Chapter 8

Structure of a Kernel-Level Driver

A kernel-level device driver supplies services to the kernel through a set of entry points in the driver. When events occur, the kernel calls these entry points. The driver takes action and returns a result code.

This chapter discusses when the driver entry points are called, what parameters they receive, and what actions they are expected to take. For a conceptual overview of the kernel and drivers, see "Kernel-Level Device Control" on page 45. For details on how a driver is compiled, linked, and added to IRIX, see Chapter 10, "Building and Installing a Driver."

Note: This chapter discusses device drivers. The entry point conventions for STREAMS drivers are covered in Chapter 19, "STREAMS Drivers."

The primary topics covered in this chapter are:

- "Summary of Driver Structure" on page 136 summarizes the entry points and how they are made known to the kernel.
- "Driver Flag Constant" on page 140 documents a public constant the driver must supply.
- "Initialization Entry Points" on page 143 documents the entry points that are called at boot time and when a loadable driver is loaded.
- "Open and Close Entry Points" on page 146 documents the entry points called by the **open()** and **close()** kernel functions.
- "Control Entry Point" on page 150 documents the entry point called by the **ioctl()** kernel function.
- "Data Transfer Entry Points" on page 151 documents the entry points called by the **read()** and **write()** kernel functions.
- "Poll Entry Point" on page 155 documents the entry point called by the **poll()** kernel function.

- “Memory Map Entry Points” on page 158 documents the entry points called by the **mmap()** kernel function.
- “Interrupt Entry Point” on page 163 documents the entry point called to handle a device interrupt.
- “Support Entry Points” on page 167 documents the entry points that support kernel operations and system administration.
- “Planning for Multiprocessor Use” on page 171 points out methods for writing a multiprocessor-aware driver.

Summary of Driver Structure

A driver consists of a binary object module in ELF format stored in the */var/sysgen/boot* directory. As a program, the driver consists of a set of functional entry points that supply services to the IRIX kernel.

The entry points that a driver supports must be named according to a specified convention. The *lboot* command uses entry point names to build tables used by the kernel. No single driver supports all possible entry points.

Entry Point Naming and lboot

The device driver makes known which entry points it supports by giving them public names in its object module. The *lboot* command links together the object modules of drivers and other kernel modules to make a bootable kernel. *lboot* recognizes the entry points by the form of their names.

Driver Name Prefix

A device driver must be described by a file in the */var/sysgen/master.d* directory (see “Master Configuration Database” on page 38). One of the items in that configuration file specifies the driver *prefix*, a string of 1 to 14 characters that is unique to that driver. For example, the prefix of the SCSI driver is *scsi_*.

The prefix string is defined in the */var/sysgen/master.d* file only. The string does not have to appear as a constant in the driver, and the name of the driver object file does not have to correspond to the prefix (although it typically has a related name).

The *lboot* command recognizes driver entry points by searching the driver object module for public names that begin with the prefix string. For example, the entry point for the **open()** operation must have a name that consists of the prefix string followed by the letters "open."

In this book, entry point names are written as follows: *pfx***open**, where *pfx* stands for the driver's prefix string.

Kernel Switch Tables

The IRIX kernel maintains tables that allow it to dispatch calls to device drivers quickly. These tables are built by *lboot* based on the device major numbers and the names of the driver entry points. The tables are named as follows:

bdevsw	Table of block device drivers
cdevsw	Table of character device drivers
fmodsw	Table of STREAMS drivers
vfssw	Table of filesystem modules (not related to device drivers)

The tables for block and character drivers have one row for each major device number, and one column for each possible driver entry point. As *lboot* loads a driver, it fills in that driver's row of a switch table with the addresses of the driver's entry points. Where an entry point is not defined, *lboot* leaves the address of a null routine that returns the ENODEV error code.

The sizes of the switch tables are fixed at boot time in order to minimize kernel data space. The table sizes are tunable parameters that can be set with *systune* (see the systune(1) reference page).

When a driver is loaded dynamically (see "Configuring a Loadable Driver" on page 237), the associated row of the switch table is not filled at link time but rather is filled when the driver is loaded. When you add new, loadable drivers, you might need to specify a larger switch table. The *IRIX Administration: System Configuration and Operation* book documents these tunable parameters.

Entry Point Summary

The names of all possible driver entry points and their purposes are summarized in Table 8-1. STREAMS drivers are covered in Chapter 19.

