or other types of buses.

CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx systems can support more I / O address space than the kernel can, so it is necessary to provide a way of allocating only the part of the I / O space that is needed into the kernel address space. Programming the I / O board / adapter registers dynamically assigns the mapping. The *edt* structure must describe the device by adapter ID and adapter bus address. The kernel uses this information to initialize the kernel virtual address.

On POWER Series workstations, ranges of VME-bus address space are mapped one-to-one with K2 segment addresses. This makes accessing the VME bus easy, but is also limiting. Only a small amount of K2 space is available for use by VME, so very little of the VME address space is made available. Even worse, for dual VME-bus systems, the space previously available is now halved because the two buses must share it.

The CHALLENGE series supports up to five VME buses. Since K2 space is a limited resource, and dividing up what is available by five makes the extra VME buses next to useless, a new approach was tried. The CHALLENGE series does not have a direct K2 address map into VME-bus space. Each VME-bus adapter has the ability to map fifteen 8 MB windows of VME-bus space into K2 space. These windows can be moved around at will to give the illusion of a much larger address space.

To access a VME space, a user must allocate a PIO map, which provides a translation between a kernel address and VME space. These mappings can be “FIXED” or “UNFIXED.” As on POWER Series platforms, a FIXED mapping is a one-to-one mapping of a range of VME-bus space into the driver’s address space. An UNFIXED window takes advantage of the sliding window ability on the CHALLENGE series, which supports both FIXED and UNFIXED mappings.

In an UNFIXED map, VME-bus space cannot be accessed directly; instead, access is provided through special **bcopy**() routines used to move data between VME space and kernel buffers. While it is not always possible to get a FIXED mapping, an UNFIXED mapping is always available. The special **bcopy**() routines work for both FIXED and UNFIXED mappings. On

POWER Series and earlier workstations, UNFIXED mappings are treated as FIXED mappings.

The PIO mapping routines also have a general interface that allows them to be used for mapping in bus spaces other than VME.

The support routines for PIO mapping are:

pio_mapalloc	Allocate a PIO map.
pio_mapaddr	Map bus space to a driver accessible address (FIXED maps only).
pio_mapfree	Free a previously allocated PIO map.
pio_badaddr	Check to see whether a bus address is equipped.
pio_wbadaddr	Check to see whether a bus address is equipped.
pio_bcopyin	Copy data from bus space to kernel buffer.
pio_bcopyout	Copy data from kernel buffer to bus space.

These PIO maps are normally set up in the driver's *drv***edtinit**() routine.

e_iospace is enhanced to be a structure of I/O base address, size and type of the I/O mapping, and kernel virtual address space. *ios_type*, *ios_iopaddr*, and *ios_size* are initialized by **lboot** from the system, and *ios* is assigned when a driver is initialized.

e_adap is added to specify the adapter number, while *e_ctlr* is for physical controller ID.

To pass the desired interrupt CPU to the driver via the *irix.sm* file, use the *VECTOR* directive. The line

```
VECTOR: module=XXX intrcpu=3
```

directs **lboot** (via **autoconfig**) to set the *v_intrcpu* field for the module's **edt** struct to 3 and the *v_setintrcpu* field to 1, indicating that *v_intrcpu* is valid. If no *intrcpu*= statement appears in the VECTOR line, *v_setintrcpu* is set to 0. The module's **edtinit** function may then use these fields to route interrupts as desired.

```
void
XXXedtinit (struct edt *ep)
{
     if (ep->setcpuintr)
               dest_cpu = ep->cpuintr;
     else
          dest_cpu = <some default>;

     ...machine-specific intr routing ...
}
```

vme_intrs Structure

In the case of a VME driver, the field *e_bus_info* will point to the *vme_intrs* structure.

Structure Definition

```
typedef struct vme_intrs {
    int     (*v_vintr)();    /* interrupt routine */
    unsigned char  v_vec;    /* vme vector */
    unsigned char  v_brl;    /* interrupt priority level */
    unsigned char  v_unit;   /* software identifier */
} vme_intrs_t;
```

The only field that must be accessed is *v_brl*, which contains the *ipl=value* from the *VECTOR* line. The *v_vec* field must be used only if the *VECTOR* line uses the *vector= directive* and your device requires a jumpered or hard-wired VME interrupt vector.

Note: Although **lboot** knows not to include a VME device driver in the kernel for a device not present, it is a good idea for your *drv***edtinit**() routine to probe for its device with **badaddr**(). This lets you write a driver that is prepared if the device is removed from the system after the kernel has been built or when the kernel runs on another system.

Continuing with this mythical VME device driver example, its *drv***edtinit**() routine could look like:

```
struct drvctlrinfo ctlrinfo[MAXCTLR];
drvedtinit(edt_t *e)
{
   int i, vec;
   struct vme_intrs   *info;
   volatile struct drvdevice *dp;
   struct drviopb      iopb;
   piomap_t *pmap;

   pmap = pio_mapalloc(e->e_bus_type,e->e_adap,&e->e_space[0],
      PIOMAP_FIXED,"DRV");

   /* make sure adapter exists and addresses are valid */
   if( pmap == 0 )
      return;
   dp = pio_mapaddr(pmap,e->e_iobase);
   /* probe for the device */
   if( badaddr(&dp->csr,sizeof(dp->csr)) ) {
      cmn_err(CE_WARN,"drv: ctlr %d not installed\n",e->e_ctlr);
      pio_mapfree(pmap);
      return;
   }

   /* save the controller's device registers pointer */
   ctlrinfo[e->e_ctlr]->devregs = dp;

   /* dynamically allocate an interrupt vector */
   vec = vme_ivec_alloc(e->e_adap);
   if( vec == -1 ) {
      cmn_err(CE_WARN,"drv: ctlr %d, no irq vector\n", e->e_ctlr);
      pio_mapfree(pmap);
      return;
   }

   /* register our interrupt routine with the kernel */
   vme_ivec_set(e->e_adap,vec,drvintr,e->e_ctlr);
   iopb.ipl = info->v_brl;
   iopb.vec = vec;
   .
   .
   .
}
```

Two new routines **vme_ivec_alloc**(*uint_t, adapter*) and **vme_ivec_set**(*adapter, vec, intr_func, arg*) are implemented to dynamically allocate an interrupt vector and register this vector into **vme_ivec**(). This scheme supports multiple vectors and loadable drivers. **vme_ivec_alloc**() and **vme_ivec_set**() are used in an **edtinit** routine; **vme_ivec_free**() can be called to free up a vector that has been allocated.

You can specify the vector in the *irix.sm* file for old VME boards with a hard-wired interrupt vector. 0x30-0x3f and 0x70-0x7f are reserved for customer boards.

VME Interrupt Handler

Your driver module must contain a routine of the form *drv***intr**(), where *drv* is the driver prefix. When the device generates an interrupt, the general VME interrupt handler calls your driver's *drv***intr**() routine.

When the VME interrupt handler calls your *drv***intr**(), it passes it the value registered with the **vme_ivec_set**() routine for the device. Within your *drv***intr**() routine, you must set flags to indicate the state of the transfer and to wake any sleeping processes waiting for the transfer to complete. Usually, the interrupt routine calls **biodone**() to indicate that an I/O transfer for the buffer is complete.

Caution: Interrupt routines (*drv***intr**()) must not try to sleep themselves by using **biowait**(), **sleep**(), **psema**(), or **delay**() kernel calls.

With the new dynamic interrupt vector allocation scheme, the argument passed to the individual *drv***intr**() is *arg*, which is set by **vme_ivec_set**() in **edtinit**(). *arg* can be an index, a controller number, the address, or any argument that the driver wants to pass to interrupt the service routine.

The IRIX 5.x and 6.0 *vme_ivec* structure is shown below:

```
struct vme_ivec {
    int   (*vm_intr)(int);
    int    vm_arg;
}vme_ivec[ADAPTER][MAX_VME_VECTS];
```

Note: Although the prototype for the VME interrupt handler routine in the **vme_intrs** (*field v_vintr*) and **vme_ivec** (*field vintr*) structures indicates that it returns an integer value, the return value is not used. The prototype should indicate that the function is of type **void***. It was left unchanged to avoid breaking existing VME device drivers.

To support the loadable device drivers, multiple adapters, and multiple interrupt vectors, the *vem_ivec* table is changed to be dynamically allocated at system boot time. The size depends on the number of VME adapters currently supported by the running system. After the table is allocated, the kernel fills the entries for the devices specified in the *irix.sm* file.

Programmed I/O (PIO)

When transferring large amounts of data, your device driver should use direct memory access (DMA). Using DMA, your driver can program a few registers, return, and put itself to sleep while it awaits an interrupt that indicates the transfer is complete. This frees up the processor for use by other processes. See the ioctl(D2) man page.

Sometimes you must write a driver for a device that does not support DMA. Even if the device does support DMA, you may not want to use DMA to transfer amounts of data so small as not to warrant the DMA overhead.

In these cases, the host processor usually copies the data from the user space to on-board memory. Your driver can then program the device registers to notify the device that the memory is ready. The device controller can then copy the data from its on-board memory to the peripheral device.

Listed below is part of a mythical VME device driver for a printer controller that does not support DMA.

To print data from the user, the driver copies data from the user's buffer to an on-board memory buffer of size *VDK_MEMSIZE*. Following the copy of each chunk, the driver programs the device register to indicate the size of valid data in memory and to tell the controller to start printing.

The driver then sleeps, waiting for an interrupt to indicate that the printing is complete and that the on-board memory buffer is available again. To prevent a race condition, in which the interrupt responds before the calling process can sleep, the driver uses the **splvme**() and **splx**() routines. See the spl(D3) man page.

```
int vdk_state;      /* flag for transfer state */

int
vdkwrite(dev_t dev, uio_t *uiop, cred_t *crp)
{
   register int size;
   register int i;
   int s;
   int err;

   /* while there is data to transfer */
   while( uiop->uio_resid > 0 ) {

      /* can only move VDK_MEMSIZE bytes at a time */
      size = MIN(uiop->uio_resid,VDK_MEMSIZE);

      if( (err = uiomove(vdk_memory,size,UIO_WRITE,uiop)) != 0 )
         return err;

      /* block interrupts until we sleep */
      s = splvme(); /* may not be sufficient on MP */

      /* start printing */
      vdk_device->count = size;
      vdk_device->command = VDK_GO;
      vdk_state = VDK_SLEEPING;

      while ( vdk_state != VDK_DONE )
         sleep(&vdk_state,PRIBIO);

      /* restore the process level after waking up */
      splx(s); /* clears any MP locks as well */
   }

   return 0;
}

void
vdkintr(int unit)
{
   ...
   /* printing is complete */
```

```
    if( vdk_state == VDK_SLEEPING ) {
        vdk_state = VDK_DONE;
        wakeup(&vdk_state);
    }
    ...
}
```

The driver's use of the *volatile* declaration informs the optimizer that this variable points to a hardware value that may change. Otherwise, the optimizer may determine that one write to *vdk_device->command* or storage of the value in a register is sufficient.

Note: If your driver uses the **sleep**() and **wakeup**() kernel routines to sleep and awaken, it is a good idea for the top half to verify that the event has occurred before awakening the sleeping process. (See sleep(D3) for details on the sleep / wakeup process synchronization mechanism.) If your driver uses the **biowait**() / **biodone**() routines or the **psema**() / **vsema**() routines to sleep and awaken, you need not worry about its awakening by accident. However, the routines **psema**() and **vsema**() are specific to IRIX and are probably not supported on other operating systems.

The **uiomove**() kernel routine is a useful procedure to call in these situations because it automatically updates *uio* and *iovec* structures while checking for valid user addresses. Remember that *uiop->uio_resid* must be left with the number of bytes remaining untransferred.

Note: **uiomove**() uses **bcopy**() to transfer data. **bcopy**() transfers data as fast as possible between locations in system memory. **bcopy**() takes advantage of CPU-specific commands to optimize performance. On the R4000 processor, **bcopy**() tries to move eight bytes at a time. Most VME-bus boards cannot move data eight bytes at a time, so using this routine directly may not work. A work-around would be to use **uiomove**() to copy the data from a user buffer into a kernel buffer, then to use **pio_bcopy**() to copy the data from the kernel buffer to the VME-bus board. **pio_bcopy**() allows the user to specify the element size being transferred.

DMA Operations

As indicated in “Programmed I/O (PIO),” use DMA (direct memory access) when the device supports it. In its simplest form, DMA is easy to use: your driver gives the device the physical memory address, and the transaction begins. Your driver can then put itself to sleep while it waits for the transfer to complete, thus freeing the processor for other tasks. When the transfer is complete, the device interrupts the processor. On most systems, when large amounts of data are involved, DMA devices obtain higher overall throughput than devices that do only PIO.

DMA operations are categorized as *DMA reads* or *DMA writes.* DMA operations that transfer from memory to a device, and hence read memory, are DMA reads. DMA operations that transfer from a device to memory are DMA writes. Thus, you may want to think of DMA operations as being named the from the point of view of what happens to memory.

There are important cache considerations for drivers using DMA. The cache architecture of the system dictates the appropriate cache operations. Write back caches require that data be written back from cache to memory before a DMA write, whereas both write back and write through caches require the cache to be invalidated before data from a DMA read is used. See “Data Cache Write Back and Invalidation” in Appendix A and the dki_dcache_wbinval(D3X) man page for a discussion of these issues.

Another concern for driver writers is that DMA buffers may require cache-line alignment. If a driver allocates a buffer for DMA, it must use the **kmem_alloc**() function, using the *KM_CACHEALIGN* flag.

The interrupt service routine then calls your *drv***intr**() routine. Your *drv***intr**() routine can check that the transfer is complete (if necessary), set flags indicating the status of the transfer, and then awaken the sleeping process.

Unfortunately, the details of how you implement the simple scheme described above is complicated by the use of virtual memory, different VME addressing modes, and a variety of device and system implementations. To sort through these potentially confusing choices, ask the following questions in order. If the answer to any question is “yes,” go on to the section indicated. Otherwise, proceed to the next question.

A32 Addressing Scatter/Gather Support

Modern computer architectures support *virtual memory*—memory in which the user's view of memory is logically contiguous, but the underlying physical pages are not. Because VME devices understand only physical page addresses, your driver would ordinarily be forced to do transfers one page at a time. At the start of each one-page transfer, your driver would have to awaken the sleeping process and compute the physical address for the next virtual page.

Because this is not efficient, many devices now provide a method to store the address mapping for the entire transfer up front. Your driver can usually do this merely by programming the device with a table of physical addresses for all of the upcoming transfer. This method of regrouping of noncontiguous physical memory is called *scatter/gather.*

If your VME device supports scatter / gather, uses A32 addressing, has less than 4 GB of physical memory, and you are not on a CHALLENGE / Onyx series system, proceed to “VME Devices with Scatter / Gather Capability.”

DMA Mapping for High-end Systems and Older Systems

Older Silicon Graphics systems and current high-end systems provide for address mapping of physical addresses so that even devices that do not support scatter / gather in the controller can transfer to and from noncontiguous physical pages with ease. This facility, called *DMA mapping*, is available on 4D / 100 through 4D / 400, Crimson, CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx series systems. Indigo, Indigo^2, and Indy workstations have no VME-bus support. DMA mapping works equally well for both VME A24 and A32 master addressing. See “Using DMA Maps” for a description of how to use DMA mapping.

Does the VME Device Perform A24 Master Addressing?

If the VME device uses A24 addressing, and your system does not support DMA mapping, the controller can access only the first 8 MB of physical memory. Because user programs may use physical pages beyond 8 MB, your device driver must do DMA into a kernel buffer and copy from that buffer to the user’s pages. See “DMA on A24 Devices with No DMA Mapping.”

A32 Addressing with No Scatter/Gather

If you are writing a driver for an A32 VME device that does not support scatter/gather on a workstation that does not support DMA mapping, see “DMA on A32 Devices with No Scatter/Gather Capability” for advice on how to implement this driver type.

VME Devices with Scatter/Gather Capability

Chapter 2, “Writing a Device Driver,” tells you to use the **physiock**() kernel routine to fault in and lock the physical pages corresponding to the user's buffer. **physiock**() also remaps these physical pages to a kernel virtual address that remains constant even when the user's virtual addresses are no longer mapped.

Internally, **physiock**() allocates a structure of type *buf* if you pass a *NULL* pointer (**physiock**() uses this structure to embody the transfer information). **physiock**() then calls your *drv***strategy**() routine and passes it a pointer to the *buf* type structure that it has allocated and primed. Your *drv***strategy**() routine must then loop through each page, starting at the kernel virtual address, and load each device scatter/gather register in turn with the corresponding physical address. Use the **kvtophys**() routine to convert a kernel virtual address to a physical address.

For example, suppose the mythical device is now an A32 VME device that supports scatter/gather. The scatter/gather registers for the device are simply a table of integers that store the physical pages corresponding to the current transfer. To start the transfer, the driver gives the device the beginning byte offset, byte count, and transfer direction. The code is:

```
#include <sys/sysmacros.h>
/* pointer to device registers */
volatile struct vdk_device *vdk_device;
vdkstrategy(bp)
struct buf *bp;
{
   int npages;
   register volatile int *sgregisters;
   register int i, v_addr;

   /* Get address of the scatter/gather registers */
```

```
    *sgregisters = vdk_device->sgregisters;

   /* Get the kernel virtual address of the data */
    v_addr = bp->b_un.b_addr;

   /* Compute number of pages received.
    * The dma_len field provides the number of pages to
    * map. Note that this may be larger than the actual
    * number of bytes involved in the transfer. This is
    * because the transfer may cross page boundaries,
    * requiring an extra page to be mapped.*/
    npages = numpages (v_addr, bp->b_bcount);

   /* Translate the virtual address of each page to a
    * physical page number and load it into the next
    * scatter/gather register.  The btoct macro
    * converts the byte value to a page value after
    * rounding down the byte value to a full page.
    */
    for (i = 0; i < npages; i++) {
      *sgregisters++ = btoct(kvtophys(v_addr));

   /*
   /* Get the next virtual address to translate.
    * (NBPC is a symbolic constant for the page
    * size in bytes.)
    */

   v_addr += NBPC;
   }

   /*
    * Provide the beginning byte offset and count to the
    * device.
    */

  vdk_device->offset = (unsigned int)bp->b_dmaaddr & (NBPC-1);
   vdk_device->count = bp->b_bcount;
   if ((bp->b_flags & B_READ) == 0)
      vdk_device->direction = VDK_WRITE;
   else
      vdk_device->direction = VDK_READ;
}
```

Using DMA Maps

On IRIS 4D / 100, 4D / 200, 4D / 300, 4D / 400, Crimson, CHALLENGE / Onyx, and POWER CHALLENGE / POWER Onyx series systems, a number of registers perform mapping from physical pages to other physical pages. Because the addresses that are mapped are really no longer “physical” addresses, they are now referred to as “bus virtual” addresses. Your driver should allocate these system mapping registers when it opens the device, remap these registers for every transfer, and then free them when it closes the device.

Internally, all the mapping routines deal with a DMA map structure. The values stored in members of the structure are subject to change from release to release. Therefore, when your driver manipulates the DMA maps, it must use the following routines only. Your driver must not try to access the structure members directly.

dma_mapalloc	Allocate a DMA Map
dma_map	Map a Virtual Address Space
dma_mapaddr	Return the Bus Virtual Address

Note: When using DMA maps, be sure that the source or destination address begins on a 32-bit word boundary.

Caution: Once you free a DMA map, it is gone and you can no longer use it. Free it only after the DMA operation has been successfully aborted.

Example Using DMA Maps

Suppose the mythical VME device is an A24 device for use with an IRIS-4D / 100 series workstation. The driver begins the transfer by giving the device the starting address, byte count, and transfer direction. Some driver excerpts could look like the following:

```
#define MAXTRANSFER  4      /* maximum transfer size in pages */
                            /* pointer to device registers */
volatile struct vdk_device*vdk_device;
dmamap_t    vdk_map;         /* pntr to DMA map */
struct buf *vdk_curbp;      /* current buffer */
caddr_t     vdk_curaddr;     /* current address to transfer */
int         vdk_curcount;
```

```
static      int vdk_vmeadap   /* computed during edtinit */

vdkopen(dev, flag, otyp, crp)
dev_t   *dev;
int     flag, otyp;
credit  *crp;
{
...
    vdk_map =
        dma_mapalloc(DMA_A24VME, vdk_vmeadap,
            MAXTRANSFER, 0);
...
}

vdkclose(dev, flag, otyp, crp)
dev_t   dev;
int     flag, otyp;
{
...
        dma_mapfree(vdk_map);
...
}

vdkstrategy(bp)
struct buf *bp
{
...

/* Save structure pointer for the interrupt  routine, vdkintr */
    vdk_curbp = bp;

    /* Set up the mapping registers */
    bp->b_resid = bp->b_bcount;
    vdk_curaddr = bp->b_dmaaddr;
    vdk_curcount = dma_map
        (vdk_map, vdk_curaddr, bp->b_resid);

/* Tell the device starting bus virtual address and count */
    vdk_device->startaddr =
            dma_mapaddr(vdk_map, vdk_curaddr);
    vdk_device->count = count;
    if (bp->b_flags & B_READ) == 0)
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
        vdk_device->command = VDK_GO;
        /* Set up for next transfer */
        vdk_curaddr += count;
        ...
}

vdkintr(unit)
int unit;
{
    int count;
    register struct buf *bp = vdk_curbp;
    ...

    if(error) {
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return( );
    }

    /*On successful transfer of last chunk, continue if necessary.*/
    bp->resid -= vdk_curcount;
    if (bp->b_resid > 0) {
        count = dma_map(vdk_map, vdk_curaddr, bp->b_resid);
        vdk_device->startaddr = dma_mapaddr(vdk_map,vdk_curaddr);
        vdk_device->count = count;
        if (bp->b_flags & B_READ) == 0)
            vdk_device->direction = VDK_WRITE;
        else
            vdk_device->direction = VDK_READ;
        vdk_device->command = VDK_GO;
        vdk_curaddr += count;
    } else {
        biodone(bp);
    }
    ...
}
```

Note: As with other examples, error checking is omitted, but should not be omitted in real code.

DMA on A24 Devices with No DMA Mapping

VME A24 addressing specifies an address space that may be smaller than all of physical memory. Silicon Graphics workstations that support the VME bus provide a DMA mapping capability so that your driver can access all of physical memory.

Some Silicon Graphics systems allow A24 masters to access only the first 8 MB of physical memory. Your driver must declare a static buffer assigned to contiguous physical pages in low memory, and it must do DMA transfers to and from this buffer only. After a transfer is complete, the driver can copy the data from this buffer to the user's memory. Because kernel static data uses contiguous physical memory pages, scatter / gather hardware is not needed. Keep this buffer no more than a few pages long; otherwise, the kernel BSS segment may be too large for the system to boot it. The limits on kernel size vary from system to system and sometimes across releases and other included kernel drivers.

Using a DMA read operation, your driver can transfer data from a device directly to physical memory. Any words in the processor data cache corresponding to this memory are now stale. To invalidate the data cache lines associated with the physical memory addresses, your driver must call the **dki_dcache_inval**() routine. If your driver calls the kernel routine, **physiock**(), it need not call **dki_dcache_inval**() because **physiock**() calls the **userdma**() routine and thus invalidates the data cache.

Using a DMA write operation, your driver can transfer data from memory to the device. Prior to the transfer, any words in the processor data cache corresponding to this memory may be more up-to-date than the corresponding memory. In this case, memory is said to be stale with respect to cache, and any words in the cache corresponding to this memory must be written back to memory before the DMA starts. Use the **dki_dcache_wbinval**() routine to write the contents of the cache back to memory. If your driver calls the kernel routine **physiock**(), it need not call **dki_dcache_wbinval**() because **physiock**() calls the **userdma**() routine and thus writes back and invalidates the data cache.

The driver below does a DMA transfer into memory that has not been prepared by **physiock**(). The driver can do this because the data is in a kernel buffer, so there is no need to lock it in memory and remap it. See

Appendix A, “System-specific Issues,” for more information on data cache management.

For example, suppose the mythical VME device is now an A24 master. The driver begins the transfer by programming the starting address, byte count, and transfer direction.

Note: On systems with multiple word cache lines, this buffer must be aligned on a cache line boundary for correct operation. Normally, some extra bytes (currently 128) must be allocated, and a pointer into the buffer, whose address is adjusted to be cache-line aligned, must be used (see *SCACHE_ALIGNED* in *sys/immu.h*).

Alternatively, allocate a buffer during the *drv***edtinit** routine with **kmem_alloc** and the *SCACHE_ALIGN* flag, and verify that the address is in the low 8 MB of physical memory with **kvtophys**().

A driver excerpt follows:

```
#define V DKBUFSIZE 4096 /* kernel buffer size */
/* pointer to device rgstrs */
volatile struct vdk_device *vdk_device;
char vdk_buffer[VDKBUFSIZE] ; /* kernel bufr */
struct buf *vdk_curbp; /* current bufr */
caddr_t vdk_curaddr;   /* current address to transfer */
caddr_t vdk_curcount   /* current count being trnsfrd */

vdkstrategy(buf *bp)
{
    ...
    bp->b_resid = bp->b_bcount;
    vdk_curaddr = bp->b_un.b_addr;
    vdk_curbp = bp;
    vdk_curcount = MIN(bp->b_resid,VDKBUFSIZE);

    /* for a write operation, copy from the user's memory
     * to the kernel buffer
     */
    if( (bp->b_flags & B_READ) == 0 )
        bcopy(vdk_curaddr,vdk_buffer,vdk_curcount);

    /* tell the device the phys address of kernel
     * buffer and count
     */
    vdk_device->startaddr = kvtophys(vdk_buffer);
    vdk_device->count = vdk_curcount;
    if( (bp->b_flags & B_READ) == 0 )
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->com = VDK _GO;

    ...
}

vdkintr(int unit)
{
    int count; error
    register struct bug *bp = vdk_curbp;

    ...
    /* check for an error */
    if( error ) {
        bioerror(bp,EIO);
        biodone(bp);
        return;
    }
    /* For a read operation, copy the data from the kernel
     * buffer to the user's pages. The bcopy routine
     * must be used with the kernel virtual address of
     * the user's buffer since the user's virtual
     * addresses aren't mapped anymore.
     *
     * Note that the data cache is flushed before
     * copying from a cached address. Ordinarily physiock()
     * does this for you. See Appendix A for when and how
     * to flush the data cache.
     */
    if( (bp->b_flags & B_READ) != 0 ) {
        dki_dcache_flush(vdk_buffer, vdk_curcount);
        bcopy(vdk_buffer, vdk_curaddr, vdk_curcount);
    }

    /* Decrement the residual count and bump up the current
     * transfer address. If there are any bytes left to
     * transfer, do it again.
     */
    bp->b_resid -= vdk_curcount;
    vdk_curaddr += vdk_curcount;
    if( bp->b_resid == 0 ) {
        biodone(bp);
        return;
    }

    vdk_curcount = MIN(bp->b_resid,VDKBUFSIZE);

    if( (bp->b_flags & B_READ) == 0 )
        bcopy(vdk_curaddr,vdk_buffer,vdk_curcount);

    vdk_device->startaddr = kvtophys(vdk_buffer);
    vdk_device->count = vdk_curcount;
    if( (bp->b_flags & B_READ) == 0 )
        vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->com = VKD_GO;
}
```

DMA on A32 Devices with No Scatter/Gather Capability

If neither your device nor your workstation provides scatter / gather capability, your driver must break up a data transfer so that no transfer crosses a page boundary. The IRIX operating system provides a utility called sgset(D3X), which simulates scatter / gather registers in software. It should not be used on systems that support DMA address mapping. (See the *IRIX Device Driver Reference Pages* for details on this routine.) Your driver can use this utility to perform the virtual-to-physical mapping up front. Or, as the example below shows, your driver can do this mapping following the transfer of each page:

```
/* pointer to device registers */
volatile struct vdk_device    *vdk_device;
struct buf   *vdk_curbp      /* current buffer */
caddr_t      vdk_curaddr;    /* current address to transfer
*/
int          vdk_curcount;
vdkstrategy(bp)
struct buf    *bp;
{
    ...
    vdk_curbp = bp;
    bp->b_resid = bp->b_bcount;
    /*
     * Initialize the current transfer address and count.
     * The first transfer must finish the rest of the
     * page, but do no more than the total byte count.
     */
    vdk_curaddr = bp->b_un.b_addr;
    vdk_curcount = NBPC -
        ((unsigned int)vdk_curaddr & (NBPC-1));
    if (bp->b_resid < vdk_curcount)
        vdk_curcount = bp->b_resid;
    /* Tell the device starting physical address, count,
     * and direction */
    vdk_device->startaddr = kvtophys(vdk_curaddr);
    vdk_device->count = vdk_curcount;
    if (bp->b_flags & B_READ) == 0)
    vdk_device->direction = VDK_WRITE;
    else
        vdk_device->direction = VDK_READ;
    vdk_device->command = VDK_GO;
    vdk_curaddr += vdk_curcount;
    biowait(bp);
    ...
}
vdkintr(unit)
int unit;
{
    int count, error;
    register struct buf *bp = vdk_curbp;
    ...
    if(error) {
        bioerror (bp,EIO);
        biodone(bp);
        return;
    }
    /* On successful transfer of last chunk, continue
     * if necessary. */
    vdk_curaddr -= vdk_curcount;
    if (bp->b_resid > 0) {
            count =
                (bp->b_resid < NBPC ? bp->b_resid : NBPC);
            vdk_device->startaddr = kvtophys(vdk_curaddr);
            vdk_device->count = count;
            if (bp->b_flags & B_READ) == 0)
                vdk_device->direction = VDK_WRITE;
            else
                vdk_device->direction = VDK_READ;
            vdk_device->command = VDK_GO;
            vdk_curaddr += count;
    } else {
            biodone(bp);
    }
    ...
}
```

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:

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.

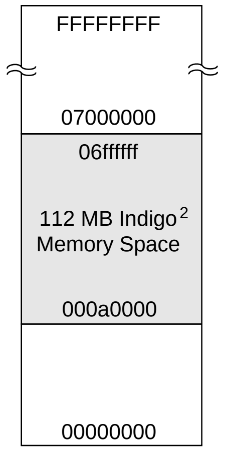


Figure 4-1 Indigo^2 Memory Space

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)
   0xzC80 = 00000111, 0xzC81h = 01000001

Figure 4-2 shows the format of the product ID (addresses 0xzC80-0xzC83).

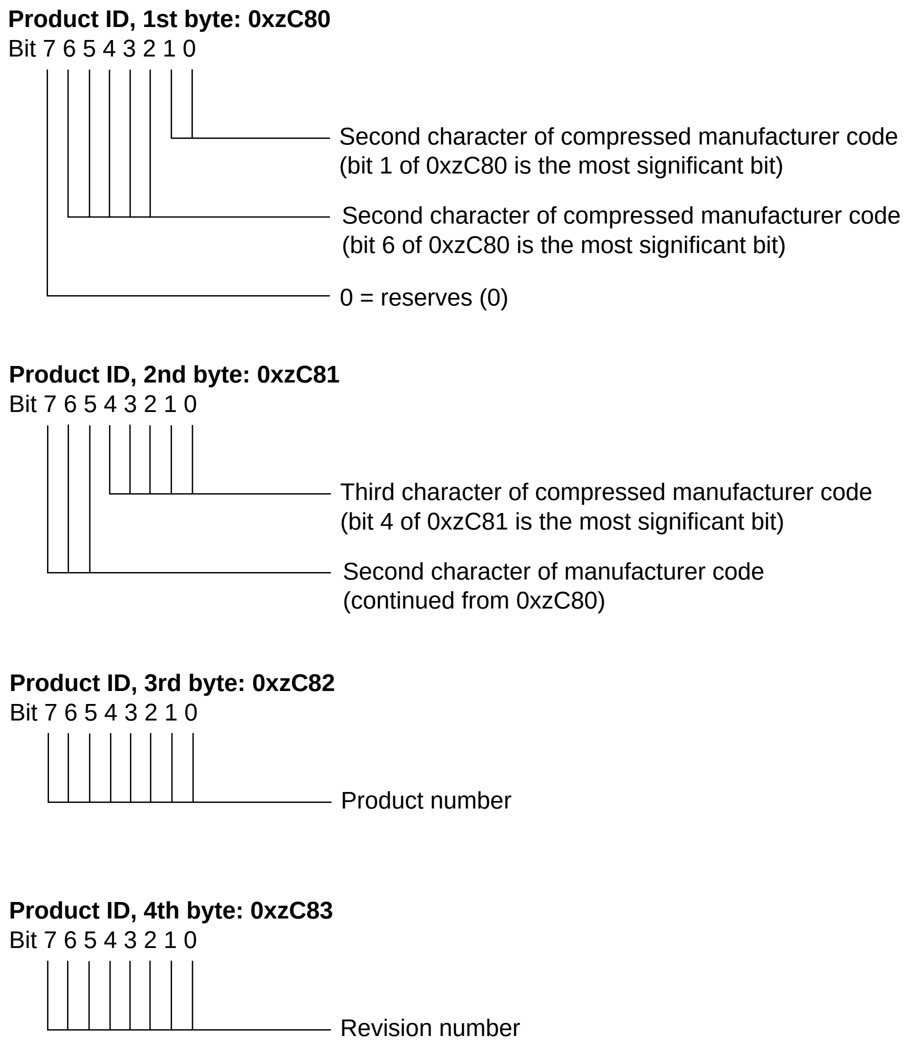

Product ID, 1st byte: 0xzC80
Bit 7 6 5 4 3 2 1 0
Second character of compressed manufacturer code (bit 1 of 0xzC80 is the most significant bit)
Second character of compressed manufacturer code (bit 6 of 0xzC80 is the most significant bit)
0 = reserves (0)
Product ID, 2nd byte: 0xzC81
Bit 7 6 5 4 3 2 1 0
Third character of compressed manufacturer code (bit 4 of 0xzC81 is the most significant bit)
Second character of manufacturer code (continued from 0xzC80)
Product ID, 3rd byte: 0xzC82
Bit 7 6 5 4 3 2 1 0
Product number
Product ID, 4th byte: 0xzC83
Bit 7 6 5 4 3 2 1 0
Revision number

Figure 4-2 Product ID Format

The manufacturer ID is located at bytes offset 0xzC80-0xzC83, 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.

Chapter 5

Writing a SCSI Device Driver

This chapter gives an overview of the SCSI device driver interface and explains how to write a user-level SCSI device driver and a kernel-level SCSI device driver. It contains the following sections:

SCSI-bus Interface Overview

SCSI (Small Computer System Interface) is an industry standard I/O bus designed to provide host computers with device independence within a class of devices. All Silicon Graphics systems provide an interface to at least a single SCSI bus for peripherals that support the SCSI standard. Your device driver can place commands on the bus by using the SCSI host adapter driver. Systems with POWERchannel I/O processor boards support two SCSI interfaces per POWERchannel board; those with POWERchannel-2 boards support two native SCSI interfaces plus as many as six additional SCSI interfaces, if mezzanine boards are used, for a maximum of eight SCSI interfaces per POWERchannel-2 board. Systems equipped with VME-SCSI (Jaguar) boards provide two buses per board.

The drivers support all three SCSI standards, SCSI-1, CCS[1], and SCSI-2. Not all optional features are supported, and different systems support different features (such as synchronous, fast, and wide).

In addition, DMA address mapping registers allow noncontiguous physical memory to appear contiguous to the SCSI host adapter. This allows your driver to handle I/O across noncontiguous pages in a single transfer (scatter-gather). The IRIX operating system provides an interface that hides much of the complexity of the SCSI bus management.

Choosing a Driver Model

Choosing between a user-level and a kernel-level device driver model usually depends on the method used to transfer data to and from the device.

Note: Silicon Graphics has a generic SCSI device driver to support its SCSI hardware. This driver has entry points that enable programs to control devices unknown to the generic Silicon Graphics SCSI device driver. In other words, there are hooks to extend the SGI SCSI device driver to manage customer SCSI devices. In the strictest sense, this is not a device driver but an extension to a device driver.

[1] The Common Command Set (CCS) standard is superseded by SCSI-2.

User-level SCSI-bus Device Driver

If you must write a driver for a device that conforms to the SCSI standard, you can often use the */dev/scsi/sc** special files and the *dslib* library of routines and macros to write a user-level program that sends SCSI commands to a device on the SCSI bus.

Although this *dslib* interface to the SCSI device differs from the IRIX standard device interface and requires that you be familiar with the SCSI protocol, you can typically write this sort of user-level SCSI device driver in a fraction of the time that it takes to write a kernel-level SCSI device driver. As a result, even if your ultimate goal is to write a kernel-level driver for a device, you can write a user-level device driver to test the device and familiarize yourself with its features.

However, there are some limits. For example, quasi-SCSI devices that do not strictly adhere to SCSI standards for reporting errors can cause problems. In earlier IRIX releases, the user-level driver did not support SCSI commands of an unusual length (commands not of *grp0*, *grp1*, or *grp2* length). IRIX 4.0 and later supports commands of 6-12 bytes, regardless of group. Finally, the *dslib* routines and macros may be somewhat less efficient (in terms of throughput) than a kernel-level SCSI driver. This lower efficiency may not be a problem when dealing with devices such as scanners and printers. DMA is used for all I/O, just as with kernel SCSI drivers.

IRIX uses the *devscsi* driver for CDROM ISO-9660 filesystems as well as for HFS (Apple® Macintosh®) and DOS™ floppy filesystems.

Kernel-level SCSI Device Driver

You may have to write a kernel-level SCSI driver for a SCSI device when you need the best throughput possible or when you must control an unusual SCSI device. See Chapter 2, “Writing a Device Driver,” for a general description of the IRIX device driver interface.

User-level SCSI Device Drivers

This section explains how to write a user-level driver for a SCSI device. It describes the SCSI interface routines for the integral SCSI controller and gives a short example.

In IRIX 5.0 and later releases, a SCSI target may, in most cases, be used by more than one driver at a time unless it is opened in exclusive mode (currently, only the *tpsc* tape driver uses exclusive mode). See “Kernel-level SCSI Device Drivers” for more information.

Note: There have been extensive changes in the kernel SCSI driver interface in the IRIX 5.x releases. In IRIX 5.x, the *devscsi* interface is available on the **wd93**, **wd95**, and **VMESCSI** (Jaguar) drivers. In prior releases, it was available only on the **wd93**. IRIX 5.2 and 5.3 remain source compatible, however.

To control a SCSI device from a user-level program, you need the routines of the *dslib* library and a device-special file created for the device. The system comes with device-special files for SCSI adapters in the directory */dev/scsi*. These files can be especially useful because they insulate the device driver writer from changes in operating system releases, and they remain valid across all Silicon Graphics platforms, independent of the low-level SCSI driver. Their limitation is that they are created for logical unit 0 only. (If the logical unit number for your device is not zero, you need to create a device-special file for the device. See “Creating Device-special Files for User-level SCSI Drivers.”)

Which special files you use depends on the SCSI ID for the device, a number from 1-7 (wide SCSI is 1-15). The SCSI ID for a device is usually controlled by switch settings or jumpers. (See the device technical specification for details.) For a SCSI device with a SCSI ID of 3 on controller 0, use */dev/scsi/sc0d3l0*; use */dev/scsi/sc0d7l0* for a SCSI device with a SCSI ID of 7, and so on. See the man page for the **ds**(7M) command for more details on device naming.

The remainder of this chapter assumes that you are familiar with the SCSI interface. For additional information on SCSI-bus operation, see the *ANSI Standards X3.131-1986* and *X3T9.2/85-52 Rev 4B.*

Creating Device-special Files for User-level SCSI Drivers

If the logical unit for the SCSI device you want to control from a user-level program is some value other than 0, you need to create a device-special file. See the **mknod**(1) and **ds**(7) man pages for more information. To create a device-special file, use the **mknod** command:

```
% mknod filename type major# minor#
```

where:

filename	The name of the device special file to create for the device.
type	The type of special file: *c* = character special file *b* = block special file
major#	The major device number. For a device that will be controlled from a user-level program, this number is 195, which is the major device number of *devscsi*.
minor#	The 14-bit (decimal) minor device. The bits are defined as shown in Figure 5-1.

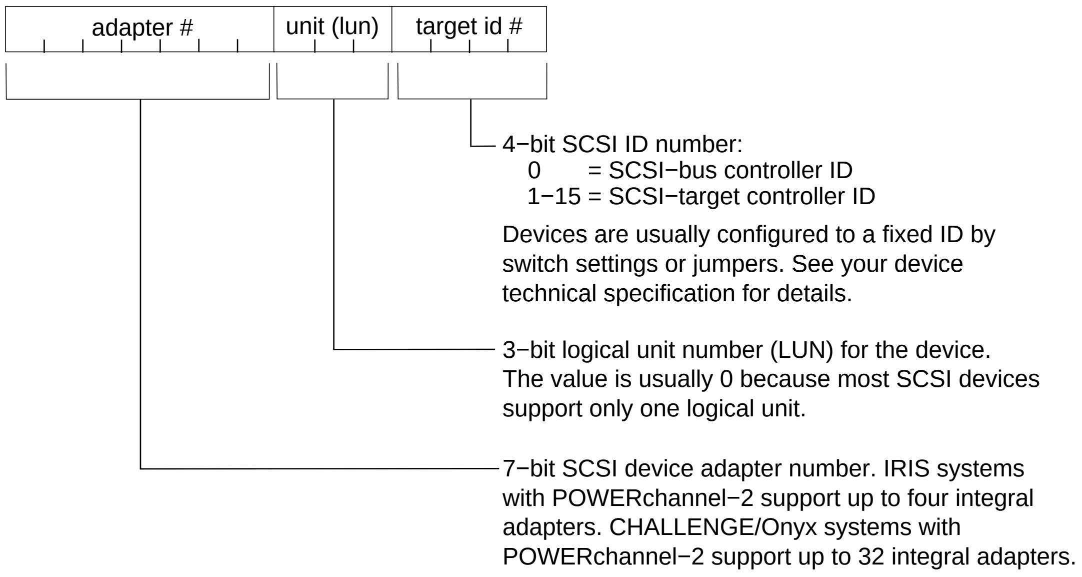

adapter #
unit (lun)
target id #
4−bit SCSI ID number:
0 = SCSI−bus controller ID
1−15 = SCSI−target controller ID
Devices are usually configured to a fixed ID by switch settings or jumpers. See your device technical specification for details.
3−bit logical unit number (LUN) for the device. The value is usually 0 because most SCSI devices support only one logical unit.
7−bit SCSI device adapter number. IRIS systems with POWERchannel−2 support up to four integral adapters. CHALLENGE/Onyx systems with POWERchannel−2 support up to 32 integral adapters.

