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.

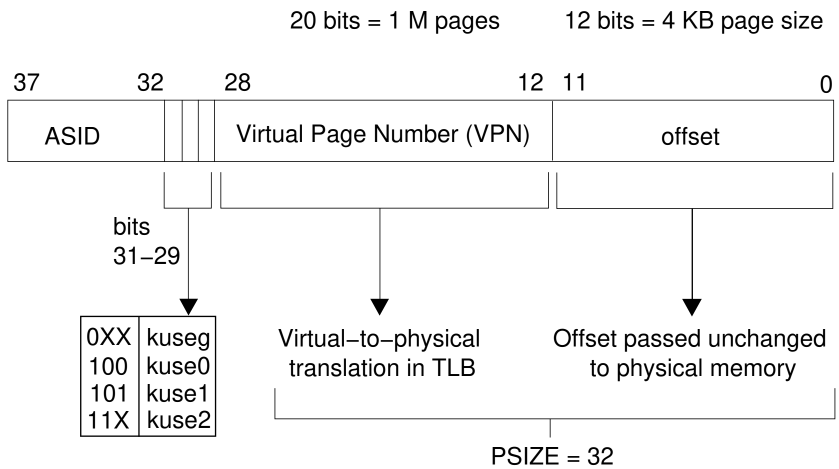

Figure 1-1 MIPS 32-bit Virtual Address Format (MIPS II Mode)

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.)

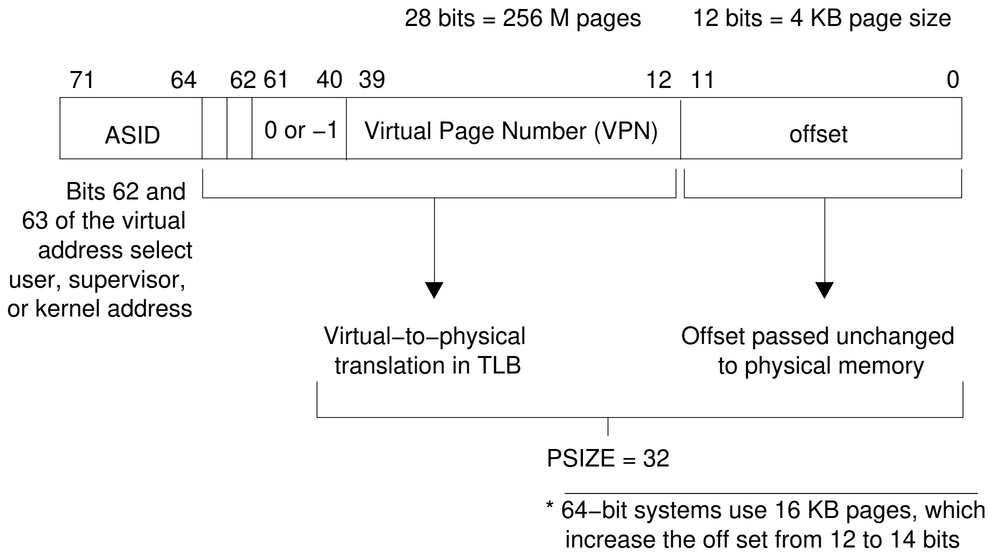

Figure 1-2 MIPS 64-bit Virtual Address Format (MIPS III Mode)

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.