Table 8-1 Entry Points in Alphabetic Order

Entry Point	Purpose	Discussion
*pfx***close**	Note the device is not in use.	page 149
*pfx***devflag**	Constant flag bits for driver features.	page 140
*pfx***edtinit**	Initialize driver from VECTOR statement.	page 144
*pfx***halt**	Prepare for system shutdown.	page 168
*pfx***init**	Initialize driver at load or boot time.	page 144
*pfx***intr**	Handle device interrupt.	page 164
*pfx***ioctl**	Implement control operations.	page 150
*pfx***map**	Implement memory-mapping (IRIX).	page 159
*pfx***mmap**	Implement memory-mapping (SVR4).	page 161
*pfx***open**	Connect a process to a device. Connect a stream module.	page 146 page 543
*pfx***poll**	Implement device event test.	page 156
*pfx***print**	Display diagnostic about block device.	page 169
*pfx***read**	Implement device input.	page 152
*pfx***rput**	STREAMS message on read queue.	page 544
*pfx***size**	Return logical size of block device.	page 169
*pfx***srv**	STREAMS service queued messages.	page 545
*pfx***start**	Initialize driver with interrupts enabled.	page 145
*pfx***strategy**	Input/output for a block device.	page 154
*pfx***unload**	Prepare loadable module for unloading.	page 167
*pfx***unmap**	Undo *pfx***mmap** memory mapping.	page 162

Table 8-1 (continued) Entry Points in Alphabetic Order

Entry Point	Purpose	Discussion
*pfx***wput**	STREAMS message on write queue.	page 544
*pfx***write**	Implement device output.	page 152

The use of entry points in different types of drivers is summarized in Table 8-2. The columns of Table 8-2 show the different types of drivers. The table cells show whether a given entry point is optional (O), required (R), or not allowed (N).

Table 8-2 Use of Driver Entry Points

Entry Point	Character	Block	Pseudo	STREAMS
*pfx***close**	R	R	R	R
*pfx***devflag**	O	O	O	O
*pfx***edtinit**	O	O	N	N
*pfx***halt**	O	O	O	O
*pfx***init**	O	O	O	O
*pfx***intr**	O	O	N	N
*pfx***ioctl**	O	O	O	N
*pfx***map**	O	O	O	N
*pfx***mmap**	O	O	O	N
*pfx***open**	R	R	R	R
*pfx***poll**	O	O	N	O
*pfx***print**	N	O	N	N
*pfx***read**	O	N	O	N
*pfx***rput**	N	N	N	R
*pfx***size**	N	R	N	N
*pfx***srv**	N	N	N	R
*pfx***start**	O	O	O	O

Table 8-2 (continued) Use of Driver Entry Points

Entry Point	Character	Block	Pseudo	STREAMS
*pfx***strategy**	N	R	N	N
*pfx***unload**	O	O	O	O
*pfx***unmap**	O	O	O	N
*pfx***wput**	N	N	N	R
*pfx***write**	O	N	O	N

As can be seen from Table 8-2, no driver supports all entry points.

- A minimal driver for a character device supports *pfx***init()**, *pfx***open()**, *pfx***read()**, *pfx***write()**, and *pfx***close()**. The *pfx***ioctl()** and *pfx***poll()** entry points are optional.
- A minimal pseudo-device driver supports *pfx***start()**, *pfx***open()**, *pfx***map()**, *pfx***unmap()**, and *pfx***close()** (the latter two as stubs).
- A minimal block device driver supports *pfx***edtinit()**, *pfx***open()**, *pfx***size()**, *pfx***strategy()**, and *pfx***close()**.

Driver Flag Constant

Any device driver or STREAMS module should define a public name *pfx***devflag** as a static integer. This integer contains a bitmask with zero or more of the following flags, which are declared in *sys/conf.h*:

D_MP	The driver is prepared for multiprocessor systems.
D_WBACK	The driver handles its own cache-writeback operations.
D_MT	The driver is prepared for a multithreaded kernel.
D_OLD	The driver implements IRIX 4.x semantics.

The flag names are declared in the header file *sys/ddi.h*. A typical definition would resemble the following:

```
int testdrive_devflag = D_MP;
```

A STREAMS module should also provide this flag, but the only relevant bit value for a STREAMS driver is D_MP (see "Driver Flag Constant" on page 542).

The flag value is saved in the kernel switch table with the driver's entry points (see "Kernel Switch Tables" on page 137).

When a driver does not define a *pfx***devflag**, *lboot* saves a word containing D_OLD by default. See the note regarding D_OLD on page 142.

Flag D_MP

You specify D_MP in *pfx***devflag** to tell *lboot* that your driver is designed to operate in a multiprocessor system. The top half of the driver is designed to cope with multiple concurrent entries in multiple CPUs. The top and bottom halves synchronize through the use of semaphores or locks and do not rely on interrupt masking for critical sections. These issues are discussed further under "Planning for Multiprocessor Use" on page 171.

When D_MP is not present in *pfx***devflag**, IRIX ensures that the driver code, including the upper-half entry points and the interrupt handler, executes only on CPU 0 of a multiprocessor. This ensures behavior similar to a uniprocessor, but can cause a performance bottleneck when either the device or CPU 0 is heavily used.

Note: This flag is only tested when loading a character driver or STREAMS driver. There is no special handling for block drivers and network drivers in multiprocessors. Block and network drivers must be multiprocessor-aware.

Flag D_WBACK

You specify D_WBACK in *pfx***devflag** to tell *lboot* that a block driver performs any necessary cache write-back operations through explicit calls to **dki_dcache_wb()** and related functions (see the dki_dcache_wb(D3) reference page).

When D_WBACK is not present in *pfx***devflag**, the **physiock()** function ensures that all cached data related to *buf_t* structures is written back to main memory before it enters the driver's strategy routine. (See the physiock(D3) reference page and "Entry Point strategy()" on page 154.)