Figure 5-1 Minor Number Bit Definitions

Example

```
% mknod /dev/scsi/sc0d1l1 c 195 17
```

The minor device number is set to 17_{10}, which is 0010001_2, thus indicating a device that has logical unit number 1 and a target ID of 1.

dsreq – User-level Driver Communication Structure

Your user-level SCSI driver communicates with a SCSI device by reading and setting the values of the members of the *dsreq* type structure. Understanding this structure is essential to writing a user-level SCSI driver. Your driver can access these fields directly; however, for many common tasks that involve controlling a SCSI device, *dslib* has simple routines or macros that can access these values for you. The advantage of using these macros and routines is that, if the structure should change in a future release, you can accommodate the change by changing the internals of the macros or simply recompiling with new macro defs in *dsreq.h*. Thus, code that uses these macros and routines is likely to be portable across releases even if the structure itself changes.

The *dsreq* type structure can be found in the *sys/dsreq.h* directory. The macros associated with a *dsreq* type structure are described below. The *dslib* routines are described in “dslib Routines Description” on page 143.

dsreq Structure

```
typedef struct dsreq {
/* devscsi prefix */

ulong          ds_flags;     /* see flags defined below */
ulong          ds_time;      /* time-out in milliseconds */
ulong          ds_private;   /* for private use by caller */
/* scsi request */

caddr_t        ds_cmdbuf;    /* command buffer */
uchar_t        ds_cmdlen;    /* command buffer length */
caddr_t        ds_databuf;   /* data buffer start */
ulong          ds_datalen;   /* total data length */
caddr_t        ds_sensebuf;  /* sense buffer */
uchar_t        ds_senselen;  /* sense buffer length */
/* miscellaneous */
dsiovec_t      *ds_iovbuf;   /* scatter-gather dma control */
ushort         ds_iovlen;    /* length of scatter-gather */
struct dsreq   *ds_link;     /* for linked requests */
ushort         ds_synch;     /* synchronous xfer control*/
uchar_t        ds_revcode;   /* devscsi version code*/

/* return portion */

uchar_t         ds_ret;       /* devscsi return code*/
uchar_t         ds_status;    /* device status byte value*/
uchar_t         ds_msg;       /* device message byte value */
uchar_t         ds_cmdsent;   /* actual length command*/
ulong           ds_datasent;  /* actual length user data*/
uchar_t         ds_sensesent; /* actual length sense data*/
} dsreq_t;
```

where:

ds_flags The bits of the value for this member are used as flags that determine what the SCSI-bus driver does after you call **doscsireq**(). Use the **FLAGS** macro to access the value of this member. The interface is designed to be implemented on a number of architectures, some of which provide more low-level control of the SCSI bus than IRIX. The file, *dsreq.h* included by *dslib.h*, currently defines this macro as:

```
#define FLAGS (dp) ((dp)->ds_flags)
```

There are symbolic constants that you can use to set the bits of the *ds_flags* value. (Not all of these flags are currently honored. Your driver can test for which flags are honored by checking the returned value of an **ioctl**() call on *devscsi*. See the **ds**(7M) man page.) These constants are defined in *sys/dsreq.h*:

DSRQ_ASYNC – No / Yes sleep until request done. A SCSI device option. Not implemented.

DSRQ_SENSE – Yes / No automatically get sense on status when a check condition occurs. A SCSI device option. All requests return only on completion. Not implemented.

DSRQ_TARGET – Target / Initiator role. A SCSI device option. Not implemented.

DSRQ_SELATN – Select with / without ATN. A select option. Not implemented.

DSRQ_DISC – Identify disconnect not-allowed / allowed. A select option. Not implemented.

DSRQ_SYNXFR – Negotiate synchronous SCSI transfer. A select option.

DSRQ_SELMSG – Send supplied / generated message. A select option. Not implemented.

DSRQ_IOV – Scatter-gather not-specified / specified. A data transfer option.

DSRQ_READ – Input data from SCSI bus to the CPU. A data transfer direction.

DSRQ_WRITE – Output data to SCSI bus from the CPU. A data transfer direction.

DSRQ_BUF – Buffered / Direct data transfer. A data transfer option.

DSRQ_CALL – Notify progress upon completion. A progress / continuation callback option. Used with *DSRQ_ASYNC*. Not implemented.

DSRQ_ACKH – Hold / Don't-hold ACK asserted. A progress / continuation callback option. Not implemented.

DSRQ_ATNH – Hold / don't-hold ATN asserted. A progress / continuation callback option. Not implemented.

DSRQ_ABORT – Send an abort message. Useful only with SCSI commands that have the immediate bit set.

DSRQ_TRACE – Trace / don’t-trace this request. A host option (and so not likely to be portable).

DSRQ_PRINT – Print / don’t-print this request. A host option (and so not likely to be portable).

DSRQ_CTRL1 – Request with host control bit 1. A host option (and so not likely to be portable).

DSRQ_CTRL2 – Request with host control bit 2. A host option (and so not likely to be portable).

DSRQ_MIXRDWR – Request can both read and write.

ds_time

This member sets the time-out value in milliseconds for the completion of a command sent to the SCSI device. You can use the **TIME** macro to access the value of this member.

The file *dsreq.h* defines this macro as:

```
#define TIME (dp) ((dp)->ds_time)
```

If you use *dslib*, you must not change this pointer or the data it references.

ds_private

To access the value of the *ds_private* member, you can use the **PRIVATE** macro currently defined in *dslib.h* as:

```
#define PRIVATE(dp) ((dp)->ds_private)
```

This is intended to be used only in the library support routines.

ds_cmdbuf

This member is a pointer to an array, the SCSI command descriptor you want to send to the device. You can use the **CMDBUF** macro to access this value. The file *dsreq.h* currently defines this macro as:

```
#define CMDBUF(dp) ((caddr_t) (dp)->ds_cmdbuf)
```

ds_cmdlen

This member is the length (in bytes) of the SCSI command pointed to by *ds_cmdbuf*. You can use the **CMDLEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define CMDLEN(dp) ((dp)->ds_cmdlen)
```

Typically, this value is 6, 10, or 12 for SCSI commands of class 0, 1, or 2, respectively.

ds_databuf

This member is a pointer to the start of a data buffer. You can use the **DATABUF** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define DATABUF(dp) ((caddr_t) (dp)->ds_databuf)
```

ds_datalen

This member is the length of the data in the buffer pointed to by the *ds_databuf* member. You can use the **DATALEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define DATALEN(dp) ((dp)->ds_datalen)
```

ds_sensebuf This member is the pointer to the start of the sense buffer. The contents written to this buffer when you request sense information from a device depend on the device. See the device-specific documentation supplied by the manufacturer. You can use the **SENSEBUF** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define SENSEBUF(dp) ((caddr_t) (dp)-
>ds_sensebuf)
```

It is used only if *DSRQ_SENSE* is set in the flags.

ds_senselen This member is the length of the data in the buffer pointed to by the *ds_sensebuf* member. You can use the **SENSELEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define SENSELEN(dp) ((dp)->ds_senselen)
```

**ds_iovbuf* This member is a pointer to a *dsiovec* type structure, a SCSI device I/O vector. This structure is used for DMA scatter-gather control. The *dsiovec* type structure (defined in *dsreq.h*) has two members: *iov_base*, a pointer to a buffer containing a table of physical addresses for the entire transfer, and *iov_len*, the length of the buffer pointed to by *iov_base*.

To access the value of the **ds_iovbuf* member, you can use the macro, **IOVBUF**. The file *dsreq.h* currently defines this macro as:

```
#define IOVBUF(dp) ((caddr_t) (dp)->ds_iovbuf)
```

ds_iovlen This member is the length, in bytes, of the data for scatter-gather transfer. You can use the **IOVLEN** macro to access the value of this member. The file *dsreq.h* currently defines this macro as:

```
#define IOVLEN(dp) ((dp)->ds_iovlen)
```

**ds_link* Not supported.

ds_synch Not supported.

ds_revcode This member is the version code for the *devscsi* driver.

ds_ret

This member is the return code for the command executed on the SCSI device. You can use the **RET** macro to access this member. The file *dsreq.h* currently defines **RET**:

```
#define RET(dp) ((dp)->ds_ret)
```

The file *dsreq.h* defines the following symbolic constants for the value pointed to by this member:

DSRT_DEVSCSI – General failure from SCSI bus.

DSRT_HOST – General host failure, typically a SCSI-bus request.

DSRT_STAI – Protocol error during status phase.

DSRT_EBSY – Busy dropped unexpectedly; protocol error.

DSRT_UNIMPL – Protocol error. Not implemented.

DSRT_CMDO – Protocol error during command phase.

DSRT_REJECT – Message rejected; protocol error.

DSRT_PARITY – Parity error on SCSI bus; protocol error.

DSRT_PROTO – Miscellaneous protocol failure.

DSRT_MEMORY – Host memory error.

DSRT_MULT – Request rejected by SCSI bus.

DSRT_CANCEL – Lower request canceled by SCSI bus.

DSRT_REVCODE – Software obsolete, must recompile.

DSRT_AGAIN – Try again, recoverable SCSI-bus error.

DSRT_NOSEL – No unit responded to select.

DSRT_SHORT – Incomplete transfer (not an error).

DSRT_OK – Completed transfer without error status.

DSRT_SENSE – Command with status, sense data successfully retrieved from SCSI host.

DSRT_NOSENSE – Command with status, error occurred while trying to get sense data from SCSI host.

DSRT_TIMEOUT – Request idled longer than requested. Command could not complete within the limit of the time-out value.

DSRT_LONG – Target over ran data bounds.

ds_status This member is the SCSI target's status byte value for the SCSI command just executed. You can use the **STATUS** macro to access this member.

The file *dsreq.h* currently defines **STATUS**:

```
#define STATUS(dsp) ((dp)->ds_status)
```

The file *dsreq.h* defines the following symbolic constants for this byte:

STA_GOOD – the target has successfully completed the SCSI command.

STA_CHECK – indicates an error, exception, or abnormal condition. If *DSRQ_SENSE* is set, request sense is automatically done. See *ds_sensebuf*.

STA_BUSY – the target is busy, so the command was not issued.

STA_IGOOD – SCSI command with link completed.

STA_RESERV – Command aborted because it tried to access a logical unit or an extent within a logical unit that reserves that type of access to another SCSI device.

ds_msg Not implemented.

ds_cmdsent The value of this *member* is the length of the SCSI command actual sent. You can use the **CMDSENT** macro to access this member. The file *dsreq.h* currently defines **CMDSENT**:

```
#define CMDSENT(dsp) ((dp)->ds_cmdsent)
```

ds_datasent The value of this member is the length of the user data actually transferred. You can use the **DATASENT** macro to access this member. The file *dsreq.h* currently defines **DATASENT**:

```
#define DATASENT(dsp) ((dp)->ds_datasent)
```

ds_sensesent The value of this member is the length of the sense data actually received. You can use the **SENSESENT** macro to access this member. The file *dsreq.h* currently defines **SENSESENT**:

```
#define SENSESENT(dsp) ((dp)->ds_sensesent)
```

dslib Routines Description

Table 5-1 and the following section describe the *dslib* routines. You can use these routines to set the values of a *dsreq* type structure and make a request of the SCSI bus that uses the information contained in it.

Table 5-1 dslib Routines

Routine	Description
dsopen	Allocate a *dsreq* type structure and open a device.
dsclose	Free the *dsreq* structure for the SCSI device and close the device.
doscsireq	Send a command to the SCSI device or to make another request.
filldsreq	Set members of a *dsreq* type structure.
fillg0cmd	Set up the *dsreq* structure to send a **group 0 SCSI** command.
fillg1cmd	Set up the *dsreq* structure to send a **group 1 SCSI** command.
inquiry12	Issue an inquiry command and retrieve information from the device concerning such things as its type.
modeselect15	Issue a **group 0 mode select** command to a SCSI device.
modesense1a	Send a **group 0 “mode sense”** command to a SCSI device to retrieve the page information from the device.
readcapacity25	Issue a **read capacity** command to a SCSI device.
readextended28	Issue a **read extended** command to a SCSI device.
requestsense03	Issue a **request sense** command and test or probe for the device.

Table 5-1 (continued) dslib Routines

Routine	Description
senddiagnostic1d	Issue a **send diagnostic** command and test whether the device or the SCSI bus is online or offline, or run a self-test on the device.
testunitready00	Issue a **test unit ready** command to the SCSI device.
vtostr	Return a pointer to a string describing a table entry value.
write0a	Issue a **group 0 write** command to the SCSI device.
writeextended2a	Issue a **write extended** command to the SCSI device.

Note: Many of the following routines take the parameter *vu*. This parameter is not yet implemented. Consequently, its value must always be zero.

SCSI Open and Close Driver Routines

dsopen – Allocate a dsreq Type Structure and Open a Device

The **dsopen**() routine is used to allocate a *dsreq* type structure for a device on the SCSI bus and to “open” that device.

Synopsis

```
struct dsreq* dsopen(char *opath, int oflags);
```

Arguments

opath	Expects the name of the special file for the device on the SCSI bus you want to open. The system comes with up to 15 device special files per adapter in the directory */dev/scsi*. These special files all assume that the logical unit for your device is 0. If the logical unit number for your device is not 0, you must create a device-special file for it. See “Creating Device-special Files for User-level SCSI Drivers.”
oflags	Expects the *oflag* value you would normally give to the standard **open**() system call when opening a device.

Returns

The returned value of this function is a pointer to a *dsreq* type structure.

Notes

This structure is the medium of communication between your user-level SCSI driver and the device on the SCSI bus, and almost every library routine expects the pointer this function returns as its first argument.

dsclose – Free the dsreq Structure and Close the Device

The **dsclose**() routine is used to free the *dsreq* structure for the SCSI device and close the device, when your driver is done with the device.

Synopsis

```
dsclose(struct dsreq *dsp);
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). Upon successful completion, **dsopen**() returns a pointer to a *dsreq* type structure as its function value.

SCSI Function-Building Routines – Group 1

The next five routines, **doscsireq**(), **filldsreq**(), **fillg0cmd**(), **fillg1cmd**(), **vtostr**() are utility routines used to construct your own SCSI functions. If the provided SCSI routines are sufficient, you will not need these routines.

doscsireq – Send a Command to the SCSI Device

The **doscsireq**() routine is used to send a command to the SCSI device or to make some other request of the SCSI bus. All data structures must have been set up before you can use this routine.

Synopsis

```
doscsireq(int fd, struct dsreq *dsp);
```

Arguments

fd	Expects the file descriptor for the special file opened by **dsopen**(). This file descriptor is stored in the *context* type structure pointed to by the *ds_private* member of the *dsreq* type structure allocated by the call to **dsopen**(). Use the **getfd**(*dsp*) macro to get *fd*.
dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). You control the behavior of **doscsireq**() by how you set the bits of the value stored in the *ds_flags* member of the *dsreq* type structure pointed to by this parameter. For more information, see the description given in “dsreq – User-level Driver Communication Structure.”

filldsreq – set Members of a dsreq Type Structure

The **filldsreq**() routine is used to set the *ds_flags*, *ds_databuf*, and *ds_datalen* members of a *dsreq* type structure.

Synopsis

```
filldsreq(struct dsreq *dsp, uchar_t data,
          long datalen, long flags)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.
data	A pointer to the start of a data buffer. The value of this parameter is written to the *ds_databuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.
datalen	The length of the data pointed to by the *data* parameter. The value of this parameter is written to the *ds_datalen* member of the *dsreq* type structure pointed to by the *dsp* parameter.
flags	The value to which you want to set the *ds_flags* member of the *dsreq* type structure pointed to by the *dsp* parameter. See the description of the *ds_flags* member given in “dsreq – User-level Driver Communication Structure.”

fillg0cmd – Set Up the dsreq Structure

The **fillg0cmd**() routine is used to set up the *dsreq* structure to send a **group 0 (6-byte) SCSI** command to the SCSI device. To actually send the command, you must call the routine **doscsireq**().

Synopsis

fillg0cmd(struct dsreq **dsp*, uchar_t **cmdbuf*, *b0*, ..., *b5*)

Arguments

dsp
A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.

cmdbuf
A pointer to a SCSI command. The value of this parameter (the pointer) is written to the *ds_cmdbuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

b0, b1, b2, b3, b4, b5
The values of the individual bytes of the **group 0 SCSI** command to be written to the string pointed to by *cmdbuf*.

fillg1cmd – Send a Group 1 SCSI Command

The **fillg1cmd**() routine is used to set up the *dsreq* structure to send a **group 1 (10-byte) SCSI** command to the SCSI device. To actually send the command, you must call the routine **doscsireq**().

Synopsis

fillg1cmd(struct dsreq **dsp*, uchar_t **cmdbuf*, *b0*, ..., *b9*)

Arguments

dsp
A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**(). The following parameters are then used to set the values of some of the members of this structure.

cmdbuf
A pointer to a SCSI command. The value of this parameter (the pointer) is written to the *ds_cmdbuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.

b0, b1, b2, b3, b4, b5, b6, b7, b8, b9	The values of the individual bytes of the ten-byte **group 1 SCSI** command that you want to write to the string pointed to by *cmdbuf*.

vtostr – Return a Pointer to a String Describing Table Entry

The **vtostr**() routine is used to look up a value in a table and return a pointer to a string describing the table entry for that value. It is normally used to print debugging information, such as when the global variable *dsdebug* is set to a nonzero value.

Synopsis

```
vtostr(long value, struct vtab *table);
```

Arguments

value	Expects the value you want to look up in the table named by the table parameter.
table	A pointer to the name of the table in which you want to look up the value specified as the *value* parameter. You have a choice of four tables are provided with the library: *dsrqnametab* – describes the *DSRQ_** flags. *dsrtnametab* – describes the *DSRT_** flags. *cmdstatustab* – describes the values returned in the *ds_status* member of the *dsreq* type structure. *cmdnametab* – describes the values used for the SCSI commands.

Note: The tables are provided in source form in the same directory as *dslib.c* in the *dstab.c* file.

SCSI Function-Building Routines – Group 2

The next ten routines implement the most frequently used SCSI commands, other than **read**. There are so many variations on **read** that a generic version of these commands would be too clumsy, so you will have to write your own. **Read** and **write** extended are more standard, and so are provided.

inquiry12 – Issue an Inquiry Command

The **inquiry12**() routine is used to issue an **inquiry** command to a SCSI device and retrieve information from the device concerning such things as its type. Much of this is device-specific information. See vendor your documentation for more details.

Synopsis

```
inquiry12(struct dsreq *dsp, caddr_t data,
          long datalen, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes the inquiry information to the buffer pointed to by this parameter. Internally, the value of this parameter is written to the *ds_databuf* member of the *dsreq* type structure pointed to by the *dsp* parameter.
datalen	The size of the buffer pointed to by the *data* parameter. Internally, the value of this parameter is written to the *ds_datalen* member of the *dsreq* type structure pointed to by the *dsp* parameter. Typically, the length must be at least 36 bytes, although 64 bytes is a more normal value for *datalen*.
vu	Not implemented.

modeselect15 – Issue a Group 0 “Mode Select” Command

The **modeselect15**() routine is used to issue a **group 0 “mode select”** command to a SCSI device. This is similar in concept to the UNIX **ioctl**() system call. It is used to control a large number of standard and vendor-specific device parameters. Typically, **modesense1A**() is used first to retrieve the current parameters; the page number(s) of interest and the length are passed in the first few bytes of the data.

Synopsis

```
modeselect15(struct dsreq *dsp, caddr_t data,
             long datalen, int save, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer containing the mode select data.
datalen	The length of the buffer in data.
save	A value that indicates whether you want the device to save the “page” information. The possible values are: 0 = do not save saveable pages. 1 = save saveable pages.
vu	Not implemented.

modesense1a – Send a Group 0 “Mode Sense” Command

The **modesense1a**() routine is used to send a **group 0 “mode sense”** command to a SCSI device to retrieve the page information from the device.

Synopsis

```
modesense1a(struct dsreq *dsp, caddr_t data, long datalen,
            char pgctrl, char pgcode, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes the “page” information to the buffer pointed to by this parameter.
datalen	The size of the buffer pointed to by the *data* parameter.
pgctrl	Expects one of four values that indicate what sort of information you want to retrieve from the page:

	0 = current values 1 = changeable values 2 = default values 3 = saved values
pgcode	Expects the value that indicates the "page" you want to see. There can be up to 0x3F pages. To return all pages, use 0x3F as the value of this parameter. The information on these pages varies from vendor to vendor. For more information, see the vendor-supplied documentation for the device.
vu	Not implemented.

readcapacity25 – Issue a Read Capacity Command

The **readcapacity25**() routine is used to issue a **read capacity** command to a SCSI device.

Synopsis

```
readcapacity25(struct dsreq *dsp, caddr_t data,
               long datalen, long lba, char pmi, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes the capacity information to the buffer pointed to by this parameter.
datalen	The size of the buffer pointed to by the *data* parameter.
lba	Unused if *pmi* is 0. Otherwise, it expects the logical block value of the track for which you want capacity information.
pmi	The value that tells the routine whether you want the capacity for the entire unit or for the current track (as specified by the logical block named in the *lba* parameter). When you ask for track information, the logical block returned (in the buffer pointed to by *data*) is the most distant logical block you can access without undue delay. The valid values for this parameter are:

	0 = return last logical block in unit. 1 = return last logical block in track.
vu	Not implemented.

readextended28 – Issue a Read Extended Command

The **readextended28**() routine is used to issue a **read extended** command to a SCSI device. This command has enough variations that it is quite possible you will need a custom version of it for your device. Do not preempt the function name.

Synopsis

```
readextended28(struct dsreq *dsp, caddr_t data,
                long datalen, long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes information from the device to the buffer pointed to by this parameter.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	Expects the logical block from which you want to read.
vu	Not implemented.

requestsense03 – Issue a Request Sense Command

The **requestsense03**() routine is used to issue a **request sense** command to a SCSI device and test or “probe” for the device. If you set *DSRQ_SENSE* in the *doscsireq* flag argument, as the included library routines do, you don’t need to use this routine.

Synopsis

```
requestsense03(struct dsreq *dsp, caddr_t data,
               long datalen, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to a buffer. Upon successful completion, this command writes the "sense" information to the buffer pointed to by this parameter.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
vu	Not implemented.

senddiagnostic1d – Issue a Send Diagnostic Command

The **senddiagnostic1d**() routine is used to issue a **send diagnostic** command to a SCSI device and test whether the device is functioning correctly. Upon completion of a self-test run by the device, the *ds_status* member of this *dsreq* type structure usually describes the results. (See the description given for this member in "dsreq – User-level Driver Communication Structure.") If you request that a self-test hold the results, you must issue a **read diagnostic** command to get the results. The *dslib* does not contain a routine to issue a **read diagnostic**, but you can use **fillg0cmd**() to create one.

Synopsis

```
senddiagnostic1d(struct dsreq *dsp, caddr_t data, long dlen,
                   long self, long dofl, long uofl, chap vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	Not used.
datalen	Not used.
self	A value that indicates whether you want the device to run a self test that holds the results or a self test that reports the results in *ds_status*. This parameter has two valid values: 0 = run a self test, hold the results 1 = run a self test, report through *ds_status*

dofl	A value that indicates whether or not you want to test if the SCSI bus is online or offline: 0 = test if bus is on-line 1 = test if bus is off-line
uofl	A value that indicates whether or not you want to test if the device is on-line or off-line: 0 = test if device is on-line 1 = test if device is off-line
vu	Not implemented.

testunitready00 – Issue a Test Unit Ready Command to a SCSI Device

The **testunitready00**() routine is used to issue a **test unit ready** command to a SCSI device.

Synopsis

```
testunitready00(struct dsreq *dsp);
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocate for the SCSI device through a call to **dsopen**(). Upon completion of the test, the *ds_status* member of this *dsreq* type structure describes the results. See the description given for this member in “dsreq – User-level Driver Communication Structure” on page 136.

write0a – Issue a Group 0 Write Command

The **write0a**() routine is used to issue a **group 0 write** command to a SCSI device. As with **readextended**(), this routine tends to be device-specific, so it is quite possible you will need to make your own custom version.

Synopsis

```
write0a(struct dsreq *dsp, caddr_t data, long datalen,
         long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer you want to write to the device.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	Expects the logical block to which you want to write. (For some devices, this may actually be a byte value.)
vu	Not implemented.

writeextended2a – Issue a Write Extended Command

The **writeextended2a**() routine is used to issue a **write extended** command to a SCSI device. As with **readextended**(), this routine tends to be device-specific, so it is quite possible you will need to make your own custom version.

Synopsis

```
writeextended2a(struct dsreq *dsp, caddr_t data,
                  long datalen, long lba, char vu)
```

Arguments

dsp	A pointer to the *dsreq* type structure that you allocated for the SCSI device through a call to **dsopen**().
data	A pointer to the buffer you want to write to the device.
datalen	A value that specifies the size of the buffer pointed to by the *data* parameter.
lba	The logical block to which you want to write.
vu	Not implemented.

Using the dslib Routines

To use the *dslib* library routines, include the header file *dslib.h* in your code and compile the code using the compiler option *-lds*. To open and close the device, your program must use the *dslib* routines **dsopen**() and **dsclose**(), rather than the **open**() and **close**() system calls. These routines set up data structures for the other library routines.

Note: This means that only specially compiled programs can use these devices; most standard programs will not be able to use them.

The reason for this departure from IRIX is that *dslib* is actually a library of routines and macros that provide an interface to the device driver for the SCSI bus. By controlling the SCSI bus, you can control any device on that bus, providing the SCSI-bus device driver contains support for it. Such a SCSI-bus device driver is available on Silicon Graphics systems. However, using the **ioctl**() of the SCSI bus device driver directly can be complex. The routines of *dslib* provide access to **ioctl**() for the SCSI bus in a safe and relatively straightforward way.

For more on the *dslib* routines, see the online man pages **dslib**(3X) and **ds**(7M). The source for *dslib* is included in *4Dgifts.sw.giftsfull* in 5.0 in the directory */usr/people/4Dgifts/examples/devices/devscsi*.

Opening a SCSI Device

To open a device on the SCSI bus, a user-level program calls **dsopen**(), which calls the **open**() of the device driver for the SCSI bus and allocates data structures. If the open succeeds, the kernel allocates a SCSI subchannel for the device on the bus that you want to control. If this succeeds, a *dsreq* type structure is allocated, which in turn sets the values of some of its members and returns a pointer to this structure as its function value. This allocated and primed *dsreq* type structure is the medium of communication between your user program and the SCSI device.

Sending Commands to a SCSI Device

To send commands to the device on the SCSI bus, your program can modify the members of the *dsreq* type structure and call the *dslib* routine **doscsireq**(). This command uses an **ioctl**() call to the SCSI bus interface driver to set the members of the *scsisubchan* type structure for the device according to the information in the *dsreq* structure.

To simplify sending commands to a SCSI device, *dslib* provides routines and macros that set members of the *dsreq* type structure and call **doscsireq**(). For example, *dslib* provides the routine **write0a**() to give you an easy way to issue a simple group 0 "**write**" command. However, since few truly general SCSI commands exist, the *dslib* provides functions such as **fillg0cmd**() to give you a relatively painless way to send vendor-specific commands to a device on the SCSI bus. You can (and are expected) to extend these as needed, since the library source is included in the **4Dgifts** subsystem.

Closing a SCSI Device

When your user-level program is done with the SCSI device, it must call **dsclose**() to close the device. This involves freeing the *dsreq* type structure and kernel data structures, among other things.

Kernel-level SCSI Device Drivers

This section shows how to write a kernel-level driver for a SCSI device. Starting with IRIX 5.x, all supported SCSI controllers use the same interface.

Configuring a Kernel-level SCSI Driver

Chapter 2, "Writing a Device Driver," gives a detailed description of how to configure a kernel to include a new driver. This section presents only the material that is unique to SCSI drivers. Recall that you must:

1. Create the object code for the driver you want to include in the kernel.
2. Move that object code to the directory */usr/sysgen/boot*.
3. Edit the *system* file. Use a directive telling **lboot**, the configuration utility, how to include your driver and specify which memory space your device will allocate. Install it in the */var/sysgen/system* directory using the driver name appended by *.sm*.
4. Create a *master* file in the directory */usr/sysgen/master.d*.

5. Create a new kernel. (To create a debuggable kernel, see “Making a Debuggable Kernel” in Chapter 10, “Driver Installation and Testing.”)

To edit the *system* file (*/usr/sysgen/system*) to include a SCSI driver, use the *INCLUDE* directive to tell **lboot** to unconditionally add the named SCSI module into the new kernel.

To create a *master* file, create an ASCII file, enter the appropriate information (described below), and move the file to the */usr/sysgen/master.d* directory. The name of the master file must correspond to the name of the file containing the object code for the driver.

Ensure that the *FLAG* field of the master file includes at least the character device flag *c* and the software driver flag *s*. You must flag all SCSI device drivers as software drivers (drivers that do not control actual hardware) because **lboot** cannot probe for SCSI devices.[1] If **lboot** tries to probe for a SCSI device, it fails, then assumes that the device is not present, and does not include your SCSI device driver in the kernel.

For example, assume that you want your kernel to include a device driver for a SCSI device that you call *sdk*. Create the object code for the device driver, and move the *sdk.o* object file to the directory */usr/sysgen/boot*. After examining */usr/include/sys/major.h*, you determine that major device number 61 is available and can be used for the device, *sdk*. Create a file *sdk.sm* with the following line, and also add it to the *system* file:

```
INCLUDE: sdk
```

You then create a master ASCII file called */usr/sysgen/master.d/sdk* and enter:

```
*
* sdk
*
*FLAG   PREFIX   SOFT   #DEV   DEPENDENCIES
 sc     sdk_     61       -     scsi
```

[1] Although **lboot** does probe for the SCSI controller, the target devices that the SCSI controller manages cannot be probed by a memory reference / access.

Under “DEPENDENCIES,” you must list “scsi.” This indicates that the SCSI interface driver must be present. The SCSI interface is described later in this chapter.

Writing a SCSI init()

Because you use the *INCLUDE* directive to include a module into the kernel, your driver object module must contain a routine of the form *drv***init**(), where *drv* is the prefix for the device that you specified in the *master* file. If the driver module includes *drv***init**(), the system calls *drv***init**() at system boot time. Your driver can use *drv***init**() for boot-time device or driver initialization. To initialize the device *sdk* every time the system boots, you can include an **sdkinit**() routine in the driver object module *sdk.o*.

The *drv***init**() routine has no parameters:

```
sdk_init( )
{
your code
};
```

SCSI Device Interface Overview

The SCSI host adapter communicates with devices, known as *targets*, on the SCSI bus. Each SCSI target can control a number of actual devices, known as *logical units*. Most SCSI targets, however, control a single device.

There are two types of SCSI drivers: device level drivers and host adapter drivers. The host adapter drivers handle the low-level communication over the SCSI interface, such as programming the SCSI interface chip or board, negotiating synchronous or wide mode, and handling disconnect/reconnect. The device drivers handle high-level communication, primarily by issuing SCSI commands and interpreting sense data.

Some examples of host adapter drivers are *wd93*, *wd95*, and *jag*. Examples of device level drivers are *dksc*, *tpsc*, and *smfd*.

There are four basic interfaces used to communicate to a host adapter driver: **scsi_info**, **scsi_alloc**, **scsi_free**, and **scsi_command**(). Each of these interfaces is implemented as an array of pointers to functions, indexed by a host adapter driver number. The host adapter driver number is determined from the adapter number. The adapter number ranges are defined in *sys/ scsi.h*, and are different on different architectures. In general, Integral SCSI controllers start at adapter 0, and Interphase VME-SCSI (Jaguar) controllers start after the last Integral adapter. Each SCSI bus is considered to be one adapter.

On a POWER Series or Crimson system, for example, Interphase VME-SCSI controller 3, bus 0, would be adapter number 10. On a CHALLENGE / Onyx system, VME-SCSI controller 3, bus 0, would be adapter number 134. See the definitions of *SCSI_SGISTART*, *SCSI_SGICOUNT*, *SCSI_JAGSTART*, and *SCSI_JAGCOUNT* in *sys/scsi.h*.

Determining Driver Number

To determine the correct number for your driver, see *scsi_driver_table* in *sys/scsi.h*. *scsi_driver_table* is an array, indexed by adapter number, that returns the adapter type, which is a constant defined by the *SCSIDRIVER_XXX* definitions. *scsi_driver_table* may be sparsely populated to accommodate possible options. This avoids the reassignment of major and minor device numbers in cases where hardware is added at a later time.

SCSI_XXXSTART definitions define the starting adapter number for the various adapter types. *SCSI_XXXCOUNT* is the number of possible adapters for a given controller type. Both *SCSIDRIVER_WD93* and *SCSIDRIVER_WD95* are part of Silicon Graphics-built SCSI controllers, and so use the *SCSI_SGISTART* and *SCSI_SGICOUNT #define*'s.

For example, on a CHALLENGE system, the second bus on Interphase VME-SCSI controller 4 would be considered adapter number 137. Counting *SCSI_JAGSTART* as 128, (controller 4 * 2 adapters per controller) + bus 1 (the second bus on the controller) gives 9 to be added to the value of *SCSI_JAGSTART* to give adapter (or bus) number 137 from the point of view of a SCSI device driver.

Typically, the major and minor numbers of a device are used to determine the adapter number, which can then be used to index into *scsi_driver_table*.

```
adap = sdk_adap_num(device);
```

```
driver_num = scsi_driver_table[adap];
```

scsi_info – Getting Information About a Device

Before your driver tries to access a device, it must call the host adapter **scsi_info** function. The host adapter driver then issues an **Inquiry** command to the given device and returns a pointer to a *struct scsi_target_info*. If the **Inquiry** is not successful — or if the *adapter, target*, or *lun* is invalid — the return value is *NULL*. The SCSI device driver must examine the inquiry data to determine whether it is the appropriate driver for the device.

Synopsis

```
struct scsi_target_info * (*scsi_info[])
                    (u_char adapter, u_char target, u_char lun);
```

Arguments

adapter	The adapter number.
target	The target number.
lun	The logical unit number.

Example

The following would be the way to use *scsi_info* to get information about target 5, LUN 0 on Interphase controller 4, bus 1 (the second bus):

```
info = (*scsi_info[SCSIDRIVER_JAG])(9, 5, 0);
```

scsi_alloc – Initializing a Connection

Before your driver can issue a command to a SCSI device through **scsi_command**, it must have the low-level driver initialize the connection. The interface to this low-level driver is through the **scsi_alloc**(), typically in the *drv***open**() routine.

Synopsis

```
int (*scsi_alloc[driver_num])
    (unsigned char adap, unsigned char target,
     unsigned char lun, int option, void (*callback)());
```

Arguments

adap The number of the SCSI adapter for the device you want to control. All IRIX systems support at least one adapter (0). Systems with POWERchannel have two or four built-in adapters and up to 16 VME-SCSI adapters (two per controller). CHALLENGE / Onyx systems can have up to 120 different built-in SCSI adapters (although one system can have only a fraction of those installed at one time) and 16 VME-SCSI adapters; Indigo^2 systems have two adapters.

target The target ID of the SCSI controller for the device your driver wants to control. This must be a value from 0 to 15, but cannot be the ID of the controller itself (typically 0 for built-in and 7 for the Jaguar). Not all host adapter implementations support SCSI device numbers 8 through 15. (Note that for this purpose, the Jaguar VME_SCSI controller has two SCSI host adapters, so only devices 0-7 are valid. Choose the bus number with *adap*.) Usually, the device is configured to a fixed ID by switch settings or jumpers. Refer to the device technical specification for details. If the system contains more than one of a particular device, the device driver normally uses the minor device number to distinguish between devices.

lun The logical unit number for the device your driver wants to control. This is often 0 because most SCSI targets support only a single logical unit.

option Two options are available to **scsi_alloc**:

SCSIALLOC_EXCLUSIVE specifies that the device driver wants to have exclusive access to the target in question. If any other device has allocated a connection, the **scsi_alloc** call fails.

SCSIALLOC_QDEPTH, the queue depth that your driver requests, is the bottom eight bits of *option*. It is considered advice only, and may or may not be followed.

callback Gets a pointer to a function that the host adapter driver calls every time it has sense data. Only one driver at a time may have a callback allocated, so, in this respect, it functions like *SCSIALLOC_EXCLUSIVE* in the option argument above. If the callback argument is not *NULL*, the host adapter driver calls it with a pointer to the sense data (an array of char) every time there is any. This allows multiple drivers to access a device while still allowing one of them to keep apprised of all sense data.

Returns

If the call to **scsi_alloc** is successful, the type of SCSI adapter being used (*SCSIDRIVER_WD93*, *SCSIDRIVER_WD95*, or *SCSIDRIVER_JAG*) is returned. Otherwise, this routine returns 0, which indicates that no connection could be established between the host adapter driver and your driver, probably because the arguments were included for the adapter.

scsi_command – Executing a SCSI Command

Once your driver has established a connection to a host adapter driver with a successful call to **scsi_alloc**[](), your driver can send SCSI commands to the device. To send SCSI commands to a device, fill out a *scsi_request* structure and then call **scsi_command**[](), passing it the address of this structure. The system uses this structure to report back on the status of the SCSI command.

Not all *scsi_request* type structure members are of interest to your device driver (some members are of interest only to the SCSI host adapter driver and may change across releases of the operating system). Following is the definition of the structure *scsi_request*:

```
struct scsi_request
{
        /* values filled in by device driver */
        u_char   sr_ctlr;
        u_char   sr_target;
        u_char   sr_lun;
        u_char   sr_tag;     /* first byte of tag message */

        u_char  *sr_command; /* scsi command */
        ushort   sr_cmdlen;  /* length of scsi command */
        ushort   sr_flags;   /* direction of data transfer */
        ulong    sr_timeout; /* in seconds */

        u_char  *sr_buffer;  /* location of data */
        uint     sr_buflen;  /* amount of data to transfer */
        u_char  *sr_sense;   /* where to put sense data in
                             /*case of CC */
        ulong    sr_senselen;/* size of buffer allocated for
                             /*sense data */
        void    (*sr_notify)(struct scsi_request *);
                             /* callback pointer */
        void    *sr_bp;      /* usually a buf_t pointer */

        /* spare pointer used by device driver */
        void    *sr_dev;

        /* spare fields used by host adapter driver */
        void    *sr_ha;      /* usually used for linked list
                             /*of req's */
        void    *sr_spare;   /* used as spare pointer, int,
                             /*etc. */

        /* results filled in by host adapter driver */
        uint     sr_status;      /* Status of command */
        u_char   sr_scsi_status; /* SCSI status byte */
        u_char   sr_ha_flags;    /* flags used by host
                                 /*adapter driver */
        short    sr_sensegotten; /* number of sense bytes
                                 /* received; -1 == err */
        uint     sr_resid;       /* amount of sr_buflen not
                                 /*transferred */
};
typedef struct scsi_request       scsi_request_t;
```

Before calling **scsi_command**[](), your driver must fill in the values indicated above. Some are optional.

sr_ctlr	Number of the adapter where the command will be transferred.
sr_target	ID of the target where the command will be transferred.
sr_lun	Number of the logical unit where the command will be transferred.
sr_tag	Type of queue tag to use (see SCSI-2 specification). Not all tag types are supported by all drivers.
sr_command	Give this member a pointer to the SCSI command descriptor block that you want to send to the device.

sr_cmdlen	The length (in bytes) of the command to which the *sr_command* member points. You can use three symbolic constants for this member (the constants are defined in *sys/scsi.h*), but other values are permitted for vendor unique commands, too: *SC_CLASS0_SZ* is a command size of 6 bytes. *SC_CLASS1_SZ* is a command size of 10 bytes. *SC_CLASS2_SZ* is a command size of 12 bytes.
sr_flags	This member must set zero or more options as specified below: *SRF_DIR_IN* — Data associated with the command will be written into memory. *SRF_FLUSH* — A cache operation is required on the data. *SRF_MAP* — The *sr_buffer* address needs to be mapped **--** it is a kernel virtual address. *SRF_MAPBP* — *sr_bp* has a pointer to a *struct buf* that is not mapped (*BP_ISMAPPED* returns false). *SRF_AEN_ACK* — This request acknowledges an error in a previous request. Once a **scsi_request** is returned with an error, all further requests are returned with *SC_ATTN*, without being acted upon, until *SRF_AEN_ACK* is set. *SRF_NEG_ASYNC* — This request attempts to negotiate *async xfer* mode on this command (if currently using *sync xfers*). *SRF_NEG_SYNC* — This request attempts to negotiate *sync xfer* mode on this command if currently asynchronous. It may be ignored by some adapter drivers, either always or if the driver has previously failed to negotiate *sync xfer* mode with this target. This overrides the *SCSIALLOC_NOSYNC* flag to **scsi_alloc** (if it has been specified).
sr_timeout	The maximum amount of time, in Hz, that a command can take.

sr_buffer	A pointer to the start of the data associated with a command. If there is no data (as with a *test_unit_ready* for instance), *sr_buffer* must be NULL.
sr_buflen	The maximum amount of data that can be transferred. This is not necessarily the amount that *will* be transferred, but is an upper limit.
sr_sense	A pointer to a buffer where request sense data can be copied in case a command gets a check condition status.
sr_senselen	The amount of space reserved for request sense data.
sr_notify	A pointer to a notification routine. The notification routine gets called when a command is complete or gets an error. This value must be set (*NULL* is not a valid value).
sr_bp	This field must point to the *struct buf* corresponding that generated the *scsi_request* if the *SRF_MAPBP* flag is set in *sr_flags*. If the *SRF_MAPBP* flag is not set, then *sr_bp* is not used by the host adapter driver.
sr_dev	This field is never used by a host adapter driver and is reserved for the use of your device driver.

After control returns from the host adapter driver to your driver, your driver must check the following members of the *scsi_request* type structure. These members are:

sr_status	This member reports the overall status of the SCSI command (this is the host adapter driver status; SCSI bus status is in *sr_scsi_status*). *sr_status* can report one of these values: *SC_GOOD* — indicates no error. The bus successfully processed the SCSI command; however, the command itself may still have failed (see *sr_scsi_status*). *SC_TIMEOUT* — indicates selection timeout. The device did not respond to selection within 250 milliseconds. *SC_HARDERR* — indicates hardware or SCSI device error, usually a SCSI bus reset, possibly caused by a bad phase or time-out on some other device.

SC_PARITY — indicates an unrecoverable parity error on the SCSI bus.

SC_MEMERR — indicates an unrecoverable parity or ECC error from host memory.

SC_CMDTIME — indicates the command did not complete before the time-out specified in *s_timeoutval* elapsed.

SC_ALIGN — indicates the buffer address did not meet the alignment requirements of the system. Most Silicon Graphics systems require starting buffer addresses to be on a four-byte boundary.