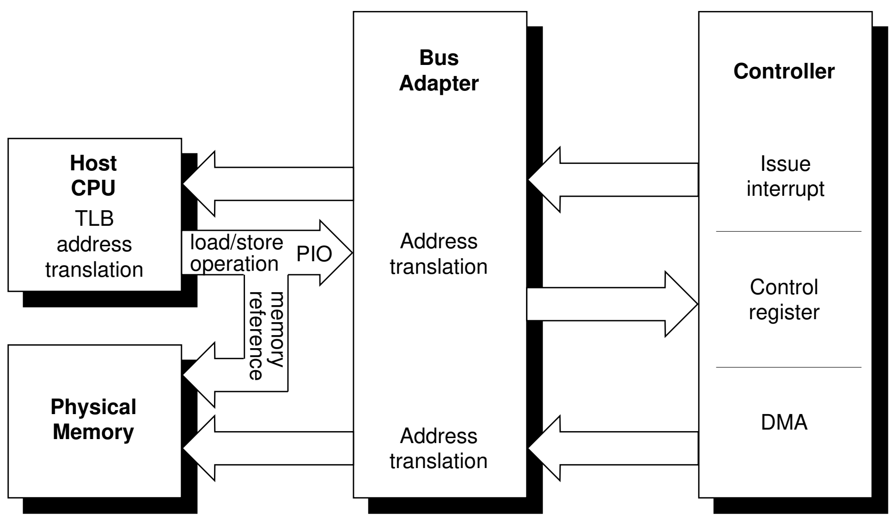

Figure 1-3 Architectural Block Diagram of Address/Data Flow

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.