Flag D_MT

This flag is defined in IRIX 6.2 but has no effect in that release. Future releases of IRIX will run driver interrupt routines as lightweight threads of control within the kernel address space. D_MT indicates that this driver understands that it can be run as one or more cooperating threads, and uses kernel synchronization primitives to serialize access to driver common data structures.

Flag D_OLD

The D_OLD flag exists only to retain compatibility with certain drivers written originally for IRIX 4.x. It changes two features of the kernel-to-driver interface:

- The first argument to the *pfx***open()** entry is a *dev_t* value instead of the pointer-to-*dev_t* that is now standard.
- The driver sets its return code by storing it into a global, *u.u_error*, instead of returning it as the result of the function call.

D_OLD is incompatible with D_MP.

When a driver has no *pfx***devflag** constant, *lboot* assumes it is a D_OLD driver.

Note: The D_OLD flag value, and the incompatible interface that it implies, is supported for compatiblity only. Support for this flag, and support for drivers that use it (or that have no *pfx***devflag** constant), will be withdrawn in the release of IRIX after 6.2. It may be removed from certain platform-specific releases of IRIX 6.2. Silicon Graphics urges you to revise any driver that depends on D_OLD to use current semantics for parameters and return codes.

Initialization Entry Points

The kernel calls a driver to initialize itself at any of three different entry points, as follows:

*pfx***init**	Initialize self-defining hardware or a pseudo-device.
*pfx***edtinit**	Initialize a hardware device based on VECTOR data.
*pfx***start**	General initialization.

Each call has different abilities. A driver may define any combination of the three entry points. It is not uncommon to define both a *pfx***start()** and one of *pfx***edtinit()** or *pfx***init()**.

When Initialization Is Performed

The initialization entry points of ordinary (nonloadable) drivers are called during system startup, after interrupts have been enabled and before the message "The system is coming up" is displayed. In all cases, interrupts are enabled and basic kernel services are available at this time. However, other loadable or optional kernel modules might not have been initialized, depending on the sequence of statements in the files in */var/sysgen/system*.

Whenever a driver is initialized, the entry points are called in the following sequence:

1. *pfx***init()** is called first.
2. *pfx***edtinit()** is called once for each VECTOR statement in reverse order of the VECTOR statements.
3. *pfx***start()** is called last.

Initialization of Loadable Drivers

When a loadable driver (see "Loadable Drivers" on page 57) is configured for autoregister, it is loaded with other drivers during system startup. (For more information on autoregister, see "Configuring a Loadable Driver" on page 237.) Such a driver is initialized at system startup time along with the nonloadable drivers.

When a loadable driver is dynamically loaded at some time after system startup, its initialization entry points are called in sequence at the time it is loaded.

Entry Point init()

The *pfx***init()** entry point is called once during system startup or when a loadable driver is loaded. It receives no input arguments; its prototype is simply:

```
void pfxinit(void);
```

You can use this entry point to initialize a hardware device that is self-defining; that is, all the information the driver needs is either coded into the driver, or can be gotten by probing the device itself. You can also use *pfx***init()** to initialize a pseudo-device driver; that is, a driver that does not have real hardware attached.

A driver that is brought into the system by a USE or INCLUDE line in a system configuration file (see "Configuring a Kernel" on page 236) typically initializes in the *pfx***init()** entry point.

Entry Point edtinit()

The *pfx***edtinit()** entry is designed to initialize devices that are configured using the VECTOR statement in the system configuration file (see "System Configuration Files" on page 39). The entry point name is a contraction of "*e*arly *d*evice *t*able *init*ialization."

The VECTOR statement specifies hardware details about a device on the VME, GIO, or EISA bus, including iospace addresses, interrupt level, and an integer parameter. The VECTOR statement can specify a "probe" parameter that lets the kernel test for the existence of the specified hardware.

When the kernel processes a VECTOR statement during bootstrap and the probe is successful (or no probe is specified), the kernel stores the VECTOR parameters in a structure of type *edt_t*. (This structure is declared in *sys/edt.h*.)

Each time the kernel loads a driver that is named in a VECTOR statement, the kernel calls the driver's *pfx***edtinit()** entry one time for each VECTOR statement that named that driver and had a successful probe (or that had no probe). VECTOR statements are processed in reverse sequence to the order in which they are coded in */var/sysgen/system* files.

The prototype of the *pfx***edtinit()** entry is

```
void pfxedtinit(edt_t *e);
```

The *edt_t* contains at least the following fields (see the system(4) reference page for the corresponding VECTOR parameters):

e_bus_type	Integer specifying the bus type; constant values are declared in *sys/edt.h*, for example ADAP_VME, ADAP_GIO, or ADAP_EISA.
e_adap	Integer specifying the adapter (bus) number.
e_ctlr	Value from the VECTOR *ctlr*= parameter; typically the device minor number.
e_space	Array of up to three I/O space structures of type *iospace_t*.

The difference between *pfx***init()** and *pfx***edtinit()** is that *pfx***edtinit()** is parameterized with information from the VECTOR line, and is called once for each VECTOR line that is associated with real hardware.

