Device Control Software

IRIX provides for two general methods of controlling devices, at the user level and at the kernel level. This chapter describes the architecture of these two software levels and points out the different abilities of each. This is important background material for understanding all types of device control. The chapter covers the following main topics:

- "User-Level Device Control" on this page summarizes five methods of device control for user-initiated processes.
- "Kernel-Level Device Control" on page 45 sets the concepts needed to understand kernel-level drivers.

User-Level Device Control

In IRIX terminology, a *user-level* process is one that is initiated by a user (possibly the superuser). A user-level process runs in an address space of its own, with no access to the address space of other processes or to the kernel's address space, except through explicit memory-sharing agreements.

In particular, a user-level process has no access to physical memory (which includes access to device registers) unless the kernel allows the process to share part of the kernel's address space. (For more on physical memory, see Chapter 1, "Physical and Virtual Memory.")

There are several ways in which a user-level process can control devices, which are summarized in the following topics:

- "EISA Mapping Support" on page 42 summarizes PIO access to the EISA bus.
- "VME Mapping Support" on page 42 summarizes PIO access to the VME bus.
- "User-Level DMA From the VME Bus" on page 43 summarizes DMA I/O managed from a user-level process.
- "User-Level Control of SCSI Devices" on page 43 summarizes DMA and command access to the SCSI bus.

- “Managing External Interrupts” on page 43 summarizes access to the external interrupt ports on Challenge and Onyx systems.
- “User-Level Interrupt Management” on page 44 summarizes the handling of some interrupts in a user-level process.

EISA Mapping Support

In systems that support the EISA bus (Indigo^2 Maximum Impact and Indigo^2, Challenge M, and their Power versions), IRIX contains a kernel-level device driver that supports memory-mapping EISA bus addresses into the address space of a user process (see “Overview of Memory Mapping” on page 50).

You can write a program that maps a portion of the EISA bus address space into the program address space. Then you can load and store from device registers directly.

For more details of PIO to the EISA bus, see Chapter 4, “User-Level Access to VME and EISA.”

VME Mapping Support

In systems that support the VME bus (Onyx, Challenge DM, Challenge L, Challenge XL, and their Power versions), IRIX contains a kernel-level device driver that supports mapping of VME bus addresses into the address space of a user process (see “Overview of Memory Mapping” on page 50).

You can write a program that maps a portion of the VME bus address space into the program address space. Then you can load and store from device registers directly.

For more details of PIO to the VME bus, see Chapter 4, “User-Level Access to VME and EISA.”

User-Level DMA From the VME Bus

The Challenge L, Challenge XL, and Onyx systems and their Power versions contain a DMA engine that manages DMA transfers from VME devices, including VME slave devices that normally cannot do DMA.

The DMA engine in these systems can be programmed directly from code in a user-level process. Software support for this facility is contained in the *udmalib* package.

For more details of user DMA, see Chapter 4, "User-Level Access to VME and EISA" and the udmalib(3) reference page.

User-Level Control of SCSI Devices

IRIX contains a special kernel-level device driver whose purpose is to give user-level processes the ability to issue commands and read and write data on the SCSI bus. By using **ioctl()** calls to this driver, a user-level process can interrogate and program devices, and can initiate DMA transfers between memory buffers and devices.

The low-level programming used with the dsreq device driver is eased by the use of a library of utility functions documented in the dslib(3) reference page.

For more details on user-level SCSI access, see Chapter 5, "User-Level Access to SCSI Devices."

Managing External Interrupts

The Challenge L, Challenge XL, and Onyx systems and their Power versions have four external-interrupt output jacks and four external-interrupt input jacks on their back panels. In these systems, the device special file */dev/ei* represents a device driver that manages access to these external interrupt ports.

Using **ioctl()** calls to this device (see “Overview of Device Control” on page 48), your program can

- enable and disable the detection of incoming external interrupts
- set the strobe length of outgoing signals
- strobe, or set a fixed level, on any of the four output ports

In addition, library calls are provided that allow very low-latency detection of an incoming signal.

For more information on external interrupt management, see Chapter 6, “Control of External Interrupts” and the ei(7) reference page.

User-Level Interrupt Management

A new facility in IRIX 6.2 allows you to receive and handle device interrupts in a function in a user-level program you write.