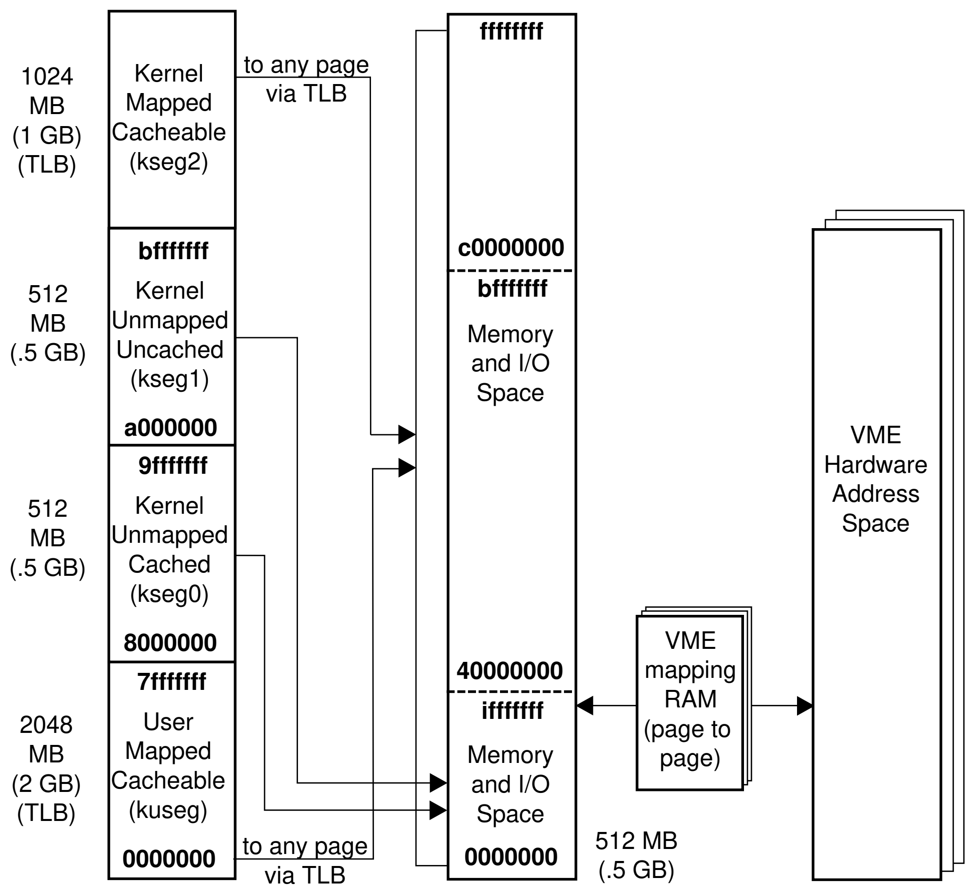

Figure 1-4 MIPS II Virtual Memory Map

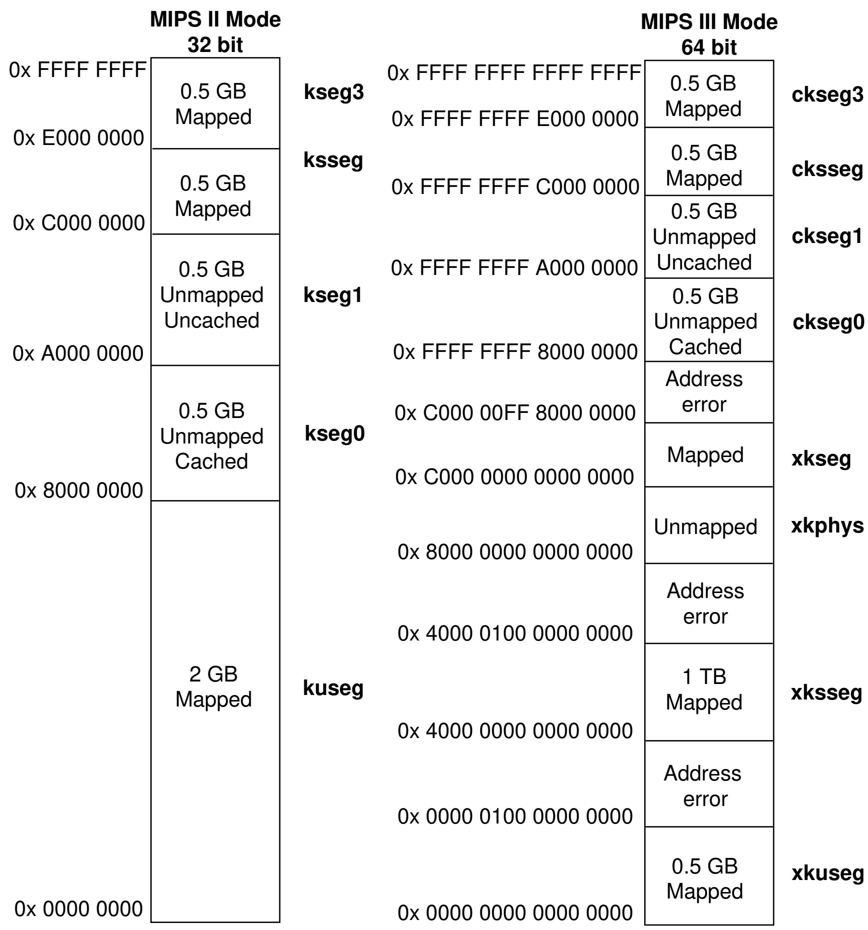

Figure 1-5 MIPS II 32-bit versus MIPS III 64-bit Kernel Mode Address Space

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.

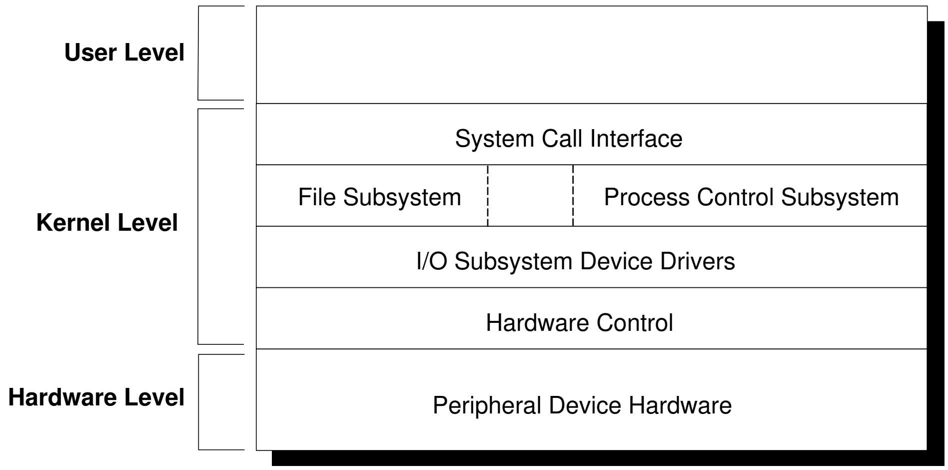

Figure 1-6 Device Driver Position in the Kernel