SC_ATTN — indicates the **scsi_request**() has been aborted by a SCSI bus reset, the device, or the driver, due to an error in another request.

SC_REQUEST — indicates the request is not a valid request. This can be because no **scsi_alloc** has been done successfully or because the **scsi_request**() has conflicting or illegal values, such as *sr_notify* being set to *NULL*.

sr_scsi_status The SCSI status byte sent by the target. *sr_scsi_status* can report any one of these values: The values of *sr_sci_status* correspond to those described in the SCSI specification:

ST_GOOD — indicates the target has successfully completed the SCSI command. On this value, *sr_sensegotten* must be checked to see if a check condition occurred on the command.

ST_CHECK — indicates that the request sense command generated by the host adapter driver in response to a check condition also got a check condition. This is a special case. If a device reports a “check condition” status, the host adapter driver automatically issues a *request sense* command and reports the status of that command in this byte. If *sr_sense* is set and *sr_senselen* is > 0, *sr_sensegotten* is set to the number of bytes of sense data received, or to -1 if an error occurs during the request sense command.

ST_COND_MET — indicates search condition satisfied.

ST_BUSY — indicates the target is busy. Normally, the driver should delay, then reissue the command.

	ST_INT_GOOD — indicates this status is reported for every command in a series of linked commands. Although they are not supported, these linked commands may sometimes work.
	ST_RES_CONF — indicates an attempt to access a logical unit or an extent within a logical unit that reserves that type of access to another SCSI device.
	This byte corresponds to the SCSI command status byte as documented in the SCSI specifications.
sr_sensegotten	The number of bytes of sense data gotten as a result of a request sense command issued in response to a check condition status from this command, if any, or -1 if a check condition occurs, but the request sense command fails.
sr_resid	The difference between *sr_buflen* and the number of bytes actually transferred.

scsi_free – Freeing the Connection

When your driver is done with a SCSI device, it must call through the array **scsi_free** to indicate to the host adapter driver that your driver no longer needs to use the device. Typically, you do this in your *drv***close**() routine.

Synopsis

```
int (*scsi_free[driver_num])
    (unsigned char adap, unsigned char target,
     unsigned char lun, void (*callback)());
```

The arguments to **scsi_free** are similar to those to **scsi_alloc** except that no option argument is used.

Using the SCSI Access Routines

"SCSI Device Interface Overview" on page 159 describes the four routines that make up the device driver interface to host adapter SCSI-bus drivers. This section describes how you can use these routines in your driver.

Your *drv***open**() routine should first use **scsi_info** to determine whether a device is present on the SCSI bus and to examine the inquiry information to see what type of device it is. Remember, SCSI target 3 may be a disk on one system but a tape drive on another. So be sure to check the inquiry information to determine whether your driver is appropriate to the device in question.

After your driver has determined that it is appropriate to talk to a device, it must then use **scsi_alloc** to initialize a connection between your device driver and the host adapter driver.

Your driver can free the connection by using **scsi_free**. Remember that if your driver uses the exclusive option, no other SCSI device driver can communicate with a target until your driver calls **scsi_free**. Similarly, if your driver uses a sense callback, no other driver can use a sense callback until your driver calls **scsi_free**, although other drivers that do not need a sense callback may still communicate with the target.

Because the IRIX SCSI interface transfers data using direct memory access (DMA) only, your driver must use the kernel routine **physio**() and a *drv***strategy**() routine for SCSI I/O to transfer data to pages that a user process is using. Internally generated requests that transfer data into kernel memory (either statically allocated at compile time in the driver or allocated through **kmem_alloc**() or similar functions) do not need to use **physio**(). However, in this case your driver must do its own cache flushing.

When filling out a *scsi_request* structure in preparation for a call to **scsi_command**, be sure to fill out the *sr_ctlr*, *sr_target*, *sr_lun*, and *sr_tag* fields with appropriate values. The *sr_command* field must point to an array of characters filled out with the SCSI command byte values, and *sr_cmdlen* must be the length of the command. *sr_flags* is used to specify the direction of data transfer, if any, whether cache flushes need to be done, and what kind of mapping needs to be done. If the data transfer involves direct mapped K0SEG or K1SEG memory, then no mapping need be specified, although the host adapter may still perform some mapping. The flags must also acknowledge an error by setting the *SRF_AEN_ACK* bit on the next command or the command will fail immediately.

sr_timeout must be the command timeout in clock ticks (defined by Hz), typically 1/100 of a second.[1] *sr_buffer* must be a pointer to the start of the buffer to be transferred to, unless there is no data transfer or the

SRF_MAPBP flag in *sr_flags* is set, in which case *sr_buffer* can be *NULL*. *sr_buflen* must be an amount equal to the maximum amount of data to be transferred in the command.

If your driver wants to examine sense data from a request sense caused by a check condition that occurs as a result of the command you issued, it can set the *sr_sense* field to point to such a buffer, with *sr_senselen* set to the size of the buffer. The *sr_notify* must point to your driver "interrupt" routine. This is not a real interrupt routine—it is called indirectly as a result of an interrupt from the interrupt handler—so it can have any name you desire. This routine is called when the host adapter driver has completed processing your command.

If the *SRF_MAPBP* flag is set, *sr_bp* must point to the buf structure that generated the SCSI request. Sometimes your driver strategy function will get a buf that is not mapped into kernel virtual memory. In this case, **BP_ISMAPPED**(*bp*), where *bp* is a pointer to the buf, will return false. If *SRF_MAPBP* is not used, *sr_bp* is free to be used by your driver for other purposes. *sr_dev* is always a spare field usable by your driver, often for linking lists of active or queued requests or to keep special information.

After your **notify** function is called, your driver must check the *sr_status*, *sr_scsi_status*, and *sr_sensegotten* fields to see whether there were any errors. See */usr/include/sys/scsi.h* for additional information on return values for the *sr_status* field. The *sr_scsi_status* field corresponds to the SCSI status byte that a target sends at the end of every command. However, if a target sends a check condition (status 2), a request sense is issued instead; otherwise *sr_scsi_status* is 0. *sr_sensegotten* is nonzero: -1 if there is an error getting sense data or the amount of sense data actually received. Otherwise, *sr_resid* will be the difference between *sr_buflen* and the amount of data actually transferred.

[1] Check the header file and include the file that defines Hz. Do not hard-code the value of Hz into a driver.

SCSI Device Driver Example

The following example shows how a driver can communicate with a direct access SCSI device, such as a disk. Note the use of **scsi_info**[]() to determine that the device is actually present and of the appropriate type.

This driver is simplified and does not do as much error checking as a real driver would do. Also, this example uses a global SCSI request structure that does not work in real drivers, since multiple reads or writes would overwrite a command in progress.

```
#include "sys/param.h"
#include "sys/types.h"
#include "sys/user.h"
#include "sys/buf.h"
#include "sys/errno.h"
#include "sys/cmn_err.h"
#include "sys/cred.h"
#include "sys/ddi.h"
#include "sys/systm.h"
#include "sys/scsi.h"

int sdk_devflag = 0;

#define ADAPT    0     /* SCSI host adapter */
#define TARGET   7     /* the disk will have target ID #7 */
#define LU       0     /* and logical unit  #0 */
#define TIMEOUT (30*HZ)/* wait 30 secs for SCSI device to
                          respond */
#define DIRECTACCESS 0 /* First byte of inqry cmnd */

unchar scsi_read[]    = {0x28, 0, 0, 0, 0, 0, 0, 0, 0, 0};
unchar scsi_write[]   = {0x2a, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int    sdk_inuse = 0;
int    sdk_driver;
struct scsi_target_info *sdk_info;
struct scsi_request sdk_req;
u_char sdk_sensebuf[SCSI_SENSE_LEN];  /* SCSI_SENSE_LEN
                                         from scsi.h */

/* forward definitions*/
int sdk_strategy(struct buf *bp);
void sdk_notify(struct scsi_request *req);
```

```
/*
 * sdk_open - Open the SCSI device exclusively.
 *
 * Issue a SCSI inquiry command upon device and ensure
 * it is a direct access device.
 */
int
sdk_open(dev_t *devp, int flag, int otyp, cred_t *crp)
{
   if (sdk_inuse)
      return EBUSY;

   /* Get driver number */
   sdk_driver = scsi_driver_table[ADAPT];

   /*
    * Call through scsi_info to get inquiry data and to
    * find out if a device is at the address we want.
    */
   sdk_info = (*scsi_info[sdk_driver])(ADAPT, TARGET, LU);
   if (sdk_info == NULL)
      return ENODEV;

   /*
    * Is it a direct access device?  We could check the
    * entire inquiry buffer to ensure it is actually the
    * correct device.
    */
   if (sdk_info->si_inq[0] != DIRECTACCESS)
      return ENXIO;

   /*
    * It's a direct access device (disk drive).  Initialize
    * the connection to the host adapter driver.
    */
   if ((*scsi_alloc[sdk_driver])
      (ADAPT, TARGET, LU, 1, NULL) == 0)
      return EBUSY;

   /*
    * We have successfully allocated a connection between
    * sdk and the host adapter driver.  Initialize the
    * scsi_request structure, and mark the driver as being
    * in use.
    */
   sdk_inuse = 1;
   bzero(&sdk_req, sizeof(sdk_req));
   sdk_req.sr_ctlr = ADAPT;
   sdk_req.sr_target = TARGET;
   sdk_req.sr_lun = LU;
   sdk_req.sr_timeout = TIMEOUT;
   sdk_req.sr_sense = sdk_sensebuf;
   sdk_req.sr_senselen = sizeof(sdk_sensebuf);
   sdk_req.sr_notify = sdk_notify;

   return 0;
}

/* sdk_close - close the device and free the subchannel. */
int
sdk_close(dev_t dev, int flag, int otyp, cred_t *crp)
{
   (*scsi_free[sdk_driver])(ADAPT, TARGET, LU, NULL);
   sdk_inuse = 0;
   return 0;
}

/*
 * sdk_read - read from the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_read(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_write - write to the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_write(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_strategy - do the dirty work of the I/O.
 * Use either the SCSI read or write command as
 * appropriate.  Modify the block number and block counts
 * within the command buffer. Simply return here;
 * physio( ) will wait for an iodone( ).
 */
int
sdk_strategy(struct buf *bp)
{
   int blkno, blkcount;

   /* Prime the subchannel communication block. */

   blkno = bp->b_blkno;
   blkcount = BTOBB(bp->b_bcount);

   sdk_req.sr_command = bp->b_flags & B_READ ?
                        scsi_read : scsi_write;
   sdk_req.sr_command[2] = (char)(blkno>>24);
   sdk_req.sr_command[3] = (char)(blkno>>16);
   sdk_req.sr_command[4] = (char)(blkno>>8);
   sdk_req.sr_command[5] = (char) blkno;
   sdk_req.sr_command[7] = (char)(blkcount>>8);
   sdk_req.sr_command[8] = (char) blkcount;

   sdk_req.sr_cmdlen = SC_CLASS1_SZ;
   sdk_req.sr_flags = bp->b_flags & B_READ ? SRF_DIR_IN : 0;

   if (BP_ISMAPPED(bp)) {
      sdk_req.sr_buffer = bp->b_dmaaddr;
      sdk_req.sr_buflen = bp->b_bcount;
      sdk_req.sr_flags |= SRF_MAP;
   }
   else {
      sdk_req.sr_buffer = NULL;
      sdk_req.sr_buflen = bp->b_bcount;
      sdk_req.sr_flags = SRF_MAPBP;
   }
   sdk_req.sr_bp = bp;   /* required for SRF_MAPBP, but a
                          * convenience in all cases */

   /* Perform the SCSI operation. */
   (*scsi_command[sdk_driver])(&sdk_req);
}

/*
 * sdk_notify - SCSI command completion notification routine
 *
 * Simply check for errors and wake up physio( ) with
 * an iodone( ) on the buffer.
 * Note that a more robust driver would be more thorough
 * about error handling by retrying errors, giving more
 * information about error types, etc.
 */
```

```
void
sdk_notify(struct scsi_request *req)
{
   register struct buf *bp = req->sr_bp;

   if ((req->sr_status != SC_GOOD) ||
       (req->sr_scsi_status != ST_GOOD) ||
       (req->sr_sensegotten < 0))
   {
      cmn_err(CE_NOTE,
         "sdk: Error: driver stat 0x%x, scsi stat 0x%x"
         " sensegotten %d\n", req->sr_status,
         req->sr_scsi_status, req->sr_sensegotten);
      bioerror(bp, EIO);
         }
   else if (req->sr_sensegotten > 0) {
      cmn_err(CE_NOTE, "sdk: Error: sensekey 0x%x\n",
         sdk_sensebuf[2] & 0x0F);
      bioerror(bp, EIO);
   }
   bp->b_resid = req->sr_resid;
   biodone(bp);
}
```

Chapter 6

Writing Kernel-level GIO Device Drivers

This chapter provides in-depth information about drivers that interface to the GIO (Graphics Input / Output) bus. It describes the system configuration for GIO device drivers and introduces several GIO-specific functions you must include in your device driver. There are several models for performing DMA operations. Which model you choose for your device driver depends on the capability of the device, which may either require a software implementation or have hardware support for scatter / gather. Memory-mapped, user-level drivers for GIO devices are not supported; all GIO drivers must be kernel-level drivers.

This chapter contains the following sections:

- “GIO-bus Architecture” on page 178
- “Determining GIO Device Addresses” on page 179
- “Including GIO Device Drivers in the Kernel” on page 180
- “Writing edtinit()” on page 183
- “GIO-specific Functions” on page 187
- “GIO Interrupt Handler” on page 189
- “Programmed I / O (PIO)” on page 190
- “DMA Operations” on page 192
- “Memory Parity Workarounds” on page 193
- “GIO Devices with Hardware-supported Scatter / Gather Capability” on page 195
- “DMA on GIO Devices Without Scatter / Gather Capability” on page 198
- “Device Driver Example” on page 200

GIO-bus Architecture

The GIO bus is a family of synchronous, multiplexed address-data buses for connecting high-speed devices to main memory and CPU for entry-level Silicon Graphics systems. The GIO bus has three varieties: GIO32, GIO32-bis, and GIO64.

The members of the GIO-bus family are all similar; however, the GIO32 and GIO64 are not compatible. A GIO32 device does not work in a GIO64 slot, but a GIO32-bis device does fit in either a GIO32 or GIO32-bis option slot. It is possible to design a board that functions in systems with either a GIO32 or GIO32-bis bus.

The form factor and bus protocol depend on the specific platform in which the device is installed. GIO32 and GIO32-bis devices can be either single or double-wide (that is, taking one or both board slots), while GIO64 boards are the size of an EISA board. Slots in Indigo^2 systems can accept either an EISA board or a GIO64 board. These two types of boards share common board dimensions but have different connectors for attaching to their respective buses.

The *GIO Bus Specification* contains more detailed information about the various types of GIO buses, from both an electrical and a mechanical point of view.

Determining GIO Device Addresses

Each GIO device has a range of GIO-bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the GIO device.[1] GIO-bus addresses cannot be mapped into user address space. GIO devices can be classified as 32-bit or 64-bit.

The address range for GIO-bus devices is determined by the slot number of the device. The hardware must be designed to determine which slot the device is in and make the appropriate adjustments to respond to that slot's address range.

Indigo, Indigo^2, and Indy systems all have two slots available for GIO devices. However, the address spaces for Indigo^2 are slightly different than for Indigo and Indy.

For Indigo and Indy, the two slots are known as *exp0* and *exp1*. The Indigo^2's slots are known as *gfx*, *exp0*, and *exp1*.

The *gfx* slot is normally used for the graphics card, but it can be used as a regular GIO card slot if the graphics can be moved up into the *exp0* slot. This slot's address space is also available on Challenge M (Indigo^2 with no graphics) systems.

Table 6-1 shows the slot names and addresses available on the Indigo, Indigo^2, and Indy platforms.

Table 6-1 Indigo, Indigo^2, and Indy Slot Names and Addresses

Slot Name	Address	Indigo/Indy	Indigo^2
gfx	0x1f000000–0x1f3fffff	N / A	Available
exp0	0x1f400000–0x1f5fffff	Available	Available
exp1	0x1f600000–0x1f9fffff	Available	N / A

[1] Unlike VME, where the class of device determines the address range, each GIO device responds to the same address range.

GIO-bus devices use only one interrupt level — interrupt 1. Interrupts 0 and 2 are used by the graphics system and may not be used by GIO-bus devices.

Since one interrupt serves multiple GIO devices, the interrupt routine in each driver must be able to deal with the various interrupt situations:

- The interrupt is for the board.
- The interrupt is for some other GIO device.
- There is no interrupt pending.

Including GIO Device Drivers in the Kernel

Chapter 2, “Writing a Device Driver,” provides general information on adding a driver to the kernel. This section describes specifics concerning GIO drivers. To add a new kernel-level GIO device driver, you must:

- Create a system file
- Create a master file
- Create the boot file

For GIO drivers, use the *INCLUDE* directive, which unconditionally adds the module to the kernel. Because **lboot** can probe for GIO devices, **lboot** can conditionally include a GIO device driver into the kernel.

Note: Because IRIX kernels cannot, as a rule, be preempted, any driver that sits in a loop waiting for some condition to be satisfied may tie a processor up for as long as it wants. Real-time processes, such as audio, are very sensitive to such delays.

Creating a System File

This file resides in the */var/sysgen/system* directory and contains the instructions that **lboot** uses to add the software module to the kernel. This normally consists of the *VECTOR* directive; in fact, the *system* file may consist of one or more *VECTOR* directives. The filename must end in *.sm* for **lboot** to recognize and include the software module. Typically, the filename is the software module’s name with the *.sm* suffix, as in *gbd.sm*.

If the current system contains the GIO device, **lboot** includes the driver; otherwise, it saves memory by leaving it out. Use the *VECTOR* directive to include a GIO device conditionally. In addition to the module name, the *VECTOR* directive requires that you fill out these fields:

vector	The interrupt vector value, as described previously. The interrupt vector for GIO devices is set using the **setgiovector**() function (see “setgiovector” on page 187). Therefore, the vector in the *VECTOR* statement must always be 0x0 for GIO devices.
unit	The device number that differentiates between more than one device of the same type. This value is related to VME-style devices. For GIO devices, this value can be anything, but for consistency, make it 0.
base	The device address(es) on the GIO bus, determined by the slot in which the board is installed. This is a K1 address (see the kvtophys(D3X) man page).
base2, base3	Additional addresses passed to driver **edtinit**() routine through the *edt* structure. These are K1 addresses (see the kvtophys(D3X) man page).
exprobe	The address read when **lboot** determines the existence of the device. This address is often the same as the base address. If you do not specify a probe address, the module is automatically included in the kernel. For GIO bus devices, the **exprobe**() call is used in place of the probe call. The fields used for this call are:

operation	read (*r*) or write (*w*)
address	address to probe
# of bytes	number of bytes to read or write
value	expected response value (the GIO ID number)
mask	mask to apply to value

Creating a Master File

The master file resides in the */var/sysgen/master.d* directory. It contains the information that **lboot** uses to create the *device switch table* as well as indicating dependencies with other kernel modules. The name of the master file must be the same as the software module. This file also contains the prefix used in building the driver entry points.

The FLAG field of the master file must include at least the character device flag *c*.

Creating a Boot File

The boot file must reside in the */var/sysgen/boot* directory. This file is the successfully compiled driver object file. The name of the boot file must end with the suffix ".o".[1]

The GBD Example

For example, to add a mythical GIO device driver to the kernel of an R4000 Indigo (IP20), you must copy the driver object file *gbd.o* to */usr/sysgen/boot*, create a master file (as shown below), and create a system file with the following *VECTOR* directive:

```
VECTOR: bustype=GIO module=gbd vector=0x0 unit=0
        base=PHYS_TO_K1(0x1f400000) base2=0xBF410000
        exprobe=(r,PHYS_TO_K1(0x1f400000),4,0x75,0xff)
```

Note that the interrupt vector (*vector*=), the base addresses, and the probe address must all be specified in hexadecimal format. The *base* address and the address in the *exprobe* must agree. In the example above, **lboot** reads four bytes at probe address PHYS_TO_K1(0x1f400000) to determine whether the device is present in slot 0. In this example, *base2* is used to point to the location of on-board memory.

[1] The ".a" suffix is used for archived files.

In actual use, it is advisable to add a second *VECTOR* line to the *system* file, to perform a probe of the other GIO slot. If only the line above had been used and the GIO device were physically placed in slot 1 rather than slot 0 as specified in the *VECTOR* line, the probe would fail, and the driver would not have been included in the kernel. Using this situation as an example, the following line must be added to the *system* file:

```
VECTOR: bustype=GIO module=gbd vector=0x0 unit=0
        base=0xBF600000 base2=0xBF610000
        exprobe=(r,0xBF600000,4,0x75,0xff)
```

This ensures that a GIO device placed in either slot will be recognized.

After examining */usr/include/sys/major.h* and looking for potential major device number conflicts in other device files in the */var/sysgen/master.d* directory, you determine that major device number 51 is available and can be used for this device. You then create a master file, *gbd*, and enter:

```
*FLAG     PREFIX    SOFT      #DEV       DEPENDENCIES
 c        gbd       51        -
```

Writing edtinit()

If you use the *VECTOR* directive to configure a driver into the kernel, your driver can use a function of the form *drv***edtinit**(), where *drv* is the driver prefix. If your device driver object module includes a *drv***edtinit**() function, the system executes the *drv***edtinit**() function when the system boots. In general, you can use your *drv***edtinit**() function to perform any device driver initialization you want.

Synopsis

```
drvedtinit(e)
struct edt *e

{
   */your code here/*
}
```

When the system calls your *drv***edtinit**() function, it hands the function a pointer to a structure of type *edt*. (This structure type is defined in the *sys/edt.h* header file.)

The definition of the *edt* type structure is:

```
#define NBASE 3

typedef unsigned long iopaddr_t;
typedef struct iospace {
    unchar    ios_type;     /* io space type on 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;

#define e_base          e_space[0].ios_vaddr
#define e_base2         e_space[1].ios_vaddr
#define e_base3         e_space[2].ios_vaddr
#define e_iobase        e_space[0].ios_iopaddr
#define e_iobase2       e_space[1].ios_iopaddr
#define e_iobase3       e_space[2].ios_iopaddr
```

The *e_bus_type* must be *ADAP_GIO* (defined in *edt.h*). The *e_ctrl*, *v_cpuintr*, and *ios_type* values must be 0. The (**e_init*)() member is not used by *drv***edtinit**(). (These fields are set by the kernel, and the driver does not interfere with them.) Your driver uses the *e_base*, *e_base2*, and *e_base3* members:

e_base, *e_base2*, *e_base3*	These members give your driver the base addresses as specified in the *VECTOR* line. Each is assigned as an unsigned long data type.

To pass the desired interrupt CPU to the driver via the *irix.sm* file, use the *VECTOR* directive. The line

```
VECTOR: module=XXX intrcpu=3
```

directs **autoconfig** (via **lboot**) to set the *v_intrcpu* field for the module’s **edt** struct to 3 and the *v_setintrcpu* field to 1, indicating that *v_intrcpu* is valid. If no *intrcpu*= statement appears in the VECTOR line, *v_setintrcpu* is set to 0. The module’s **edtinit** function may then use these fields to route interrupts as desired.

```
void
XXXedtinit (struct edt *ep)
{
     if (ep->setcpuintr)
             dest_cpu = ep->cpuintr;
         else
             dest_cpu = <some default>;

     ...machine-specific intr routing ...
}
```

Note: Although **lboot** knows not to include in the kernel any GIO device driver for a device that is not present, it is a good idea for your *drv***edtinit**() function to probe for its device with **badaddr_val**(). This allows you to write a driver that is prepared if the device has been removed from the system after the kernel has been built or when the kernel runs on another system.

Continuing with this mythical GIO device driver example, its *drv***edtinit**() function could look like:

```
/* equipped device table initialization function. The edt
 * structure is defined in edt.h.
 */
void
gbdedtinit(struct edt *e)
{
   int slot, val;

   /* Check to see if the device is present */
   if(badaddr_val(e->e_base, sizeof(int), &val) ||
         (val && GBD_MASK) != GBD_BOARD_ID) {
      if (showconfig)
         cmn_err (CE_CONT,
            “gbdedtinit: board not installed.”);
         return;
   }

   /* figure out slot from base on VECTOR line in
   /* system file*/
   if(e->e_base == (caddr_t)PHYS_TO_K1(0x1f400000))
      slot = GIO_SLOT_0;
   else if(e->e_base == (caddr_t)0xBF600000)
      slot = GIO_SLOT_1;
   else {
      cmn_err (CE_NOTE,
      "ERROR from edtinit: Bad base address %x\n", e-
>e_base);
      return;
   }

#ifdef IP12      /* For Indigo R3000, set up board as a
                 /* realtime bus master  */

   setgioconfig(slot,0);

#endif

#ifdef IP20     /* For Indigo R4000, set up board as a
                /* realtime bus master */

   setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);

#endif

#ifdef IP22      /* For Indigo2, set up board as a pipelined,
                 /* realtime bus master  */

   setgioconfig(slot,GIO64_ARB_EXP0_RT |
                 GIO64_ARB_EXP0_PIPED) ;

#endif

   /* Save the device addresses, because
    * they won't be available later.
    */

   gbd_device[slot == GIO_SLOT_0 ? 0 : 1] =
            (struct gbd_device *)e->e_base;
   gbd_memory[slot == GIO_SLOT_0 ? 0 : 1] =
            (char *)e->e_base2;
              /* Where "unit_#" is any parameter passed to
              /* the interrupt handler (gbdintr) */
   setgiovector(GIO_INTERRUPT_1,slot,gbdintr,unit_#);
}
```

GIO-specific Functions

The GIO-specific support functions **setgiovector**(), **setgioconfig**(), and **splgio**(n) must be included in the **init**() or **edtinit**() section of any GIO driver.

setgiovector

The **setgiovector**() function registers an interrupt service function for a GIO-bus device interrupt with the kernel's interrupt dispatcher.

Synopsis

```
setgiovector(int level, int slot, long (*func)
            (long),int arg)
```

Arguments

level	Specifies which interrupt is used by the device. For GIO boards, this must always be *GIO_INTERRUPT_1*, since *GIO_INTERRUPT_0* and *GIO_INTERRUPT_2* are used by the graphics system.
slot	Specifies which physical slot the GIO-bus board is plugged into; must be either *GIO_SLOT_0* or *GIO_SLOT_1*.
func	Pointer to the interrupt service routine called when the associated interrupt occurs. Note that *func* may be called even when there is no pending interrupt from the particular slot specified, in which case it should simply return. The interrupt handler therefore needs to be able to determine when its device is actually interrupting and when it is not, in a timely, nondestructive manner.
arg	Passed to the interrupt service routine when it is called and may contain any value. The interrupt service routine is called with the processor interrupt mask set to disable further interrupts from the device.

splgio0, splgio1, splgio2

These functions set the processor interrupt mask to block GIO-bus interrupts.

Synopsis

```
long splgio0();
long splgio1();
long splgio2();
```

setgioconfig

setgioconfig (2K) sets up the GIO-bus arbitration mode for the GIO slot specified by the *slot* parameter. The arbitration mode is specified in the *flags* parameter as a bit-wise OR of the flags documented below.

Synopsis

`setgioconfig(int` *slot*`, int` *flags*`)`

Arguments

slot Specifies which physical slot the GIO-bus board is plugged into; must be either *GIO_SLOT_0* or *GIO_SLOT_1*.

flags Flags that indicate the configuration for the GIO board. The flags are defined as follows:

For R3000-based systems using the GIO32 bus, these defines are found in */usr/include/sys/IP12.h:*

GIO_CONFIG_LONG
Configure board as a long burst device; otherwise it will be a real-time device.

GIO_CONFIG_SLAVE
Configure board as a bus slave; otherwise it will be a bus master.

For R4000-based systems using the GIO32-bis or GIO64 bus, these defines are found in */usr/include/sys/mc.h*:

GIO64_ARB_EXP0_SIZE_64
Configure slot for 64-bit transfers; otherwise transfers will be 32-bit. For Indigo, this must not be set.

GIO64_ARB_EXP0_RT
Configure slot as a real-time device; otherwise it will be a long burst device.

GIO64_ARB_EXP0_MST
Configure slot as a bus master; otherwise it will be a slave.

GIO64_ARB_EXP0_PIPED
Configure slot as a pipelined device, otherwise it will be a non-pipelined device. For Indigo^2 systems, this must be set. For Indigo, this must *not* be set.

On R4000-based Indigo and Indigo^2 systems, **setgioconfig**() uses the slot argument to determine the location of boards.

GIO Interrupt Handler

Your driver module must contain an interrupt routine. The name of this routine does not need to be *drv***intr**(), since GIO uses **setgiovector**() to register interrupt routines. When the device generates an interrupt, the general GIO interrupt handler calls your driver's interrupt routine and passes it the unit number for the device. Within your interrupt routine, you must set flags to indicate the state of the transfer and wake up sleeping processes (if any) waiting on the transfer to complete. Usually, the interrupt routine calls **iodone**() to indicate that a block type I/O transfer for the buffer is complete.

Caution: Interrupt routines must not try to sleep themselves by calling **iowait**(), **sleep**(), **psema**(), or **delay**() kernel calls, nor should they try to access the per-process global variables in the *u* type structure directly. The *u* type structure they access may not be that of the process that made the I/O request.

Programmed I/O (PIO)

When transferring large amounts of data, your device driver should use direct memory access (DMA). Using DMA, your driver can program a few registers, return, and put itself to sleep while it awaits an interrupt that indicates the transfer is complete. This frees up the processor for use by other processes.

However, sometimes you must write a driver for a device that does not support DMA. Even if a device does support DMA, you may not want to use DMA to transfer amounts of data so small that the DMA overhead is not warranted.

In these non-DMA cases, the processor is used to copy data from user space to the device. Some device make additional operations on the data. These operations may then be triggered by writing to a register on the device, such as a printer or disk controller. Most such devices have a status register that is used to verify completion. Polling on an interrupt can be used, but it is expensive and should be used sparingly.

Listed below is part of a mythical GIO device driver for a printer controller that does not support DMA. To print data from the user, the driver copies a number of bytes (as specified by *uio_resid*) from the *uio_iov* array to an on-board memory buffer of size GBD_MEMSIZE. Following the copy of each chunk, the driver programs the device registers to indicate the size of valid data in the memory and to tell the controller to start the printing.

Note: Beginning with IRIX 5.0, direct access of the user structure *u* is not allowed. Instead, user data may be accessed only through the *uio* structure. For example, note that the field *u.u_count* is not found in this driver, and references are made to the *uio_resid* field only.

The driver then sleeps, waiting for an interrupt to indicate that the printing is complete and that the on-board memory buffer is available again. To prevent a race condition, in which the interrupt responds before the calling process can sleep, the driver uses the **splgio1**() routine.

```
/* device write routine entry point (for character devices)*/
volatile int
gbdwrite(dev_t dev, uio_t *uio)
{
   int unit = geteminor(dev)&1;
   int size, err=0, s;

   /* while there is data to transfer */
   while((size=uio->uio_resid) > 0) {

      /* Transfer no more than GBD_MEMSIZE bytes
      * to the device */
      size = size < GBD_MEMSIZE ? size : GBD_MEMSIZE;

      /* decrements size, updates uio fields, copies data */
      if(err=uiomove(gbd_memory[unit], size, UIO_WRITE, uio))
         break;

      /* prevent interrupts until we sleep */
      s = splgio1();

      /* Transfer is complete; start output */
      gbd_device[unit]->count = size;
      gbd_device[unit]->command = GBD_GO;
      gbd_state[unit] = GBD_SLEEPING;
      while (gbd_state[unit] != GBD_DONE) {
         sleep(&gbd_state[unit], PRIBIO);
      }
      /* restore the interrupt level after waking up */
      splx(s);
   }
   return err;
}
```

The driver’s use of the *volatile* declaration informs the optimizer that this register points to a hardware value that may change. Otherwise, the optimizer may determine that one write to *gbd_device->command* is sufficient.

Note: If your driver uses the **sleep**() and **wakeup**() kernel routines to sleep and awaken, it is a good idea for the *drv***intr**() to verify that the actual event has occurred before actually awakening the sleeping process. (See **sleep**() for details on the sleep / wakeup process synchronization mechanism.) If your driver uses the **iowait**() / **iodone**() routines or the **psema**() / **vsema**() routines to sleep and awaken, you need not worry about it awakening by accident. However, the routines **psema**() and **vsema**() are specific to IRIX and are probably not supported on other operating systems.

The **uiomove**() kernel routine is a useful procedure to call in these situations because it automatically updates the fields in the *uio* structure and uses **copyout**() (or **copyin**()) to check for invalid user addresses. Recall that *uio_resid* must be left with the number of bytes left untransferred.

DMA Operations

Use DMA (direct memory access) when the device supports it. In its simplest form, DMA is easy to use: your driver gives the device the physical memory address, and the transaction begins. Your driver can then put itself to sleep while it waits for the transfer to complete, thus freeing the processor for other tasks. When the transfer is complete, the device interrupts the processor. On most systems, when large amounts of data are involved, DMA devices obtain higher overall throughput than devices that do only PIO.

DMA operations are categorized as *DMA reads* or *DMA writes*. DMA operations that transfer from memory to a device, and hence read memory, are DMA reads. DMA operations that transfer from a device to memory are DMA writes. Thus, you may want to think of DMA operations as being named the point of view is that of memory.

There are some cache considerations for drivers using DMA. The cache architecture of the system dictates the appropriate cache operations. Write back caches require that data be written back from cache to memory before a DMA read, whereas both write back and write through caches require the cache to be invalidated before data from a DMA write is used. See “Data Cache Write Back and Invalidation” in Appendix A and the dki_dcache_wbinval(D3X) man page for a discussion of these issues.

Another concern for driver writers is that DMA buffers may require cache-line alignment. To this end, when a driver allocates a buffer for DMA, it must use the **kmem_alloc**() function with the *KM_CACHEALIGN* flag to obtain a buffer that is properly aligned.The interrupt service routine then calls your *drv***intr** routine. Your *drv***intr** routine can confirm that the transfer is complete (if necessary), set flags indicating the status of the transfer, and then awaken the sleeping process.

The GIO bus does not provide any address mapping registers. Any DMA operation that requires scatter / gather must be supported by GIO board hardware or a software implementation of scatter / gather.

Memory Parity Workarounds

If you are writing a GIO device driver for Indigo R4000, Indigo^2, POWER Indigo^2, or Indy systems, please make note of the following changes introduced in IRIX 5.3 (or IRIX 5.2 with Patch4, the Memory Parity Patch).

Beginning with IRIX 5.3, parity checking is enabled on the SysAD bus (see Figure 6-1). Unfortunately, with certain GIO cards, errors can occur if memory reads complete before the Memory Controller (MC) finishes calculating parity.

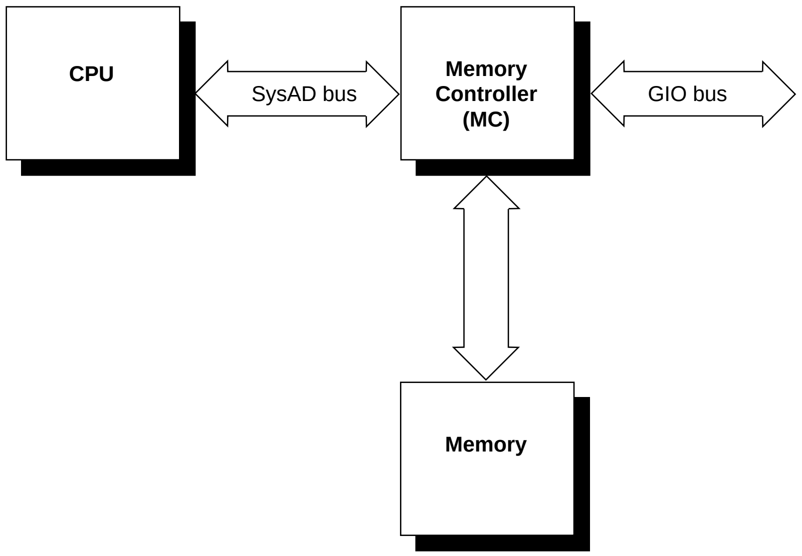


Figure 6-1 The SysAD Bus in Context

Some GIO cards do not drive all 32 GIO data lines during CPU PIO reads. These reads from the GIO card are either 8-bit (byte) or 16-bit (short word), so the lines are left floating. The problem is that to generate parity bits for the SysAD bus, the Memory Controller (MC) must calculate parity for all 32 bits. Since the calculation must occur before the CPU read completes, it is possible that one (or more) of the floating bits may change while parity is being calculated. Thus, when the CPU read completes, it may receive a parity error on the SysAD bus.

Caution: Even on GIO boards that do not drive all data lines, this problem may not show up on every transaction. It occurs only when one of the floating data lines changes state between the start of the MC parity calculation and the completion of the CPU read. Even if a driver appears to function correctly, the system may panic due to a parity error.

If you are writing a driver for a GIO card that does not drive all 32 data lines, even when fewer bits are being read, you must either:

1. Disable SysAD parity checking completely.
   This reduces the system's ability to recover from a parity error in main memory, but it is both reliable and easy to program. The way to implement this is simply to put a call to **disable_sysad_parity**() at the beginning of your driver's **init** (or **edtinit**) routine before the driver attempts any PIO reads from the GIO device.
2. Disable SysAD parity checking only when your driver is actually performing PIOs.
   The advantage here is that the software recovery procedures for memory parity errors are almost always in effect, but it requires a bit more work during driver development. Put wrappers around your driver's PIO transactions to disable SysAD parity checking before the transactions and re-enable it after the PIOs complete, as in the following code fragment:

```
{
 int was_enabled = is_sysad_parity_enabled();

 if (was_enabled)
 disable_sysad_parity();

 /* do driver PIO transactions */

 if (was_enabled)
```

```
    enable_sysad_parity();

    }
```

GIO Devices with Hardware-supported Scatter/Gather Capability

Chapter 2, “Writing a Device Driver,” tells you to use the **physio**() kernel routine to fault in and lock the physical pages corresponding to the user's buffer. **physio**() also remaps these physical pages to a kernel virtual address that remains constant even when the user's virtual addresses are no longer mapped.

Internally, **physio**() allocates a structure of type *buf* if you pass a *NULL* pointer. (**physio**() uses this structure to embody the transfer information.) **physio**() then calls your *drv***strategy**() routine and passes it a pointer to the *buf* type structure that it has allocated and primed. Your *drv***strategy**() routine must then loop through each page, starting at the kernel virtual address, and load each device scatter/gather register in turn with the corresponding physical address. Use the **kvtophys**() routine to convert a kernel virtual address to a physical address.

For example, suppose the mythical device is now a GIO device that has hardware-supporting scatter/gather. The scatter/gather registers for the device are simply a table of integers that store the physical pages corresponding to the current transfer. To start the transfer, the driver gives the device the beginning byte offset, byte count, and transfer direction. The code is:

```
/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters;
   int i, v_addr;

   /* Get address of the scatter/gather registers */
```

```
sgregisters = gbd_device[unit]->sgregisters;

/* Get the kernel virtual address of the data; note
* b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
* indicates false; in that case, the field bp->b_pages
* is a pointer to a linked list of pfdat structure
* pointers; that saves creating a virtual mapping and
* then decoding that mapping back to physical addresses.
* BP_ISMAPPED will never be false for character devices,
* only block devices.
*/
if(!BP_ISMAPPED(bp)) {
   cmn_err(CE_WARN,
      “gbd driver can’t handle unmapped buffers”);
   bioerror(bp, EIO);
   biodone(bp);
   return;
}

v_addr = bp->b_dmaaddr;

/* Compute number of pages received.
* The dma_len field provides the number of pages to
* map. Note that this may be larger than the actual
* number of bytes involved in the transfer. This is
* because the transfer may cross page boundaries,
* requiring an extra page to be mapped. Limit to
* number of scatter/gather registers on board.
* Note that this sample driver doesn’t handle the
* case of requests > than # of registers!
* numpages() is a macro declared in sys/sysmacros.h
*/
npages = numpages (v_addr, bp->b_dmalen);
/*
* Provide the beginning byte offset and count to the
* device.
*/
gbd_device[unit]->offset =
      (unsigned long)bp->b_dmaaddr & (NBPC-1);
if(npages > GBD_NUM_DMA_PGS) {
   npages = GBD_NUM_DMA_PGS;
   cmn_err(CE_WARN,
   “request too large, only %d pages max”, npages);
   if(gbd_device[unit]->offset)
      gbd_device[unit]->count = NBPC -
```

```
            gbd_device[unit]->offset + (npages-1)*NBPC;
         else
            gbd_device[unit]->count = npages*NBPC;
         bp->b_resid = bp->b_count - gbd_device[unit]->count;
      }
      else
         gbd_device[unit]->count = bp->b_count;

      /* Translate the virtual address of each page to a
      * physical page number and load it into the next
      * scatter/gather register. btop()
      * converts the byte value to a page value after
      * rounding down the byte value to a full page.
      */
      for (i = 0; i < npages; i++) {
         *sgregisters++ = btop(kvtophys(v_addr));

         /*
         /* Get the next virtual address to translate.
         * (NBPC is a symbolic constant for the page
         * size in bytes)
         */

         v_addr += NBPC;
      }

      if ((bp->b_flags & B_READ) == 0)
         gbd_device[unit]->direction = GBD_WRITE;
      else
         gbd_device[unit]->direction = GBD_READ;
      gbd_device[unit]->command = GBD_GO;   /* start DMA */

      /* and return; upper layers of kernel wait for iodone(bp)
   */
   }
```

DMA on GIO Devices Without Scatter/Gather Capability

If your device does not provide scatter / gather capability, it must break up a data transfer so that DMA transfer targets in physical memory are physically contiguous to the DMA engine (this assures that no transfer crosses a page boundary). The IRIX operating system provides a utility, sgset(D3X), that simulates scatter / gather registers in software. (See the *IRIX Device Driver Reference Pages* for details on this routine.) Your driver can use this facility to perform the virtual-to-physical mapping up front; or, as the example below shows, your driver can do this mapping following the transfer of each page:

```
/* Actual device setup for DMA, etc., if your board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;

   /* any checking for initial state here. */

   /* Get the kernel virtual address of the data; note
   * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
   * indicates false; in that case, the field bp->b_pages
   * is a pointer to a linked list of pfdat structure
   * pointers; that saves creating a virtual mapping and
   * then decoding that mapping back to physical addresses.
   * BP_ISMAPPED will never be false for character devices,
   * only block devices.
   */
   if(!BP_ISMAPPED(bp)) {
      cmn_err(CE_WARN,
         “gbd driver can’t handle unmapped buffers”);
      bioerror(bp, EIO);
      biodone(bp);
      return;
   }

   gbd_curbp[unit] = bp;
   /*
   * Initialize the current transfer address and count.
   * The first transfer should finish the rest of the
   * page, but do no more than the total byte count.
   */
   gbd_curaddr[unit] = bp->b_dmaaddr;
   gbd_totcount[unit] = bp->b_count;
   gbd_curcount[unit] = NBPC -
      ((unsigned int)gbd_curaddr[unit] & (NBPC-1));
   if (bp->b_count < gbd_curcount[unit])
      gbd_curcount[unit] = bp->b_count;
   /* Tell the device starting physical address, count,
   * and direction */
   gbd_device[unit]->startaddr = kvtophys(gbd_curaddr[unit]);
   gbd_device[unit]->count = gbd_curcount[unit];
   if (bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */

   /* and return; upper layers of kernel wait for iodone(bp)
*/
}
```

Device Driver Example

The following pages contain the complete driver code for the mythical *gbd* GIO device. Note that it includes strategy routines for devices that have hardware support for scatter/gather as well as for those devices that have no hardware scatter/gather support.