Your program calls a library function to register the interrupt-handling function. When the device generates an interrupt, the kernel branches directly into your handler. Because the handler runs as a subroutine of the kernel, it can use only a very limited set of system and library functions. However, it can refer to variables in the process address space, and it can wake up a process that is blocked, waiting for the interrupt to occur.

Combined with PIO, user-level interrupts allow you to test most of the logic of a device driver for a new device in user-level code.

In IRIX 6.2, support for user-level interrupts is limited to VME devices and to external interrupts in the Challenge L, Challenge XL,and Onyx systems and their POWER versions.

For more details on user-level interrupts, see Chapter 7, “User-Level Interrupts” and the uli(3) reference page.

Memory-Mapped Access to Serial Ports

The Audio/Serial Option (ASO) board for the Challenge and Onyx series provides six high-performance serial ports, each of which can be set to run at speeds as high as 115,200 bits per second. The features and administration of the Audio/Serial Option board are described in the *Audio/Serial Option User's Guide* (document 007-2645-001).

The serial ports of the ASO board can be accessed in the usual way, by opening a file to a device in the */dev/tty** group of names. However, for the minimum of latency and overhead, a program can open a device in the */dev/aso_mmap* directory. These device files are managed by a device driver that permits the input and output ring buffers for the port to be mapped directly into the user process address space. The user-level program can spin on the input ring buffer pointers and detect the arrival of a byte of data in microseconds after the device driver stores it.

The details of the memory-mapping driver for ASO ports are spelled out in the asoserns(7) reference page (available only when the ASO feature has been installed).

Kernel-Level Device Control

IRIX supports the conventional UNIX architecture in which a user process uses a kernel service to request a data transfer, and the kernel calls on a device driver to perform the transfer.

Kinds of Kernel-Level Drivers

There are three distinct kinds of kernel-level drivers:

- A *character device driver* transfers data as a stream of bytes of arbitrary length. A character device driver is called as a direct result of a user process issuing a system function call such as **read()** or **ioctl()**.
- A *block device driver* transfers data in blocks of fixed size, and is called from the kernel to support filesystem or paging operations.

- A STREAMS driver is not a device driver, but rather can be dynamically installed to operate on the flow of data to and from any character device driver.

Overviews of the operation of STREAMS drivers are found in Chapter 19, “STREAMS Drivers.” The rest of this discussion is on character and block device drivers.

Typical Driver Operations

There are five different kinds of operations that a device driver can support:

- The open interaction is supported by all drivers; it initializes the connection between a process and a device.
- The control operation is supported by character drivers; it allows the user process to modify the connection to the device or to control the device.
- Programmed I/O (PIO) is used by character drivers to transfer small quantities of data synchronously.
- Memory mapping enables the user process to perform PIO for itself.
- Direct memory access (DMA) is used by block device drivers to transfer larger quantities of data asynchronously under device control.

The following topics present a conceptual overview of the relationship between the user process, the kernel, and the kernel-level device driver. The software architecture that supports these interactions is documented in detail in Part III, “Kernel-Level Drivers,” especially Chapter 8, “Structure of a Kernel-Level Driver.”

Overview of Device Open

Before a user process can use a kernel-controlled device, the process must open the device as a file. A high-level overview of this process, as it applies to a character device driver, is shown in Figure 3-1.

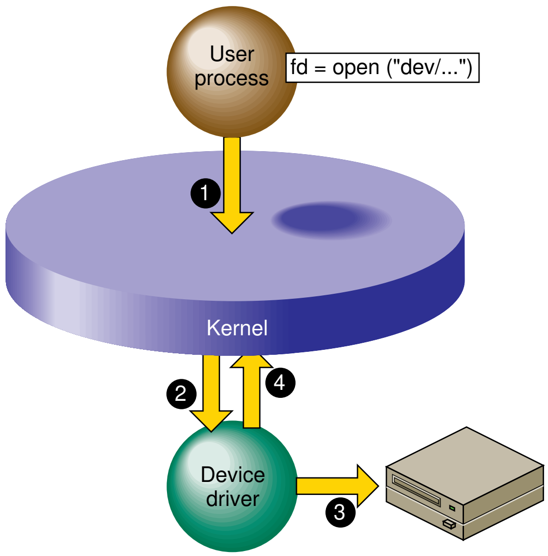

Figure 3-1 Overview of Device Open

The steps illustrated in Figure 3-1 are:

1. The user process calls the **open()** kernel function, passing the name of a device special file (see "Device Special Files" on page 32 and the open(2) reference page).
2. The kernel notes the device major and minor numbers from the inode of the device special file (see "Device Representation" on page 33). The kernel uses the major device number to select the device driver, and calls the driver's open entry point, passing the minor number and other data.
3. The device driver verifies that the device is operable, and prepares whatever is needed to operate it.
4. The device driver returns a return code to the kernel, which returns either an error code or a *file descriptor* to the process.

It is up to the device driver whether the device can be used by only one process at a time, or by more than one process. If the device can support only one user, and is already in use, the driver returns the EBUSY error code.

The **open()** interaction on a block device is similar, except that the operation is initiated from the filesystem code responding to a **mount()** request, rather than coming from a user process **open()** request (see the mount(1) reference page).

There is also a **close()** interaction so a process can terminate its connection to a device.

Overview of Device Control

After the user process has successfully opened a character device, it can request control operations. Figure 3-2 shows an overview of this operation.

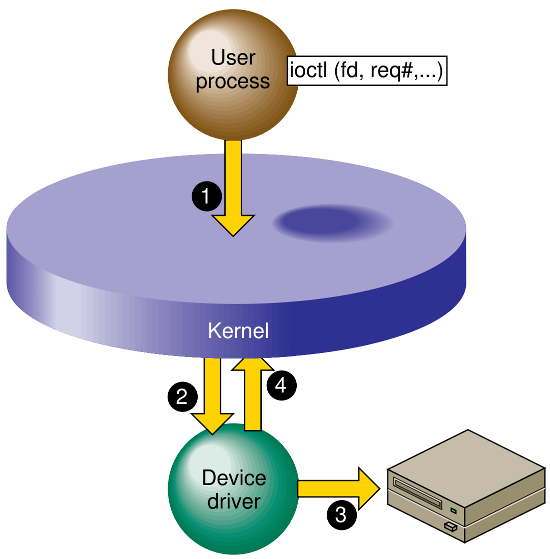

Figure 3-2 Overview of Device Control

The steps illustrated in Figure 3-2 are:

1. The user process calls the **ioctl()** kernel function, passing the file descriptor from open and one or more other parameters (see the ioctl(2) reference page).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number, the request number, and an optional third parameter from **ioctl()**.
3. The device driver interprets the request number and other parameter, notes changes in its own data structures, and possibly issues commands to the device.

4. The device driver returns an exit code to the kernel, and the kernel (then or later) redispatches the user process.

Block device drivers are not asked to provide a control interaction. The user process is not allowed to issue **ioctl()** for a block device.

The interpretation of ioctl request codes and parameters is entirely up to the device driver. For examples of the range of ioctl functions, you might review some reference pages in volume 7, for example, termio(7), ei(7), and arp(7P).

Overview of Programmed Kernel I/O

Figure 3-3 shows a high-level overview of data transfer for a character device driver that uses programmed I/O.

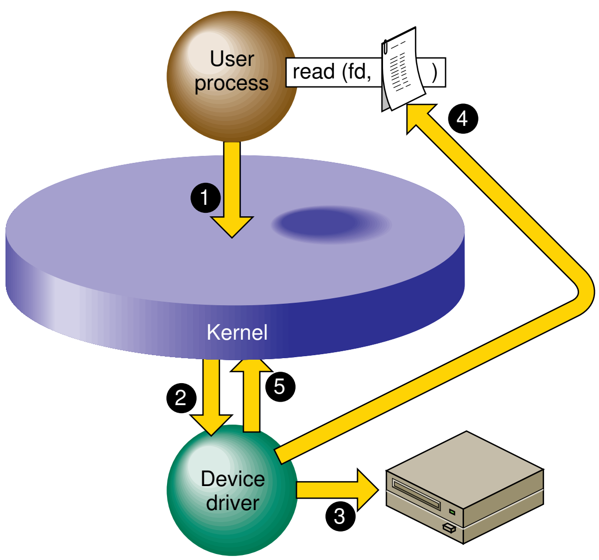

Figure 3-3 Overview of Programmed Kernel I/O

The steps illustrated in Figure 3-3 are:

1. The user process invokes the **read()** kernel function for the file descriptor returned by **open()** (see the read(2) and write(2) reference pages).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and other information.