A driver that uses *pfx***edtinit()** needs to save the *edt_t* information in a data structure. If the driver supports multiple devices—that is, if it can be called for multiple VECTOR statements—it needs to allocate an array or chain of structures, and save new data on each entry.

Entry Point start()

The *pfx***start()** entry point is called at system startup, and whenever a loadable driver is loaded. It is called after *pfx***edtinit()** and *pfx***init()**, but before any other entry point such as *pfx***open()**. The *pfx***start()** entry point receives no arguments; its prototype is simply

```
void pfxstart(void);
```

The *pfx***start()** entry point is a suitable place to allocate a poll-head structure using **phalloc()**, as discussed in "Use and Operation of poll(2)" on page 155.

Open and Close Entry Points

The *pfx***open()** and *pfx***close()** entries for block and character devices are called when a device comes into use and when use of it is finished. For a conceptual overview of the **open()** process, see "Overview of Device Open" on page 46.

Entry Point open()

The kernel calls a device driver's *pfx***open()** entry when a process executes the **open()** system call on any device special file (see the open(2) reference page). It is also called when a process executes the **mount()** system call on a block device (see the mount(2) reference page). (For the *pfx***open()** entry point of a STREAMS driver, see "Entry Point open()" on page 543.)

The prototype of *pfx***open()** is as follows:

```
int pfxopen(dev_t *devp, int oflag, int otyp, cred_t *crp);
```

The argument values are

**devp* — Pointer to a *dev_t* value from which you can extract both the major and minor device numbers.

otyp — An integer flag specifying the source of the call: a user process opening a character device or block device, or another driver.

oflag — Flag bits specifying user mode options on the **open()** call.

crp — A *cred_t* object—an opaque structure for use in authentication. Standard access privileges to the special device file have already been verified.

Note: When the driver's *pfx***devflag** entry contains D_OLD or when *pfx***devflag** is not defined, the first argument to *pfx***open()** is a *dev_t* value, not a pointer to a *dev_t* value. See "Flag D_OLD" on page 142.

In general, the driver is expected to verify that this user process is permitted access in the way specified in *otyp* (reading, writing, or both) for the device specified in **devp*. If access is not allowable, the driver returns a nonzero error code from *sys/errno.h*, for example ENOMEM or EBUSY.

Use of the Device Number

When the driver supports a single device with no logical unit divisions, the device number is of little interest except for diagnostic displays. When the driver supports multiple devices, or a device with multiple logical units, the minor device number is the key to locating the device information. The device number can also encode device options, as discussed under "Minor Device Number" on page 35.

When the driver supports the *pfx***edtinit()** entry, the driver needs a way to associate the different *edt_t* structures passed to *pfx***edtinit()** with the device numbers passed to *pfx***open()** and other routines. One solution is to require that the *ctlr*= value from the VECTOR statement—which is passed in the *e_ctlr* field of *edt_t*—must be the same as the device minor number.

Use of the Open Type

The *otyp* flag distinguishes between the following possible sources of this call to *pfx***open()** (the constants are defined in *sys/open.h*).

- a call to open a character device (OTYP_CHR)
- a call to open a block device (OTYP_BLK)
- a call to a mount a block device as a filesystem (OTYP_MNT)
- a call to open a block device as swapping device (OTYP_SWP)
- a call direct from a device driver at a higher level (OTYP_LYR)

Typically a driver is written only to be a character driver or a block driver, and can be called only through the switch table for that type of device. When this is the case, the *otyp* value has little use.

It is possible to have the same driver treated as both block and character, in which case the driver needs to know whether the **open()** call addressed a block or character special device. It is possible for a block device to support different partitions with different uses, in which case the driver might need to record the fact that a device has been mounted, or opened as a swap device.

With all open types except OTYP_LYR, *pfx***open()** is called for every open or mount operation, but *pfx***close()** is called only when the last close or unmount occurs. The OTYP_LYR feature is used almost exclusively by drivers distributed with IRIX, like the host adapter SCSI driver (see "Host Adapter Concepts" on page 355). For each open of this type, there is one call to *pfx***close()**.

Use of the Open Flag

The interpretation of the open mode flags is up to the designer of the driver. Four modes can be requested (declared in *sys/file.h*):

FREAD	Input access wanted.
FWRITE	Output access wanted (both FREAD and FWRITE may be set, corresponding to O_RDWR mode).
FNDELAY or FNONBLOCK	Return at once, do not sleep if the open cannot be done immediately.
FEXCL	Request exclusive use of the device.

You decide which of the flags have meaning with respect to the abilities of this device. You can return an EINVAL error when an unsupported mode is requested.

A key decision is whether the device can be opened only by one process at a time, or by multiple processes. If multiple opens are supported, a process can still request exclusive access with the FEXCL mode.

When the device can be used by only one process, or when FEXCL access is supported, the driver must keep track of the fact that the device is open. When the device is busy, the driver can test the FNDELAY and FNONBLOCK flags; if either is set, it can return EBUSY. Otherwise, the driver should sleep until the device is free; this requires coordination with the *pfx***close()** entry point.