Commonly, a single set of source files is used for multiple target machines. C preprocessor *defines* are used to define differences conditionally. Command line compile options expose the correct values. The following examples are interesting:

For an Indigo (R3000) system:

```
% cc -DIP12 -DR3000 -cckr -c gbd.c
```

For an Indigo (R4000) system:

```
% cc -DIP20 -DR4000 -cckr -c gbd.c
```

For an Indigo^2 (R4000) or Indy system:

```
% cc -DIP22 -DR4000 -cckr -c gbd.c
```

Note: For R8000 systems, omit the `-cckr` argument.

For more information on compile directives, see */var/sysgen/Makefile.kernio*.

```
/* Source for a mythical GIO board device; it can be compiled
 * for devices that support DMA (with or without scatter/
 * gather support), or for PIO mode only. This version is
 * designed for IRIX 5.1 or later.
 * Dave Olson, 5/93
 */

/* defines for compilation; would normally be passed on
compilation
 * line via Makefile */
#define _K32U32   1
#define _KERNEL   1

#define IP20   1   /* define cpu type */
```

```
#if IP20 || IP22
#define R4000   1
#elif IP12
#define R3000   1
#endif
/* end of ‘normal’ compilation definitions */
/* The following definitions choose between PIO vs DMA
 * supporting boards, and if DMA is supported, whether
 * hardware scatter/gather is supported. */
#define GBD_NODMA 0  /* non-zero for PIO version of driver */
#define GBD_NUM_DMA_PGS 4 /* 0 for no hardware scatter/gather
                           * support, else number of pages of
                           * scatter/gather supported per
                           * request */

#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/cpu.h>
#include <sys/buf.h>
#include <sys/cred.h>
#include <sys/uio.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/edt.h>

   /* NOTE: This sample driver ignores the possiblity that
    * the board might be busy handling some earlier request.
    * Any real device must deal with that possiblity, of
    * course, before changing the board registers.
    */

/* these defines and structures would normally be in
 * a separate header file */

#define GBD_BOARD_ID   0x75
#define GBD_MASK   0xff  /* Use 0xff if using only first byte
                          * of ID word; use 0xffff if using
                          * whole ID word.
                          */

#define GBD_MEMSIZE 0x8000

/* command definitions */
```

```
#define GBD_GO 1

/* state definitions */
#define GBD_SLEEPING 1
#define GBD_DONE 2

/* direction of DMA definitions */
#define GBD_READ 0
#define GBD_WRITE 1

/* status defines */
#define   GBD_INTR_PEND   0x80

/* “gbd” is device prefix; also in master.d/xxx file */

/* devices interface to the board */
struct gbd_device {
   int command;
   int count;
   int direction;
   off_t offset;
   unsigned *sgregisters; /* if scatter/gather supported */
   caddr_t startaddr; /* if no scatter/gather on board */
   unsigned status;   /* errors, interrupt pending, etc. */
};

/* These are used for no scatter/gather case only, and assume
 * (since they aren’t protected!) that the driver is
 * completely single threaded. */
struct buf   *gbd_curbp[2];  /* current buffer */
caddr_t      gbd_curaddr[2]; /* current address to transfer
*/
int          gbd_curcount[2];
int          gbd_totcount[2];

/* pointer to on-board registers */
volatile struct gbd_device *gbd_device[2];

char *gbd_memory[2];   /* pointer to on-board memory */

static int gbd_state[2];    /* flag for transfer state
             * (PIO driver) */

void gbdintr(int);
extern int splgio1(void);
```

```
/* equipped device table initialization routine.  The edt
 * structure is defined in edt.h.
 */
void
gbdedtinit(struct edt *e)
{
   int slot, val;

   /* Check to see if the device is present */
   if(badaddr_val(e->e_base, sizeof(int), &val) ||
         (val && GBD_MASK) != GBD_BOARD_ID) {
      if (showconfig)
         cmn_err (CE_CONT,
            “gbdedtinit: board not installed.”);
         return;
   }

/* figure out slot from base on VECTOR line in
    * system file */
   if(e->e_base == (caddr_t)PHYS_TO_K1(0x1f400000))
      slot = GIO_SLOT_0;
   else if(e->e_base == (caddr_t)0xBF600000)
      slot = GIO_SLOT_1;
   else {
      cmn_err (CE_NOTE,
      “ERROR from edtinit: Bad base address %x\n”, e-
>e_base);
      return;
   }

#if IP12      /* For Indigo R3000 system, set up board as a
               * realtime bus master.
               */

   setgioconfig(slot,0);

#endif

#if IP20 /* For Indigo R4000 system, set up board as a
          * realtime bus master.
          */

   setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
```

```
#if IP22  /* for Indigo2 system, set up board as a pipelined,
          * realtime bus master */

   setgioconfig(slot,GIO64_ARB_EXP0_RT |
                 GIO64_ARB_EXP0_PIPED);

#endif

   /* Save the device addresses, because
    * they won’t be available later. */

   gbd_device[slot == GIO_SLOT_0 ? 0 : 1] =
            (struct gbd_device *)e->e_base;
   gbd_memory[slot == GIO_SLOT_0 ? 0 : 1] =
            (char *)e->e_base2;
   setgiovector(GIO_INTERRUPT_1,slot,gbdintr,unit_#);
}
/* minor number used to indicate which slot; open does
 * nothing but check that board is present. */
/* ARGSUSED */
gbdopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
   if(!gbd_device[geteminor(*devp)&1])
      return ENXIO;   /* board not present */
   return 0;   /* OK */
}

/* ARGSUSED */
gbdclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
   return 0;   /* nothing to do */
}

#ifdef GBD_NODMA

/* device write routine entry point (for character devices)
*/
int
gbdwrite(dev_t dev, uio_t *uio)
{
   int unit = geteminor(dev)&1;
   int size, err=0, s;

   /* while there is data to transfer */
   while((size=uio->uio_resid) > 0) {

      /* Transfer no more than GBD_MEMSIZE bytes
       * to the device */
      size = size < GBD_MEMSIZE ? size : GBD_MEMSIZE;

      /* decrements count and updates uio fields,
       * and copies data */
      if(err=uiomove(gbd_memory[unit], size, UIO_WRITE, uio))
         break;

      /* prevent interrupts until we sleep */
      s = splgio1();

      /* Transfer is complete; start output */
      gbd_device[unit]->count = size;
      gbd_device[unit]->command = GBD_GO;
      gbd_state[unit] = GBD_SLEEPING;
      while (gbd_state[unit] != GBD_DONE) {
         sleep(&gbd_state[unit], PRIBIO);
      }
      /* restore the process level after waking up */
      splx(s);
   }
   return err;
}

/* interrupt routine for PIO only board, just wake up
 * upper half of driver
 */
/* ARGSUSED1 */
void
gbdintr(int unit)
{
   /* Read your board’s registers to determine if there are
    * any errors or interrupts pending. If no interrupts
    * are pending, return without doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if (gbd_state[unit] == GBD_SLEEPING) {
      /* Output is complete; wake up top half
       * of driver, if it is waiting. */
      gbd_state[unit] = GBD_DONE;
      wakeup(&gbd_state[unit]);
   }

   /* Do anything else to board to tell it we are done
    * with transfer and interrupt here. */
return;    /* could just fall through */
}

#else   /* DMA version of driver */

void gbd_strategy(struct buf *);

/* device write routine entry point (for character devices).
 * Does nothing but call uiophysio to setup passing a pointer
 * to the gbd_strategy routine, which does most of the work.
 */
int
gbdwrite(dev_t dev, uio_t *uiop)
{
   return uiophysio((int (*)())gbd_strategy, 0, dev,
B_WRITE, uiop);
}

#if GBD_NUM_DMA_PGS > 0

/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters;
   int i, v_addr;

   /* Get address of the scatter/gather registers */
    sgregisters = gbd_device[unit]->sgregisters;

   /* Get the kernel virtual address of the data; note
    * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
    * indicates false; in that case, the field bp->b_pages
    * is a pointer to a linked list of pfdat structure
    * pointers; that saves creating a virtual mapping and
```

```
     * then decoding that mapping back to physical addresses.
     * BP_ISMAPPED will never be false for character devices,
     * only block devices.
     */
     if(!BP_ISMAPPED(bp)) {
       cmn_err(CE_WARN,
          “gbd driver can’t handle unmapped buffers”);
       bioerror(bp, EIO);
       biodone(bp);
       return;
    }

    v_addr = bp->b_dmaaddr;

    /* Compute number of pages received.
     * The dma_len field provides the number of pages to
     * map. Note that this may be larger than the actual
     * number of bytes involved in the transfer. This is
     * because the transfer may cross page boundaries,
     * requiring an extra page to be mapped. Limit to
     * number of scatter/gather registers on board.
     * Note that this sample driver doesn’t handle the
     * case of requests > than # of registers!
     */
    npages = numpages (v_addr, bp->b_dmalen);

 /*
     * Provide the beginning byte offset and count to the
     * device.
     */
    gbd_device[unit]->offset =
          (unsigned int)bp->b_dmaaddr & (NBPC-1);
    if(npages > GBD_NUM_DMA_PGS) {
       npages = GBD_NUM_DMA_PGS;
       cmn_err(CE_WARN,
           “request too large, only %d pages max”, npages);
       if(gbd_device[unit]->offset)
          gbd_device[unit]->count = NBPC -
              gbd_device[unit]->offset + (npages-1)*NBPC;
       else
          gbd_device[unit]->count = npages*NBPC;
       bp->b_resid = bp->b_count - gbd_device[unit]->count;
    }
    else
       gbd_device[unit]->count = bp->b_count;
```

```
   /* Translate the virtual address of each page to a
    * physical page number and load it into the next
    * scatter/gather register. btop()
    * converts the byte value to a page value after
    * rounding down the byte value to a full page.
    */
    for (i = 0; i < npages; i++) {
      *sgregisters++ = btop(kvtophys(v_addr));

      /*
      /* Get the next virtual address to translate.
       * (NBPC is a symbolic constant for the page
       * size in bytes)
       */

      v_addr += NBPC;
   }

   if ((bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */
  /* and return; upper layers of kernel wait for iodone(bp)*/
}
/* not much to do in this interrupt routine, since we are
 * assuming for this driver that we can never have to do
 * multiple DMA’s to handle the number of bytes requested...
 */
void
gbdintr(int unit)
{
   int error;

   /* Read your board’s registers to determine if
    * there are any errors or interrupts pending.
    * If no interrupts are pending, return without
    * doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if(error)
      bioerror(bp, EIO);
   biodone(bp);   /* we are done, tell upper layers */

   /* do anything else to board to tell it we are done
    * with transfer and interrupt here */
}

#else /*  GBD_NUM_DMA_PGS == 0; no hardware
       *  scatter/gather support */

/* Actual device setup for DMA, etc., if your board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the gbdwrite() routine via physio().
 */
void
gbd_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;

   /* any checking for initial state here. */

   /* Get the kernel virtual address of the data; note
    * b_dmaaddr may be NULL if the  BP_ISMAPPED(bp) macro
    * indicates false; in that case, the field bp->b_pages
    * is a pointer to a linked list of pfdat structure
    * pointers; that saves creating a virtual mapping and
    * then decoding that mapping back to physical addresses.
    * BP_ISMAPPED will never be false for character devices,
    * only block devices.
    */
    if(!BP_ISMAPPED(bp)) {
     cmn_err(CE_WARN,
        "gbd driver can't handle unmapped buffers");
     bioerror(bp, EIO);
     biodone(bp);
     return;
   }

   gbd_curbp[unit] = bp;
   /*
    * Initialize the current transfer address and count.
    * The first transfer should finish the rest of the
    * page, but do no more than the total byte count.
    */
   gbd_curaddr[unit] = bp->b_dmaaddr;
   gbd_totcount[unit] = bp->b_count;
   gbd_curcount[unit] = NBPC -
      ((unsigned int)gbd_curaddr[unit] & (NBPC-1));
   if (bp->b_count < gbd_curcount[unit])
      gbd_curcount[unit] = bp->b_count;
   /* Tell the device starting physical address, count,
    * and direction */
   gbd_device[unit]->startaddr = kvtophys(gbd_curaddr[unit]);
   gbd_device[unit]->count = gbd_curcount[unit];
   if (bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */

   /* and return; upper layers of kernel wait for iodone(bp)
*/
}

/* more complicated interrupt routine, not necessarily
because
 * board has DMA, but more typical of boards that do have
 * DMA, since they are typically more complicated.
 * Also more typical of devices that support block i/o, as
 * opposed to character i/o.
 */
void
gbdintr(int unit)
{
   int error;
   register struct buf *bp = gbd_curbp[unit];

   /* read your board’s registers to determine if
    * there are any errors or interrupts pending.
    * If no interrupts are pending, return without
    * doing anything.
    */
   if(!gbd_device[unit]->status & GBD_INTR_PEND)
      return;

   if(error) {
      bioerror(bp, EIO);
      biodone(bp);   /* we are done, tell upper layers */
   }
   else {
```

```
            /* On successful transfer of last chunk, continue
             * if necessary */
            gbd_curaddr[unit] += gbd_curcount[unit];
            gbd_totcount[unit] -= gbd_curcount[unit];
            if(gbd_totcount[unit] <= 0)
               biodone(bp);
                  /* we are done, tell upper layers */
            else {
            /* else more to do, reprogram board and
             * start next dma */
                gbd_curcount[unit] =
                   (gbd_totcount[unit] < NBPC
                         ? gbd_totcount[unit] : NBPC);
                gbd_device[unit]->startaddr =
                         kvtophys(gbd_curaddr[unit]);
                gbd_device[unit]->count = gbd_curcount[unit];
                if (bp->b_flags & B_READ) == 0)
                   gbd_device[unit]->direction = GBD_WRITE;
                else
                   gbd_device[unit]->direction = GBD_READ;
                gbd_device[unit]->command = GBD_GO;
                     /* start next DMA */
            }
         }

         /* Do anything else to board to tell it we are done
          * with transfer and interrupt here. */
      }
      #endif /*  GBD_NUM_DMA_PGS */

      #endif /* GBD_NODMA */
```

Chapter 7

Writing Kernel-level General Memory-mapping Device Drivers

This chapter explains how to write kernel-level general memory-mapping device drivers.

It contains the following sections:

This chapter describes how a kernel-level device driver can map VME device hardware or main (kernel) memory to user space. It introduces two system calls, **mmap**(2) and **munmap***(2),* as well as describing two IRIX driver functions, *drv***map**() and *drv***unmap**()[1]. Because your driver allows the user to map the device user space, your driver may not need to include *drv***read**() and *drv***write**() functions.

[1] SVR4 also uses a *drv***mmap**() function and an optional *drv***unmap**() function.

Including a Memory-mapping Device Driver in the Kernel

For a VME device driver, refer to Chapter 3, “Writing a VME Device Driver,” for details on adding a VME device driver to the kernel.

For a EISA device driver, see Chapter 4, “Writing an EISA Device Driver,” for details on including this type of driver.

For a SCSI device driver, refer to Chapter 5, “Writing a SCSI Device Driver,” for details on including this type of driver to the kernel.

For a GIO device driver, see Chapter 6, “Writing Kernel-level GIO Device Drivers,” for details.

Mapping and Unmapping Functions

The functions for mapping memory and registers are **mmap**(), *drv***mmap(), munmap**(), and **v_mapphys**().

mmap – Mapping the Device
*drv*mmap – Mapping the Device

When a user-level program wants to map device memory into its address space, the user program opens the special file corresponding to the particular device and uses the **mmap**() system call. Your driver needs to call *drv***mmap**() — which you need to write — when a **mmap** call maps into a user’s address space. *drv***mmap** is logically similar to a *drv***open** routine; make sure it does whatever work your device requires. (See Chapter 2, “Writing a Device Driver,” and the mmap(D2) man page for more details.)

The section of a user-level program that maps device memory into its own addressing space could look like:

```
#include "fcntl.h"
#include "sys/mman.h"

fd = open(special_file, O_RDWR);
addr = mmap(0, len, PROT_READ|PROT_WRITE,
            MAP_PRIVATE, fd, off);
```

After the kernel performs basic sanity checking on the system call arguments, if the file descriptor passed to the **mmap**(2) system call represents a special file, the kernel looks in your driver object module and calls that device mapping function.

Synopsis

```
drvmap(dev, vt, off, len, prot)
       dev_t     dev;      /* device number*/
       vhandl_t  *vt;      /* handle to caller's
                           /* virtual address space */
       off_t     off;      /* offset into device */
       int       len;      /* # of bytes to map */
       int       prot;     /* protections */
```

Arguments

dev	Gives your *drv***map**() function the device major and minor numbers. Use the **major** and **minor** macros to extract this information from *dev.*
vt	Gives your *drv***map** function a pointer to the kernel-level data structure that describes the virtual space to which the device memory will be mapped. Your driver needs this pointer when calling certain kernel service functions.
	Caution: Your driver must treat this pointer as an “opaque” handle and try not to set any of the member values directly. The specifics of this structure are likely to change from release to release. Your *drv***map**() function may change the member values of this structure indirectly, but only by calling kernel service functions.
off	This offset within device memory, at which mapping begins, gives your *drv***map**() function the kernel’s virtual address for the device.
len	Gives your *drv***map**() function the length of the device memory to be mapped into the user’s address space.
prot	Gives your *drv***map**() function the protections that the user program specified when it called **mmap**().

munmap – Unmapping the Device

To unmap a device, the user program calls the **munmap**(2) system call:

munmap (*addr*, *len*);

where *addr* is the device virtual address returned by the **mmap**(2) function and *len* is the length of the mapped area. After performing device-independent unmapping in the user's space, the **munmap**(2) system call calls your driver's *drv***unmap**() function if it is defined as an entry in the device switch table.

Synopsis

```
drvunmap(dev, vt);
dev_t     dev;              /* device number */
vhandl_t  *vt;              /* handle to caller's virtual
                               address space */
```

The *vt* and *dev* parameters are the same as for *drv***map**(), above.

Note: If a driver provides a mapping function but does not provide an unmapping function, the **munmap**() system call returns the ENODEV error condition to the user. Therefore, it is a good idea for your driver to provide a dummy unmapping function even if your driver does not need to perform any action to unmap the device.

v_mapphys – Mapping Device Control Registers and On-board Memory

Your driver can allow the user to map device registers and local memory from the user's virtual space to physical memory if your driver's *drv***map**() function calls **v_mapphys**().

Synopsis

```
int v_mapphys(vt, addr, len);
              vhandl_t *vt;
              caddr_t addr;
              int len;
```

Arguments

vt	Your *drv***map**() function must give this parameter the opaque handle to the user's virtual address space. (The internals of the structure for the opaque handle are likely to change from release to release.)
addr	Your *drv***map**() function must give this parameter the kernel virtual address by which the device is accessed. (See "VME Slave Addressing" in Appendix A.) The system takes this address from the user's call to **mmap**(2) and hands it to the *off* parameter of your *drv***map**() function.
len	Your *drv***map**() function must give this parameter the number of bytes to be mapped. The system takes this value from the user's call to **mmap**(2) and hands it to the *len* parameter of your *drv***map**() function.

If successful, **v_mapphys**() returns 0. If **v_mapphys**() fails, it sets *errno* and returns -1. After a successful call to **v_mapphys**(), the device's registers at *addr* are mapped into the user's address space as designated by *vt*. You do not need any special unmapping, so the *drv***unmap**() function does not need actions specified within it.

Caution: Your driver must be very careful when it maps device registers to a user process. It must carefully check the range of addresses that the user requests and make sure that the request references only the requested device. Because protection is available only up to a page boundary, configure the addresses of I/O cards so that they do not overlap a page. If they are allowed to overlap, an application process may be able to access more than one device, possibly a system device (for example, a disk controller or Ethernet). This can cause system secuity problems or other problems that are hard to diagnose.

Sharing Kernel Memory with a User Program

To allocate memory that is shared between the driver and the application process, your *drv***map**() function must do steps 1 and 2 of the procedure below. To free that memory later, your *drv***unmap**() function must do step 3.

1. Use the **kmem_alloc**() kernel function to allocate some memory pages in the kernel:

   ```
   caddr_t *kaddr = kmem_alloc (len , KM_CACHEALIGN);
   ```

2. Map the memory pages into the user's address space by calling **v_mapphys**():

   ```
   v_mapphys (vt, kaddr, nbytes)
   ```

3. To free the memory, your driver's unmapping function, *drv***unmap**(), must call **kmem_free**():

   ```
   kmem_free(kaddr, len);
   ```

kmem_alloc() allocates physical memory and returns a kernel virtual address associated with that memory. The physical memory is not subject to paging. **v_mapphys**() returns 0 upon success and -1 upon failure. The parameters for these calls are:

vt	Your *drv***map**() function must give this parameter the opaque handle to the user virtual address space. (The internals of this structure are likely to change from release to release.)
kaddr	Your *drv***map**() function must give this parameter the virtual address returned by **kmem_alloc**(). Your *drv***unmap**() function must give this parameter the kernel virtual address returned by **kmem_alloc***()*.
len	Your *drv***map**() function must give this parameter the number of bytes to be mapped. (This value must not be greater than the number of pages allocated in step 1.)

Example Program

Suppose the mythical VT device wants to share memory with a user program. Its *drv***map**() and *drv***unmap**() functions would look something like this:

```
#include <sys/sysmacros.h>
struct mpd {
   unsigned int d_id;    /* id of memory segment */
   caddr_t  d_addr;      /* address of allocated memory */
   int      d_npages;    /* number of pages allocated */
   struct mpd *d_last, *d_next;   /* links */
};

struct mpd vdk_list;    /* at init, this becomes a doubly
                        /*linked ring */

int vdkmap(dev_t dev, vhandl_t *vt, off_t off, int len,
           int prot)
{
   struct mpd *d;

   /* initial sanity checking (not shown) */
   ...

   /* allocate some temporary storage */
   if( (d = kmem_alloc(sizeof(struct mpd)), 0 )
      == NULL )
      return ENOMEM;

   d->d_npages = btoc(len);
   if( (d->d_addr = kmem_alloc(ctob(d-
>d_npages))KM_CACHEALIGN) == NULL ) {
      kmem_free(d,sizeof(struct mpd));
      return ENOMEM;
   }

   /* map it into the user's address space */
   if( v_mapphys(vt,d->d_addr,len) ) {
      kmem_free(d->d_addr,ctob(d->d_npages));
      kmem_free(d,sizeof(struct mpd));
      return ENOMEM;
   }

   d->d_id = v_gethandle(vt);
   /* initialize the memory */
   bzero(d->d_addr,ctob(d->d_npages));

   /* add to the list */
   d->d_next = vdk_list.d_next;
   d->d_last = &vdk_list;
   d->d_next->d_last = d->d_last->d_next = d;

   return 0;
}
```

See “Returning Opaque Handle Data,” for a description of the **v_gethandle**() system function. In the **vdkunmap**() function, you can find the piece of memory to be deallocated by searching the above list. The driver can then call the **kmem_free**() kernel function with the address and length of this section of memory:

```
int vdkunmap(dev_t dev, vhandl_t *vt)
{
   struct mpd *d;
   int id;

   id = v_gethandle(vt);

   /* Find chunk of memory corresponding to it. */
   for(d = vdk_list.d_next; d != &vdk_list ; d = d->d_next )
      if( d->d_id == id )
         break;

   /* Make sure we found it. */
   if( d == &vdk_list )
      return 0;

   /* remove from list */
   d->d_next->d_last = d->d_last;
   d->d_last->d_next = d->d_next;

   /* free up resources */
   kmem_free(d->d_addr,ctob(d->d_npages));
   kmem_free(d,sizeof(struct mpd));

   return 0;
}
```

Returning Opaque Handle Data

Use the **v_gethandle** macro to get the unique identifier associated with *vt*, the opaque handle to the user's virtual address space. (The term “opaque” indicates that your code does not directly deal with the members of this structure, which is likely to change from release to release.)

```
#include "sys/region.h"
unsigned  v_gethandle(vt);
vhandl_t  *vt;
```

Because the virtual handle points into the kernel stack, it is likely to be overridden. Use **v_gethandle** if your driver must “remember” several virtual handles. Various other macros are also defined in *sys/region.h*, including macros that get the user virtual address to which the device space is mapped; macros that get the inode associated with the special file; and macros that get the length (in pages) of the user's mapped space.

Chapter 8

Writing Multiprocessor Device Drivers

This chapter addresses questions particular to device drivers that run on multiprocessor workstations. It contains the following sections:

By default, all upper-half device driver functions—**open**(), **close**(), **ioctl**(), **read**(), **write**(), and **strategy**()— and the interrupt function are forced onto processor 0 on a multiprocessor (MP) system. Therefore, a device driver written for a single processor will work unmodified on a multiprocessor system. To avoid context switches to processor 0 for every I/O call, you can modify a device driver to run on any processor. The process of making a device driver MP-safe is often called *semaphoring*, or *multi-threading*, a driver, although the preferred method relies, strictly speaking, on locks rather than on semaphores.

Note: The driver interface now uses the SVR4 MP DDI/DKI interface except for the Silicon Graphics-specific routines, such as **pio_map**() and **dma_map**(). For example, entry points such as **open**(), **close**(), **read**(), and **write**() all have slightly different arguments and, in some cases, different procedure types in 5.x than they had in earlier versions.

Preliminary Considerations

The best way to develop a multiprocessor driver is to follow these steps:

1. Implement and test a single-threaded version of the driver.
2. Make the driver MP-safe with the use of the spinlock and semaphore calls described in this chapter.
3. Test and debug this version of the driver. It must still work perfectly when forced onto processor 0.
4. Add *D_MP* to the *drv***devflag**, recompile the driver, and rebuild the kernel with **lboot**.

   This builds a kernel that no longer forces the upper-half routines onto processor 0; they are allowed to run on the current CPU instead.
5. Run the device driver routines on an arbitrary processor to test and fix any bugs that have been exposed.

Unfortunately, making a device driver MP-safe is not a mechanized procedure, but one that requires a good understanding of the driver and its data structures. When a driver is flagged as *semaphored*, the driver writer must modify the code to prepare for two new scenarios:

- An upper-half routine may be executing on one processor while the interrupt procedure executes on another processor.
- Upper-half routines may run concurrently on different processors. For example, a process on one processor can be executing an **open** procedure while another processor executes the **strategy** function.

Shared Data between Upper-half and Interrupt Routines

If the upper-half and interrupts use **biowait** and **biodone** for synchronization, the driver will perform as desired on both single-processor and multiprocessor systems. This is because these routines have already been made multiprocessor-safe. Often, however, an interrupt routine will synchronize with upper-half procedures by using the **sleep/wakeup** functions and a shared flag word. The following is a common scenario:

Upper-half routine:

```
s = splvme();
flag |= WAITING;
while (flag & WAITING) {
   sleep(&flag, PZERO);
}
splx(s);
```

Interrupt routine:

```
if (flag & WAITING) {
   wakeup(&flag);
   flag &= ~WAITING;
}
```

The **splvme** call is used to protect *flag* from being modified in an interrupt routine. The **splvme** call raises the interrupt priority level only on the current processor and is, thus, insufficient on a multiprocessor. In this case, semaphores can be used for synchronization. When you initialize a semaphore to 0, the first **psema** call puts the calling process to sleep. A subsequent vsema(D3X) call will put the process back on the run queue. See the spl(D3) man page.The following code can replace the upper-half and interrupt scenario above:

Initialization function:

```
initnsema(&driversema, 0, "driver");
```

Upper-half routine:

```
psema(&driversema, PZERO);
```

Interrupt routine:

```
vsema(&driversema);
```

Since the semaphore functions themselves are multiprocessor-safe, no additional locking is necessary.

There may be other cases within the driver where data not specifically pertaining to synchronization is shared between an upper-half routine and an interrupt routine. You can identify these cases easily by searching for **spl***N***/splx** calls and identifying the data actually being protected against concurrent access. In such cases, it is often useful to employ spinlocks. (They are called spinlocks because the locking functions actually loop until a test-and-set value is **unset**.) Replace the **spl***N* call with a **LOCK**(D3) call, and replace the **splx** call with a corresponding **UNLOCK**(D3) call. These replacements allow the driver to perform as desired on a single processor while providing locking on a multiprocessor. See the spl(D3) man page.

Caution: Data and cache interactions must be considered.Variables that might be in registers must be declared volatile and protected as well in multiprocessor device drivers.

Protecting Shared Data Among Upper-half Routines

Since instances of the device **open**, **close**, **ioctl**, **read**, **write**, and **strategy** functions may execute concurrently on a number of processors, all data that is shared among these routines must be protected. Unfortunately, you need to identify this data by careful examination of the driver code. It is not possible to look for all instances of certain procedure calls, as it is with the interrupt routine.

You may use spinlock calls to protect shared data where the locks are held only for a short period of time. If a lock must be held for a longer period of time, you may use a semaphore initialized to 1. If this is the case, the first call to **psema** is not blocked, but all succeeding calls are. When the lock is to be freed, a vsema(D3X) call allows at most one process waiting for the semaphore to proceed. Semaphores involve slightly more overhead than spinlocks if the lock is free, and a great deal more overhead if the lock is held and the calling process must sleep. This may sound undesirable at first, but keep in mind that waiting on a locked spinlock ties up the processor from other work, while a process that puts itself to sleep allows the CPU to execute other processes. Thus, spinlock calls should be used only in situations where the lock will be held for a short duration.

Semaphore and Spinlock Calls

The remainder of this chapter is a listing of semaphore and spinlock calls. In each case, an example of the call precedes a brief explanation:

semap

```
#include <sys/types.h>
#include <sys/sema.h>
void initnsema(sema_t *semap, int value, char *name);
```

Allocate and initialize a semaphore addressed by **semap**, given *value* and *name* (for debugging).

freesema

```
void freesema(sema_t *semap);
```

Free the semaphore addressed by **semap**.

psema

```
int psema(sema_t *semap, int priority);
```

Decrement the current semaphore value by 1; if the semaphore value becomes less than 0, sleep at the given priority. The priority is the same as that given to **sleep**. The flag bit *PCATCH* may be bit-wise ORed into the priority if the sleep is breakable (greater than PZERO) and it is desired to catch the signal (as is usually the case).

The call may be prefixed with *ap* if the call is to be a NOP on single processors. This is often the case when the semaphore is used for locking.

This function returns 0 in normal operation or -1 if *PCATCH* is specified and a signal interrupted the sleep.

vsema

```
int vsema(sema_t *semap);
```

Increment the current semaphore value by 1; if the result is less than or equal to 0, place a process sleeping on the semaphore onto the run queue. As above, the call may be prefixed with *ap* if the call is to be a NOP on single processors.

This function returns 0 if no process is waiting on the semaphore, or 1 if a process is awakened.

cpsema

```
int cpsema(sema_t *semap);
```

This call conditionally provides the functionality of the **psema** operation. If the semaphore count is already less than 0, the function does not affect the semaphore value and simply returns 0. Otherwise, the semaphore count is decremented.

Note: In no case does the calling process sleep; this function can be useful to test whether a given lock has been acquired.

cvsema

```
int cvsema(sema_t *semap);
```

This function wakes up a process on the semaphore if there is one. More precisely, if the semaphore count is less than 0, it increments the semaphore count, places a process on the run queue, and returns 1; otherwise, the semaphore is unaffected and the function returns 0.

LOCK_ALLOC

```
lock_t *LOCK_ALLOC(uchar_t hierarchy, pl_t min_pl
        lkinfo_t *lkinfop, int flag);
```

This call dynamically allocates and initializes a basic lock. The lock is initialized to the unlocked state. Silicon Graphics does not support the compilation option **_LOCKTEST**, but does provide **splockmeter** for debugging purpose.

LOCK_DEALLOC

```
void LOCK_DEALLOC(lock_t *lockp);
```

This call frees an instance of a basic lock.

LOCK

```
int LOCK(lock_t, lock, int (*splr)());
```

On multiprocessor systems, this call acquires the given spinlock, *lock*. The interrupt priority level is set to at least *splr* while the lock is acquired.

On single processor systems, this calls the **spl** function **splr**.

This function returns the old priority level.

UNLOCK

```
void UNLOCK(lock_t lock, int s);
```

On multiprocessor systems, this call releases the given spinlock **lock** and restores the interrupt priority level to **s**.

On single-processor systems, restore the interrupt priority level to **s**. This is the value returned to **LOCK** above.

SLEEP_LOCK

```
void SLEEP_LOCK(sleep_t *lockp, int priority);
```

This call acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

The caller is not interrupted by signals while sleeping inside **SLEEP_LOCK**. See **psema**(D3X).

SLEEP_LOCK_SIG

```
boolean_t SLEEP_LOCK_SIG(sleep_t *lockp, int priority);
```

This function acquires the sleep lock specified by *lockp*. If the lock is not immediately available, the caller is put to sleep (the caller's execution is suspended and other processes may be scheduled) until the lock becomes available to the caller, at which point the caller wakes up and returns with the lock held.

SLEEP_LOCK_SIG may be interrupted by a signal, in which case it may return early without acquiring the lock.

If the function is interrupted by a job control stop signal (such as **SIGSTOP**, **SIGTSTP, SIGTTIN, SIGTTOU**), which results in the caller entering a stopped state, the **SLEEP_LOCK_SIG** function transparently retries the lock operation upon continuing (the call will not return without the lock).

If the function is interrupted by a signal other than a job control stop signal, or by a job control stop signal that does not result in the caller stopping (because the signal has a non-default disposition), the **SLEEP_LOCK_SIG** call returns early without acquiring the lock.

SLEEP_UNLOCK

```
void SLEEP_UNLOCK(sleep_t *lockp);
```

This function releases the sleep lock specified by *lockp*. If there are processes waiting for the lock, one of the waiting processes is awakened. See vsema(D3X).

SLEEP_TRYLOCK

```
boolean_t SLEEP_TRYLOCK(sleep_t *lockp);
```

This function tries to acquire a sleep lock. See cpsema(D3X).

TRYLOCK

```
int TRYLOCK(lock_t *lockp, pl_t pl);
```

This function tries to acquire a basic lock.

Caution: Drivers that reacquire multiple locks may deadlock when an asynchronous processor obtains a needed lock and does not free it because a lock held by another processor is also looking for a lock.

Multiprocessing STREAMS Drivers

In IRIX, all STREAMS activity is single-threaded through use of a STREAMS monitor. The kernel takes care of acquiring the monitor before running any of the regular STREAMS entry point routines. However, the device driver writer needs to take care of the interrupt entry points (hardware interrupt and timeouts) with streams_interrupt(D3X) and STREAMS_TIMEOUT(D3X). For more detailed information, see the man pages for these two calls.

STREAMS Monitor

The STREAMS monitor ensures mutually exclusive access to STREAMS on multiprocessor systems. The STREAMS put, service, open, and close functions are guaranteed to have the monitor upon entry and, thus run with assured mutual exclusion. The kernel handles all monitor interactions for these procedures, although it does not acquire the monitor for STREAMS driver interrupt routines, which must acquire the monitor explicitly.

All STREAMS drivers must acquire the monitor before performing any interaction with STREAMS from interrupt level, such as **quenable**(), **getq**(), or **putq**(). To obtain the monitor from an interrupt routine, the driver should call:

```
int streams_interrupts (func,arg1,arg2,arg3)
```

This routine either:

1. Acquires the monitor and runs **func** with arguments of *arg1*, *arg2*, and *arg3,* then releases the monitor and returns 1

   or

2. Queues the function on the monitor for execution once the current owner of the monitor releases it, and immediately returns 0. The example below shows how a STREAMS driver could use **streams_interrupt**().

There are additional changes for STREAMS drivers that use calls to **timeout**() and **delay**(), which corrupt the mutual exclusion of the monitor. To make these calls safe, the device driver writer must replace them with macros defined in the include file *sys/strmp.h.*

Replace all calls to **timeout**() with the macro **STREAMS_TIMEOUT**(); replace all calls to **delay**() with the macro **STREAMS_DELAY**(). For example, if the single-processor version of a driver contains the following calls:

```
timeout(watchdog,unit,HZ/10);
```

and

```
delay(100);
```

they should be replaced by:

```
STREAMS_TIMEOUT(watchdog,unit,HZ/10;
```

and

```
STREAMS_DELAY(100);
```

These macros revert to the original **timeout**() and **delay**() calls in the single processor case. The include file *sys/strmp.h* also defines the constant *MP_STREAMS* if the multiprocessor version of STREAMS is in use. This is useful for performing conditional compilation of sections of the STREAMS driver for multiprocessor systems. In any case, the use of these macros and definitions makes the driver machine-dependent.

STREAMS Example

```
#include “sys/strmp.h”

static void interrup_handler;

/* Actual interrupt routine that is called on an interrupt */
driverintr(unit)
int unit:
{
    /* Check to see if interrupt is valid */
    if (driver[unit]->intrmask !=0 {
        /* Call the interrupt handler that interacts
         * with STREAMS */
        streams_interrupt(interrupt_handler,unit);
    }
    else {
        /* Stray interrupt! */
        driver[unit]->stray++;
    }
    return;
}

/* Second-level interrupt handler that interacts with
 *  STREAMS.  This guarantees mutually exclusive access
 *  to STREAMS */
static void
interrupt_handler(unit)
int unit;
{
    register mblk_t *bp;

    if ((bp = allocb(128,BPRI_HI)) ==0) {
        /* Unable to allocate STREAMS block */
        driver[unit]->allocb_fail++;
        return
    }

    /* Copy data into message block */
    bcopy(driver[unit]->data,bp->wptr,128);

    /* Put onto our read queue for additional processing */
    putq(driver[unit]->rq,bp);
    return;
}
```

Chapter 9

Writing Network Device Drivers

This chapter addresses questions particular to device drivers that run on networked workstations, and is based on the assumption that network device driver writers are familiar with BSD conventions. In particular, it describes how to write an IRIX kernel **ifnet** interface networking device driver. Only issues specific to IRIX are covered here; this section does not describe the complete **ifnet** programmatic interfaces to the system although the sources for a sample skeleton **ifnet** device driver are included at the end of this section.Refer to the following books for more complete information on the BSD kernel protocol stack and device driver conventions:

- Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX® Device Driver.* John Wiley & Sons, 1992.
- Hines, Robert M., and Spence Wilcox. *Device Driver Programming*, UNIX SVR4.2. Englewood Cliffs, New Jersey: UNIX Press, 1992.
- Leffler, Samuel J., et alia. *The Design and Implementation of the 4.3BSD UNIX® Operating System*. Palo Alto, California: Addison-Wesley Publishing Company, 1989.

This chapter contains the following sections:

Preliminary Discussion

This chapter deals with requirements that go beyond STREAMS, namely, how to allow a board to communicate directly with IRIX's native protocol stack.

It is recommended that device driver writers review Chapter 8, "Writing Multiprocessor Device Drivers" before writing a network driver.

IRIX 5.3 and IRIX 6.0, although divergent, accept device drivers that have run on IRIX 5.x.

Note: A forthcoming release, IRIX 6.x, will represent a convergence of the two operating systems (32- and 64-bit). It is expected to be easier, as well as more time-efficient, in many cases, to postpone the development of new network device drivers until IRIX 6.x becomes available.

Caution: Information in this chapter is subject to change without notice.

IRIX Kernel Networking Design

The IRIX kernel networking design is based on the kernel networking framework in 4.3BSD. If you are familiar with the 4.3BSD kernel networking design, then you are already familiar with the IRIX kernel networking design because they are basically the same.

The IRIX networking design is based on the socket interface: *mbufs* are used to exchange messages within the kernel, and device drivers support the TCP/IP internet protocol suite by supporting the *ifnet* interface.

Since the kernel BSD-based networking framework and TCP/IP internet protocol suite implementation have changed little from previous releases of IRIX, porting your *ifnet* device driver to IRIX 5.3 from earlier releases of IRIX should be simple and straightforward.

Figure 9-1 displays the basic IRIX kernel networking architecture.

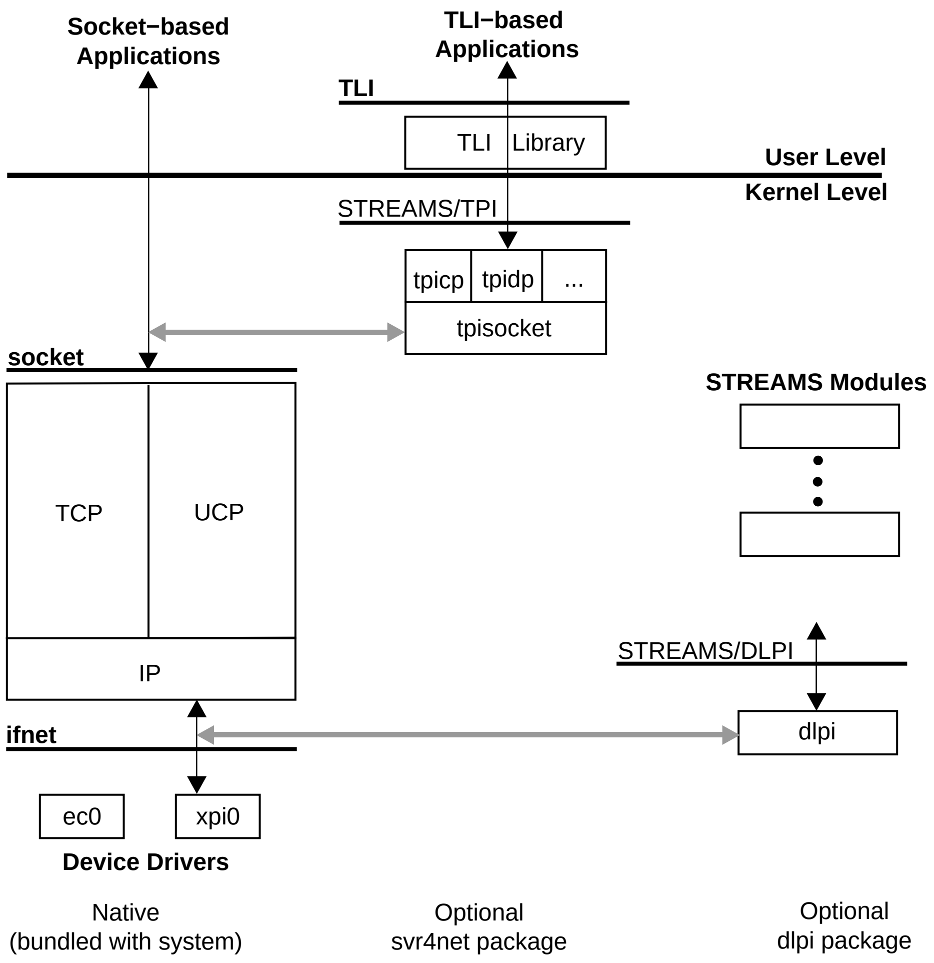


Figure 9-1 IRIX 5.3 Kernel Network Architecture

The left side of the figure shows the native socket-based TCP / IP protocol code, socket layer, and *ifnet*-based device drivers. This portion comes bundled in the basic IRIX system. Socket-based applications such as **rlogin**, **rcp**, NFS client and server, and the socket-based RPC library operate directly over this native networking framework.