3. The device driver directs the device to operate by storing into its registers in physical memory.
4. The device driver retrieves data from the device registers and uses a kernel function to store the data into the buffer in the address space of the user process.
5. The device driver returns to the kernel, which (then or later) dispatches the user process.

The operation of **write()** is similar. A kernel-level driver that uses programmed I/O is conceptually simple since it is basically a subroutine of the kernel.

Overview of Memory Mapping

It is possible to allow the user process to perform I/O directly, by mapping the physical addresses of device registers into the address space of the user process. Figure 3-4 shows a high-level overview of this interaction.

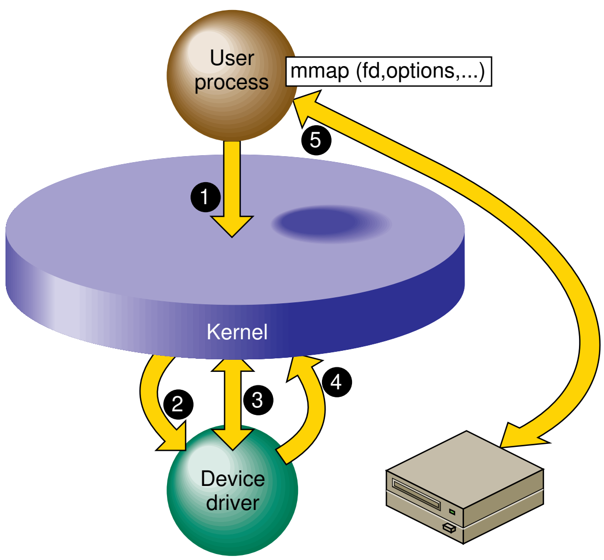

Figure 3-4 Overview of Memory Mapping

The steps illustrated in Figure 3-4 are:

1. The user process calls the **mmap()** kernel function, passing the file descriptor from open and various other parameters (see the mmap(2) reference page).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and certain other parameters from **mmap()**.
3. The device driver validates the request and uses a kernel function to map the necessary range of physical addresses into the address space of the user process.
4. The device driver returns an exit code to the kernel, and the kernel (then or later) redispatches the user process.
5. The user process accesses data in device registers by accessing the virtual address returned to it from the **mmap()** call.

Memory mapping can be supported by either a character or a block device driver. When a block device driver supports it, the memory mapping request comes from the filesystem when it responds to an **mmap()** call to map a file into memory. The filesystem calls the driver to map different pages of the file into memory, as required.

Memory mapping by a character device driver has the purpose of making device registers directly accessible to the process as memory addresses.

The Silicon Graphics device drivers for the VME and EISA buses support memory mapping. This enables user-level processes to perform PIO to devices on these buses, as described under "EISA Mapping Support" on page 42 and "VME Mapping Support" on page 42.

A memory-mapping character device driver is very simple; it needs to support only **open()**, **mmap()**, and **close()** interactions. Data throughput can be higher when PIO is performed in the user process, since the overhead of the **read()** and **write()** system calls is avoided.

It is possible to write a kernel-level driver that only maps memory, and controls no device at all. Such drivers are called *pseudo-device* drivers. For examples of psuedo-device drivers, see the prf(7) and imon(7) reference pages.

Overview of DMA I/O

Block devices and block device drivers normally use DMA (see "Direct Memory Access" on page 11). With DMA, the driver can avoid the time-consuming process of transferring data between memory and device registers. Figure 3-5 shows a high-level overview of a DMA transfer.