Use of the cred_t Object

The *cred_t* object passed to *pfx***open()**, *pfx***close()**, and *pfx***ioctl()** can be used with the **drv_priv()** function to find out if the effective calling user ID is privileged or not (see the drv_priv(D3) reference page). Do not examine the object in detail, since its contents are subject to change from release to release.

Saving the Size of a Block Device

In a block device driver, the *pfx***size()** entry point will be called soon after *pfx***open()** (see “Entry Point size()” on page 169). It is typically best to calculate or read the device capacity at open time, and save it to be reported from *pfx***size()**.

Saving the User ABI

If your driver is, or might be, compiled to the 64-bit model for use with a 64-bit IRIX kernel, and if it supports the *pfx***ioctl()** or *pfx***poll()** entry points, the driver should test and save the user process’s programming model during an open. For details, see “Handling 32-Bit and 64-Bit Execution Models” on page 170.

Entry Point close()

The kernel calls the *pfx***close()** entry when the last process calls **close()** or **umount()** for the device special file. It is important to know that when the device can be opened by multiple processes, *pfx***close()** is not called for every **close()** function, but only when the last remaining process closes the device and no other processes have it open.

The function prototype and arguments of *pfx***close()** are

```
int pfxclose(dev_t dev, int flag, int otyp, cred_t *crp);
```

The arguments are the same as were passed to *pfx***open()**. However, the flag argument is not necessarily the same as at any particular call to **open()**.

It is up to you to design the meaning of “close” for this type of device:

- If the device is opened and closed frequently, you may decide to retain dynamic data structures.
- If the device can perform an action such as “rewind” or “eject,” you decide whether that action should be done upon close. Possibly the choice of acting or not acting can be set by an **ioctl()** call; or possibly the choice can be encoded into the device minor number—for example, the no-rewind-on-close option is encoded in certain tape minor device numbers.

- If the *pfx*__open()__ entry point supports exclusive access, and it can be waiting for the device to be free, *pfx*__close()__ must release the wait.

The *pfx*__close()__ entry can detect an error and report it with a return code. However, the file is closed or unmounted regardless.

Control Entry Point

The *pfx*__ioctl()__ entry point is called by the kernel when a user process executes the **ioctl()** system call (see the ioctl(2) reference page). This entry point is allowed in character drivers only. Block device drivers do not support it, and STREAMS drivers pass control information as messages.

For an overview of the relationship between the user process, kernel, and the control entry point, see "Overview of Device Control" on page 48.

The prototype of the entry point is

```
int pfxioctl(dev_t dev, int cmd, void *arg,
            int mode, cred_t *crp, int *rvalp);
```

The argument values are

dev A *dev_t* value from which you can extract the major and minor device numbers.

cmd The request value specified in the **ioctl()** call.

arg The optional argument value specified in the **ioctl()** call, or NULL if none was specified.

mode Flag bits specifying the **open()** mode, as associated with the file descriptor passed to the **ioctl()** system function.

crp A *cred_t* object—an opaque structure for use in authentication, describing the process that is in-context. Standard access privileges to the special device file have already been verified.

**rvalp* The integer result to be returned to the user process.

It is up to the device driver to interpret the *cmd* and *arg* values in the light of the *mode* and other arguments. When the *arg* value is a pointer to data in the process address space, the driver uses the **copyin()** kernel function to copy the data into kernel space, and the **copyout()** function to return updated values. (See the copyin(D3) and copyout(D3) reference pages, and also “Transferring Data” on page 192.)

Choosing the Command Numbers

The command numbers supported by *pfx***ioctl()** are arbitrary; but the recommended practice is to make sure that they are different from those of any other driver. One method to achieve this is suggested in the ioctl(D2) reference page.

Supporting 32-Bit and 64-Bit Callers

The **ioctl()** entry point may need to interpret a structure prepared in the user process. In a 64-bit system, the user process can be either a 32-bit or a 64-bit program. For discussion of this issue, see “Handling 32-Bit and 64-Bit Execution Models” on page 170

User Return Value

The kernel returns 0 to the **ioctl()** system function unless the *pfx***ioctl()** function returns an error code. In the event of an error, the kernel returns the code the driver places in **rvalp*, if any, or -1. To ensure that the user process sees a specific error code, set the code in **rvalp*, and return that value.

Data Transfer Entry Points

The *pfx***read()** and *pfx***write()** entry points are supported by character device drivers and pseudo-device drivers that allow reading and writing. They are called by the kernel when the user process calls the **read()**, **readv()**, **write()**, or **writev()** system function.

The *pfx***strategy()** entry point is required of block device drivers. It is called by the kernel when either a filesystem or the paging subsystem needs to transfer a block of data.

Entry Points read() and write()

The *pfx***read()** and *pfx***write()** entry points are similar to each other—only the direction of data transfer differs. The prototypes of the functions are

```
int pfxread (dev_t dev, uio_t *uiop, cred_t *crp);
int pfxwrite(dev_t dev, uio_t *uiop, cred_t *crp);
```

The arguments are

dev — A *dev_t* value from which you can extract both the major and minor device numbers.

**uiop* — A *uio_t* object—a structure that defines the user's buffer memory areas.

crp — A *cred_t* object—an opaque structure for use in authentication. Standard access privileges to the special device file have already been verified.