The middle of the figure shows the optionally installed **svr4net** package, which provides compatibility support for user-level applications written to the STREAMS Transport Layer Interface (TLI). **tpisocket** is a kernel library module used by protocol-specific STREAMS pseudo-drivers, such as **tpitcp**, **tpiudp**, and so on, providing a TPI interface above the native kernel sockets-based network protocol stack.

The right side of the figure shows the optionally installed **dlpi** package which provides a STREAMS pseudo-driver that supports the Data Link Provider Interface (DLPI) for STREAMS-based kernel protocol stacks.

Refer to the *IRIX Network Programming Guide* and the SVR4 man pages for **STREAMS**, **TLI**, and **DLPI** programming information.

ifnet Driver Interfaces

The interface definitions and contents of the following *#include* files are subject to change without notice. While the policy is to avoid or minimize driver modifications required as new releases of IRIX become available, no guarantees of source or binary compatibility between releases of the operating system are made for networking drivers.

The primary *ifnet* data structure and routines to manipulate this are defined in *net/if.h*. They are augmented with interface types defined in *net/if_types.h*.

Functions and macros to allocate, manipulate, and free *mbufs* are defined in *sys/mbuf.h*.

The function **schednetisr** to schedule a kernel software interrupt routine, related macros, and a declaration for the IP input queue *ipintrq* are defined in *net/netisr.h*.

Constants and structures for support of the raw protocol family are defined in *net/raw.h*.

Routines defining a generic filter for use by network interfaces whose devices cannot perfectly filter multicast packets are declared in *net/multi.h*.

DLPI interface support routines and structure definitions are in *sys/dlsap_register.h*.

Socket interface **ioctl** definitions are in *net/soioctl.h*.

Ethernet and ARP-related data structures and function prototypes are provided in *netinet/if_ether.h*.

Multiprocessor Issues

Prior to IRIX 5.3, the kernel BSD framework code and TCP / IP protocol executed under a single kernel lock on multiprocessor systems making it a *single-threaded* implementation. In IRIX 5.3, the BSD framework and TCP / IP protocol suite have been *multi-threaded* to support symmetric multiprocessing by the addition of kernel locks protecting critical sections. It now supports multiple, concurrent threads of execution within the TCP / UDP / IP protocol suite, kernel socket layer, and bundled networking device drivers.

These changes are transparent to user-level programs, but, if you've written your own *ifnet*-based networking driver, it requires minor source-level changes in order to run in IRIX 5.3.

In a multi-threaded kernel, raising the processor interrupt level (IPL) by calling one of the **spl** routines, such as **splimp**() or **splnet**(), blocks interrupts from occurring on the local processor; it does not prevent interrupts from occurring on other processors in the system, nor does it prevent other processes on other processors from executing code in your critical section.

Under BSD networking, drivers interface with the protocol stacks by queueing the incoming packets on a per-protocol input queue. On multiprocessor systems, this protocol input queue must be protected by the locking macros defined in the file *net/if.h*.

All the locking macros that protect the input queue are assumed to be called at the proper processor masking level, **splimp**. All input queue locking macros also take an input parameter *ifq*, which is a pointer to the protocol input queue that must be defined as a `struct ifqueue`.

Compilation Flags for MP TCP/IP

For IRIX 5.3, the following flag must be defined in order to enable the macros necessary to run under multi-threaded TCP/IP:

```
-D_MP_NETLOCKS -DMP
```

IFNET_LOCK(*ifp, s*)

Network driver interrupt handlers should call this macro to protect critical data structures against system calls that try to access the same data structures on other processors. This macro acquires the lock that is part of the driver **ifnet** structure pointed to by *ifp*. *s* is the return value of the processor interrupt mask. The lock is held at **splimp** level. The multiprocessor TCP/IP locking scheme in IRIX 5.3 automatically holds this lock when the system enters the driver via a system call. The lock is be released automatically upon returning from the driver to synchronize simultaneous accesses to the driver from multiple processors.

IFNET_UNLOCK(*ifp*,*s*)

This macro is the reverse of **IFNET_LOCK**, **IFNET_UNLOCK** is used to release the lock. *s* is the value previously returned by **IFNET_LOCK**().

IFNET_LOCKNOSPL(*ifp*)

This macro is similar to **IFNET_LOCK** but assumes that **splimp** has been called previously.

IFNET_UNLOCKNOSPL(*ifp*)

This macro is similar to **IFNET_UNLOCK** but does not lower the **spl**.

IFQ_LOCK/UNLOCK(*ifq*)

This macro acquires/releases the lock, which is part of the input queue.

IF_ENQUEUE(*ifq*, *m*)

This macro acquires the lock on *ifq* and appends the packet pointed to by the mbuf *m*. The lock is released upon return.

Input Queueing Example

This is a code fragment of an interrupt handler that queues an input packet pointed to by *m* onto the **ip** input queue. **schednetisr**() is called to schedule processing of that packet.

The code assumed to be already at **splimp**().

```
{
    ...

      ifq = &ipintrq; /* the ip protocol queue */

 /*
 * If queue is full, we drop the packet.
 */
 IFQ_LOCK(ifq);
 if (IF_QFULL(ifq)) {
 m_freem(m);
 IF_DROP(ifq);
 IFQ_UNLOCK(ifq);
 return(-1);
 }

 IF_ENQUEUE_NOLOCK(ifq, m);
 schednetisr(NETISR_IP); /* schedule ip interrupt */
 IFQ_UNLOCK(ifq);
 return(0);
}
```

Interrupt Handler Example

The following is an example of an Ethernet interrupt handler.

```
/*
 * Ethernet interface interrupt.
 */
if_etintr(int unit)
{
 ETIO io;
 struct et_info *ei;
 register int s = splimp(); /* get the spin lock */

 ASSERT(unit == 0);
 ei = &et_info;
 io = ei->ei_io;

 if (io == 0) { /* ignore early interrupts */
     printf(“et0: early interrupt\n”);
     splx(s);
     return 1;
 }
 IFNET_LOCKNOSPL(&ei->ei_if);
 et_poll(ei);
 IFNET_UNLOCKNOSPL(&ei->ei_if);
 splx(s);
}
```

ifnet Device Driver Example

This is a skeleton *ifnet* driver for IRIX 5.3 meant to demonstrate *ifnet* driver entry points, data structures, required **ioctls**, address format conventions, kernel utility routines, and locking primitives.

Note: These kernel data structures and routines are subject to change without notice. “XXX” is used to designate places where device-specific, bus-specific, or driver-specific code sections are required.

```
/*
 * Locking strategy:
 * IFNET_LOCK() and IFNET_UNLOCK() acquire/release the
 * lock on a given ifnet structure. IFQ_LOCK() and
 * IFQ_UNLOCK() acquire/release the lock on a given ifqueue
 * structure. The ifnet or ifqueue lock must be held while
 * modifying any fields within the associated data
 * structure. The ifnet lock is also held to singlethread
 * portions of the device driver. The driver xxinit,
 * xxreset, xxoutput, xxwatchdog, and xxioctl entry points
 * are called with IFNET_LOCK() already acquired thus only
 * a single thread of execution is allowed in these
 * portions of the driver for each interface. It is the
 * driver’s responsibility to call IFNET_LOCK() within its
 * xxintr() and other private routines to singlethread any
 * other critical sections.  It is also the driver’s
 * responsibility to acquire the ifq lock by calling
 * IFQ_LOCK() before attempting to enqueue onto the IP
 * input queue “ipintrq”.
 *
 * Notes:
 * - don’t forget appropriate machine-specific cache flushing operations
 *    (refer to IRIX Device Driver Programming guide)
 * - declare pointers to device registers as “volatile”
 * - compile on multiprocessor systems with “-D_MP_NETLOCKS -DMP”
 *
 * Caveat Emptor:
 * No guarantees are made wrt correctness nor completeness
 * of this source.
 *
 * Copyright 1994 Silicon Graphics, Inc.  All rights reserved.
 */
#ident “$Revision: 1.0$”
```

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/edt.h>
#include <sys/errno.h>
#include <sys/tcp-param.h>
#include <sys/mbuf.h>
#include <sys/immu.h>
#include <sys/sbd.h>
#include <sys/ddi.h>
#include <sys/cpu.h>
#include <sys/invent.h>
#include <net/if.h>
#include <net/if_types.h>
#include <net/netisr.h>
#include <netinet/if_ether.h>
#include <net/raw.h>
#include <net/multi.h>
#include <netinet/in_var.h>
#include <net/soioctl.h>
#include <sys/dlsap_register.h>
XXX

/*
 * driver-specific and device-specific data structure
 * declarations and definitions might go here.
 */

#define    SK_MAX_UNITS    8
#define    SK_MTU        4096
#define    SK_DOG        (2*IFNET_SLOWHZ) /* watchdog duration in seconds */
#define    SK_IFT        (IFT_FDDI)    /* refer to <net/if_types.h> */
#define    SK_INV        (INV_NET_FDDI)    /* refer to <sys/invent.h> */

#define    INV_FDDI_SK    (23)        /* refer to <sys/invent.h> */

#define    IFF_ALIVE        (IFF_UP|IFF_RUNNING)
#define    iff_alive(flags)    (((flags) & IFF_ALIVE) == IFF_ALIVE)
#define iff_dead(flags)        (((flags) & IFF_ALIVE) != IFF_ALIVE)

#define    SK_ISBROAD(addr)    (!bcmp((addr), &skbroadcastaddr, SKADDRLEN))
#define    SK_ISGROUP(addr)    ((addr)[0] & 01)
```

```
/*
 * XXX media-specific definitions of address size and header format.
 */

#define    SKADDRLEN    (6)
#define    SKHEADERLEN    (sizeof (struct skheader))

/*
 * Our fictional media has an IEEE 802-looking header..
 */
struct skaddr {
    u_int8_t sk_vec[SKADDRLEN];
};
struct skheader {
    struct skaddr sh_dhost;
    struct skaddr sh_shost;
    u_int16_t sh_type;
};
struct skaddr skbroadcastaddr = {
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff
};

/*
 * Each interface is represented by a private
 * network interface data structure that maintains
 * the device hardware resource addresses, pointers
 * to device registers, allocated dma_alloc maps,
 * lists of mbufs pending transmit or reception, etc, etc.
 * XXX We use ARP and have an 802 address.
 */
struct sk_info {
    struct arpcom si_ac;        /* common ifnet and arp */
    struct skaddr si_ouraddr;    /* our individual media address */
    struct mfilter si_filter;    /* AF_RAW sw snoop filter */
    struct rawif si_rawif;        /* raw snoop interface */
    int si_unit;
    int si_flags;
    int si_initdone;
    XXX
};
struct sk_info sk_info[SK_MAX_UNITS];

#define    si_if    si_ac.ac_if
XXX
```

```
#define    sktoifp(si) (&(si)->si_ac.ac_if)
#define ifptosk(ifp)((struct sk_info *)ifp)

#define    WORDALIGNED(p)    (p & (sizeof(int)-1) == 0)

/*
 * The start of an mbuf containing an input frame
 */
struct sk_ibuf {
    struct ifheader sib_ifh;
    struct snoopheader sib_snoop;
    struct skheader sib_skh;
};
#define    SK_IBUFSZ    (sizeof (struct sk_ibuf))

/*
 * Multicast filter request for SIOCADDMULTI/SIOCDELMULTI .
 */
struct mfreq {
    union mkey *mfr_key;    /* pointer to socket ioctl arg */
    mval_t    mfr_value;    /* associated value */
};

static void skedtinit(struct edt *e);
static int sk_init(int unit);
static void sk_reset(struct sk_info *si);
static void sk_intr(int unit);
static int sk_output(struct ifnet *ifp, struct mbuf *m, struct sockaddr *dst);
static void sk_input(struct sk_info *si, struct mbuf *m, int totlen);
static int sk_ioctl(struct ifnet *ifp, int cmd, void *data);
static void sk_watchdog(int unit);
static void sk_stop(struct sk_info *si);
static int sk_start(struct sk_info *si, int flags);
static int sk_add_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_del_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_dahash(char *addr);
static int sk_dlp(struct sk_info *si, int port, int encap, struct mbuf *m, int
len);
XXX

extern struct ifqueue ipintrq;    /* ip input queue */
extern struct ifnet loif;    /* loopback driver if */

/*
```

```
 * EDT initialization routine.
 */
static void
skedtinit(struct edt *e)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int    unit;
    XXX

    /*
     * Refer to writing xxedtinit() routine descriptions
     * in VME/GIO sections of the Device Driver Programming
     * guide for:
     *
     * - probing the device
     * - configuring the slot config register
     * - registering our interrupt handler
     */
    XXX

    /*
     * Driver-specific actions that might go here:
     *
     * - allocate an unused unit number and initialize
     *   that sk_info structure.
     * - call sk_reset to disable the device
     * - allocate shared host/device memory
     * - allocating VME dma mapping registers
     * -
     */
    XXX

    if (showconfig)
        printf("sk%d: hardware MAC address %s\n",
            si->si_unit,
            sk_sprintf(si->si_ouraddr));

    /*
     * XXX your address translation protocol goes here.
     * Save a copy of our MAC address in the arpcom structure.
     */
    bcopy((caddr_t)&si->si_ouraddr, (caddr_t)si->si_ac.ac_enaddr,
        SKADDRLEN);
```

```
/*
 * Initialize ifnet structure with our name, type, mtu size,
 * supported flags, pointers to our entry points,
 * and attach to the available ifnet drivers list.
 */
ifp = sktoifp(si);
ifp->if_name = “sk”;
ifp->if_unit = unit;
ifp->if_type = SK_IFT;
ifp->if_mtu = SK_MTU;
ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST | IFF_NOTRAILERS;
ifp->if_init = (int (*)(int))sk_init;
ifp->if_output = sk_output;
ifp->if_ioctl = (int (*)(struct ifnet*, int, void*))sk_ioctl;
ifp->if_watchdog = sk_watchdog;
if_attach(ifp);

/*
 * Allocate a multicast filter table with an initial
 * size of 10.  See <net/multi.h> for a description
 * of the support for generic sw multicast filtering.
 * Use of these mf routines is purely optional -
 * if you’re not supporting multicast addresses or
 * your device does perfect filtering or you think
 * you can roll your own better, feel free.
 */
if (!mfnew(&si->si_filter, 10))
    cmn_err(CE_PANIC, “sk_edtinit: no memory for frame filter\n”);

/*
 * Initialize the raw socket interface.  See <net/raw.h>
 * and the man pages for descriptions of the SNOOP
 * and DRAIN raw protocols.
 */
rawif_attach(&si->si_rawif, &si->si_if,
    (caddr_t) &si->si_ouraddr,
    (caddr_t) &skbroadcastaddr,
    SKADDRLEN,
    SKHEADERLEN,
    structoff(skheader, sh_shost),
    structoff(skheader, sh_dhost));

/*
 * for hinv
 */
```

```
    add_to_inventory(INV_NETWORK, SK_INV, INV_FDDI_SK, unit, 0);
}

static int
sk_init(int unit)
{
    struct sk_info *si;
    struct    ifnet *ifp;
    XXX

    si = &sk_info[unit];
    ifp = sktoifp(si);

    ASSERT(IFNET_ISLOCKED(ifp));

    /*
     * Reset the device first, ask questions later..
     */
    sk_reset(si);

    /*
     * - free or reuse any pending xmit/recv mbufs
     * - initialize device configuration registers, etc.
     * - allocate and post receive buffers
     *
     * Refer to Device Driver Programming guide for
     * descriptions on use of kvtophys() (GIO) or
     * dma_map/dma_mapaddr() (VME) routines for
     * obtaining DMA addresses and system-specific
     * issues like flushing caches or write buffers.
     */

    /*
     * enable if_flags device behavior (IFF_DEBUG on/off, etc.)
     */
    XXX

    ifp->if_timer = SK_DOG;    /* turn on watchdog */

    /* turn device “on” now */
    XXX

    return 0;
}
```

```
/*
 * Reset the interface.
 */
static void
sk_reset(struct sk_info *si)
{
    struct ifnet *ifp = sktoifp(si);

    ifp->if_timer = 0;    /* turn off watchdog */

    /*
     * - reset device
     * - reset device receive descriptor ring
     * - free any enqueued transmit mbufs
     * - create device xmit descriptor ring
     */
}

static void
sk_intr(int unit)
{
    register struct sk_info *si;
    struct ifnet *ifp;
    struct mbuf *m;
    struct    ifqueue *ifq;
    int totlen;
    int s;
    int error;
    int port;

    si = &sk_info[unit];
    ifp = sktoifp(si);

    /*
     * Ignore early interrupts.
     */
    if ((si->si_initdone == 0) || iff_dead(ifp->if_flags)) {
        sk_stop(si);
        return;
    }

    IFNET_LOCK(ifp, s);    /* acquire interface lock */

    /*
     * disable device and return if early interrupt
     */
    XXX

    /*
     * test and clear device interrupt pending register.
     */
    XXX

    /*
     * process any received packets.
     */
    while (/* XXX received packets available */) {

        /*
         * Do device-specific receive processing here.
         * Allocate and post a replacement receive buffer.
         */
        XXX

        sk_input(si, m, totlen);
    }

    while (/* XXX mbufs completed transmission */) {

        /*
         * Reclaim any completed device transmit resources
         * freeing completed mbufs, checking for errors,
         * and maintaining if_opackets, if_oerrors,
         * if_collisions, etc.
         */
        XXX
    }

    IFNET_UNLOCK(ifp, s);
}

/*
 * Transmit packet.  If the destination is this system or
 * broadcast, send the packet to the loop-back device if
 * we cannot hear ourself transmit.  Return 0 or errno.
 */
static int
sk_output(
    struct ifnet    *ifp,
    struct mbuf *m0,
    struct sockaddr *dst)
{
    struct    sk_info    *si = ifptosk(ifp);
    struct skaddr *sdst, *ssrc;
    struct skheader *sh;
    struct mbuf *m, *m1, *m2;
    struct mbuf *mloop;
    int error;
    u_int16_t type;
    XXX

    ASSERT(IFNET_ISLOCKED(ifp));

    mloop = NULL;

    if (iff_dead(ifp->if_flags)) {
        error = EHOSTDOWN;
        goto bad;
    }

    /*
     * If snd queue full, try reclaiming some completed
     * mbufs.  If it’s still full, then just drop the
     * packet and return ENOBUFS.
     */
    if (IF_QFULL(&si->si_if.if_snd)) {
        while (/* XXX xmits done */) {
            /*
             * Reclaim completed xmit descriptors.
             */
            XXX

            IF_DEQUEUE_NOLOCK(&si->si_if.if_snd, m);
            m_freem(m);
        }
        if (IF_QFULL(&si->si_if.if_snd)) {
            m_freem(m0);
            si->si_if.if_odrops++;
            IF_DROP(&si->si_if.if_snd);
            return (ENOBUFS);
        }
    }

    switch (dst->sa_family) {
    case AF_INET: {
```

```
/*
 * Get room for media header,
 * use this mbuf if possible.
 */
if (!M_HASCL(m0)
    && m0->m_off >= MMINOFF+sizeof(*sh)
    && (sh = mtod(m0, struct skheader*))
    && WORDALIGNED((u_long)sh)) {
    ASSERT(m0->m_off <= MSIZE);
    m1 = 0;
    --sh;
} else {
    m1 = m_get(M_DONTWAIT, MT_DATA);
    if (m1 == NULL) {
        m_freem(m0);
        si->si_if.if_odrops++;
        IF_DROP(&si->si_if.if_snd);
        return (ENOBUFS);
    }
    sh = mtod(m1, struct skheader*);
    m1->m_len = sizeof (*sh);
}

bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);

/*
 * translate dst IP address to media address.
 */
if (!ip_arpresolve(&si->si_ac, m0,
    &((struct sockaddr_in *)dst)->sin_addr,
    (u_char*)&sh->sh_dhost)) {
    m_freem(m1);
    return (0);    /* just wait if not yet resolved */
}

if (m1 == 0) {
    m0->m_off -= sizeof (*sh);
    m0->m_len += sizeof (*sh);
} else {
    m1->m_next = m0;
    m0 = m1;
}

/*
 * Listen to ourself, if we are supposed to.
```

```
         */
        if (SK_ISBROAD(&sh->sh_shost)) {
            mloop = m_copy(m0, sizeof (*sh), M_COPYALL);
            if (mloop == NULL) {
                m_freem(m0);
                si->si_if.if_odrops++;
                IF_DROP(&si->si_if.if_snd);
                return (ENOBUFS);
            }
        }
        break;
    }

    case AF_UNSPEC:
#define     EP     ((struct ether_header *)&dst->sa_data[0])
        /*
         * Translate an ARP packet using RFC-1042.
         * Require the entire ARP packet be in the first mbuf.
         */
        sh = mtod(m0, struct skheader*);
        if (M_HASCL(m0)
            || !WORDALIGNED((u_long)sh)
            || m0->m_len < sizeof(struct ether_arp)
            || m0->m_off < MMINOFF+sizeof(*sh)
            || EP->ether_type != ETHERTYPE_ARP) {
            printf("sk_output: bad ARP output\n");
            m_freem(m0);
            si->si_if.if_oerrors++;
            IF_DROP(&si->si_if.if_snd);
            return (EAFNOSUPPORT);
        }
        ASSERT(m0->m_off <= MSIZE);
        m0->m_len += sizeof(*sh);
        m0->m_off -= sizeof(*sh);
        --sh;

        bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
        bcopy(&EP->ether_dhost[0], &sh->sh_dhost, SKADDRLEN);

        sh->sh_type = EP->ether_type;
# undef EP
        break;

    case AF_RAW:
        /* The mbuf chain contains the raw frame incl header.
```

```
          */
         sh = mtod(m0, struct skheader*);
         if (M_HASCL(m0)
             || m0->m_len < sizeof(*sh)
             || !WORDALIGNED((u_long)sh)) {
             m0 = m_pullup(m0, SKHEADERLEN);
             if (m0 == NULL) {
                 si->si_if.if_odrops++;
                 IF_DROP(&si->si_if.if_snd);
                 return (ENOBUFS);
             };
             sh = mtod(m0, struct skheader*);
         }
         break;

     case AF_SDL:
#define     SCKTP     ((struct sockaddr_sdl *)dst)
         /*
          * Send an 802 packet for DLPI.
          * mbuf chain should already have everything
          * but MAC header.
          */

         /* sanity check the MAC address */
         if (SCKTP->ssdl_addr_len != SKADDRLEN) {
             m_freem(m0);
             return (EAFNOSUPPORT);
         }

         sh = mtod(m0, struct skheader*);
         if (!M_HASCL(m0)
             && m1->m_off >= MMINOFF+SCKTP_HLEN
             && WORDALIGNED(sh)) {
             ASSERT(m0->m_off <= MSIZE);
             m0->m_len += SCKTP_HLEN;
             m0->m_off -= SCKTP_HLEN;
         } else {
             m1 = m_get(M_DONTWAIT,MT_DATA);
             if (!m1) {
                 m_freem(m0);
                 si->si_if.if_odrops++;
                 IF_DROP(&si->si_if.if_snd);
                 return (ENOBUFS);
             }
             m1->m_len = SCKTP_HLEN;
```

```
                m1->m_next = m0;
                m0 = m1;
                sh = mtod(m0, struct skheader*);
            }
            sh->sh_type = htons(ETHERTYPE_IP);
            bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
            bcopy(SCKTP->ssdl_addr, &sh->sh_dhost, SKADDRLEN);
            break;
# undef SCKTP

        default:
            printf("sk_output:  bad af %u\n", dst->sa_family);
            m_freem(m0);
            return (EAFNOSUPPORT);
        }

        /*
         * Check whether snoopers want to copy this packet.
         */
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif, (caddr_t)sh, m0->m_len)) {
            struct mbuf *ms, *mt;
            int len;        /* m0 bytes to copy */
            int lenoff;
            int curlen;

            len = m_length(m0);
            lenoff = 0;
            curlen = len + SK_IBUFSZ;
            if (curlen > MCLBYTES)
                curlen = MCLBYTES;
            ms = m_vget(M_DONTWAIT, MAX(curlen, SK_IBUFSZ), MT_DATA);
            if (ms) {
                IF_INITHEADER(mtod(ms,caddr_t), &si->si_if, SK_IBUFSZ);
                curlen = m_datacopy(m0, lenoff, curlen - SK_IBUFSZ,
                    mtod(ms,caddr_t) + SK_IBUFSZ);
                mt = ms;
                for (;;) {
                    lenoff += curlen;
                    len -= curlen;
                    if (len <= 0)
                        break;
                    curlen = MIN(len, MCLBYTES);
                    m1 = m_vget(M_DONTWAIT, curlen, MT_DATA);
                    if (0 == m1) {
```

```
                m_freem(ms);
                ms = 0;
                break;
            }
            mt->m_next = m1;
            mt = m1;
            curlen = m_datacopy(m0, lenoff, curlen,
                        mtod(m1, caddr_t));
        }
    }
    if (ms == NULL) {
        snoop_drop(&si->si_rawif, SN_PROMISC,
               mtod(m0,caddr_t), m0->m_len);
    } else {
        (void)snoop_input(&si->si_rawif, SN_PROMISC,
                  mtod(m0, caddr_t),
                  ms,
                  (lenoff > SKHEADERLEN)?
                  (lenoff - SKHEADERLEN) : 0);
    }
}

/*
 * Save a copy of the mbuf chain to free later.
 */
IF_ENQUEUE_NOLOCK(&si->si_if.if_snd, m0);

/*
 * Start DMA on the msg.
 * - allocate device-specific xmit resources  (need max
 *   of twice the number of mbufs in the mbuf chain
 *   if we're using physical memory addresses for
 *   GIO assuming worst case that each mbuf crosses
 *   a page boundary.
 */
XXX

if (error)
    goto bad;

ifp->if_opackets++;

if (mloop) {
    si->si_if.if_omcasts++;
    (void) looutput(&loif, mloop, dst);
    } else if (SK_ISGROUP(sh->sh_dhost.sk_vec))
        si->si_if.if_omcasts++;

    return (0);

bad:
    ifp->if_oerrors++;
    m_freem(m);
    m_freem(mloop);
    return (error);
}

/*
 * deal with a complete input frame in a string of mbufs.
 * mbuf points at a (struct sk_ibuf), totlen is #bytes
 * in user data portion of the mbuf.
 */
static void
sk_input(struct sk_info *si,
    struct mbuf *m,
    int totlen)
{
    struct sk_ibuf *sib;
    struct ifqueue *ifq;
    int snoopflags = 0;
    uint port;

    /*
     * set ‘snoopflags’ and ‘if_ierrors’ as appropriate
     */
    XXX

    ifq = NULL;
    sib = mtod(m, struct sk_ibuf*);
    IF_INITHEADER(sib, &si->si_if, SK_IBUFSZ);

    si->si_if.if_ibytes += totlen;
    si->si_if.if_ipackets++;

    /*
     * If it is a broadcast or multicast frame,
     * get rid of imperfectly filtered multicasts.
     */
    if (SK_ISGROUP(sib->sib_skh.sh_dhost.sk_vec)) {
        if (SK_ISBROAD(sib->sib_skh.sh_dhost.sk_vec))
```

```
            m->m_flags |= M_BCAST;
        else {
            if (((si->si_ac.ac_if.if_flags & IFF_ALLMULTI) == 0)
            && !mfethermatch(&si->si_filter,
                sib->sib_skh.sh_dhost.sk_vec, 0)) {
                if (RAWIF_SNOOPING(&si->si_rawif)
                && snoop_match(&si->si_rawif,
                    (caddr_t) &sib->sib_skh, totlen))
                    snoopflags = SN_PROMISC;
                else {
                    m_freem(m);
                    return;
                }
                m->m_flags |= M_MCAST;
            }
        }
        si->si_if.if_imcasts++;
    } else {
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif,
                (caddr_t) &sib->sib_skh,
                totlen))
            snoopflags = SN_PROMISC;
        else {
            m_freem(m);
            return;
        }
    }

    /*
     *  Set ‘port’ .  For us, just sh_type.
     */
    port = ntohs(sib->sib_skh.sh_type);

    /*
     * do raw snooping.
     */
    if (RAWIF_SNOOPING(&si->si_rawif)) {
        if (!snoop_input(&si->si_rawif, snoopflags,
                (caddr_t)&sib->sib_skh,
                m,
                (totlen>sizeof(struct skheader)
                 ? totlen-sizeof(struct skheader) : 0))) {
        }
        if (snoopflags)
            return;

    } else if (snoopflags) {
        goto drop;    /* if bad, count and skip it */
    }

    /*
     * If it is a frame we understand, then give it to the
     * correct protocol code.
     */
    switch (port) {
    case ETHERTYPE_IP:
        ifq = &ipintrq;
        break;

    case ETHERTYPE_ARP:
        arpinput(&si->si_ac, m);
        return;

    default:
        if (sk_dlp(si, port, DL_ETHER_ENCAP, m, totlen))
            return;
        break;
    }

    /*
     * if we cannot find a protocol queue, then flush it down the
     * drain, if it is open.
     */
    if (ifq == NULL) {
        if (RAWIF_DRAINING(&si->si_rawif)) {
            drain_input(&si->si_rawif,
                    port,
                    (caddr_t)&sib->sib_skh.sh_dhost.sk_vec,
                    m);
        } else
            m_freem(m);
        return;
    }

    /*
     * Put it on the IP protocol queue.
     */
    if (IF_QFULL(ifq)) {
        si->si_if.if_iqdrops++;
```

```
        si->si_if.if_ierrors++;
        IF_DROP(ifq);
        goto drop;
    }
    IF_ENQUEUE(ifq, m);
    schednetisr(NETISR_IP);
    return;

drop:
    m_freem(m);
    if (RAWIF_SNOOPING(&si->si_rawif))
        snoop_drop(&si->si_rawif, snoopflags,
               (caddr_t)&sib->sib_skh, totlen);
    if (RAWIF_DRAINING(&si->si_rawif))
        drain_drop(&si->si_rawif, port);

}

/*
 * See if a DLPI function wants a frame.
 */
static int
sk_dlp(struct sk_info *si,
    int port,
    int encap,
    struct mbuf *m,
    int len)
{
    dlsap_family_t *dlp;
    struct mbuf *m2;
    struct sk_ibuf *sib;

    if ((dlp = dlsap_find(port, encap)) == NULL)
        return (0);

    /*
     * The DLPI code wants the entire MAC and LLC headers.
     * It needs the total length of the mbuf chain to reflect
     * the actual data length, not to be extended to contain
     * a fake, zeroed LLC header which keeps the snoop code from
     * crashing.
     */
    if ((m2 = m_copy(m, 0, len+sizeof(struct skheader))) == NULL)
        return (0);
    if (M_HASCL(m2)) {
        m2 = m_pullup(m2, SK_IBUFSZ);
        if (m2 == NULL)
            return (0);
    }
    sib = mtod(m2, struct sk_ibuf*);

    /*
     * The DLPI code wants the MAC address in canonical bit order.
     * Convert here if necessary.
     */
    XXX

    /*
     * The DLPI code wants the LLC header, if present,
     * not to be hidden with the MAC header.  Decrement
     * LLC header size from ifh_hdrlen if necessary.
     */
    XXX

    if ((*dlp->dl_infunc)(dlp, &si->si_if, m2, &sib->sib_skh)) {
        m_freem(m);
        return (1);
    }
    m_freem(m2);
    return (0);
}

/*
 * Process an ioctl request.
 * Return 0 or errno.
 */
static int
sk_ioctl(
    struct ifnet *ifp,
    int cmd,
    void *data)
{
    struct sk_info *si;
    int error = 0;
    int flags;
    XXX

    ASSERT(IFNET_ISLOCKED(ifp));
```

```
si = ifptosk(ifp);

switch (cmd) {
case SIOCSIFADDR:
{
    struct ifaddr *ifa = (struct ifaddr *)data;

    switch (ifa->ifa_addr.sa_family) {
    case AF_INET:
        sk_stop(si);
        si->si_ac.ac_ipaddr = IA_SIN(ifa)->sin_addr;
        sk_start(si, ifp->if_flags);
        break;

    case AF_RAW:
        /*
         * Not safe to change addr while the
         * board is alive.
         */
        if (!iff_dead(ifp->if_flags))
            error = EINVAL;
        else {
            bcopy(ifa->ifa_addr.sa_data,
                si->si_ac.ac_enaddr, SKADDRLEN);
            error = sk_start(si, ifp->if_flags);
        }
        break;

    default:
        error = EINVAL;
        break;
    }
    break;
}

case SIOCSIFFLAGS:
{
    flags = ((struct ifreq *)data)->ifr_flags;

    if (((struct ifreq*)data)->ifr_flags & IFF_UP)
        error = sk_start(si, flags);
    else
        sk_stop(si);
    break;
}
```

```
    case SIOCADDMULTI:
    case SIOCDELMULTI:
    {
#define MKEY ((union mkey*)data)
        int allmulti;

        /*
         * Convert an internet multicast socket address
         * into an 802-type address.
         */
        error = ether_cvtmulti((struct sockaddr *)
            data, &allmulti);
        if (0 == error) {
            if (allmulti) {
                if (SIOCADDMULTI == cmd)
                    si->si_if.if_flags |= IFF_ALLMULTI;
                else
                    si->si_if.if_flags &= ~IFF_ALLMULTI;
                /* XXX enable hw all multicast addrs */
                XXX
            } else {
                bitswapcopy(MKEY->mk_dhost, MKEY->mk_dhost,
                    sizeof (MKEY->mk_dhost));
                if (SIOCADDMULTI == cmd)
                    error = sk_add_da(si, MKEY, 1);
                else
                    error = sk_del_da(si, MKEY, 1);
            }
        }
        break;
#undef MKEY
    }

    case SIOCADDSNOOP:
    case SIOCDELSNOOP:
    {
#define SF(nm) ((struct skheader*)&(((struct snoopfilter *)data)->nm))
        /*
         * raw protocol snoop filter.  See <net/raw.h>
         * and <net/multi.h> and the snoop(7P) man page.
         */
        u_char *a;
        union mkey key;
```

```
            a = &SF(sf_mask[0])->sh_dhost.sk_vec[0];
            if (!SK_ISBROAD(a)) {
                /*
                 * cannot filter on device unless mask is trivial.
                 */
                error = EINVAL;
            } else {
                /*
                 * Filter individual destination addresses.
                 * Use a different address family to avoid
                 * damaging an ordinary multi-cast filter.
                 * XXX You'll have to invent your own
                 * mulicast filter routines if this doesn't
                 * fit your address size or needs.
                 */
                a = &SF(sf_match[0])->sh_dhost.sk_vec[0];
                key.mk_family = AF_RAW;
                bcopy(a, key.mk_dhost, sizeof (key.mk_dhost));

                if (cmd == SIOCADDSNOOP)
                    error = sk_add_da(si, &key, SK_ISGROUP(a));
                else
                    error = sk_del_da(si, &key, SK_ISGROUP(a));
            }
            break;
        }

        /*
         * XXX add any driver-specific ioctls here.
         */

        default:
            error = EINVAL;
        }

        return (error);
    }

    /*
     * Add a destination address.
     * Add address to the sw multicast filter table and to
     * our hw device address (if applicable).
     */
    static int
    sk_add_da(
        struct sk_info *si,
        union mkey *key,
        int ismulti)
    {
        struct mfreq mfr;

        /*
         * mfmatchcnt() looks up key in our multicast filter
         * and, if found, just increments its refcnt and
         * returns true.
         */
        if (mfmatchcnt(&si->si_filter, 1, key, 0))
            return (0);

        mfr.mfr_key = key;
        mfr.mfr_value = (mval_t) sk_dahash(key->mk_dhost);
        if (!mfadd(&si->si_filter, key, mfr.mfr_value))
            return (ENOMEM);

        /* poke this hash into device's hw address filter */
        XXX

        return (0);
    }

    /*
     * Delete an address filter. If key is unassociated, do nothing.
     * Otherwise delete software filter first, then hardware filter.
     */
    sk_del_da(
        struct sk_info *si,
        union mkey *key,
        int ismulti)
    {
        struct mfreq mfr;

        /*
         * Decrement refcnt of this address in our multicast filter
         * and reclaim the entry if refcnt == 0.
         */
        if (mfmatchcnt(&si->si_filter, -1, key, &mfr.mfr_value))
            return (0);
        mfdel(&si->si_filter, key);

        /* disable this hash value from the device if necessary */
        XXX

        return (0);
    }

    /*
     * compute a hash value for destination addr
     */
    static int
    sk_dahash(char *addr)
    {
        int    hv;

        hv = addr[0] ^ addr[1] ^ addr[2] ^ addr[3] ^ addr[4] ^ addr[5];
        return (hv & 0xff);
    }

    /*
     * Periodically poll the device for input packets
     * in case an interrupt gets lost or the device
     * somehow gets wedged.  Reset if necessary.
     */
    static void
    sk_watchdog(int unit)
    {
        struct sk_info *si;
        struct ifnet *ifp;
        int s;

        si = &sk_info[unit];
        ifp = sktoifp(si);

        ASSERT(IFNET_ISLOCKED(ifp));

        XXX
    }

    /*
     * Disable the interface.
     */
    static void
    sk_stop(struct sk_info *si)
    {
        struct ifnet *ifp = sktoifp(si);
        ASSERT(IFNET_ISLOCKED(ifp));

        ifp->if_flags &= ~IFF_ALIVE;

        /*
         * Mark an interface down and notify protocols
         * of the transition.
         */
        if_down(ifp);

        sk_reset(si);
    }

    /*
     * Enable the interface.
     */
    static int
    sk_start(
        struct sk_info *si,
        int flags)
    {
        struct ifnet *ifp = sktoifp(si);
        int    error;

        ASSERT(IFNET_ISLOCKED(ifp));

        error = sk_init(si->si_unit);
        if (error || (ifp->if_addrlist == NULL))
            return error;
        ifp->if_flags = flags | IFF_ALIVE;

        /*
         * Broadcast an ARP packet, asking who has addr
         * on interface ac.
         */
        arpwhohas(&si->si_ac, &si->si_ac.ac_ipaddr);

        return (0);
    }
```

Chapter 10

Driver Installation and Testing

This chapter explains how to use **symmon**, the kernel debugger. It contains the following sections:

Note: This program requires an ASCII terminal on the first serial port.

Using the Kernel Debugger

Like all software, the IRIX kernel needs a reliable debugging tool. This chapter describes the kernel debugger, **symmon**(1), an indispensable tool for debugging device driver software. **symmon**(1) and the other kernel debugging tools are not automatically loaded during installation. To put these tools onto your system, install the subsystem *eoe2.sw.kdebug* manually from your software release CD-ROMs.

You can load **symmon**(1) coresident with a standalone program or operating system, then use **symmon**(1) to control the execution of that program or operating system. **symmon**(1) gives you commands to examine and alter memory and registers, set breakpoints, and execute programs.

Making a Debuggable Kernel

To make a kernel that you can debug from **symmon**(1), you need to create a kernel slightly different from the standard one. Use the following steps (many of which you used when you added a device driver to the kernel) to create a debuggable kernel.

1. Become root:

   ```
   % su
   ```

2. Edit */var/sysgen/system/irix.sm*.

   - Change the line:

     ```
     EXCLUDE idbg
     ```

     to

     ```
     INCLUDE idbg
     ```

   - Find the two lines containing *LDOPTS* toward the end of the file. Comment out the uncommented lines and remove the comment marker for the currently commented out lines. (To comment out a line, put an asterisk character in the first column.)
   - Make a note of the change. When you are finished debugging, undo the change so that you can make a standard kernel.

3. Copy the current kernel to a safe place before rebooting.

   ```
   # cp /unix /unix.orig
   ```
   or
   ```
   # ln /unix /unix.orig
   ```

4. Run */etc/autoconfig* and answer yes if prompted.

5. Use **halt** to bring the system down. When you issue the **halt** command, the system overwrites the current kernel, */unix*, with the kernel you have just created, */unix.install*.

   ```
   # halt
   ```

6. On IRIS-4D / 100, 4D / 200, 4D / 300, and 4D / 400 Series workstations, set the front panel dip switches as shown in Figure 10-1.

   Caution: For these workstations, you must not use any odd-numbered port connected to a CPU for data transfer because an inadvertent **`<ctrl-A>`** in the data stream can put the system into **symmon**. This is true of any odd-numbered port not connected to a 6-port serial card.

   Personal IRIS, Indigo, Indigo^2, Indy, Crimson, CHALLENGE / Onyx, and POWER CHALLENGE / POWER Onyx systems do not require physical switches for kernel debugging.

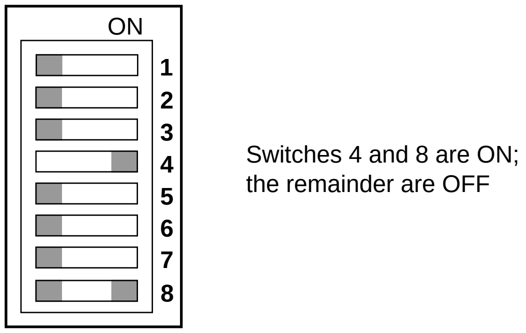


Figure 10-1 IRIS Front Panel Dip Switches

7. After halting the system, push the **`<Reset>`** button to force the processor to read the switches.

Making an ASCII Terminal the Console

To use the kernel debugger, you must install an ASCII terminal as the system console. If you do not install an ASCII terminal as the system console, you can still use the newly built kernel under your normal configuration, but you cannot use the kernel debugger. Some systems (IRIS-4D / 20, -4D / 25, -4D / 30, -4D / 35, Indigo, Indigo^2, Indy, and Crimson) allow you to use the graphics text port when the window system is not running. **symmon** may fail occasionally in this configuration, but it can be useful when no ASCII terminal is available.

To install an ASCII terminal as the system console:

1. Connect the terminal to Serial Port 1 for a single processor (or to the appropriate port number for multiprocessor systems).
2. Shut the system down in an orderly fashion.
3. Select the Command Monitor mode from the Bootstrap menu.
4. Type the following from the Command Monitor mode:

   ```
   setenv console d
   init
   ```

At this point, all console input and output occur on the ASCII terminal. Boot the system as usual. The kernel loads and, because the kernel includes the module **idbg**(), **symmon**() loads automatically from the *root* disk volume header partition for those systems where it is not part of the PROM.

Note: Although kernel output normally appears in a graphics window once graphics are started, the debugger still uses the ASCII terminal. The interactive version of **symmon**, called **idbg**, uses the same commands as **symmon** but does not stop the system from processing and does not require an ASCII terminal. You can use **idbg** to look at variables but not to set them. (See “Using symmon’s Kernel Print Command” on page 296.)

Invoking symmon

Once **symmon**(1) is loaded, control is transferred to the kernel. Control transfers to **symmon**(1) in any of the following ways:

- From the console, press **`<ctrl-A>`**

  Note: **symmon** can also be invoked from any odd-numbered CPU port on IP5 and IP7 systems, except for the IRIS-4D / 210.

- Set the *dbgstop* environment variable from command mode before booting the kernel. (You cannot use kernel symbols at this point, but all other functions work.) To do so, set the *dbgstop* environment variable:

  ```
  > setenv dbgstop 1
  ```

  This transfers control to **symmon**(1) early in system startup. (Symbolic debugging is not enabled at this point.)

- Set the *symstop* environment variable from command mode before booting the kernel:

  ```
  > setenv symstop 1
  ```

  This transfers control to symmon(1) after the symbol table has been loaded but early in the system startup procedure.

- Insert a call to the **debug(uchar_t *msg**) service from a known location within your kernel.
- The IRIX kernel executes a breakpoint instruction.
- A system panic occurs.

Displaying and Changing Registers

symmon(1) provides commands that allow you to display and alter the processor and coprocessor general-purpose registers. To identify a general-purpose register (there are 32 registers, numbered 0 through 31), you can use names such as "r0" or "r31," or you can use the compiler usage names (in some cases, you may need to prepend a "$"). The compiler names and the associated "r" names are listed in Table 10-1.

Table 10-1 Processor and Coprocessor General-purpose Registers

Compiler	Processor	Usage
zero	r0	Wired zero
at	r1	Assembler temporary
v0	r2	Function value registers
v1	r3	
a0	r4	Argument registers
a1	r5	
a2	r6	
a3	r7	
t0	r8	Caller saved registers
t1	r9	
t2	r10	
t3	r11	
t4	r12	
t5	r13	
t6	r14	
t7	r15	
s0	r16	Callee saved
s1	r17	

Table 10-1 (continued) Processor and Coprocessor General-purpose Registers

Compiler	Processor	Usage
s2	r18	
s3	r19	
s4	r20	
s5	r21	
s6	r22	
s7	r23	
t8	r24	Caller saved
t9	r25	
k0	r26	Kernel temporary
k1	r27	
gp	r28	Global pointer
sp	r29	Stack pointer
fp / s8	r30	Callee saved
ra	r31	Return address

You can refer to special R2000 / 3000 / 4000 / 8000 registers and system coprocessor registers by using the names listed in Table 10-2, Table 10-3, and Table 10-4.

Table 10-2 R2000-R4000 Processor and Coprocessor Special Registers

Name	R2000/3000/4000 Register
mdlo	Mul / div register lower word
mdhi	Mul / div register higher word
pc epc	Exception PC
sr	Status register
cause	Cause register
tlbhi entryhi	TLB entry hi register
tlblo entrylo	TLB entry lo register
badvaddr	Bad virtual address
index inx	TLB index register
context ctxt	Context register
random	Random register

Table 10-3 R4000-only System Coprocessor Registers

Name	R4000Series System Coprocessor Register
tlblo0	TLB entrylo0 register
tlblo1	TLB entrylo1 register
pagemask	TLB pagemask register
wired	TLB wired register
count	Timer count register
compare	Timer compare register
watchlo	WatchLo register
watchhi	WatchHi register

Table 10-3 R4000-only System Coprocessor Registers

Name	R4000Series System Coprocessor Register
ecc	Ecc register
cacherr	Cache error and status register
errepc	Cache ErrorEpc register
taglo	Cache tag register
config	Configuration register

Table 10-4 R8000-only Special Registers

Name	R8000 Series Special Register
tlbset	TLBset register (index into a TLB entry's set)
trapbase	Trapbase register (base address of trap vectors)
ubase	UBase register
pbase	PBase register
gbase	GBase register
shiftamt	ShiftAmt register
wired	Wired register
badpaddr	BadPAddr register

Using symmon Commands

This section describes the **symmon** commands. "symmon's dbgmon Mode Commands" describes the general **symmon** commands, which are referred to as **dbgmon** mode commands. "Using symmon's Kernel Print Command" describes the **symmon** command **kp**. The **kp** command briefly accesses another mode of the kernel debugger, *Kernel Print* mode (also called *KP* mode). After **symmon** executes the **kp** command, it returns to **dbgmon** mode.

Use **symmon**’s *dgbmon* mode when you need ordinary debugger commands such as **brk**, **dump**, **get**. Use **symmon**’s *KP* mode when you want to view kernel structures and other information.

Help for symmon Commands

To see a listing of messages concerning the **symmon** commands while running **symmon**, type a question mark (**?**) and press **<Enter>**. In addition, commands that require arguments give a usage summary if you enter the command without arguments.

symmon’s dbgmon Mode Commands

This section describes the commands of **symmon**’s *dbgmon* mode. Each heading lists the name of the command, its option flags, and its arguments. The headings use square brackets to indicate optional arguments and option flags. Following each heading is text that describes the purpose of the command. Following the command description are descriptions of each argument or option flag (if any). Different systems and operating system releases may have slightly different commands or options. For example, **symmon** is part of the PROM on some systems, while it is loaded at boot time on others. (See Table 10-5.)

Table 10-5 symmon’s dbgmon Mode Commands

Command	Description
brk [*addresslist*]	Mark breakpoints at specified addresses.
bt	Show the backtrace leading up to entry to the debugger. Also see the **kp ubt** command.
c	Leave the debugger environment and continue execution.
cacheflush *range*	Flush both the instruction and the data caches over the range of addresses given.
clear	Clear the screen.
call [*pc*] [*arg1, arg2, arg3, arg4*]	Execute the code starting at the address specified.
dis *range*	Disassemble MIPS assembly instructions for the specified range of memory locations.

Table 10-5 symmon's dbgmon Mode Commands

Command	Description
dump [-*Bcdoux*] [-*bhw*] *range*	Get a formatted display of an area of memory.
g [-*bhw*] *location*	Display the contents of a memory location or a register.
goto *list*	Continue execution of the client process from the location indicated by the client pc register to the location specified.
help	List a short summary of the built-in commands.
hx *namelist*	Convert a name or list of names into their equivalent hexadecimal address values.
lkaddr *address*	Print symbols "near" the given address.
lkup *name*	Print address of the specific partial name.
nm *addresslist*	Display the equivalent symbol name of a hexadecimal address or list of hexadecimal addresses.
p [-*bhw*] *location value*	Set the contents of the register or memory location to a value.
s [*count*] *and S* [*count*]	Execute one or more instructions of client code.
sleep	Put a processor into the waiting loop on multiprocessing systems.
string *address* [*maxlen*]	Display memory as a null-terminated ASCII string.
tlbdump [*range*]	Display the current contents of an R2000 / 3000 / 4000 address translation buffer.
tlbflush [*range*]	Flush mappings from R2000 / 3000 / 4000 address translation buffer.
tlbmap [-*i index*] [-*ndgv*] *vaddress paddress*	Establish a virtual-to-physical address map in the R2000 / 3000 / 4000 translation buffer.
tlbpid [*pid*]	Get or set the process identifier (*pid*).
tlbptov *physaddr*	Display the *tlb* entries that map a physical address.
tlbvtop *vaddress* [*pid*]	Display the R2000 / 3000 / 4000 translation buffer entries that map the specified virtual address.

Table 10-5 symmon's dbgmon Mode Commands

Command	Description
unbrk *bpnumlist*	Remove breakpoints.
wake	Wake up slave processors.
wpt [*r* | *w* | *rw*] [*0* | *physaddr*]	Set a read, write, or read / write watch point at physical address *physaddr*, using the R4000 watch point registers.

brk Command

Use **brk** to mark breakpoints at specified addresses. If you do not specify an argument, **brk** shows all currently set breakpoints.

Synopsis

`brk` [*addresslist*]

Arguments

addresslist	Use this parameter to specify the addresses at which you want breakpoints. You can enter addresses either numerically or symbolically. If you enter the addresses numerically, **brk** assumes that all address values are in base 10 unless you specify otherwise. To enter a hexadecimal value, precede the value with **`0x`**. To enter an octal value, precede the value with **`0`**. If you enter the addresses as symbolic names, enter the addresses according to the format *symbol* or *symbol+hexval*, where *hexval* is a word-aligned hexadecimal value.

bt Command

bt shows the backtrace leading up to the entry to the debugger. However, you will probably find the information from *kp ubt* to be more useful. In general, backtraces do not go past the last interrupt or exception.

Synopsis

```
bt
```

c Command

c allows you to leave the debugger environment and continue execution. *c* directs **symmon** to continue execution from the location indicated by the current value of the client program counter register. This command is the counterpart of the **`<ctrl-a>`** character combination that returns control to the debugger. See also *q*(uit).

Synopsis

```
c
```

call Command

call executes the code starting at the address specified.

Synopsis

`call [pc] [`*arg1 arg2 arg3 arg4*`]`

Arguments

[*pc*]	Use this argument to specify the starting address of a client routine. If no argument is specified, the saved *pc* is used.
[*args*]	Use these optional arguments to specify the arguments (up to four) that you want to pass to the routine pointed to by pc.

dis Command

dis disassembles MIPS assembly instructions for the specified range of memory locations.

Synopsis

`dis` *range*

Arguments

range	Use this argument to specify the range of memory you want to display. You can specify the *range* argument in one of the following three formats: *baserange* is a base address. Use this format to disassemble the contents of a single location. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. *base#count range* is a base address, followed by a number "#" character, followed by a count value. Use this format to disassemble a range of *count* words, starting at the base address. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. However, the *count* value is a hexadecimal value. *base:limit range* is a base address, followed by a colon ":" character, followed by an upper limit address. *base* is a hexadecimal address or a symbol name. Use this format to disassemble the contents of those words whose addresses are greater than or equal to the *base* address, but less than the *limit* address. The value given as the *base* address or as the *limit* can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. **Note:** In all the formats described above, the *base* address must be word-aligned.

dump Command

Use **dump** to get a formatted display of an area of memory.

Synopsis

`dump [-`*Bcdoux*`] [-`*bhw*`]` *range*

Arguments

[-*Bcdoux*]	Use these options to set the format in which dump displays the contents of a memory location. The default format is hexadecimal. The formats associated with these options are: *x* = hexadecimal *o* = octal *d* = decimal *u* = unsigned decimal *c* = ASCII *B* = binary
[-*bhw*]	Use these options to specify the size of the memory location. The default is word. The associated sizes are: *b* = byte *h* = half-word *w* = word
range	Use *range* to specify the amount of memory to be displayed. You can specify range in one of the following formats: *base range* is a base address. Use this format to disassemble the contents of a single location. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. *base#count range* is a base address, followed by a number "#" character, followed by a count value. Use this format to disassemble a range of *count* words, starting at the base address. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. However, the *count* value is a hexadecimal value.

base:limit range is a base address, followed by a colon ":" character, followed by an upper limit address. *base* is a hexadecimal address or a symbol name. Use this format to disassemble the contents of those words whose addresses are greater than or equal to the *base* address, but less than the *limit* address. The value given as the *base* address or as the *limit* can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset.

g Command

Use **g** *(*or **get**) to display the contents of a memory location, general-purpose register, special-purpose register, or a system coprocessor register.

Synopsis

`g [-`*bhw*`]` *location*

Arguments

[-*bhw*]	Use these options to specify the size of the location you want to display. The default size is a word. The sizes associated with these options are: *b* = byte *h* = half-word *w* = word
location	Use *location* to specify the location or register you want to get. The format for a location or register can be a hexadecimal value or the symbolic name (plus a hexadecimal offset) of the address or the client register you want to display. To specify one of the 32 general-purpose registers, 0 through 31, use the names **r0** through **r31**, or use the compiler-usage names. See Table 10-1 for a list of such registers.

goto Command

goto continues the execution of the client process from the location indicated by the client program counter (pc register) to the instruction at the location(s) you specify. Use this command to set a list of temporary breakpoints.

Synopsis

`goto` *list*

Arguments

list	Use this parameter to specify a list of hexadecimal addresses and / or names of locations at which you want to set temporary breakpoints. These temporary breakpoints are automatically removed once they have been encountered.

hx Command

Use **hx** to convert a name or list of names into their equivalent hexadecimal address values.

Synopsis

`hx` *namelist*

Arguments

namelist	Use this argument to specify the list of names you want to convert. This argument can be one or more symbol names (plus hexadecimal offset).

The next two items (**lkaddr** and **lkup**) are not available in older versions:

lkaddr Command

lkaddr prints symbols “near” the given address.

Synopsis

`lkaddr` *address*

Arguments

address	Use this argument to specify the starting address of the string you want to display.

lkup Command

name may be only a partial name, such as “init.”. Symbols containing *name* are printed with their respective addresses.

Synopsis

`lkup` *name*

Arguments

name	*name* is a filename.

nm Command

Use **nm** to display the equivalent symbol name (plus hexadecimal offset) of a hexadecimal address or list of hexadecimal addresses.

Synopsis

`nm` *addresslist*

Arguments

addresslist	Use this parameter to specify the addresses at which you want breakpoints. You can enter addresses either numerically or symbolically.

p Command

Use *p (*or *put*) to set the contents of the register or memory location to a value.

Synopsis

`p [-bhw]` *location value*

Arguments

[-*bhw*]	Use these options to specify the size of the memory location or register value you want to set. The default size is word. The sizes associated with each of these options are: *b* = byte *h* = half-word *w* = word
location	Use this argument to specify the memory location or register you want to set. To specify a memory location, use the hexadecimal name of the memory location, or the symbolic name (plus a hexadecimal offset) of a memory location you want to set. To specify any of the 32 general-purpose registers, use the names *r0*, *r1, r2* through *r31,* or use the compiler mnemonics for these registers (see Table 10-1). You can also set special-purpose registers and the system coprocessor registers (see Table 10-2). To specify the pc register, use the entry point of a client routine as the *location a*rgument.
value	Use this argument to specify the hexadecimal value you want to write to the specified memory location or register (*value* is always assumed to be hexadecimal.)

sleep Command

Use **sleep** to put a processor into the waiting loop on multiprocessing systems. To awaken the sleeping processor, use the **wake** command. On IRIS-4D 100 / 200 / 300 / 400 Series workstations, do not use this command on CPU 0.

Synopsis

```
sleep
```

s [count] and S [count]

Both **s** and **S** allow you to execute one or more instructions of client code. These two commands differ only in the way they handle *jal* and *bal*, instructions that execute subroutines.

When **S** executes an instruction that calls a subroutine, it executes the entire subroutine as a single instruction, up to and including the *return* instruction. **S** fails to regain control if the subprocedure does not return.

When **s** executes an instruction that calls a subroutine, it counts the jump to the subroutine as a single instruction, then executes instructions within that subroutine up to the number you have specified (minus the one counted when executing the jump instruction). If you specify only one instruction (the default), and the next instruction calls a subroutine, **s** jumps to the subroutine and stops.

Synopsis

```
s [count]
S [count]
```

Arguments

[*count*]	Use this optional argument to specify the number of instructions you want s or S to execute. The default is 1.

string Command

Use this command to display memory as a null-terminated ASCII string. This argument escapes non-printable characters with the backslash character, just as is done in the C programming language.

Synopsis

`string` *address* `[`*maxlen*`]`

Arguments

address	Use this argument to specify the starting address of the string you want to display.
[*maxlen*]	Use this optional argument to specify the length of the string. The default value for this argument is 70.

tlbdump Command

Use **tlbdump** to display the current contents of an R2000 / 3000 / 4000 address translation buffer.

Synopsis

`tlbdump [`*range*`]`

Arguments

[*range*]	Use this optional argument to select a range of tlb entries. The default behavior is to display the entire contents of tlb. The three formats for range are:
	baserange is a hexadecimal value. An index into *tlb*. *tlbdump* displays the contents of this single location.
	base#countrange is a table index followed by a number "#" character, followed by a count value. **tlbdump** displays the contents of a range of *count tlb* entries starting at the *tlb* entry whose index is *base*. Both *base* and *count* are hexadecimal values.
	base:limitrange is a table index followed by a colon ":" character, followed by an upper limit index. Use this format to display a range of *tlb* entries whose indices are greater than or equal to *base*, but less than *limit*. These index values are all hexadecimal values.

tlbflush Command

Use **tlbflush** to flush mappings from the R2000 / 3000 / 4000 address translation buffer, thus making them invalid, not matching any possible **address/pid** pair. You can use this command to clear translations for both **symmon** and the client process.

Synopsis

`tlbflush [`*range*`]`

Arguments

[*range*]	Use this optional argument to specify the range of tlb entries that you want to flush. If you do not specify a range, tlbflush defaults to clearing all tlb entries. To specify a [*range*], use one of the following formats:

Note: Multiprocessor systems have an address translation buffer associated with each processor. A command that dumps or changes translation buffers affects only the processor associated with the debug terminal from which such commands issued.

tlbmap Command

Use **tlbmap** to establish a virtual to physical address map in the R2000 / 3000 / 4000 translation buffer. You can use **tlbmap** to establish mappings for both **symmon** and the client process.

Synopsis

`tlbmap [-i` *index*`] [-`*ndgv*`] [-`*dgv*`] [-c` *algo*`]` *vaddress paddress*

Arguments

[-*i index*]	Use the -i option to specify the particular tlb entry, index, that you want to use to contain the mapping. If you do not specify a *tlb* index, **tlbmap** uses a random *tlb* entry, at an index ranging 8 to 63.
[-*ndgv*]	(R2000 / 3000 only)
[-*dgv*]	(R4000 only) Use these options to set bits in the *tlb* entry for the map. Each option controls a single bit. By default, these bits are unset (zero). The significance of setting a bit (and the associated options) are: *n* = non-cacheable (R2000 / 3000 only) *d* = data *g* = global *v* = valid
[-*c algo*]	(R4000 only) Use this to set the cache algorithm in the *tlb* entry for the map. The algorithms are specified by a number. The designations are: 0: reserved 1: reserved 2: uncached 3: cacheable, non-coherent 4: cacheable, coherent exclusive 5: cacheable, coherent exclusive on write 6: cacheable, coherent update on write 7: reserved

vaddress	Use this argument to specify the virtual address side of the map.
paddress	Use this argument to specify the physical address side of the map.

tlbpid Command

Use **tlbpid** to get or set the process identifier (*pid*), a value in the R2000 / 3000 / 4000 system coprocessor register, **tlbhi**. The **tlbpid** command affects only the process identifier used by **symmon** and client code executed through the **call** command. **tlbpid** does not affect the process identifier used when client code is executed by single stepping or when continuing execution.

Synopsis

`tlbpid [`*pid*`]`

Arguments

[*pid*]	Use this optional argument to indicate the value to which you want **tlbpid** to set the *pid* value in the R2000 / 3000 / 4000 coprocessor register, **tlbhi**.

If you specify no [*pid*] argument, **tlbpid** displays the *pid* value currently in the R2000 / 3000 / 4000 coprocessor register, **tlbhi**.

tlbptov Command

Use **tlbptov** to display the *tlb* entries that map a physical address. **tlbptov** searches the R2000 / 3000 / 4000 translation buffer, looking for translations that map the specified physical address. **tlbptov** displays all matches, whether valid or invalid.

Synopsis

`tlbptov` *physaddr*

Arguments

physaddr	Use this argument to specify the physical address for which you want to find *tlb* entries.

tlbvtop Command

Use **tlbvtop** to display the R2000 / 3000 / 4000 translation buffer entries that map the specified virtual address.

Synopsis

`tlbvtop` *vaddress* [*pid*]

Arguments

vaddress	Use this argument to specify the virtual address for which you want to find *tlb* entries.
[*pid*]	Use this optional argument to specify the pid associated with the *tlb* entry. If you do not specify a pid, **tlbvtop** defaults to using the *pid* value currently in the coprocessor register **tlbhi**.

unbrk Command

Use **unbrk** to remove breakpoints.

Synopsis

`unbrk` *bpnumlist*

Arguments

bpnumlist	Use this argument to specify the ordinal of a particular breakpoint you want to remove. Use the **brk** command to get the ordinal of a particular breakpoint.

wake Command

Use **wake** to wake up slave processors. **wake** brings all slave processors into a waiting loop, whether those processors are scanning for a keystroke on the console to drop into **symmon** or looking for the address to which to jump to execute the **boot**() routine.

Synopsis

```
wake
```

wpt Command

Use **wpt** to set a read, write, or read / write watch point on a physical address, using the R4000 watch point registers.

Synopsis

```
wpt [r|w|rw|] [0|physaddr]
```

Arguments

r	Read
w	Write
rw	Read / write
physaddr	Double word aligned address. The watch point will trip on any access within the next eight bytes.
0	An argument of 0 clears the watch point.

Using symmon's Kernel Print Command

The kernel print mode command, **kp**, allows the user to print kernel structures or information summarized from kernel structures. If you do not know the names of the kernel structures or summaries that you want to see, just issue the **kp** command without specifying an argument. If you do not specify an argument for **kp**, the command displays the names of all its subcommands. The **??** command lists the names of all the **kp** subcommands with short descriptions. For systems that do not have **symmon** in PROM, you may omit the leading "kp".

Some of the information you can view using the **kp** command is not of interest to you when debugging device drivers. In the following sections, we describe only those structures and summaries useful when debugging a device driver. **kp** commands that display data are also used with the **idbg** command (without the **kp** keyword).

Note: The following sections mention a number called the "process table entry index." This number is not the process ID. To find the process table entry index that corresponds to a particular process ID, issue the command: **kp plist** *proc_id*, where *proc_id* is the process ID for which you want to find the process table entry index.

kp buf [index/address] – View a buf Structure

The kernel can contain any number of **buf** type structures. Use these structures to manage data transfers to and from a device. To view all of these structures, use **kp buf** without specifying [*index/address*]. To see a particular buf type structure, specify either the buffer index number for the structure or the address for the **buf** type structure as the [*index/address*] argument.

kp eframe address

This displays the exception frame at the given address. The exception frame holds the contents of the general purpose registers at the time the process last executed. If the address is a small integer, the exception frame of the process with that process table index is used.

kp inode number/address – View an inode Structure

Use **kp inode** to view the contents of an **inode** structure. To specify which inode structure you want to see, provide the number or the address of the **inode** structure as the [*number/address*] argument. **kp inode** is obsolete in 5.x. See *vnode* structures.

Synopsis

`kp inode` *number/address*

kp kill addr

Send **SIGKILL** to a process, where *addr* is the process id.

kp mlist

List all dynamically loaded and registered kernel modules.

kp msyms *id*

Use *ID* to print symbols for the dynamically loaded module **ID**, which can be found using the **ml list** command or the **lboot -V** command.

kp pb — Dump Console Print Buffer

This prints the contents of the console print buffer, which can be useful when an important message scrolls off the screen.

kp plist [#] – List Process Summaries

Without an argument, the **kp** command describes all the processes on the system. To see the *plist* information that applies to a particular process, specify the process ID as the [#] argument. Thus, to see the *plist* information that applies to a process with a *pid* of 16672, enter the command:

```
% kp plist 16672
```

The information that *kp plist* [#] displays for a process consists of items such as:

- process table entry number, or index (also known as the process slot number). You use this number, instead of the *pid*, when selecting process-specific information from other kernel structures via **kp**, so be careful not to confuse it with the *pid*.
- process ID
- process status (such as sleep)

kp pda [index] – View a pda Structure

The **pda** structure contains information about a processor's private data area. If you do not specify [*index*], *kp pda* gives you the **pda** structure for all the processors. To see the **pda** structure for a specific processor only, use the [*index*] argument to specify the processor array index for that processor. The value of this argument must be 0 on a single-processor system or a number between 0 and *n* on a multiprocessor system.

kp proc index/address – View a proc Structure

A **proc** structure contains process-specific information not listed in the plist summaries. **kp** requires that you specify the **proc** structure you want to see. You can specify a particular **proc** structure by giving either the process table entry number (*index*) for the process or the address (*address*) for the **proc** structure (in *u.u_proc.p* for the currently mapped process).

kp qbuf device

Dump the contents of buffers queued for the given device. The *device* argument is given as the major / minor device number of the desired device.

kp runq – View the runq Structure

The **runq** structure contains information describing the processes in the run queue. These are processes that are not running but that could run. **kp runq** displays all the information for every process in the **runq** structure.

kp sema address – View a Semaphore Structure

The system uses **sema** type structures to manage the information associated with a semaphore (for example, the semaphore's name, the number of times it was used, whether it is free or locked). Use **kp sema** *address* to list the contents of a particular **sema** structure.

kp slpproc – View Summary of Sleeping Processes

Use this command to view information on all sleeping processes. This information includes such items as the process name, the process address, and the name and address of the semaphore upon which the process is sleeping. **kp slpproc** lists all the information in the *slpproc* structure.

kp ubt [index] – View Subroutine Backtrace Summary

Use this command to view subroutine backtrace information for a user process. This information includes such items as the stack pointer, the caller, and the called function. If you do not specify [**index**], this command reports on the currently mapped user process. To specify a particular user process (one that may or may not currently be mapped) give **kp ubt** [**index**], the index into the process table for the user process in which you are interested.

Caution: If a process is currently running on a CPU, you *must* issue the **ubt** command on the CPU the process is running on. Using **ubt #** on currently running process will produce erroneous results.

kp ubt [*index*] is analogous to the *dbx* command, *where*.

kp user [index] – View a User Structure

The **user** structure contains information that describes a user area. If you do not specify [*index*]**, kp user** displays the user structure for the currently mapped user. To view the user structure for a particular process (one that may or may not currently be mapped), give **kp user** [*index*], the index into the process table for the user process in which you are interested.

kp wd – View SCSI Information

Use **kp WD93** to view information concerning **WD93** SCSI devices.

Synopsis

`kp wd [`*0*`, `*1*`, `*2*`, `*address*`]`

Arguments

0	Use this option to display the contents of the WD93 registers. No equivalent yet exists for WD95.
1	Use this option to display the host structure and active *scsi_request_t*.
2	Use this option to show all allocated subchannels.
address	Use this option to show the subchannel at the specified address.

Multiprocessor Debugging

Debugging multiprocessor device drivers with **symmon** is similar to debugging multiprocess code in general. There are two phases: debugging multiprocess code forced to run on a single processor, followed by debugging the same code as it runs on multiple processors.

Configuring the System Software

To force a multiprocess device driver to run on one processor only, you must deactivate all but one processor. Currently, network load is spread across CPUs by default. To preempt network packet processing, see the **rtnetd**(1M) and **network**(1M) man pages.

Preparing the System Hardware

On IRIS-4D and POWER Series systems, multiprocessor debugging requires that you connect an ASCII terminal to each processor. On a two-processor system, the second terminal connects to Serial Port 3. For four-processor systems, the additional terminals connect to ports 5 and 7. Additionally, you must set the MODE switches 4 and 8 (located at the front of the 4D system panel) to "open." When these MODE switches are open, power-on diagnostics display on the terminal attached to each processor. **symmon** loads automatically on all non-console processors, and each prints a message to its terminal. The mode switches are depicted in Table 10-1.

Note: Indigo, Indigo^2, Indy, Crimson, CHALLENGE / Onyx, POWER CHALLENGE / POWER Onyx systems do not require physical manipulation of MODE switches.

symmon and Multiprocessor Debugging

On multiprocessor systems, a separate **symmon** runs on each processor. Breakpoints, however, are shared. When a processor reaches a breakpoint instruction, control is transferred to the **symmon** associated with that processor. Other processors may continue to run independently until they also happen to hit that breakpoint, which can happen if you set a breakpoint in a kernel function or anywhere in a multiprocessor device driver.

At this point, before the kernel is loaded (through a **boot** command from command mode), you can disable processors not associated with the console (the console processor is referred to as the *master processor*) by pressing the **`<Enter>`** key on the keyboard. Disabling processors is useful during the initial debugging phase of a multiprocess program because it is easier to debug a multiprocess program first in single-processor mode. This way, you eliminate the usual sorts of bugs before you have to deal with bugs associated specifically with multiprocessing (such as synchronization bugs).

(See also M. Maekawa, A. Oldehoeft, and R. Oldehoeft. *Operating Systems Advanced Concepts*. The Benjamin / Cummings Publishing Company, Inc., 1987 and A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*, Third Edition. Addison Wesley Publishing Company, 1991.)

By default, all device driver software runs only on the master processor. A driver that understands the multiprocessor environment can indicate that it contains semaphored code through a flag in the master file. However, it is still easier to debug drivers initially on a single-processor system.

If you do not disable the non-master processors, processes run on them as usual. If a breakpoint is encountered on that (non-master) processor, or if a **`<ctrl-A>`** character sequence is entered on that (non-master) processor's terminal, control transfers to **symmon** on that processor. Other processors continue to execute kernel (and user) code until they detect a **`<ctrl-A>`** character sequence, reach a **break** instruction, or hit a lock or semaphore held by the stopped CPU.

Debugging on CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx systems requires only a single ASCII terminal. All symmon output for all CPUs is multiplexed to the one port. To talk to an individual CPU, use the **cpu#** command. CPUs must be stopped (with the **stop** command) before you can connect to them. To restart all CPUs, use the **C all** command.

Note: It may be necessary to use **`<ctrl-A>`** several times when entering symmon.

Forcing System Memory Dumps

Because the code to force all CPUs, other than the one that does the dump, to spin is at a very high level, the best way to force a memory dump is to use **symmon** to call **panic** with the address of some string in the kernel, as in:

```
kp call panic addr
```

On CHALLENGE / Onyx and POWER CHALLENGE / POWER Onyx series systems, an alternative is to use the NMI button.

Chapter 11

Kernel-level Dynamically Loadable Modules (DLMs)

This chapter describes how kernel modules can be loaded dynamically. It contains the following sections:

IRIX supports dynamic loading and unloading of modules into a running kernel. Kernel modules can be registered and then loaded automatically by the kernel when the corresponding device is opened, or they can be loaded using either the **lboot** or the **ml** command. Similarly, dynamically loaded modules can be unloaded from the kernel automatically or manually if the module includes an “unload” entry point. Loadable kernel modules that are supported include: character, block, and STREAMS device drivers; STREAMS modules; library modules; and the *idbg.o* module. This chapter describes how to configure and use dynamically loadable kernel modules.

Module Configuration

Each loadable module must contain the string shown below, where *M_VERSION* is defined in the *mload.h* header file that must be included by the loadable module:

```
char *drvmversion = M_VERSION;
```

A loadable module must be compiled and linked with the following **cc** options before it can be loaded into the kernel:[1]

```
% cc -non_shared -coff -Wx,-G0 -Wc,-pic0 -r -d -c -Wc,-jalr
```

-*non_shared*	Produce a static executable. The output object created does not use any shared objects during execution.
-*coff*	Produce a COFF object.
-*Wx,-G0*	Disable global pointer. Not supported for loadable modules.
-*Wc,-pic0*	Do not allocate extra stack space that is not necessary for *non_shared* coff objects.
-*r*	Retain relocation entries in the output file.
-*d*	Force definition of common storage and define loader-defined symbols. Without this option, space is not allocated in *bss* for common variables.

[1] No driver written for IRIX 6.0 should use the -coff flag.

-*c*	Suppress the compilation loading phase and force an object file to be produced even if only one program is compiled.
-*Wc,-jalr*	Force the compiler to produce **jalr** instructions rather than **jal** instructions. A **jal** instruction has a 26-bit target, so if a module is loaded into *k2seg*, for example, it could not call a kernel function in *k0seg*.

A loadable module must not be dependent on any loadable module, other than a library module. In order to load a module comprised of multiple object files, the object files must be linked into a single object file, using *ld* with the following options:[1]

```
% ld -non_shared -coff -G0 -r -d
```

For more information, see the **cc**(1) and **ld**(1) man page entries.

Using a Dynamically Loadable Kernel Module

You can use either **lboot** or the **ml** command to load, register, unload, unregister, and list loadable kernel modules. The **lboot** command parses module type, prefix, and major number information from the module's master file in the */var/sysgen/master.d* directory. The loadable object file is expected to be found in the */var/sysgen/boot* directory. The **ml** command also provides a means of loading, registering, and unloading loadable modules, without the having to create a master file or reconfigure the kernel.

Load

When a module is loaded, the following operations take place:

1. The object file's header is read.
2. Memory is allocated for the module's text, data, and bss.
3. The module's text and data are read.
4. The module's text and data are relocated, and unresolved references into the kernel are resolved.

5. A symbol table is created for the module; the module is added into the appropriate kernel switch table.
6. The module's *drv***init**() function is called.

Space allocated for the module's text, data, and bss is located in either *k0seg* or *k2seg*. Static buffers in loadable modules are not necessarily physically contiguous in memory.

A module can be loaded using the following **lboot** command:

```
% lboot -L master
```

A module can also be loaded using the following **ml** command:

```
% ml ld -[v] -[cbBfm] module.o -p prefix \
 [-s major major...] [-a modname]
```

If a major number is specified by the **ml** command, that major number must match the major number used to create the corresponding device in */dev*. If a major number is not specified, a device needs to be created in */dev* with the major number selected by the **ml** command. The major number selected by the **ml** command can be found by using the **ml list** command described below. For more information about creating devices, see the **mknod**(1M) man page entry.

If a module is loaded successfully by the **ml** command, an *id* number, which can be used to unload the module, is returned.

Register

A register command is used to register a module for loading when its corresponding device is opened. When a module is registered, a stub function is entered into the appropriate kernel switch table. When the corresponding device is opened, the stub function loads the module into the kernel.

A module can be registered with the following **lboot** command:

```
% lboot -R <master>
```

A module can also be registered with the following **ml** command:

```
% ml ld [-v] -[cbBfm] module.o -p prefix [-s major...] \
 [-a modname] [-t autounload_delay]
```

If a module is registered successfully with the *ml* command, an *id* number, which can be used to unregister the module, is returned.

Unload

A module can be unloaded only if it provides an “unload” entry point, described in “Module Entry Points.” A module can be unloaded using the following **lboot** command:

```
% lboot -U <id>
```

A module can also be unloaded using the following **ml** command:

```
% ml unld id [id id ...]
```

Unregister

A module can be unregistered with the following **lboot** command:

```
% lboot -W id [id id ...]
```

A module can also be unregistered with the following **ml** command:

```
% ml unreg id [id id ...]
```

List

Loaded and / or registered modules can be listed with the following **lboot** command:

```
% lboot -V
```

Loaded and/or registered modules can also be listed with the following **lboot** command:

```
% ml list [-rlb]
```

For more information, see the **lboot**(1M) and **ml**(1M) man page entries.

Master File Configuration

If a dynamically loadable module has an associated master file, the master file must include a *d* in the *FLAG* field. The *d* flag indicates to **lboot** that the module is a dynamically loadable kernel module. If the *d* flag is present, **lboot** parses the module's master file but does not fill in the entry in the corresponding kernel switch table for the module. All global data defined in the master file are included in the *master.c* file generated by **lboot**. The kernel must be configured with master files that contain the *d* option for each module that will be a dynamically loadable module, if **lboot** is used to load, register, unload, unregister, or autoregister the module. If the **ml** command is used, then it is not necessary to create a master file for the module.

Auto Registration

Loadable modules can be registered by **lboot** automatically at system startup, when **autoconfig** is run. For a module to be auto-registered, its master file must contain an *R* in the *FLAG* field in addition to *d*, which indicates that the module is loadable. When **lboot** runs at system startup, it registers each module that contains an *R* in its master file.

Auto Unload

All registered modules that include an **unload** routine are automatically unloaded after last close unless they have been configured not to do so. By default, modules are unloaded five minutes after last close. You can change the default auto-unload delay by using **systune** to modify the *module_unld_delay* variable. For more information about **systune**, see the **systune** (1M) man page entry. To configure a particular module with a specific auto-unload delay, use the **ml** command. To configure a module so that it is not auto-unloaded, place an *N* in the flags filed of its *master.d* file, if it is registered using **lboot**; otherwise, use **ml** to register the module, then use the -*t* option.

Kernel Configuration

A kernel that supports loadable modules must be configured so that the kernel switch tables generated by **lboot** contain “extra” entries for the loadable modules. Extra entries are generated by **lboot** based on the values of the kernel-tunable parameters shown in Table 11-1.

Table 11-1 Kernel-tunable Parameters

Name	Default	Minimum	Maximum
bdevsw_extra	21	1	254
cdevsw_extra	23	3	254
fmodsw_extra	20	0	0
vfssw_extra	5	0	0

These tunable parameters are found in the */var/sysgen/mtune/kernel* kernel file and are set to the defaults listed above. It is not necessary to change the defaults unless you want the kernel to allow a greater or lesser number of loadable modules. For more information about changing tunable parameters, see the **mtune**(4) and **systune**(1M) manual entries or Chapter 11 of the *IRIX Advanced Site and Server Administration Guide.*

Module Entry Points

Loadable device drivers must conform to the UNIX System V Release 4 *DDI/DKI* standard. In addition to the entry points specified by the standard, if a loadable module is to be unloaded, the module needs to contain an unload entry point:

```
int drvunload (void)
```

An **unload**() routine should be treated as an interrupt routine. It should not call any routines that would cause it to sleep, such as **biowait**(), **sleep**(), **psema**(), or **delay**().

Module Initialization

After a module is loaded and linked into the kernel, and sanity checking is done, the module's initialization routines, *drv***init**(), *drv***edtinit**(), and *drv***start**(), are called, if they exist. For more information on these routines, refer to the *SVR4 DDI/DKI Reference Manual* and the *IRIX Device Driver Reference Pages*.

Edt Type Drivers

Drivers that have an *edt***init** entry point are passed a pointer to an *edt* structure. You must use **lboot** to load these drivers. Add a vector line to the *system* file for the driver. When the module is loaded using **lboot**, **lboot** parses the vector line from the *system* file to create an *edt* structure, which is passed through the kernel and to the driver's *edt***init** routine. The driver's *edt***init** routine is called after the driver is successfully loaded, after the driver's **init** routine is called. For more information, see the *system*(4) man page.

Library Modules

A library module is a module that contains a collection of functions and data that other loaded modules can link against. A library module that contains an **init** function calls it automatically after the module is loaded and linked into the kernel. To load a library module, use the **ml** command:

```
% ml ld [-v] -[l] library.o
```

A library module must be loaded before other modules that link against it are loaded. Library modules cannot be unloaded, registered, or unregistered. Only regular *COFF* object files are supported as loadable library modules.

Loadable Modules and Hardware Inventory

Many device drivers add to the hardware inventory in their **init** or *edt***init** routines. If a driver is a dynamically loadable driver and is auto-registered, it will not show up in the hardware inventory until the driver has been loaded on the first open of the corresponding device. If a clean install or a diskless install is done, a */dev* entry will not be created by **MAKEDEV** for such a driver, since it does not appear in the hardware inventory. If this situation arises, the *D master.d* flag can be used to indicate that the driver should be loaded, then unloaded, by **autoconfig**. If the *R master.d* flag, which indicates that the driver should be auto-registered, is also used, then the driver will be auto-registered as usual. A startup script can then be added to run **MAKEDEV** after **autoconfig**, if necessary. For an example, see the startup script in */etc/init.d/chkdev.*

Run-time Symbol Table

lboot creates a run-time symbol table from the tables in *master.d/rtsymtab* and *master.d/*.exports*. The run-time symbol table contains kernel routines and global data that modules can link against. Only routines and globals that are always present in the kernel should be added to *master.d/rtsymtab.*

If a module contains a routine or global that could be configured out of the kernel, add it to an *xxx.exports* file that will not be included in the kernel if the module is configured out. See *master.d/idev.exports* for an example.

If a loadable module contains globals in its *master.d* file, you must create an *xxx.exports* file to include those globals so that they can be added to the run-time symbol table. For more information, see *master.d/rtsymtab.*

Debugging Loadable Kernel Modules

symmon supports debugging of loadable kernel modules. **symmon** commands that do a symbol table lookup, such as *brk, lkup, lkaddr, hx*, and *nm*, also search the symbol tables created for loadable modules. The **msyms** command can also be used to list the symbols for a particular loaded module:

```
% msyms id
```

Use the *mlist* command to list correctly loaded and registered modules:

```
% mlist
```

For more information about using **symmon**, refer to Chapter 10, “Driver Installation and Testing”.

The **ml** command contains a debug option that can be used to turn verbose error reporting on or off; it may also be used to disable the loading and registering of modules:

```
% ml debug [-vns]
```

-*v*	the *verbose* option turns verbose error reporting on.
-*n*	*no load or register* disallows the loading or registering of modules.
-*s*	*silence* silences verbose error reporting and enables loading a registering of modules.

Load/Register Failures

If a registered module fails to load, unregister and reload it with **ml**, **ld**, or **lboot -L** to get a more detailed error message about the failure. The kernel will fail to load or register a module for any of the following reasons:

- The major number specified either in the master file or by the *ml* command is already in use.
- The object file is not compiled with the correct options.
- The module is an "old style" driver, with either *xxxdevflag* set to *D_OLD*, or if no *xxxdevflag* exists in the driver.
- If the object file is corrupted, it may cause "invalid JMPADDR" errors from the relocation code in the kernel.
- The kernel did not resolve all of the module's symbols.
- All major numbers are in use.
- The switch table is full.
- Required entry points for the particular type of module are not found in the loaded object file.

Example 1

The following example lists the steps necessary to build a kernel and load a character device driver, called *dlkm*, using **lboot**:

1. Add *d* to the *dlkm* master file:

*Flag	Prefix	Soft	#Dev	Dependencies
cd	dlkm	38	2	

2. Make sure that the *cdevsw_extra* kernel tuneable parameter allows for extra entries in the *cdevsw* table. The default settings in */var/sysgen/mtune/kernel* are:

cdevsw_extra	23	3	254

   The **systune** command also lists the current values of all of the tunable parameters. If the kernel is not configured to allow extra entries in the *cdevsw* table, use the **systune** command to change the *cdevsw_extra* parameter:

   ```
   # systune -i
   systune-> cdevsw_extra 3
   systune-> quit >
   ```

3. Build a new kernel and boot the target system with the new kernel.
4. Compile the *dlkm.c* driver:

   ```
   # cc -non_shared -coff -G0 -r -d -Wc,-jalr -c dlkm.c
   ```

5. Copy *dlkm.o* to */var/sysgen/boot*.
6. Load the driver into the kernel:

   ```
   # lboot -L dlkm
   ```

7. List the currently loaded modules to verify that the module is loaded:

   ```
   # lboot -V
   ```

Example 2

The following example lists the steps necessary to load a character device driver called *dlkm*, using the **ml** command:

1. Make sure that the *cdevsw_extra* kernel tunable parameter allows for extra entries in the *cdevsw* table. The default settings in */var/sysgen/mtune/kernel* are:

cdevsw_extra	23	3	254

   The **systune** command also lists the current values of all of the tunable parameters. If the kernel is not configured to allow extra entries in the *cdevsw* table, use the **systune** command to change the *cdevsw_extra* parameter:

   ```
   # systune -i
   systune-> cdevsw_extra 3
   systune-> quit >
   ```

2. Compile the *dlkm.c* driver:

   ```
   # cc -non_shared -coff -G0 -r -d -Wc,-jalr -c dlkm.c
   ```

3. Load the driver into the kernel:

   ```
   # ml ld -c dlkm.o -p dlkm -s38
   ```

4. List the currently loaded modules to verify that the module was loaded:

   ```
   # ml list
   ```

System-specific Issues

This appendix lists the Silicon Graphics functions available for writing device drivers and how those function differ from the functions listed in the *IRIX Device Driver Reference Pages* .

It contains the following sections:

Although all Silicon Graphics systems share similar architectural elements, there are several significant differences you must recognize when writing device drivers. Despite these differences, it is possible to write a device driver that runs on all Silicon Graphics systems. This appendix outlines the various CPU types used by Silicon Graphics systems and describes the CPU features that can vary. Also listed are the VME-bus addresses and interrupt vectors available for customer use.

Whenever possible, this appendix promotes the use of those IRIX kernel functions that are supported on all Silicon Graphics architectures. The hardware features that can differ across architectures are:

- Data cache write back and invalidation
- Write buffer flushing
- Hardware spinlocks (test and set variables)

CPU Types

This appendix describes the CPU types used in Silicon Graphics workstations and servers. These CPU types are summarized in Table A-1.

Table A-1 CPUs Used in Silicon Graphics Computer Systems

CPU Type	Processor	Clock Speed	System/Series
IP4	R2000	8 / 12.5MHz 33 MHz	4D / 50, 4D / 70, and 4D / 80 systems (Originally known as Single-Board Computer or “SBC”)
IP5	R2000	16.7 MHz	4D / 100 series multiprocessor systems
IP6	R2000	12.5 / 20 MHz	Personal IRIS (4D / 20, 4D / 25) systems
IP7	R3000	25 / 33 MHz	4D / 200 series multiprocessor systems
IP9	R3000	25MHz	4D / 210 series systems
IP11	R3000	33 MHz	4D / 300 series systems
IP12	R3000	30 MHz	4D / 30, 4D / 35 and older Indigo series systems. Indigo series systems have no VME interface, but they do support GIO.
IP15	R3000	40 MHz	4D / 400 series systems
IP17	R4000 R4400	100 MHz 150MHz	Crimson series single-processor systems
IP19	R4400	150 MHz	CHALLENGE L / XL and Onyx multiprocessor systems
IP20	R4000	100 MHz	Newer Indigo series systems
IP21	R8000	75 MHz	POWER CHALLENGE and POWER / Onyx multiprocessor systems
IP22	R4000 R4000 R4400 R4400 R4400 R4600	100 MHz 100 MHz 150 MHz 150MHz 150MHz 133 MHz	Indigo^2 systems (with GIO and EISA) Indy systems Indigo^2 systems (with GIO and EISA) Indy systems (with GIO but not EISA) Indigo systems Indigo systems
IP26	R8000	75 MHz	POWER CHALLENGE M and POWER Indigo^2 systems

To determine which CPU a Silicon Graphics system uses, type:

```
% uname -m
```

at a shell prompt. This command reports the CPU type. For details on this command, see the **uname**(1) man page.

For a more detailed listing of a Silicon Graphics system's hardware configuration, type:

```
% hinv
```

at a shell prompt. See the **hinv**(1M) man page.

Data Cache Write Back and Invalidation

All CPUs use the MIPS R2000, R3000, R4000, or R8000 (formerly known as "TFP") series processor chips (R4400 and R4600 are the same as the R4000 for all practical purposes). These chips use a data cache to maximize the efficiency of fetching of heavily used memory.

The IP5, IP7, IP11, IP15, IP19, and IP21 multiprocessor CPUs have *bus-watching caches* that automatically invalidate the data cache when the system performs DMA (direct memory access) into physical memory. For these CPUs, no data cache write back or invalidation is required by software because these functions are performed by hardware.

DMA operations are categorized as *DMA reads* or *DMA writes*. DMA operations that transfer from memory to a device, and hence read memory, are DMA reads. DMA operations that transfer from a device to memory are DMA writes. Thus, you may want to think of DMA operations as being named from the point of view of what happens to memory.

The single-processor CPUs based on the R2000 or R3000 employ a cache architecture known as a *write through cache*. This means that all stores generated by the processor to a cached memory location go into the cache and into memory at the same time. Therefore, the data caches on these systems never contain data that is more recent than memory. However, after the system performs DMA to physical memory, the data lines in the processor's cache corresponding to this physical memory contain data that is *stale* with respect to memory. Therefore, a driver running on a single-processor CPU must explicitly invalidate the data cache before reading from the corresponding cached address, after the DMA completes.

R4000s and R8000s employ a cache architecture known as a *write back cache*. This means that stores generated by the processor go only into cache; they are written back from cache to memory only when a cache miss causes that cache line to be replaced. For this type of cache, the cache contains data that is newer than the corresponding memory locations. Memory is then *stale* with respect to the cache. Drivers that perform DMA reads from memory to device must specifically cause the cache to be written back to memory before the DMA starts. On IP19 and IP21 platforms, DMA pulls data from the processor caches, if necessary, thus providing *coherent I/O*.

Recall the code examples that read kernel data in Chapter 3, “Writing a VME Device Driver,” and Chapter 5, “Writing a SCSI Device Driver.” Before reading the data, the driver code used the **dki_dcache_inval**() function to invalidate the appropriate data cache lines. When the data cache lines are invalidated, accessing the kernel data causes a cache miss and, thus, forces a read from physical memory. Therefore, to ensure driver portability, your driver must always use **dki_dcache_inval**() to invalidate the data cache. This is the case even though the **dki_dcache_inval**() functions defined for the IP5 and IP7 use stub functions that do not do any actual work, although they use **dki_dcache_wb**() before starting a DMA from memory to device. See Table A-2 for a summary of cache line sizes for various MIPS processors.

If your driver uses the functions **userdma**() or **physio**() (**physio** calls **userdma**() internally), the data cache is automatically written back and invalidated for you no matter what system you are using. If your driver does not use these functions for a DMA write into cached memory, your driver must use **dki_dcache_inval**() to invalidate the data cache explicitly after the DMA completes.[1] Further, if your driver does a DMA read from memory to device, it must use **dki_dcache_wb**() to write back the data cache explicitly before the DMA is started.

[1] On systems with write through caches, and on IP5, IP7, IP9, IP11, and so on, the **dki_dcache_inval**() functions are stub functions that perform no actual work.

Table A-2 Cache Line Sizes by Processor Type

	Primary Cache			Secondary Cache		
Processor Type	**Size (D/I)**	**Type**	**Line Size (D/I)**	**Size (D/I)**	**Type**	**Line Size (D/I)**
R3000 (IP12)	32K / 32K	D	4 / 64	None	None	None
R3000 (IP7)	64K / 64K	D	4 / 64	None	None	None
R4000PC (IP20)	8K / 8K	D	32 / 32	None	None	None
R4000SC (IP17)	8K / 8K	D	32 / 32	1-4 MB	D	128 / 128
R4400MP (IP19)	16K / 16K	D	32 / 32	1-4 MB	D	128 / 128
R4600PC (IP22)	16K / 16K	2	32 / 32	None	None	None
R4600SC (IP22)	16K / 16K	2	32 / 32	512K	2	128
R8000 (IP21)	16K / 16K	D	32 / 32	4 MB	4	512
R8000 (IP26)	16K / 16K	D	32 / 32	2 MB	4	128

Another consideration worth mentioning is that of buffer alignment for DMA. The R4000 processor implements a secondary cache line size of between 4 and 32 words (the secondary cache line size is dependent upon the CPU board implementation). Buffers used for DMA must be aligned on a byte boundary that is equal to the cache line size. To accomplish this, use the **kmem_alloc**() function with the *KM_CACHEALIGN* flag. This returns a buffer with the necessary alignment for the system.

Note: The R8000 has the same DMA alignment problems in general as the R4000. This is true for all systems with write back caches.

Why is this alignment necessary? Suppose you have a variable, *X*, followed by a buffer you are going to use for DMA write. If you invalidate the buffer prior to the DMA write, but then reference the variable *X*, the resulting cache miss brings part of the buffer back into the cache. When the DMA write completes, the cache is stale with respect to memory. If, however, you invalidate the cache after the DMA write completes, you destroy the value of the variable *X*.

Flushing the Write Buffer

You may, in some cases, want to call **flushbus**() to ensure that any writes in the write buffer have actually been flushed to the system bus. This is sometimes necessary when a device requires delays between PIOs, particularly between a write and a read, since they might otherwise arrive at the device back-to-back.

For example, you write to the device, delay, write to the device, delay. These writes may be buffered regardless of the delay and still be sent to the device in quick succession.

Registers and Register Optimization

Variables not declared volatile may be optimized to registers in the processor, since there are multiple processors. Multiple inconsistent copies of a variable may exist in registers if volatile is not declared.

Note: Use the *-O* compiler flag to turn optimization on. The *-g* compiler flag for debugging disables optimization.

Reliable Multiprocessor Spinlocks

The multiprocessor CPUs implement test-and-set variables in hardware. These variables are known as *spinlocks*. Your code can use functions such as **LOCK_ALLOC**(D3) and **LOCK**(D3) to take advantage of these variables to protect a critical region of code or data on one processor from interference from other processors.

All kernels for all Silicon Graphics CPUs support the spinlock functions (although these functions perform no actual work on single-processor systems). Therefore, a fully semaphored multiprocessor device driver can run unmodified on a single-processor system.

Note: Because IRIX kernels cannot, as a rule, be preempted, any driver that sits in a loop waiting for some condition to be satisfied may tie a processor up for as long as it wants. Real-time processes, such as audio, are very sensitive to such delays.

VME Slave Addressing

Recall that these base addresses are passed to the driver through the device **edtinit**() function. Therefore, while you may need a different *system* file for each Silicon Graphics platform, the driver code itself can still be portable.

Table A-3, Table A-4, and Table A-5 show the mapping of kernel virtual address, physical address, and VME-bus address for each addressing mode. For IP6 and IP12 systems, block mode and burst mode transfers are supported only during DMA, where the VME device is the master. No predefined addresses are available on CHALLENGE/Onyx series systems.

Note: When consulting Table A-3 through Table A-7, note that VME space is set up by software on various hardware platforms. Always use the macro #defines in */usr/include/sys/vmereg.h* to refer to the various VME resource address ranges by their logical names.

Table A-3 A24 Kernel/VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address VME0	Kernel Address VME1	Size (MB)	VME Address Range
IP4/6/12	0x3D	0xBC000000 – 0xBCFFFFFF	NA	16	0x0 – 0xFFFFFF
IP4/6/12	0x39	0xBE000000 – 0xBEFFFFFF	NA	16	0x0 – 0xFFFFFF
IP5/7/9/17	0x39	0xB2000000 – 0xB2FFFFFF	F0000000 – F0FFFFFF	16	0x0 – 0xFFFFFF
IP5/7/9/17	0x3D	0xB3000000 – 0xB3FFFFFF	F1000000 – F1FFFFFF	16	0x0 – 0xFFFFFF

Table A-4 A32 Kernel / VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address	Size (MB)	VME Address Range
R2300	0x09	0xB8000000 – 0xBBFFFFFF	192	0x18000000 – 0x1BFFFFFF
IP4 / 6 / 12	0x09	0xB0000000 – 0xBBFFFFFF	192	0x10000000 – 0x1BFFFFFF
IP5 / 7 / 9 / 17	0x09	0xD0000000 – 0xDFFFFFFF	256	0x10000000 – 0x1FFFFFFF
IP5 / 7 / 9 / 17	0x0D	0xE0000000 – 0xEFFFFFFF	256	0x10000000 – 0x1FFFFFFF

Table A-5 Dual VME-bus Address Mapping

System CPU	VME Modifier	Kernel Address VME0	Kernel Address VME1	Size (MB)	VME Address Range
IP5 / 7 / 9 / 17	0x9	0xD8000000– DFFFFFFF	D0000000– D7FFFFFF	128 128	18000000– 1FFFFFFF
IP5 / 7 / 9 / 17	0xD	0xE8000000– EFFFFFFF	E0000000– E8000000	128 128	18000000– 1FFFFFFF

VME Master Addressing

When a VME device uses DMA to access main memory, it acts as a "VME master." Silicon Graphics systems support both A24 and A32 VME master addressing. Although there are a few minor differences in the addressing modes supported (non-privileged versus supervisor), the main difference is that the 4D / 100, 4D / 200, 4D / 300, 4D / 400, and Crimson series systems support dynamic address mapping. This allows IP5, IP7, IP9, and IP17 CPUs to access all of physical memory for A24 addressing and allows scatter-gather for both A24 and A32 addressing.

You can still write an IRIS-portable device driver if the **dma_mapalloc**() function does not return –1 and if the device driver can take advantage of the DMA mapping functions described in Chapter 5, "Writing a SCSI Device Driver." Otherwise, you must use one of the other DMA methods.

Table A-6 describes the mapping between the address generated by a VME device and physical memory. You can perform A32 master addressing on the IP5, IP7, IP9, and IP17 in either mapped or unmapped mode.

Table A-6 A32 VME-bus/Physical Address Mapping

System CPU	VME Address	Physical Address	Size (MB)	VME Modifier	Map
IP4/6/12	0x0 – 0x0FFFFFFF	0x0 – 0x0FFFFFFF	256	0x9	No
IP5/7/9/17	0x0 – 0x0FFFFFFF	0x0 – 0x0FFFFFFF	256	0x9	No
IP5/7/9/17	0x80000000 – 0x8FFFFFFF	0x – 0x0FFFFFFF	256	0x9	Yes
IP19/21	Must be mapped by the driver	Must be mapped by the driver			

On POWER Series workstations, ranges of VME-bus address space were mapped one-to-one with K2 segment addresses. This made accessing the VME bus easy but was also limiting. Only a small amount of K2 space is available for use by VME, so very little of the VME address space was made available. Even worse, for dual VME-bus systems, the space previously available was now cut in half as it was shared between the two buses.

The CHALLENGE series supports up to five VME buses. Since K2 space is a limited resource, and dividing up what is available by five would make the extra VME buses next to useless, a new approach was tried. The CHALLENGE series does not have a direct K2 address map into VME-bus space. Each VME bus adapter has the ability to map fifteen 8 MB windows of VME-bus space into K2 space. These windows can be slid around at will to give the illusion of a much larger address space.

To access a VME space, a user must allocate a PIO map, which provides a translation between a kernel address and VME space. These mappings can be “FIXED” or “UNFIXED.” As on POWER Series platforms, a FIXED mapping is a one-to-one mapping of a range of VME-bus space into the driver’s address space. An UNFIXED window takes advantage of the sliding window ability on the CHALLENGE series, which supports both FIXED and UNFIXED mappings.

In an UNFIXED map, VME-bus space cannot be accessed directly; instead, access is provided through special **bcopy**() routines used to move data between VME space and kernel buffers. While it is not always possible to get a FIXED mapping, an UNFIXED mapping is always available. The special **bcopy**() routines work for both FIXED and UNFIXED mappings. On POWER Series and earlier workstations, UNFIXED mappings are treated as FIXED mappings.

The PIO mapping routines also have a general interface that allows them to be used for mapping in bus spaces other than VME.

The support routines for PIO mapping are:

pio_mapalloc	Allocate a PIO map.
pio_mapaddr	Map bus space to a driver accessible address (FIXED maps only).
pio_mapfree	Free a previously allocated PIO map.
pio_badaddr	Check to see whether a bus address is equipped.
pio_wbadaddr	Check to see whether a bus address is equipped.
pio_bcopyin	Copy data from bus space to kernel buffer.
pio_bcopyout	Copy data from kernel buffer to bus space.

These PIO maps are normally set up in the driver’s *drv***edtinit**() routine.

VME-bus Space Reserved for Customer Drivers

Table A-7 shows the VME-bus space reserved for customers.

Table A-7 VME-bus Space Reserved for Customer Drivers

Space	VME Modifier	VME Address
A16	0x2D	0x6000 – 0x7FFF
A16	0x29	0x6000 – 0x7FFF
A24	0x3D	0x800000 – 0x9FFFFF
A24	0x39	0x000000 – 0xFFFFFF
A32	0x09	0x1A000000 – 0x1BFFFFFF
A32	0x09	0x1E000000 – 0x1FFFFFFF (IP5/7/9/17 Only)
A32	0x0D	0x1E000000 – 0x1BFFFFFF (IP5/7/9/17 Only)
A32	0x09	0x20000000 – 0x3FFFFFFF (IP19 and IP21 only)
A32	0x0D	0x20000000 – 0x3FFFFFFF (IP19 and IP21 only)

Note: For information that may not have been available when this guide went to press, refer to the comments in the */var/sysgen/system/irix.sm* file.

POWER Indigo^2 and POWER CHALLENGE M Drivers

For POWER Indigo^2 or POWER CHALLENGE M processors, uncached (K1 segment) writes to main memory must be double-word (64-bit) writes, and they must be aligned on a double-word boundary. The reason for this has to do with the POWER Indigo^2's hardware support for ECC-protected memory. Since the ECC calculation requires that data be written to main memory in 64-bit (8-byte) chunks, smaller (1-, 2-, and 4-byte) uncached writes to main memory tend to produce memory corruption. Cached accesses (both read and write), as well as uncached read operations do *not* cause problems; neither do accesses that are not to main memory, such as device register reads and writes.

Any device driver that performs uncached writes to main memory must always do writes in 8-byte quantities.[1] If the driver needs to write a smaller piece of memory, then it must do the write as a read-modify-write operation of an 8-byte piece of memory. For example, a driver with code like the following (assuming that the structure is being accessed uncached), would corrupt memory:

```
struct foo_s {
    char    b0;
    char    b1;
    char    b2;
    char    b3;
    char    b4;
    char    b5;
    char    b6;
    char    b7;
} bar1;

driver_write_bar1_b3 ()
{
    bar1.b3 = 5;
}
```

Instead, it would have to be modified to perform a read-modify-write operation on the whole double-word containing *b3*, as in the following example (on a big-endian system):

[1] Access to device registers does not fall under this restriction.

```
struct foo_s {
    union {
        __uint64    dw;    /* A 64-bit quantity */
        struct {
            char    b0;    /* READ ONLY */
            char    b1;    /* READ ONLY */
            char    b2;    /* READ ONLY */
            char    b3;    /* READ ONLY */
            char    b4;    /* READ ONLY */
            char    b5;    /* READ ONLY */
            char    b6;    /* READ ONLY */
            char    b7;    /* READ ONLY */
        } byte;
    } dw0;
} bar1;

driver_write_bar1_b3 ()
{
    bar1.dw0.dw = (bar1.dw0.dw & ~(0xff<<32)) | (5<<32);
}
```

If at all possible, use cached accesses to memory, along with cache coherency operations where necessary, instead of uncached operations. Cache coherency operations are necessary only for data shared by the CPU and a device that needs to perform DMA. See “Data Cache Write Back and Invalidation” on page 321 for more information on use of cache coherency operations.

Sharing Data Between CPU and Peripheral Devices

To avoid memory contamination, the CPU and any other device capable of DMA must not both be allowed to write data to separate parts of a double-word in memory. Since CPU accesses to smaller portions of a double-word are performed as read-modify-write operations, they are not atomic, and could be interleaved with data written by a device between the CPU read operation and the CPU write operation. Make sure that data written by a device does not fall within the same double-word as data written by the CPU.

Using mmap()

If you have a driver that allows a process to access main memory uncached through the **mmap**() call, that process must not write the memory using operations that write less than 8 bytes at a time. Again, this restriction does not apply to accesses to device registers, or to accesses to cached memory.

Appendix B

SCSI Controller Error Messages

This appendix lists many common error messages. It contains the following sections:

This appendix lists the error strings printed by the device drivers *tpsc*, *dksc*, and, in some cases, *devscsi*.

Introduction

In early IRIX releases, the differential SCSI dual-channel controller board and the *dksc* driver printed the information differently; in IRIX 4.0.1, they began to use the same form. In IRIX 5.x and later releases, the differential SCSI dual-channel controller board driver reports the error message in the same format as the integral SCSI controller driver.

The error message format for IRIX 3.x and 4.x was:

`sense codes.` **key**`%x` **asc**`%x` **asq**`%x`

Arguments

key	the number from Table B-1.
asc	(additional sense code) from Table B-2.
asq	(additional sense qualifier) sometimes provides additional information.

Note: Sometimes, there is only one possible *asq* for a given *asc*, and many SCSI devices return nonstandard *asq* values.

The *asq* tends to be more vendor-specific, although the IEEE SCSI 2 specification defines the "standard" sense qualifiers.

For IRIX 5.x and 6.x, the *integral* SCSI controller on your system normally prints messages in the forms below, corrected for the two Western Digital bus controllers:

```
WD93 Bus # tarsct # lun # message
```

or

```
wd95_(bus)d(target); sense key {num} ({string}) asc{num}
```

Arguments

- the first *#* (or *d* for the wd95) is the SCSI adapter involved (0 for all systems except those with the IO3 (input/output board), which supports up to four adapters, numbered 0-3).
- the second *#*, *#* pair is printed only if you know which device is causing the problem.

In a number of cases, a phase and, possibly, a state are printed. These error codes come from the files */usr/include/sys/scsidev.h* and */usr/include/sys/scsi.h*.

The state and phase meanings are listed in Table B-5 and Table B-6. A few comments have been added. Some of the messages are also included.

Sense Key Information

Table B-1 and Table B-2 map error codes to sense keys.

Table B-1 Primary Sense Key Information

Message	Sense Key	Most Common Cause(s)
`No sense`	0x0	No error information available
`Recovered error`	0x1	The device recovered by itself
`Device not ready`	0x2	No media or not spun up
`Media error`	0x3	An actual media problem
`Device hardware error`	0x4	Usually a device hardware error
`Illegal request`	0x5	Invalid command or data issued
`Unit attention`	0x6	Device was reset or power-cycled
`Data protect error`	0x7	Usually device is write protected
`Unexpected blank media`	0x8	Tried to read at end of a tape
`Vendor unique error`	0x9	Varies
`Copy aborted`	0xa	Copy cmd aborted by host (not used)
`Aborted command`	0xb	Target aborted command
`Search data successful`	0xc	Search data command OK (not used)
`Volume overflow`	0xd	Tried to write past EOT on tape
`Reserved`	(0xE)	0xe should not be seen
`Reserved`	(0xF)	0xf should not be seen

While Table B-1 helps to identify an error, Table B-2 provides further information on the cause of an error. The ASQ (additional sense qualifier) is printed numerically when its value is not 0 (in 4.0; in 3.3.3, it is always printed by the differential SCSI dual-channel controller). Missing numerical values are not printed either because they are not defined or because the drivers treat them specially.

Table B-2 is provided primarily so you can look up the additional sense codes in the device manual. Some are self-explanatory, others quite obscure.

Table B-2 Additional Sense Code

Addition Sense Qualifier Message	**Additional Sense Code**
`No index/sector signal`	0x01
`No seek complete`	0x02
`Write fault`	0x03
`Not ready to perform command`	0x04
`Unit does not respond to selection`	0x05
`No reference position`	0x06
`Multiple drives selected`	0x07
`LUN communication error`	0x08
`Track error`	0x09
`Error log overflow`	0x0a
`Write error`	0x0c
`ID CRC or ECC error`	0x10
`Unrecovered data block read error`	0x11
`No address mark found in ID field`	0x12
`No address mark found in Data field`	0x13
`No record found`	0x14
`Seek position error`	0x15
`Data sync mark error`	0x16

Table B-2 (continued) Additional Sense Code

Addition Sense Qualifier Message	**Additional Sense Code**
`Read data recovered with retries`	0x17
`Read data recovered with ECC`	0x18
`Defect list error`	0x19
`Parameter overrun`	0x1a
`Synchronous transfer error`	0x1b
`Defect list not found`	0x1c
`Compare error`	0x1d
`Recovered ID with ECC`	0x1e
`Invalid command code`	0x20
`Illegal logical block address`	0x21
`Illegal function`	0x22
`Illegal field in CDB`	0x24
`Invalid LUN`	0x25
`Invalid field in parameter list`	0x26
`Media write protected`	0x27
`Media change`	0x28
`Device reset`	0x29
`Log parameters changed`	0x2a
`Copy requires disconnect`	0x2b
`Command sequence error`	0x2c
`Update in place error`	0x2d
`Tagged commands cleared`	0x2f
`Incompatible media`	0x30
`Media format corrupted`	0x31

Table B-2 (continued) Additional Sense Code

Addition Sense Qualifier Message	Additional Sense Code
`No defect spare location available`	0x32
`Media length error`	0x33[a]
`Toner/ink error`	0x36
`Parameter rounded`	0x37
`Saved parameters not supported`	0x39
`Medium not present`	0x3a
`Forms error`	0x3b
`Invalid ID msg`	0x3d
`Self config in progress`	0x3e
`Device config has changed`	0x3f
`RAM failure`	0x40
`Data path diagnostic failure`	0x41
`Power on diagnostic failure`	0x42
`Message reject error`	0x43
`Internal controller error`	0x44
`Select/reselect failed`	0x45
`Soft reset failure`	0x46
`SCSI interface parity error`	0x47
`Initiator detected error`	0x48
`Inappropriate/illegal message`	0x49
`Command phase error`	0x4a
`Data phase error`	0x4b
`Failed self configuration`	0x4c
`Overlapped commands attempted`	0x4e

Table B-2 (continued) Additional Sense Code

Addition Sense Qualifier Message	**Additional Sense Code**
`Media load/unload failure`	0x53
`Unable to read table of contents`	0x57
`Generation (optical device) bad'`	0x58
`Updated block read (optical device)`	0x59
`Operator request or state change`	0x5a
`Logging exception`	0x5b
`RPL status change`	0x5c
`Self diagnostics predict unit will fail soon`	0x5d
`Lamp failure`	0x60
`Video acquisition error/focus problem`	0x61
`Scan head positioning error`	0x62
`End of user area on track`	0x63
`Illegal mode for this track`	0x64
`Decompression error`	0x70[b]

a. Specified as tape only

b. DAT only; may be in SCSI3

SCSI Driver Error Messages

Table B-3 lists the messages that are printed by the *wd93* SCSI driver. (Messages for the wd95 SCSI driver are similar). After the message is printed, the driver resets the SCSI bus. These messages are from IRIX 5.1, but similar ones are printed by earlier releases.

Table B-3 SCSI Driver Error Messages

Error Message	Comments

`No memory for wd93 device array`
`Not enough memory for WD93 data structures`
`Not enough memory for WD93 DMA maps`
	These messages occur during boot if something is seriously wrong, and memory can't be allocated.

`wd93 SCSI Bus=%d ID=%d LUN=%d: error during abort message, resetting bus`	An upper level driver tried to issue an ABORT message, but the expected bus phases were not followed.
`wd93 SCSI Bus=%d ID=%d LUN=%d: SYNC negotiation error, resetting bus`	An error occurred while trying to negotiated synchronous SCSI rates (usually during an open or mount); the device is left in async mode.
`timeout after %d %ssec`	Any SCSI command that doesn't terminate within the time limit set by the upper level driver will result in this message and a SCSI bus reset. The “%ssec” part will either be “msec” or “sec”, depending on whether it is an integral number of seconds or not (timeouts are passed as HZ values). This can be caused by anything from driver errors to hardware errors to SCSI bus problems. The latter is the most common cause. This message is always paired with the standard “wd93 SCSI Bus=%d...” message.
`wd93 controller %d didn't reset correctly`	An attempt to reset the controller chip failed; this is a catastrophic (usually) hardware error.

Table B-3 SCSI Driver Error Messages

Error Message	Comments
`wd93 SCSI Bus=%d ID=%d LUN=%d: SCSI cmd=0x%x <MSG>. Resetting SCSI bus wd93 SCSI Bus=%d: <MSG>. Resetting SCSI bus`	Used with a number of other messages when a SCSI bus timeout or other error on the SCSI bus occurs. The SCSI bus is reset. The long form (with target, and the first byte of the SCSI command) is shown when the driver is connected to a known target. The short form is shown when no target is connected (referred to as "cmdabort" in other messages).
`Spurious wd93 interrupt, no connected channel`	Occurs when the driver is responding to a SCSI bus phase where some device should be connected (active), but in fact, none is.
`wd93 SCSI Bus=%d ID=%d LUN=%d: SCSI bus parity error`	A SCSI bus parity error was detected during a data transfer. Usually a cabling problem.
`wd93 SCSI Bus=%d ID=%d LUN=%d: host memory parity error during DMA`	On command completion (normal or error), the DMA hardware tells us that a parity error occurred during data transfer to memory. Usually a system hardware problem.

SCSI Driver Debugging Messages

The information listed in Table B-4 is sometimes useful for debugging drivers (kernel or **devscsi**). It dumps out information on the current command and the sense info obtained, after a SCSI "check condition" status. Printed only when the variable *wd93_printsense* is set non-zero in *master.d/ wd93*.

Table B-4 Error Messages Useful for Debugging

Error Message	Comments
`wd93 SCSI Bus=%d ID=%d LUN=%d: check condition start request sense`	The check condition was detected, a request sense is started.
`wd93 SCSI Bus=%d ID=%d LUN=%d: sense failed wd93 status %d, scsi status 0x%x`	The request sense command failed. Usually bad device firmware, or SCSI bus problems.
`wd93 SCSI Bus=%d ID=%d LUN=%d: sense key=0x%x (%s) ASC=0x%x (%s)`	The request sense succeeded, the driver status and the SCSI status are printed. The ASC is printed if valid, and the ascii strings corresponding to the sense key and the ASC (additional sense code) are also printed, if they are known to the driver.
`Hex sense data:`	If `wd93_printsense is > 1`, then the raw data returned by request sense are dumped in hex with this header.
`reselect without ID`	The SCSI bus phase indicates a reselection, but the reselecting device's ID could not be determined. Usually a cabling problem. Printed with the "cmdabort" message.
`illegal disconnection interrupt: phase %x`	A SCSI bus disconnect was detected at an unexpected point. The wd93 phase register (see *sys/wd93.h*) is printed. Printed with the "cmdabort" message.

Table B-4 Error Messages Useful for Debugging

Error Message	Comments
`unexpected message in %x, phase %x`	A SCSI bus message in phase was found, but the message byte was not expected. Printed with the “cmdabort” message.
`Hardware error`	The wd93 reported no active phase, but should have. Normally a hardware problem. Printed with the “cmdabort” message.
`Too much data %s (probable SCSI bus cabling problem)`	Too many REQ's were received for the amount of data programmed into the SCSI controller. Usually a SCSI bus cabling problem, but can also occur when the byte count passed to the wd93 driver doesn't match the way that the device interprets the bytes in the SCSI command. Followed by either “requested” or “sent”, depending on direction of transfer. Printed with the “cmdabort” message.
`Unexpected info phase %x, state %x`	Another unexpected bus phase. The wd93 phase and state registers are printed (see *sys/wd93.h*). Printed with the “cmdabort” message.
`wd93 SCSI Bus=%d ID=%d LUN=%d: Unexpected extended msgin type %x, len %x`	A special case of an unexpected message. Extended messages occur when the first byte is 001; subsequent bytes indicate the length and type. The only extended messages currently handled are synchronous negotiation initiated by the target.
`unexpected reselection`	A reselection of the host was attempted, but the host doesn't think that any target is both disconnected and active. Usually a SCSI bus problem, but might be a firmware bug also. Printed with the “cmdabort” message.

Table B-4 Error Messages Useful for Debugging

Error Message	Comments
`wd93 SCSI Bus=%d ID=%d LUN=%d: I/O address %x not correctly aligned, can't DMA disconnected on non-word boundary (addr=%x, 0x%x left), can't DMA`	Silicon Graphics DMA hardware requires word (32-bit) alignment at start of any DMA (low two bits must be 0 in the address). If not, one of the following two messages is issued, depending on whether this is the start of a command or a data phase continued on a reselection. The latter case can occur if a device disconnects and reselects, even if it doesn't go to data phase, if the DMA count remaining is non-zero. This is because it is too difficult to handle the case where the device disconnects and then reselects just to go to status phase. Such devices are inefficient at best, and, fortunately, are rare. If you must use such a device, your only option is to disable SCSI disconnects altogether (in *master.d/wd93*).
`ID=%d LUN=%d not found in active list`	Typically occurs due to SCSI bus problems or to driver bugs. A reselection occurred with valid data and bus phases at the same time as the driver attempted to select a device to initiate a command, but the reselecting device does not appear to have a command active.

SCSI States and Phases

Several of the SCSI states and phases are listed in Table B-5. There are other possible states and phases, but they rarely occur. The SCSI states and phases are listed in the files */usr/include/sys/wd93.h* and */usr/include/sys/wd95.h* and perhaps in *scsi.h*. The comments below have been extracted from these files and supplemented with additional information.

Note: "Out" is from the CPU to the SCSI device in these descriptions, and "receive" and "send" are also from the SCSI device point of view, since the target controls all the bus phases except for initial selection.

Table B-5 SCSI State Error Messages

State Message	Sense Key	Comments
`ST_RESET`	0x00	SCSI chip reset by reset command or power-up.
`ST_SELECT`	0x11	Selection of target complete (after C93SELATN).
`ST_SATOK`	0x16	Select-And-Transfer completed successfully, that is, all phases have completed in a normal manner.
`ST_TR_DATAOUT`	0x18	Transfer cmd done, target requesting data.
`ST_TR_DATAIN`	0x19	Transfer cmd done, target sending data.
`ST_TR_STATIN`	0x1b	Target is sending status in.
`ST_TR_MSGIN`	0x1f	Transfer cmd done, target sending msg.
`ST_TRANPAUSE`	0x20	Transfer cmd has paused with ACK.
`ST_SAVEDP`	0x21	Save Data Pointers message during SAT normal state when device is disconnecting from the bus.
`ST_A_RESELECT`	0x27	Reselected after disc (93A).
`ST_UNEXPDISC`	0x41	An unexpected disconnect device disconnected without sending a disconnect message; sometimes happens when devices with removable media have had the media removed during a transfer.
`ST_PARITY`	0x43	Cmd terminated due to parity error on the SCSI bus.

Table B-5 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
`ST_PARITY_ATN`	0x44	Cmd terminated due to parity error (ATN is asserted so that host can send a message to device; the transfer is just aborted).
`ST_TIMEOUT`	0x42	Time-out during Select or Reselect, that is, the device never responded to an attempt to select it; normally seen only during hardware inventory probing, but sometimes happens after a SCSI bus reset if device takes a long time to recover from the reset or is powered off.
`ST_INCORR_DATA`	0x47	Incorrect message or status byte.
`ST_UNEX_RDATA`	0x48	Unexpected receive data phase device tried to send more data than the SCSI chip is programmed to expect. This can be OK, as when a high-level request is made to transfer more data than the DMA hardware can map on a single request. In this case, simply reprogram the DMA hardware for the next chunk of data and restart the transfer (but don't send a new SCSI command to the device). When printed as part of an error message, it can sometimes be caused by a SCSI cabling problem, or (particularly with devscsi user drivers) by a mismatch in the byte count given to the driver and the byte count implied by the SCSI command sent to the device.
`ST_UNEX_SDATA`	0x49	Unexpected send data phase (same as above, but device is asking for more data).
`ST_UNEX_CMDPH`	0x4a	Unexpected cmd phase

Table B-5 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
`ST_UNEX_SSTATUS`	0x4b	Unexpected send status phases occur at the end of SCSI command (that is, byte count remaining is 0); if they happen at other times, the chip interrupts. This can happen when you ask a device for more data than it can give you, and in this case, you just return a short I/O count to the caller. When printed as part of an error message, it usually implies a cabling or termination problem.
`ST_UNEX_RMESGOUT`	0x4e	Unexpected request message out phase usually indicates a SCSI cabling problem.
`ST_UNEX_SMESGIN`	0x4f	Unexpected send message in phase usually indicates a SCSI cabling problem; also happens when device sends a disconnect message in normal use when preparing to disconnect from the bus.
`ST_RESELECT`	0x80	WD33C93 has been reselected.
`ST_93A_RESEL`	0x81	Reselected while idle (93A).
`ST_DISCONNECT`	0x85	Disconnect has occurred.
`ST_NEEDCMD`	0x8a	Target is ready for a cmd.
`ST_REQ_SMESGOUT`	0x8e	REQ signal for send message out.
`ST_REQ_SMESGIN`	0x8f	REQ signal for send message in above 3 usually seen only during sync negotiations.

Table B-6 lists phases during Select-and-Transfer commands.

Table B-6 Phases During a Select-and-Transfer Command

Phase Message	Sense Key	Comments
`PH_NOSELECT`	0x00	Selection not successful.
`PH_SELECT`	0x10	Selection successful.
`PH_IDENTSEND`	0x20	Identify message sent (during selection when sending initial command to a device). Phase 30 indicates none of the cmd bytes have yet been sent; every cmd byte sent increments that by one.
`PH_CDB_START`	0x30	Start of CDB transfers.
`PH_CDB_6`	0x36	6th cmd byte sent.
`PH_CDB_10`	0x3a	0xAth cmd byte sent.
`PH_CDB_12`	0x3c	0xCth cmd byte sent.
`PH_SAVEDP`	0x41	Save data pointers.
`PH_DISCRECV`	0x42	Disconnect message received.
`PH_DISCONNECT`	0x43	Target disconnected.
`PH_RESELECT`	0x44	Original target reselected.
`PH_IDENTRECV`	0x45	Correct identify (right LUN) message received (during reselection).
`PH_DATA`	0x46	Data transfer completed (expect status next).
`PH_STATUSRECV`	0x50	Status byte received (expect cmd complete next).
`PH_COMPLETE`	0x60	Command complete message received; SCSI command is finished, and SCSI bus is free.

Appendix C

Device Driver Migration Notes

This appendix explains how to make an IRIX 4.x device driver compliant with the IRIX 5.3 and 6.0 environments.

This appendix contains the following sections:

Introduction

Intended Audience and Prerequisites

This appendix is intended to aid in the planning of device driver migration from previous versions of IRIX to the IRIX 5.3 and IRIX 6.0 environments. It assumes you are familiar with the existing material on developing IRIX device drivers.

Background

Before you upgrade a perfectly good device driver that runs on an older version of IRIX, compute the time it will take to rewrite your driver to run under the new operating systems against the actual performance improvement you hope to achieve. In some cases, it may be to your advantage to continue using an older driver by recompiling its source code under the new operating system. For those cases where it is to your advantage to modify an existing driver, the following notes are provided.

IRIX 5.x

IRIX 5.x is a binary-compatible upgrade to IRIX 4.0.x at the user program level. However, due to the extensive internal changes to the IRIX kernel, device drivers and other kernel components, such as STREAMS modules and file systems, must be revised and recompiled for the IRIX 5.x environment. It is possible that changes to IRIX 5.3 will become effective after the publication of this manual.

Most of the changes in the IRIX kernel result from supporting the SVR4 interfaces for both applications programs and for certain kernel internal interfaces. The IRIX 5.x kernel interfaces are:

- SVR4 Device Driver Interface / Driver Kernel Interface (SVR4 DDI / DKI) described in the *IRIX Device Driver Reference Pages.* The IRIX 5.x operating system implementation uses a multiprocessor version of DDI / DKI developed for Silicon Graphics platforms.
- SVR4 STREAMS Interface, which is documented in the *UNIX® System V Release 4 STREAMS Programming Guide.*
- IRIX 5.x Data Link Provider Interface (DLPI) (relevant to STREAMS protocol stacks and to network drivers). This standard interface, defined by IEEE Standard 802.3, permits flexible integration protocol stacks and their "peaceful coexistence" with the TCP / IP stack. Protocol stacks register themselves with DLPI, as do the network drivers present in the system. The new protocol stack is thus isolated from the specific network hardware support present in the system.

IRIX 6.0

IRIX 6.0 is the 64-bit operating system for all Silicon Graphics systems that use MIPS R8000 series microprocessors. Because the IRIX 6.0 kernel is itself a 64-bit object, device drivers must be ported to the 64-bit operating system.

Migration Overview

The migration of drivers from an IRIX 4.0.5 environment to an IRIX 5.x environment is straightforward. The migration of several typical IRIX drivers required changes to less than 10% of the source lines. Most of these changes were in declarations. Naturally, extremely complex drivers that reached into the IRIX kernel to access services not typically employed in device drivers will have to change, as many of these interfaces have been replaced or altered. In any case, *you* are responsible for getting your device drivers to work across operating system and platform upgrades.

The changes to drivers fall in a number of broad categories:

- Changes in drivers that are a result of changes in the SVR4 API (Application Programmatic Interface) as compared to the SVR3 API.
- Changes to the declaration of the DKI interface, such as additional arguments to the driver calling interfaces and, in some cases, different procedure typing (mostly to make them void).
- Changes to use the DDI interface instead of the now obsolete IRIX 4.0.x interfaces for kernel service invocation. This may include changes that result from the changes in the semantics of the DDI. There are also some SGI interfaces defined for services omitted by DDI, such as cache flushing and address map setup.
- Changes to accommodate the architecture of the CHALLENGE/Onyx family, especially in the richness of its bus structure.
- Changes to accommodate the architecture of the Indigo^2, notably the EISA bus. Since this is the first Silicon Graphics system to use this bus, there are no migration issues.
- Changes in network device drivers and protocol stacks to make use of the Data Link Provider Interface (DLPI). See the **dlpi**(7) man page.
- Changes to take advantage of the IRIX 5.x dynamically loadable kernel module support. This permits device drivers to be loaded into a running system without rebooting. This is discussed in the **mload**(4) man pages.
- Changes to the SCSI driver interface to unify all supported SCSI controllers. See Chapter 5, “Writing a SCSI Device Driver,” for details.

IRIX 4.x to 5.x Migration

This section serves as a preliminary guide to planning the migration of device drivers from the IRIX 4.0.x environment to IRIX 5.x. The discussion below assumes that you already have the device driver working in IRIX 4.0.x, and need only to migrate it to IRIX 5.x.

SVR4 API Changes

The SVR4 API changes the size of a number of system structures to accommodate the changes in Expanded Fundamental Types. (Briefly, a number of fields, such as the user and group ID and device numbers are changed from 16 to 32 bits.) For driver work, the most important impact of these API changes is in the access to the major and minor numbers. The device number is now a 32-bit quantity, by changing the underlying type of the *typedef dev_t*. Access to the major and minor numbers can no longer be done with the 8-bit shift-and-mask technique common in prior versions of UNIX. (This is especially true as the split of the 32 bits between major and minor is a “hidden” parameter.) DDI defines a series of functions / macros to provide access to the major and minor numbers.

STREAMS Changes

For STREAMS, there are the standard SVR3 (4.x) to SVR4 (5.x) STREAMS changes, most notably in the arguments to the driver’s **open** routine. For further details, please refer to *STREAMS Modules and Drivers*, UNIX SVR4.2. UNIX Press, 1992.

DDI Changes

The *IRIX Device Driver Reference Pages* documents the routines that DDI supports. Generally, these have analogs to routines supported in prior versions of IRIX, but the syntax and semantics may differ, and the driver may need to have minor logic updates to accommodate these changes. In some drivers, the changes to support DDI simplify the driver by invoking common service interfaces in DDI instead of performing the work in the driver.

As it stands, DDI is insufficient to write a driver using *only* the routines in DDI. The lacks are in areas specific to a particular architecture or family, such as cache flushing and device register addressability in complex bus architectures. SGI has defined a number of additional service interfaces to support these needs.

DKI Changes

The changes from the new DKI affect the driver routines called by the rest of the IRIX kernel, both the **devsw* entry points and the interrupt handlers. The details of these interfaces are described in the *IRIX Device Driver Reference Pages* . The primary changes are:

- Device numbers are passed as a *dev_t* rather than as an *int* in prior IRIX versions. As mentioned elsewhere, the DDI contains new functions to access the major and minor numbers. Note that *open* takes a *dev_t **, while **read** and **write** take a *dev_t* (no indirection).
- Many of the entry points (**open**, **close**, **read**, **write**) take a new argument that is an opaque credentials structure. (It is opaque in the sense that the device driver never examines the internal makeup of the structure.) This is used to call the DDI **drv_priv**() routine, which takes the place of checking *u.u_uid == 0*. This permits different models of privileges to be used, for example, in a trusted system.
- The **read** and **write** routines take a *uio_t **, a pointer to a structure that defines the addresses and lengths of the data in the user space. (The *IRIX Device Driver Reference Pages* describe this structure. It is defined in */usr/include/sys/uio.h*.) Previously, this information was computed in the driver from the *u* vector. The driver typically passes this structure to a DMA setup routine such as the DDI **uiophysio**() or the Silicon Graphics enhancement, **biophysio**(), which is nearly identical to **iophysio**(), except that it takes a block number instead of a file offset).
- Generally, drivers can no longer access the *u* vector directly. DDI defines access routines for fields in the *u* vector. Direct access to the *u* vector tends to make a driver more dependent on specific aspects of the system than is desirable. This also extends to access *u.u_error*. DKI expect to return the error value (0 indicates no error), and are not supposed to set *u.u_error*.
- Interrupt routines are now of type *void*.

Addressability

The driver needs to arrange for addressability of the device registers in the CHALLENGE / Onyx family and, in the interest of having common code for VME drivers between the POWER Series™ and these systems, drivers must set up the appropriate mappings. This is done by the new **pio_map**() routines, which have similar calling interfaces to the **dma_map**() routines from prior versions.

CHALLENGE/Onyx Family Support

The CHALLENGE / Onyx family architecture has added new features and restrictions on the drivers, configuration, and other tools that could not be dealt with by the existing software framework. Supporting these systems with the common source used for all Silicon Graphics products necessitated changes in the way kernels are configured and support in for the new hardware features.

Some of the CHALLENGE series architecture differences include:

- Multiple types of bus support (such as VME, GIO, SCSI).
- VME buses are not automatically mapped into known K2 segment addresses.
- VME-bus controllers cannot access physical memory directly.
- Only a portion of the VME-bus A32 space can be mapped into the kernel at any one time.

These differences cause the following software changes:

1. The need for a general mechanism for probing for devices on multiple and different types of buses caused changes to the VECTOR line to specify which type of bus, which bus of that type, which address space on that particular bus, which address on that bus, and so on. Since there are no automatic K2 addresses, the addresses are abstracted and more meaningful than they were in the past. Different buses require different types of information, as well. VME has interrupts specifying both a

vector and IPL, whereas EISA does not. So, a bus-specific portion was added to the *edt* structure. This caused the need for new VECTOR line probing specifications, as well.

2. Since there are no known K2 addresses that correspond to VME-bus locations, user-level VME drivers can no longer be performed through */dev/mmem*, but instead, they now have their own */dev/vme* device interface. This also affects kernel drivers. Kernel drivers must map in the address space of their controllers before they can access them. This is done with special pio_mapping routines, similar to existing DMA mapping routines. This is further complicated by the fact that, for VME A32 space, pio_mapping registers are a limited resource. Only 13*8 MB can be mapped into the kernel at any one time; so 12 of these registers can be locked down and the thirteenth used as a floater to be shared by whoever needs it, from drivers to ***lboot***.

   Note: A32 pio_mappings start and end on 8 MB boundaries. The address you map plus the length of the mapping must not cross an 8 MB boundary.

3. Since VME-bus controllers cannot access physical memory directly, you can no longer pass the controller a K0 address and expect it to work. EVERYTHING has to be *DMA* mapped. For example, in the past, IOPB addresses were normally converted to K0 addresses and passed to the controller. Now the IOPBs must be *DMA* mapped.

4. Some new functionality has been added to the VME driver interface. Since most VME boards can have their IRQ vector and ipl programmed through software, it is now possible to allocate VME vectors dynamically, so they need not be specified on the VECTOR line. This frees you from worrying about finding one that is not already in use. Simply call **vme_ivec_alloc**() to allocate a free vector, then call **vme_ivec_set**() to register your interrupt routine.

5. The driver interface now uses the SVR4 MP DDI / DKI interface except for the Silicon Graphics-specific routines, such as **pio_map**() and **dma_map**(). For example, entry points such as **open**, **close**, **read**, and **write** all have slightly different arguments and, in some cases, different procedure types, in 5.x than they had in earlier versions.

6. The DDI / DKI *drvrflags* variable, not the flag in the *master.d* file, is used for the driver to indicate that it is MP-safe.

Kernel Configuration Issues

The highly flexible architecture of the CHALLENGE family requires extensions to the descriptions of devices in the IRIX kernel configuration process, specifically to the VECTOR lines in the system configuration files. The new VECTOR line for VME would look like the following:

```
VECTOR: bustype=VME module=jag ipl=1 ctlr=0 adapter=0
iospace=(A16S, 0, 0x800) probe_space=(A16S, 0, 1)
```

Note: The VECTOR line is still all one line. It is broken here to fit on the page.

New fields are *bustype*, which in this case is *VME*. The *ctlr* field takes the place of *unit* in 4.0.x. The *adapter* field specifies which VME bus. (CHALLENGE systems can be configured to have multiple VME buses).

The *iospace* triple is used to pass in the address of the controller. The first argument defines the address space. Valid values are *A16S*, *A16NP*, *A24S*, *A24NP*, *A32S*, *A32NP*, *A64S*, *A64NP*. The second argument is the address with the specified address space. The third argument is the length of the mapping. The *probe_space* line performs a **badaddr** like function (that is, it tries to read the specified address and catch any errors) on the specified address. In this case, the arguments are the address space, address within that space, and the size of the read.

There is also an *exprobe_space* extended probe space defined as follows:

```
exprobe_space=(r,A16S,0,2,0xfdd1,0xffff)
```

The difference between it and the 4.0.x version of *exprobe* is in the specification of the address to test.

Notice that no vector is specified. The old *vector*= primitive is still supported for boards that are jumpered. Generally, drivers would use the **vme_ivec_alloc**() and **vme_ivec_set**() routines to allocate and set the vector.

Equipped Device Table (EDT) Changes

The form of the structures for the Equipped Device Table (EDT) have changed for the same reasons as the VECTOR line in the configuration (the VECTOR information is used as initialization values for the EDT entries).

The following information is from */usr/include/sys/edt.h*:

```
#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)();  /* device init/run-time probe */
   iospace_t e_space[NBASE];
} edt_t;

#define    e_base e_space[0].ios_vaddr
#define    e_base2 e_space[1].ios_vaddr
#define    e_base3 e_space[2].ios_vaddr
#define    e_iobase e_space[0].ios_iopaddr
#define    e_iobase2 e_space[1].ios_iopaddr
#define    e_iobase3 e_space[2].ios_iopaddr

The e_bus_info field points to the following structure:

typedef struct vme_intrs {
   int (*v_vintr)();        /* interrupt routine */
   unsigned    char v_vec;  /* vme vector */
   unsigned    char v_brl;  /* interrupt priority level */
   unsigned    char v_unit; /* software identifier */
} vme_intrs_t;
```

The following fragment illustrates an *edt***init** routine using the new structures and the new **pio_*** routines.

```
mydrvredtinit(struct edt *e)
{
   piomap_t *piomap;
   vme_intrs_t *intrs = e->e_bus_info;
[...]
 piomap = pio_mapalloc(e->e_bus_type, e->e_adap,
   &e->e_space[0], PIOMAP_FIXED, ”mydrvr”);

 if (piomap == 0) {
      /* This could fail because the adapter isn’t valid
       * or invalid addresses or there are no more
       * fixed mappings available in the case of A32.
       */
      printf(“mydrvr not installed\n”);
      return;
   }

 e->e_base = pio_mapaddr(piomap, e->e_iobase);

   /* You can now use e->e_base as a normal address
    * to access your controller.
    */
[...]
/* Now allocate a VME IRQ vector and register
    * the interrupt routine.
    */
 ipl = intrs->v_brl;


 vec = vme_ivec_alloc(e->e_adap);

 if (vec == -1) {
 cmn_err(CE_WARN,”mydrvredtinit: no interrupt vector\n”);
 pio_mapfree(piomap);
 return;
 }
 vme_ivec_set(e->e_adap, vec, mydrvrintr, e->e_ctlr);
[...]
}
```

IRIX 5.2 to 5.3 Migration

Note: This information is preliminary and subject to change. While likely to be correct, it is based on extracted **diff** listings of actual drivers migrating between 5.2 and 5.3, so there may be errors or omissions. Please use the information with caution.

All IRIX 5.2 kernel components, including drivers and STREAMS modules, require some amount of work to migrate to IRIX 5.3. In the simplest cases, no source changes are required, but changes in the size of certain kernel structures make it necessary to recompile the same source in a 5.3 build environment.

In certain device drivers, three types of changes from the IRIX 5.2 kernel to the IRIX 5.3 kernel require some work:

1. Support for a multi-threaded version of TCP/IP.

   This requires change in network drivers to use the new blocking scheme instead of the older **splnet**() blocking scheme.

2. Incorporation of hooks for Trusted IRIX/B to allow the use of a least privilege model.

   The changes in drivers are to replace any calls to **suser**() or explicit checks of **u.u_uid == 0** with a corresponding call to one of the capability check routines.

3. **mbuf** management scheme.

   The **mbuf** pool is now initialized early in the system life, so drivers do not need to call **mbinit**().

Standard Device Drivers

The simplest migration is for standard (non-network and non-STREAMS) device drivers. Generally, these drivers require only recompilation, and do not require source changes. If these drivers check for root to protect privileged operations, however, the Trusted IRIX capabilities mechanism requires changes similar to the following fragment:

Note: If Trusted IRIX is not installed, these map to a stub that does a simple **u.u_uid == 0** check.

```
*** 30,35 ****
--- 30,36 ----
  #include “sys/pio.h”
  #include “sys/strsubr.h”
  #include “sys/ddi.h”
+ #include “sys/capability.h”

  extern time_t lbolt;

*** 1447,1453 ****
     * reasons.*/
    if (((cdp->cd_cflag & CLOCAL) ^ (cflag & CLOCAL)) &&
!       !suser()) {
            u.u_error = 0;  /* XXXrs */
            cflag &= ~CLOCAL;
            cflag |= cdp->cd_cflag & CLOCAL;
--- 1448,1454 ----
     * reasons.*/
    if (((cdp->cd_cflag & CLOCAL) ^ (cflag & CLOCAL)) &&
!       !_CAP_ABLE(CAP_DEVICE_MGT)) {
            u.u_error = 0;  /* XXXrs */
            cflag &= ~CLOCAL;
            cflag |= cdp->cd_cflag & CLOCAL;
```

Refer to the file *sys/capability.h* for a list of all capabilities.

STREAMS Modules

STREAMS modules and drivers resemble standard drivers in their modification requirements. In general, these need only to be recompiled. A STREAMS module should not, however, contain privilege checks because it does not have a valid user context in which to make them.

ifnet Drivers

ifnet-based networking device drivers that queue or dequeue packets on the **ipintrq** or **if_snd** queue must be modified to use the appropriate **IFNET_LOCK** macro. Definitions of the new locking macros are in *net/if.h* in an IRIX 5.3 system. Refer to Chapter 9, “Writing Network Device Drivers,” for more information.

Context diff of Token Ring ifnet Driver

The following fragment illustrates the changes made in one such driver (this is for the token ring, a driver that was otherwise unchanged during these modifications).

```
>>> add include file for capabilities.
*** 105,110 ****
--- 105,111 ----
  #endif
  #include “sys/dlsap_register.h”
  #endif    /* _IRIX4 */
+ #include “sys/capability.h”

  #ifdef QDBG
  struct tr_qs fvqs;
```

Board Initialization

This is in the board initialization routine, where the interface lock has been set by the caller.

```
*** 547,552 ****
--- 548,554 ----
  {
      struct fv_info *fv = &fv_info[unit];
      struct ifnet *ifp = &fv->fv_if;
+     ASSERT(IFNET_ISLOCKED(ifp));

      if ((ifp->if_flags & IFF_RUNNING) != 0) {
          DP((“if_fvinit%d: already running\n”, fv-
>fv_unit));

>>> also in the init routine.  Release the interface lock
>>> before sleeping, reacquire it after the sleep returns.
*** 553,560 ****
          return(0);
      }

!     if (fv->fv_state < FV_STATE_OK)
          sleep((caddr_t)&fv->fv_state, PZERO|PCATCH);
      if (fv->fv_state != FV_STATE_OK) {
          return(EIO);
      }
```

```
--- 555,565 ----
            return(0);
        }

!       if (fv->fv_state < FV_STATE_OK) {
!           IFNET_UNLOCKNOSPL(ifp);
            sleep((caddr_t)&fv->fv_state, PZERO|PCATCH);
+           IFNET_LOCKNOSPL(ifp);
+       }
        if (fv->fv_state != FV_STATE_OK) {
            return(EIO);
        }

>>> this is in the driver’s ioctl routine. ASSERT that the
>>> ioctl routine was called with interface lock held.
*** 894,899 ****
--- 899,905 ----
        struct fv_info *fv = &fv_info[ifp->if_unit];

        ASSERT(&fv->fv_ac == (struct arpcom*)ifp);
+       ASSERT(IFNET_ISLOCKED(ifp));
        switch (cmd) {
        case SIOCSIFADDR: {
            struct ifaddr *ifa = (struct ifaddr *)data;

>>> TrIRIX change for privilege check.
*** 1167,1173 ****
        case SIOC_TR_ARM: {
            TR_SIOC *sioc = (TR_SIOC*)data;

!           if (!suser()) {
                error = EPERM;
                break;
            }
--- 1173,1179 ----
        case SIOC_TR_ARM: {
            TR_SIOC *sioc = (TR_SIOC*)data;

!           if (!_CAP_ABLE(CAP_NETWORK_MGT)) {
                error = EPERM;
                break;
            }
```

Interrupt Handler

This is in the interrupt handler. Acquire the interface lock. *fv* is a unit info structure pointer, one element of which is the pointer to **struct ifnet**, the interface structure shared with IP.

```
*** 1231,1236 ****
--- 1237,1243 ----
          printf(“fv%d: early interrupt\n”, unit);
          goto fvintr_ret;
      }
+     IFNET_LOCKNOSPL(&fv->fv_if);
      QDBGUP(fvints.tot,1);
      mem = fv->fv_mem;
  iloop:

 More of the interrupt handler.  Free the lock as we exit.
*** 1237,1242 ****
--- 1244,1250 ----
      /* get the type of interrupt */
      cmdsts = io->sifcmd_stat;
      if ((cmdsts&TR_STAT_INTR) == 0) {
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          goto fvintr_ret;
      }
      found++;

>>> still more of the interrupt handler.
*** 1300,1305 ****
--- 1308,1314 ----
          fv->fv_state = FV_STATE_SICK;
          }
          QDBGUP(fvints.buf,1);
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          goto fvintr_ret;

      case TR_INT_SCB_CLEAR:

>>> frame receive handler.  Lock the IP input queue to add a
packet to it.
*** 2031,2044 ****

      switch (port) {
      case ETHERTYPE_IP:
-         schednetisr(NETISR_IP);
```

```
          ifq = &ipintrq;
                 if (IF_QFULL(ifq)) {
              IF_DROP(ifq);
              fv->fv_if.if_iqdrops++;
              goto drop;
          }
!                IF_ENQUEUE(ifq, m0);
          goto read_ret;
      case ETHERTYPE_ARP:
          if (sri) {
--- 2041,2057 ----

      switch (port) {
      case ETHERTYPE_IP:
          ifq = &ipintrq;
+         IFQ_LOCK(ifq);
                 if (IF_QFULL(ifq)) {
              IF_DROP(ifq);
              fv->fv_if.if_iqdrops++;
+             IFQ_UNLOCK(ifq);
              goto drop;
          }
!         IF_ENQUEUE_NOLOCK(ifq, m0);
!         IFQ_UNLOCK(ifq);
!         schednetisr(NETISR_IP);
          goto read_ret;
      case ETHERTYPE_ARP:
          if (sri) {

>>> Output routine, assert caller has if structure
>>> locked for us.
*** 2269,2274 ****
--- 2282,2288 ----
      ASSERT((ifp->if_unit >= 0) && (ifp->if_unit <
FV_MAXBD));
      fv = &fv_info[ifp->if_unit];
      ASSERT(0 != fv->FVIO && ifp == &fv->fv_if);
+     ASSERT(IFNET_ISLOCKED(ifp));

      /* 2: make sure board has been initialized properly */
      if (fv->fv_state != FV_STATE_OK || iff_dead(ifp-
>if_flags)) {

>>> close routine.  ASSERT caller locked interface,
>>> release and reacquire lock around sleep.
```

```
*** 3750,3761 ****
--- 3765,3780 ----
      FVMEM *mem = fv->fv_mem;

      DP((“fv%d: close%\n”, fv->fv_unit));
+     ASSERT(IFNET_ISLOCKED(&fv->fv_if));
+
      if (fv->fv_state != FV_STATE_OK)
          goto close_ret;

      while ((fv->cmd_Flags[TR_CMD_CLOSE]&CMD_BUSY) != 0) {
          fv->cmd_Flags[TR_CMD_CLOSE] |= CMD_PENDING;
+         IFNET_UNLOCKNOSPL(&fv->fv_if);
          sleep((caddr_t)&fv-
>cmd_Flags[TR_CMD_CLOSE],PZERO|PCATCH);
+         IFNET_LOCKNOSPL(&fv->fv_if);
      }
      fv->cmd_Flags[TR_CMD_CLOSE] |= CMD_BUSY;


>>> later in the close routine, another release/reacquire
    around sleep.
*** 3776,3782 ****
--- 3795,3803 ----
      io->sifcmd_stat = TR_CMD_INT_ADAPT | TR_CMD_EXECUTE |
                  TR_CMD_SCB_REQUEST | TR_STAT_INTR;

+ IFNET_UNLOCKNOSPL(&fv->fv_if);
  sleep((caddr_t)&fv->cmd_Status[TR_CMD_CLOSE],
PZERO|PCATCH);
+ IFNET_LOCKNOSPL(&fv->fv_if);
  if (fv->cmd_Status[TR_CMD_CLOSE] == 0) {
      fv->fv_state = FV_STATE_CLOSE;
      fv->fv_if.if_flags &= ~(IFF_UP|IFF_RUNNING);
```

mbuf Manager Changes

```
>>> delete the mbinit() call, no longer needed.
*** 3174,3183 ****
      IDP((“fv%u: IVEC set!\n”, unit));
  #endif /* !_IRIX4 */

-     /* 3: start the mbufs */
-     mbinit();
```

```
!     /* 4: setup PRIVATE data.*/
      /* TBD: allocate mcast filter table.
       *    Probably, ALLMULTI(via ffffffff) should be
used        *    and locally calculate correct filter.
--- 3174,3181 ----
      IDP((“fv%u: IVEC set!\n”, unit));
  #endif /* !_IRIX4 */


!     /* setup PRIVATE data.*/
      /* TBD: allocate mcast filter table.
       *    Probably, ALLMULTI(via ffffffff) should be used
       *    and locally calculate correct filter.
```

Migration to IRIX 6.0

The following issues are important when attempting to convert a device driver for a 32-bit kernel to a 64-bit kernel driver. For details on drivers for POWER Indigo^2 or POWER CHALLENGE M, see “POWER Indigo2 and POWER CHALLENGE M Drivers” in Appendix A.

Virtual Page Size

The virtual page size for 64-bit kernels is currently 16 KB, while it is 4 KB for 32-bit systems. However, various I/O hardware items (such as DMA map registers on CHALLENGE/Onyx platforms) still deal with 4 KB pages for I/O, requiring that the driver use a different set of constants or procedures when dealing with I/O pages.

Most of the following new I/O-related items already exist without the *IO_* prefix and refer to virtual memory page size rather than I/O page size.

Constants

IO_NBPP	number of bytes in an I/O page
IO_PNUMSHFT	number of bits to shift I/O address to page number
IO_POFFMASK	mask for offset into I/O page

Macros

io_pnum(x)	I/O page number from address
io_poff(x)	I/O page offset from address
io_numpages(addr, len)	number of I/O pages to span address range starting at *addr* for *len* bytes
io_ctob(x)	convert I/O pages to bytes
io_btoc(x)	convert byte count to number of pages (rounded up)
io_btoct(x)	convert byte count to number of pages (rounded down)

To convert an existing driver, a good starting point would be to examine all uses of **NBPP**, **PNUMSHFT**, **POFFMASK**, **pnum**, **poff**, **ctob**, **btoc**, and **btoct** and consider:

- whether they refer to virtual memory pages and should be left alone

  or

- whether they refer to I/O pages and need to be converted.

ioctl Support

In addition to the usual 64-bit conversion worries (*longs* and pointers becoming 64-bits, *ints* remaining 32-bits), drivers may need to support ioctls from user programs. Since user programs may be 32-bit or 64-bit, data items from those programs may need to be converted into the appropriate internal form for the driver. (The driver needs to know whether a pointer or a *long* from a user program should be treated as a 32-bit quantity or a 64-bit quantity.) To ease the driver conversion, a new system intrinsic that informs the driver of the size of various types for the currently executing user has been supplied. See the ioctl(D2) man page.

The following definitions are available in *sys/types.h* and *sys/ddi.h*:

```
/* Since device drivers may need to know which ABI the
/* current user process is running under in order to use the
/* correct types, we provide the following structure. See
/* ddi.h for the definition of userabi().
 * All sizes are in bytes. */

typedef struct __userabi {
    short uabi_szint;
    short uabi_szlong;
    short uabi_szptr;
    short uabi_szlonglong;
} __userabi_t;

/* function: userabi(__userabi_t *)
 * purpose: determine the size int bytes of various C types
 * in the ABI under which the current user process is
 * running. 0 indicates success,nonzero indicates failure.
 * This function must only be called from a user's
 * context, and the values copied into __userabi_t are only
 * valid for the process executing when userabi is called.
 */
int
userabi(__userabi_t *currentabi)
```

Pointers

Pointers in memory buffers need to be double-word aligned to avoid address errors when the pointers are loaded. Both pointers and longs must be aligned on 8-byte boundaries; neither should be cast to int because the int structure is only 32-bits wide.

Hardware Data Copying

When copying data to or from a hardware device, drivers should use the functions **hwcpin** and **hwcpout** rather than **bcopy**. The reason is that the kernel **bcopy** routine is optimized for memory access and may use double-word **loads** and **stores**, which may not be supported by the hardware device. The routines **hwcpin** and **hwcpout** perform only word (or byte and half-word) operations.

Glossary

ABI
Application Binary Interface.

adjmsg
Trim bytes from a message.

allocb
Allocate a message block.

ASSERT
Program verification macro.

badaddr
Check for bus error when reading an address.

bcanput
Test for flow control in a specified priority band.

bcopy
Copy data between address locations in the kernel.

big-endian
The default for a byte order.

biodone
Release buffer after block I/O and wakeup processes.

bioerror
Manipulate error field within a buffer header.

biowait

Suspend processes pending completion of block I/O.

block driver

A device driver, such as for magnetic tape or disk drives, that transfers data in blocks through the buf structure.

bp_mapin

Allocate virtual address space for buffer page list.

bp_mapout

Deallocate virtual address space for buffer page list.

brelse

Return a buffer to the system's free list.

btod

Convert from bytes to disk sectors.

btop

Convert size in bytes to size in pages (rounded down).

bptophys

Get physical address of buffer data.

btopr

Return number of memory pages contained in the specified number of bytes, rounded up.

buf

Block I/O data transfer structure, the basic data structure for block I/O transfers.

bufcall

Call a function when a buffer becomes available.

bus-watching cache

When an IP5, IP7, or IP19 system performs a DMA write into physical memory, the bus-watching cache automatically invalidates the data cache. This hardware function eliminates the need for data cache write back or invalidation in software.

bzero

Clear memory for a given number of bytes.

canput

Test for flow control in a stream.

character device

A device driver, such as a terminal or printer, that transfers data character by character. See also block device.

character driver

A device driver, such as for a terminal or printer, that transfers data characters between the device and the user program. Note that block devices, such as magnetic tape or disk drives, also support character access.

close

Relinquish access to a device. The user process invokes the **close**() system call when it is finished with a device, but the system does not necessarily execute your *drv***close**() entry point for that device.

clrbuf

Erase the contents of a buffer.

cmn_err

Display an error message or panic the system.

copyb

Copy a message block.

copyin

Copy data from user process virtual space to kernel virtual space.

copymsg
Copy a message.

copyout
Copy data from kernel virtual space to user process virtual space.

copyreq
STREAMS transparent **ioctl**() copy request structure – data necessary to process transparent ioctls.

copyresp
STREAMS transparent **ioctl**() copy response – data in response to a prior copy request necessary to continue processing transparent ioctls.

cpsema
Conditionally perform a "P" or wait semaphore operation.

cvsema
Conditionally perform a "V" or wait semaphore operation.

data structure
The memory storage area used to hold data types such as integers, strings, or an array of integers. The data structures associated with drivers are used as buffers for holding data being moved between the user data area and the device.

datab
STREAMS data block structure that describes the data of a STREAMS message.

datamsg
Test whether a message is a data message.

DDI/DKI
Device Driver Interface / Device Kernel Interface.

delay
Delay process execution for a specified number of clock ticks.

devflag

Driver flags – Silicon Graphics only supports flags D_MP, D_WBACK and D_OLD device.

device driver

A software routine that manages a hardware device; it brings the device into and out of service, sets hardware parameters in the device, transmits data from the kernel to the device, receives data from the device and passes data back to the kernel, and handles I/O errors.

Device Driver Interface (DDI)

The set of structures, routines, and optional functions used to implement a device driver.

device types

There are two types of devices available on any UNIX system: software and hardware. A software device is usually a section of memory and is referred to as a pseudo-device. A pseudo-device may provide access to system structures that are unavailable at the user level. For example, a pseudo-device such as a RAM disk could provide fast access to files. Some examples of hardware devices are disk drives, tape drives, printers, scanners, and terminals.

dki_dcache_inval

Invalidate the data cache for a given range of virtual addresses.

dki_dcache_wb

Write back the data cache for a given range of virtual addresses.

dki_dcache_wbinval

Write back and invalidate the data cache for a given range of virtual addresses.

dma_map

Load DMA mapping registers for an imminent transfer.

dma_mapaddr

Return the "bus virtual" address for a given map and address.

dma_mapalloc

Allocate a DMA map. See the dma_map(D3X) man page.

dma_mapfree

Free a DMA map. See the dma_map(D3X) man page.

downstream

The direction of STREAMS messages flowing through a write queue from the user process to the driver.

Driver-Kernel Interface (DKI)

A defined service interface for the entry point routines and utility functions specified for communications between the driver and the kernel. It does not include the driver/hardware or the driver/boot software interface.

drv_getparm

Retrieve kernel state information

drv_hztousec

Convert clock ticks to microseconds.

drv_priv

Determine whether credentials are privileged.

drv_setparm

Set kernel state information.

drv_usectohz

Convert microseconds to clock ticks.

drv_usecwait

Busy-wait for specified interval.

dupb

Duplicate a message block.

dupmsg

Duplicate a message.

edtinit

Initialize a device at boot time.

EISA bus

Enhanced Industry Standard Architecture bus.

EISA Product Identifier (ID)

EISA expansion boards have a four-byte product identifier (z=0 for the system board).

eisa_dma_disable

Disable recognition of hardware requests on a DMA channel.

eisa_dma_enable

Enable recognition of hardware requests on a DMA channel.

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 buffer descriptor.

eisa_dma_get_cb

Allocated a DMA command block.

eisa_dma_prog

Program a DMA operation for a subsequent software request.

eisa_dma_stop

Stop software-initiated DMA operation on a channel and release it.

eisa_dma_swstart

Initiate a DMA operation via software request.

enableok

Allow a queue to be serviced.

Enhanced Industry Standard Architecture

The EISA bus specification.

errnos

Error numbers.

esballoc

Allocate a message block using an externally supplied buffer.

esbbcall

Call a function when an externally supplied buffer can be allocated.

etoimajor

Convert external to internal major device number.

flushband

Flush messages in a specified priority band.

flushbus

Make sure contents of the write buffer are flushed to the system bus.

flushq

Flush messages on a queue.

freeb

Free a message block.

freemsg

Free a message.

freerbuf

Free a raw buffer header.

freesema

Free the resources associated with a semaphore.

free_rtn
STREAMS driver's message free routine structure.

fubyte
Fetch (read) a byte from user space.

fuword
Fetch (read) a word from user space.

geteblk
Get an empty buffer.

getemajor
Get external major device number.

geteminor
Get external minor device number.

geterror
Retrieve error number from a buffer header.

getmajor
Get internal major device number.

getminor
Get internal minor device number.

getq
Get the next message from a queue.

getrbuf
Get a raw buffer header.

GIO bus
Graphics I/O bus used on Indigo, Indigo^2, and Indy workstations.

halt
Shut down the driver when the system shuts down.

I/O operations

Services that provide access to shared input/output devices and to the global data structures that describe their status. I/O operations open and close files and devices, read data from and write data to devices, set the state of devices, and read and write system data structures.

info

STREAMS driver and module information.

init

Initialize a device.

initnsema

Allocate a semaphore and initialize it to a given value.

insq

Insert a message into a queue.

inter-process communication

These are system calls that allow a process to send information to another process. There are several ways of sending information to another process: signals, pips, shared memory, message queues, semaphores, or streams and sockets.

interrupt level

A driver interrupt routine that is started when an interrupt is received from a hardware device. The system accesses the interrupt vector table, determines the major number of the device, and passes control to the appropriate interrupt routine.

interrupt priority level

The interrupt priority level at which the device requests that the CPU call an interrupt process. This priority can be overridden in the drivers's interrupt routine for critical sections of code with the spl function.

intr

Process a device interrupt after a transfer terminates (either normally upon completion or abnormally due to some error).

iocblk

STREAMS ioctl structure.

ioctl

Control a character device. Character devices may include a "special function" entry point, *drv***ioct**().

iovec

Data storage structure for I/O using uio.

IRQ

See Interrupt Request Input.

itimeout

Execute a function after a specified length of time.

itoemajor

Convert internal to external major device number.

k0

Virtual address range that is cached but not mapped by translation look-aside buffers. Also *kseg0*.

k1

Virtual address range that is neither cached nor mapped. Also *kseg1*.

k2

Virtual address range that can be both cached and mapped by translation look-aside buffers. Also *kseg2*.

kern_calloc

Allocate storage for objects of a specified size.

kern_free

Free kernel memory space

kern_malloc

Allocate kernel virtual memory.

kmem_alloc

Allocate space from kernel free memory.

kmem_free

Free previously allocated kernel memory.

kmem_zalloc

Allocate and clear space from kernel free memory.

kvtophys

Get physical address of buffer data.

linkb

Concatenate two message blocks.

linkblk

STREAMS multiplexor link structure – data needed by a multiplexing driver to set up or take down a multiplexor link.

LOCK

Acquire a basic lock Silicon Graphics LOCK function returns int instead of pl_t.

LOCK_ALLOC

Allocate and initialize a basic lock. Silicon Graphics doesn't support compilation option _LOCKTEST. splockmeter is provided for debugging purpose by Silicon Graphics.

LOCK_DEALLOC

Deallocate an instance of a basic lock

makedevice

Make device number from major and minor numbers.

map

Support virtual mapping for memory-mapped device.

max

Return the larger of two integers.

messages

STREAMS messages.

min

Return the lesser of two integers.

mmap

Check virtual mapping for memory-mapped device. (Silicon Graphics also supports map and unmap entry routines).

mmapped device driver

Memory-mapped device drivers are those in which the hardware is memory mapped into a user's address space; no interrupt or DMA service routine is available to the user process.

module

A STREAMS module consists of two related queue structures, one for upstream messages and one for downstream messages. One or more modules may be pushed onto a stream between the stream head and the driver, usually to implement and isolate a communication protocol or a line discipline.

module_info

STREAMS driver and module information – identification and limit values used to initialize the module's or driver's queues.

msgb

STREAMS message block structure.

msgdsize

Return number of bytes of data in a message.

ngeteblk

Get an empty buffer of the specified size.

noenable

Prevent a queue from being scheduled.

open

Gain access to a device. The kernel calls drvopen() when the user process issues an **open**() system call.

OTHERQ

Get pointer to queue's partner queue.

pcmsg

Test whether a message is a priority control message.

phalloc

Allocate and initialized a pollhead structure.

phfree

Free a pollhead structure.

physiock

Validate and issue raw I/O request.

PIO

Programmed I/O.

pio_badaddr

Check for bus error when reading an address.

pio_bcopyin

Copy data from VME bus address to kernel's virtual space.

pio_bcopyout

Copy data from kernel's virtual space to VME bus address.

pio_mapaddr

Used with FIXED maps to generate a kernel pointer to VME bus space.

pio_mapalloc

Allocate a PIO map.

pio_mapfree

Free up a previously allocated PIO map.

pio_wbadaddr

Check for bus error when writing to an address.

poll

Poll entry point for a non-stream character driver. Silicon Graphics currently does not support **POLLRDNORM**, **POLLWRNORM**, **POLLRDBAND**, and **POLLWRBAND**. A character device driver may include a *drv***poll**() entry point so that users can use **select**(2) or **poll**(2) to poll the file descriptors opened on such devices.

pollwakeup

Inform polling processes that an event has occurred.

prefix

Driver prefix. Throughout this manual, the prefix *drv* preceding a function, routine, or entry point represents the name of the device driver you are writing.

primatives

C operations from which more complex operations can be constructed.

print

Display a driver message on the system console.

process control

These are system calls that allow a process to control its own execution. A process can allocate memory, lock itself in memory, set its scheduling priorities, wait for events, execute a new program, or create a new process.

proc_ref

Obtain a reference to a process for signaling.

proc_signal

Send a signal to a process.

proc_unref

Release a reference to a process.

psema

Perform a "P" or wait semaphore operation.

pseudo-device

A section of memory that emulates the functionality of a hardware device in software. Pseudo-devices may provide access to system structures that are unavailable at the user level. For example, a pseudo-device such as a RAM disk could provide fast access to files.

ptob

Convert size in pages to size in bytes.

put

Receive messages from the preceding queue.

putbq

Place a message at the head of a queue.

putctl

Send a control message with a one-byte parameter to a queue.

putctl1

Send a control message with a one-byte parameter to a queue.

putnext

Send a message to the next queue.

putq

Put a message on a queue.

qenable

Schedule a queue's service routine to be run.

qinit

STREAMS queue initialization structure – pointers to processing procedures and default values for a **queue**().

qreply

Send a message in the opposite direction in a stream.

qsize

Find the number of messages on a queue.

queue

STREAMS queue structure – pointers to processing procedures, the next queue in the stream, flow control parameters, and messages.

RD

Get a pointer to the read queue.

read

Read data from a device. The kernel executes the drvread() or drvwrite() entry points whenever a user process calls the **read**() system call.

rmalloc

Allocate space from a private space management map.

rmallocmap

Allocate and initialize a private space management map.

rmalloc_wait

Allocate space from a private space management map.

rmfree

Free space into a private space management map.

rmfreemap

Free private space management map.

rmvb

Remove a message block from a message.

rmvq

Remove a message from a queue.

SAMESTR

Test whether the next queue is of the same type.

SCSI

Small Computer System Interface.

SCSI bus

See Small Computer System Interface.

SCSI driver interface

A collection of machine-independent input/output controls, functions, and data structures, that provide a standard interface for writing a SCSI driver.

scsi_alloc

Allocate communication channel between host adapter driver and a kernel level SCSI device driver

scsi_command

Issue a command to a SCSI device

scsi_free

Free communication channel between host adapter driver and a kernel level SCSI device driver

scsi_info

Get information about a SCSI device

sgset

Assign physical addresses to a vector of software scatter/gather registers.

signals

Signal numbers.

size

Return size of logical block device.

sleep

Suspend process execution pending occurrence of an event.

SLEEP_ALLOC

Allocate and initialize a sleep lock Silicon Graphics doesn't support compilation option _MPSTATS.

SLEEP_DEALLOC

Deallocate an instance of a sleep lock.

SLEEP_LOCK

Acquire a sleep lock. Always pass -1 as priority.
`void SLEEP_LOCK(sleep_t *lockp, -1)`

SLEEP_LOCKAVAIL

Query whether a sleep lock is available.

SLEEP_LOCK_SIG

Acquire a sleep lock The valid values for priority are as follows: PUSER, PCATCH, PSLEP, PPIPE, PVFS, and PWAIT SLEEP_TRYLOCK Try to acquire a sleep lock.

SLEEP_TRYLOCK

Try to acquire a sleep lock.

SLEEP_UNLOCK

Release a sleep lock.

socket

A software structure that represents one endpoint in a two-way communications link. Created by **socket**(2).

spl

Block / allow interrupts on a processor.

srv

Service queued messages.

start

Start initialize a device at system start-up.

strategy

Perform block I/O strategy.

strcat

Concatenate strings.

strcpy

Copy a string.

Stream

A linked list of kernel data structures that provide a full-duplex data path between a user process and a device. Streams are supported by the STREAMS facilities in UNIX System V Release 3 and later.

stream head

The stream head, which is inserted by the STREAMS subsystem, processes STREAMS-related system calls and performs data transfers between user space and kernel space. It is the component of a stream closet to the user process. Every stream has a stream head.

STREAMS

A kernel subsystem used to build a stream, which is a modular, full-duplex data path between a device and a user process. In IRIX 5.x and later, the TCP/IP stack sits on top of the STREAMS stack. The Transport Layer Interface (TLI) is fully supported.

streamstab

STREAMS driver and module declaration structure.

streams_interrupt

Synchronize interrupt-level function with STREAMS mechanism.

STREAMS_TIMEOUT

Synchronize timeout with STREAMS mechanism.

strlog

Submit messages to the log driver.

stroptions

STREAMS head option structure.

strqget

Get information about a queue or band of the queue.

strqset

Change information about a queue or band of the queue.

subyte

Set (write) a byte to user space.

suword

Set (write) a word to user space.

TCP/IP

Transmission Control Protocol / Internet Protocol.

TFP

SGI's pre-release, internal code name for the MIPS R8000 processor.

TLI

Transport Interface Layer.

TRYLOCK

Try to acquire a basic lock.

uio

scatter / gather I / O request – describes an I / O request that can be broken into different data storage areas.

uiomove

Copy data using uio structure.

uiophysio
Set up user data space for I/O.

unbufcall
Cancel a pending bufcall request.

undma
Unlock physical memory in user space.

unlinkb
Remove a message block from the head of a message.

unload
Clean up a loadable kernel module.

UNLOCK
Release a basic lock

unmap
Support virtual unmapping for memory-mapped device

untimeout
Cancel previous timeout request.

untimeout_func
Cancel a previous invocation of timeout by function.

ureadc
Copy a character to space described by uio structure.

userdma
Lock, unlock physical memory in user space

uwritec
Return a character from space described by uio structure.

valusema
Return the value associated with a semaphore.

VME bus

VERSA Module Eurocard bus.

VME-bus adapter

A hardware conduit that translates host CPU operations to VME-bus operations and decodes some VME-bus operations to translate them to the host side.

vme_adapter

Determine VME adapter.

vme_ivec_alloc

Allocate a VME bus interrupt VECTOR.

vme_ivec_free

Free up a VME bus interrupt VECTOR.

vme_ivec_set

Register a VME bus interrupt handler.

volatile

Inform the compiler of volatile variables.

vpsema

Perform an atomic "V" and "P" semaphore operation on two semaphores.

vsema

Perform a "V" or signal semaphore operation.

v_getaddr

Get the user address associated with virtual handle.

v_gethandle

Get unique identifier associated with virtual handle.

v_getlen

Get length of user address space associated with virtual handle.

v_mapphys

Map physical addresses into user address space.

wakeup

Resume suspended process execution.

wbadaddr

Check for bus error when writing to an address.

WR

Get a pointer to the write queue.

write

Write data to a device. The kernel executes the drvread() or drvwrite() entry points whenever a user process calls the **read**() or **write**() system calls.

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0911-050.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389