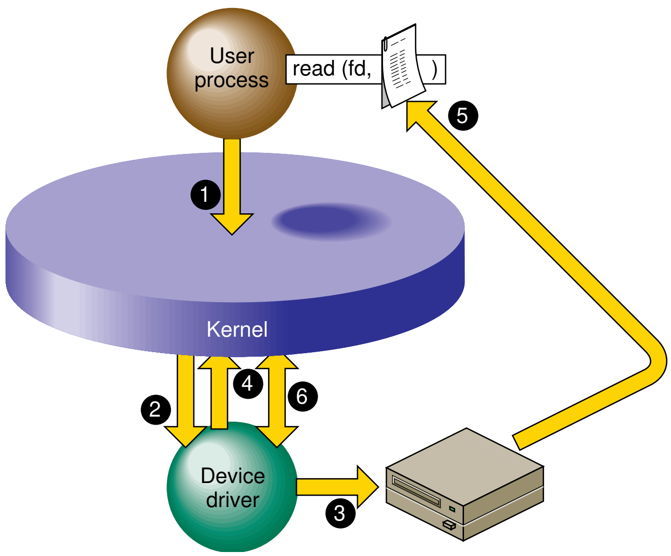

Figure 3-5 Overview of DMA I/O

The steps illustrated in Figure 3-5 are:

1. The user process invokes the **read()** kernel function for a normal file descriptor (not necessarily a device special file). The filesystem (not shown) asks for a block of data.
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and other information.
3. The device driver uses kernel functions to locate the filesystem buffer in physical memory; then programs the device with target addresses by storing into its registers.
4. The device driver returns to the kernel after telling it to block the user process from running.

5. The device itself stores the data to the physical memory locations that represent the buffer in the user process address space. During this time the kernel may dispatch other processes.
6. When the device presents a hardware interrupt, the kernel invokes the device driver. The driver notifies the kernel that the user process can now resume execution. The filesystem code moves the requested data into the user process buffer.

DMA is fundamentally asynchronous. There is no necessary timing relation between the operation of the device performing its operation and the operation of the various user processes. A DMA device driver has a more complex structure because it must deal with such factors as

- programming a device to store into a page buffer in physical memory
- scheduling operations, such as blocking a user process and waking it up when the operation is complete
- asynchronous interrupts from the device
- the possibility that requests from other processes can occur while the device is operating
- the possibility that a device interrupt can occur while the driver is handling a request

When a DMA driver permits multiple processes to open its device, it must be able to cope with the possibility that it can receive several requests from different processes while the device is busy handling one operation. This implies that the driver must implement some method of queuing requests until they can be serviced in turn.

The mapping between physical memory and process address space can be complicated. For example, the buffer can span multiple pages, and the pages need not be in contiguous locations in physical memory. If the device supports *scatter/gather*, it can be programmed with the starting addresses and lengths of each page in the buffer. If it does not support scatter/gather, the device driver has to program a separate DMA operation for each page or part of a page—or else has to obtain a contiguous buffer in the kernel address space, do the I/O from that buffer, and copy the data from that buffer to the process buffer.

The reward for the extra complexity of DMA is the possibility of much higher performance. The device can store or read data from memory at its maximum rated speed, while other processes can execute in parallel.

Upper and Lower Halves

When a device can produce hardware interrupts, its kernel-level device driver has two distinct logical parts, called the "upper half" and the "lower half" (although the upper "half" usually has much more than half the code).

Driver Upper Half

The upper half of a driver comprises all the parts that are invoked as a result of user process calls: the driver entry points that execute in response to **open()**, **close()**, **ioctl()**, **mmap()**, **read()** and **write()**.

These parts of the driver are called on behalf of a specific process. This is referred to as "having user context," which means that they are executed under the identity of a specific process.

As a result, code in the upper half of the driver is allowed to request kernel services that can be delayed, or "sleep." For example, code in the upper half of a driver can call **kmem_alloc()** to request memory in kernel space, and can specify that if memory is not available, the driver can sleep until memory is available. Also, code in the upper half can wait on a semaphore until some event occurs, or can seize a lock knowing that it may have to sleep until the lock is released.

In each case, the entire kernel does not "sleep." The driver upper half sleeps under the identity of the user process; the kernel dispatches other processes to run. When the blocking condition is removed—when memory is available, the semaphore is posted, or the lock is released—the driver is scheduled for execution and resumes.

Driver Lower Half

The lower half of a driver comprises the code that is called to respond to a hardware interrupt. An interrupt can occur at almost any time, including large parts of the time when the kernel is executing other services, including driver upper halves, and even driver lower halves for devices with lower-priority interrupts.