Data Transfer for a PIO Device

A character device driver using PIO transfers data in the following steps:

1. If there is a possibility of a timeout, start a timeout delay (see “Waiting for Time to Pass” on page 214).
2. Initiate the device operation as required.
3. Transfer data between the device and the buffer represented by the *uio_t* (see “Transferring Data Through a uio_t Object” on page 194).
4. If it is necessary to wait for an interrupt, put the process to sleep (see “Waiting and Mutual Exclusion” on page 204).
5. When data transfer is complete, or when an error occurs, clear any pending timeout and return the final status of the operation. If the return code is 0, the final state of the *uio_t* determines the byte count returned by the **read()** or **write()** call.

Calling Entry Point strategy() From Entry Point read() or write()

A device driver that supports both character and block interfaces must have a *pfx***strategy()** routine in which it performs the actual I/O. For example, the Silicon Graphics disk drivers support both character and block driver interfaces, and perform all I/O operations in the *pfx***strategy()** function. However, the *pfx***read()**, *pfx***write()** and *pfx***ioctl()** entries supported for character-type access also need to perform I/O operations. They do this by calling the *pfx***strategy()** routine indirectly, using the kernel function **physiock()** or **uiophysio()** (see the physiock(D3) and uiophysio(D3) reference pages, and see "Waiting for Block I/O to Complete" on page 217).

Both the **physiock()** and **uiophysio()** functions takes care of the housekeeping needed to interface to the *pfx***strategy()** entry, including the work of allocating a buffer and a *buf_t* structure, locking buffer pages in memory and waiting for I/O completion. The **physiock()** function is preferable because it is compatible with SVR4. Example 8-1 shows the skeleton of a hypothetical driver in which the *pfx***read()** entry does its work through the *pfx***strategy()** entry.

Example 8-1 Hypothetical pfxread() entry in a Character/Block Driver

```
hypo_read (dev_t dev, uio_t *uiop, cred_t *crp)
{
   // ...validate the operation... //
   return physiock(hypo_strategy, /* our strategy entry */
                   0,   /* allocate temp buffer & buf_t */
                   dev, /* dev_t arg for strategy */
                   B_READ, /* direction flag for buf_t */
                   uiop);
}
```

The *pfx***write()** entry would be identical except for passing B_WRITE instead of B_READ.

This dual-entry strategy is required only in a driver that supports both character and block access. **physiock()** returns an error if the length requested in the *uio_t* object is not a multiple of the standard sector length, so a different approach is required for a device that permits odd-length I/O.

Entry Point strategy()

A block device driver does not directly support user process calls. Instead, it provides services to a filesystem such as EFS, or to the memory paging subsystem of IRIX. These subsystems call the *pfx***strategy()** entry point to read data in whole blocks.

Calls to *pfx***strategy()** are not directly related in time to system functions called by a user process. For example, a filesystem may buffer many blocks of data in memory, so that the user process may execute dozens or hundreds of **write()** calls without causing an entry to the device driver. When the user function closes the file or calls **fsync()—**or when for unrelated reasons the filesystem needs to free some buffers—the filesystem calls *pfx***strategy()** to write numerous blocks of data.

In a driver that supports the character interface as well, the *pfx***strategy()** entry can be called indirectly from the *pfx***read()**, *pfx***write()** and *pfx***ioctl()** entries, as described under "Calling Entry Point strategy() From Entry Point read() or write()" on page 153.

The prototype of the *pfx***strategy()** entry point is

```
int pfxstrategy(struct buf *bp);
```

The argument is the address of a *buf_t* structure, which gives the strategy routine the information it needs to perform the I/O:

- The major and minor device numbers
- The direction of the transfer (read or write)
- The location of the buffer in kernel memory
- The amount of data to transfer
- The starting block number on the device

For more on the contents of the *buf_t* structure, see "Structure buf_t" on page 183 and the buf(D4) reference page.

The driver uses the information in the *buf_t* to validate the data transfer and programs the device to start the transfer. Then it stores the address of the *buf_t* where the interrupt handler can find it (see "Entry Point intr()" on page 164) and calls **biowait()** to wait for the operation to complete. For the next step, see "Completing Block I/O" on page 166 (see also the biowait(D3) reference page).

Poll Entry Point

The *pfx***poll()** entry point is called by the kernel when a user process calls the **poll()** or **select()** system function asking for status on a character special device. To implement it, you need to understand the IRIX implementation of **poll()**.

Use and Operation of poll(2)

The IRIX version of **poll()** allows a process to wait for events of different types to occur on any combination of devices, files, and STREAMS (see the poll(2) and select(2) reference pages). It is possible for multiple processes to be waiting for events on the same device.

It is up to you as the designer of a driver to decide which of the events that are documented in poll(2) are meaningful for your device. Other requested events simply never happen to the device.

Much of the complexity of **poll()** is handled by the IRIX kernel, but the kernel requires the assistance of any device driver that supports **poll()**. The driver is expected to allocate and hold a *pollhead* structure (declared in *sys/poll.h*) for each minor device that it supports. Allocation is simple; the driver merely calls the **phalloc()** kernel function. (The *pfx***start()** entry point is a suitable place for this call; see "Entry Point start()" on page 145.)

There are two phases to the operation of **poll()**. When the system function is called, the kernel calls the *pfx***poll()** entry point to find out if any requested events are pending at this time. If the kernel finds any event s pending (on this or any other polled object), the **poll()** function returns to the user process. Nothing further is required.

However, when no requested event has happened, the user process expects the **poll()** function to block until an event has occured. The kernel cannot implement this delay by repeatedly testing for events; that would be too inefficient. The kernel must rely on device drivers to notify it when an event has occurred.

Use of pollwakeup()

A device driver that supports *pfx***poll()** is required to notify the kernel whenever an event that the driver supports has occurred. The driver does this by calling a kernel function, **pollwakeup()**, passing the *pollhead* structure for the affected device, and bit flags for the events that have taken place. In the event that one or more user processes are blocked in a **poll()**, waiting for an event from this device, the call to **pollwakeup()** will release the sleeping processes. For an example, see "Calling pollwakeup()" on page 166.

Use of pollwakeup() Without Interrupts

If the device in question does not support interrupts, the driver cannot support **poll()** unless it can somehow get control to discover an event and report it to **pollwakeup()**. One possibility is that the driver could simulate interrupts by setting a succession of **itimeout()** delays. On each timeout the driver would test its device for a change of status, call **pollwakeup()** when an event has occurred; and schedule a new delay. (See "Waiting for Time to Pass" on page 214.)

Entry Point poll()

The prototype for *pfx***poll()** is as follows:

```
int pfxpoll(dev_t dev, short events, int anyyet,
            short *reventsp, struct pollhead **phpp);
```

The argument values are

dev	A *dev_t* value from which you can extract the major and minor device numbers.
events	Bit-flags for the events the user process is testing, as passed to **poll()** and declared in *sys/poll.h*..
reventsp	A field to receive the bit-flags of events that have occurred, or to receive 0x0000 if no requested events have occurred..
anyyet and *phpp*	When *anyyet* is nonzero and no events have occurred, the kernel requires the address of the pollhead structure for this minor device to be returned in **phpp*.

Example 8-2 shows the *pfx***poll()** code of a hypothetical device driver. Only three event tests are supported: POLLIN and POLLRDNORM (treated as equivalent) and POLLOUT. The device driver maintains an array of *pollhead* structures, one for each supported minor device. These are presumably allocated during initialization.

Example 8-2 *pfx*poll() Code for Hypothetical Driver

```
struct pollhead phds[MAXMINORS];
#define OUR_EVENTS (POLLIN|POLLOUT|POLLRDNORM)
hypo_poll(dev_t dev, short events, int anyyet,
          short *reventsp, struct pollhead **phpp)
{
   minor_t dminor = geteminor(dev);
   short happened = 0;
   short wanted = events & OUR_EVENTS;
   if (wanted & (POLLIN|POLLRDNORM))
   {
      if (device_has_data_ready(dminor))
         happened |= (POLLIN|POLLRDNORM);
   }
   if (wanted & POLLOUT)
   {
      if (device_ready_for_output(dminor))
         happened |= POLLOUT;
   }
   if (device_pending_error(dminor))
      happened |= POLLERR;
   if (0 == (*reventsp = happened))
   {
      if (anyyet) *phpp = phds[dminor]
   }
   return 0;
}
```

The code in Example 8-2 begins by discarding any unsupported event flags that might have been requested. Then it tests the remaining flags against the device status. If the device has an uncleared error, the code inserts the POLLERR event. If no events were detected, and if the kernel requested it, the address of the *pollhead* structure for this minor device is returned.

Memory Map Entry Points

A user process requests memory mapping by calling the system function **mmap()**. When the mapped object is a character device special file, the kernel calls the *pfx***mmap()** or *pfx***map()** entry to validate and complete the mapping. To understand these entry points, you must understand the **mmap()** system function.

Concepts and Use of mmap()

The purpose of the **mmap()** system function (see the mmap(2) reference page) is to make the contents of a file directly accessible as part of the virtual address space of the user process. The results depend on the kind of file that is mapped:

- When the mapped object is a normal file, the process can load and store data from the file as if it were an array in memory.
- When the mapped object is a character device special file, the process can load and store data from device registers as if they were memory variables.
- When the mapped object is a block of memory owned and prepared by a pseudo-device driver, the process gains access to some special piece of memory data that it would not normally be able to access.

In all cases, access is gained through normal load and store instructions, without the overhead of calling system functions such as **read()**. Furthermore, the same mapping can be executed by other processes, in which case the same memory, or file, or device is shared by multiple, concurrent processes. This is how shared memory segments are achieved.

Use of mmap()

The **mmap()** system function takes four key parameters:

- the file descriptor for an open file, which can be either a normal disk file or a device special file
- an offset within that file at which the mapped data is to start. For a normal file, this is a file offset; for a device file, it represents an address in the address space of the device or the bus
- the length of data to be mapped
- protection flags, showing whether the mapped data is read-only or read-write

When the mapped object is a normal file, the filesystem implements the mapping. The filesystem does not call the block device driver for assistance in mapping a file. It does call the block device driver *pfx***strategy()** entry to read and write blocks of file data as necessary, but the mapping of pages of data into pages of memory is controlled in the filesystem code.