The kernel is not in a known state when executing a driver lower half, and there is no process context. Several things follow from this fact:

- It is very important that the interrupt be handled in the absolute minimum of time, since it may be delaying a kernel service or even the handling of a lower-priority interrupt.
- The lower-half code may not use any kernel service that can sleep (because there is no dispatchable process to be blocked and dispatched again later). Every authorized kernel service is documented as to whether it can sleep or not.

Relationship Between Halves

Each half has its proper kind of work. In general terms, the upper half performs all validation and preparation, including allocating and deallocating memory and copying data between address spaces. It initiates the first device operation of a series and queues other operations. Then it waits on a semaphore.

The lower half verifies the correct completion of an operation. If another operation is queued, it initiates that operation. Then it posts the semaphore to awaken the upper half, and exits.

Layered Drivers

IRIX allows for "layered" device drivers, in which one driver operates the actual hardware and the driver at the higher layer presents the programming interface. This approach is implemented for SCSI devices: actual management of the SCSI bus is delegated to a set of Host Adapter drivers. Drivers for particular kinds of SCSI devices call the Host Adapter driver through an indirect table to execute SCSI commands. SCSI drivers and Host Adapter drivers are discussed in detail in Chapter 15, "SCSI Device Drivers."

Combined Block and Character Drivers

A block device driver is called indirectly, from the filesystem, and it is not allowed to support the **ioctl()** entry point. In some cases, block devices can also be thought of as character devices. For example, a block device might return a string of diagnostic information, or it might be sensitive to dynamic control settings.

It is possible to support *both* block and character access to a device: block access to support filesystem operations, and character access in order to allow a user process (typically one started by a system administrator) to read, write, or control the device directly.

For example, the Silicon Graphics disk device drivers support both block and character access to disk devices. This is why you can find every disk device represented as a block device in the */dev/dsk* directory and again as a character device in */dev/rdsk* ("r" for "raw," meaning character devices).

Drivers for Multiprocessors

Many Silicon Graphics computers have multiple CPUs that execute concurrently. The CPUs share access to the single main memory, including a single copy of the kernel address space. In principle, all CPUs can execute in the kernel code simultaneously. In principle, the upper half of a device driver could be entered simultaneously by as many different processes are there are CPUs in the system (up to 36 in a Challenge or Onyx system).

A device driver written for a uniprocessor system cannot tolerate concurrent execution by multiple CPUs. For example, a uniprocessor driver has scalar variables whose values would be destroyed if two or more processes updated them concurrently.

In order to make uniprocessor drivers work in multiprocessors, IRIX by default uses only CPU 0 to execute calls to upper-half code of character and STREAMS drivers. This ensures that at most one process executes in any upper half at one time. (Network and block device drivers do not receive this service.)

It is not difficult to design a kernel-level driver to execute safely in any CPU of a multiprocessor. Each critical data object must be protected by a lock or semaphore, and particular techniques must be used to coordinate between the upper and lower halves. These techniques are discussed in "Planning for Multiprocessor Use" on page 171.

When you have made a driver multiprocessor-safe, you compile it with a particular flag value that IRIX recognizes. From then on, the driver upper half is executed on any CPU of a multiprocessor. This can improve performance, since processes that use the driver are not required to wait for CPU 0 to be available.

Loadable Drivers

Some drivers are needed whenever the system is running, but others are needed only occasionally. IRIX allows you to create a kernel-level device driver or STREAMS driver that is not loaded at boot time, but only later when it is needed.

A loadable driver has the same purposes as a nonloadable one, and uses the same interfaces to do its work. A loadable driver can be configured for automatic loading when its device is opened. Alternatively it can be loaded on command using the *ml* program (see the ml(1) and mload(4) reference pages).

A loadable driver remains in memory until its device is no longer in use, or until the administrator uses *ml* to unload it. A loadable driver remains in memory indefinitely, and cannot be unloaded, unless it provides a *pfx***unload()** entry point (see “Entry Point unload()” on page 167).

There are some small differences in the way a loadable driver is compiled and configured (see “Configuring a Loadable Driver” on page 237).

One operational difference is that a loadable driver is not available in the miniroot, the standalone system administration environment used for emergency maintenance. If a driver might be required in the miniroot, it can be made nonloadable, or it can be configured for “autoregistration” (see “Registration” on page 239).