When the mapped object is a device special file, the **mmap()** parameters are passed to the device driver at either its *pfx***mmap()** or *pfx***map()** entry point. The device driver interprets the parameters in the context of the device, and uses a kernel function to create the mapping.

Persistent Mappings

Once a device or kernel memory has been mapped into some user address space, the mapping persists until the user process terminates or calls **unmap()** (see the unmap(2) reference page). In particular, the mapping does not end simply because the device special file is closed. You cannot assume, in the *pfx***close()** or *pfx***unload()** entry points, that all mappings to devices have ended.

Entry Point map()

The *pfx***map()** entry point can be defined in either a character or a block driver (it is the only mapping entry point that a block driver can supply). The function prototype is

```
int pfxmap(dev_t dev, vhandl_t *vt,
          off_t off, int len, int prot);
```

The argument values are

dev — A *dev_t* value from which you can extract both the major and minor device numbers.

vt — The address of an opaque structure that describes the assigned address in the user process address space. The structure contents are subject to change.

off, *len* — The offset and length arguments passed to **mmap()** by the user process.

prot — Flags showing the access intentions of the user process.

The first task of the driver is to verify that the access specified in *prot* is allowed. The next task is to validate the *off* and *len* values: do they fall in the valid address space of the device?

When the device driver approves of a mapping, it uses a kernel function, **v_mapphys()**, to establish the mapping. This function (documented in the v_mapphys(D3) reference page) takes the *vhandle_t*, an address in kernel cached or uncached memory, and a length. It makes the specified region of kernel space a part of the address space of the user process.

For example, a pseudo-device driver that intends to share kernel virtual memory with user processes would first allocate the memory:

```
caddr_t *kaddr = kmem_alloc (len , KM_CACHEALIGN);
```

It would then use the address of the allocated memory with the *vhandle_t* value it had received to map the allocated memory into the user space:

```
v_mapphys (vt, kaddr, len)
```

Note: There are no special precautions to take when mapping cached memory into user space, or when mapping device registers or bus addresses. However, you should almost never map *uncached memory* into user space. The effects of uncached memory access are hardware dependent and differ between multiprocessors and uniprocessors. Among uniprocessors, the IP26 CPU module has highly restrictive rules for the use of uncached memory (see “Uncached Memory Access in the IP26 CPU” on page 27). In general, mapping uncached memory makes a driver nonportable and is likely to lead to subtle failures that are hard to resolve.

Example 8-3 contains an edited fragment of code from a Silicon Graphics device driver. This pseudo-device driver, whose prefix is *flash_*, provides access to “flash” PROM in certain computer models. It allows a user process to map the PROM into user space.

Example 8-3 Edited Fragment of flash_map()

```
int flash_map(dev_t dev, vhandl_t *vt, off_t off, long len)
{
   long offset = (long) off; /*Actual offset in flash prom*/
/* Don't allow requests which exceed the flash prom size */
   if ((offset + len) > FLASHPROM_SIZE)
      return ENOSPC;
/* Don't allow non page-aligned offsets */
   if ((offset % NBPC) != 0)
      return EIO;
/* Only allow mapping of entire pages */
   if ((len % NBPC) != 0)
      return EIO;
   return v_mapphys(vt, FLASHMAP_ADDR + offset, len);
}
```

When the driver allocates some memory resource associated with the mapping, and when more than one mapping can be active at a time, the driver needs to tag each memory resource so it can be located when the *pfx***unmap()** entry point is called. One answer is to use the **vt_gethandle()** macro defined in *sys/region.h*. This macro takes a pointer to a *vhandle_t* and returns a unique pointer-sized integer that can be used to tag allocations. No other information in *sys/region.h* is supported for driver use.

Entry Point mmap()

The *pfx***mmap()** (note: *two* letters "m") entry can be used only in a character device driver. The prototype is

```
int pfxmmap(dev_t dev, off_t off, int prot);
```

The argument values are

dev A *dev_t* value from which you can extract both the major and minor device numbers.

off The offset argument passed to **mmap()** by the user process.

prot Flags showing the access intentions of the user process.

The function is expected to return the page frame number (PFN) that corresponds to the offset *off* in the device address space. A PFN is an address divided by the page size. (See “Working With Page and Sector Units” on page 197 for page unit conversion functions.)

This entry point is supported only for compatibility with SVR4. When the kernel needs to map a character device, it looks first for *pfx***map()**. It calls *pfx***mmap()** only when *pfx***map()** is not available. The differences between the two entry points are as follows:

- This entry point receives no *vhandl_t* argument, so it cannot use **v_mapphys()**. It has to calculate a page frame number, which means that it has to be aware of the current page size (obtainable from the **ptob()** kernel function, see the ptob(D3) reference page).
- This entry point does not receive a length argument, so it has to assume a default length for every map (typically the page size).
- When a mapping is created using this entry point, the *pfx***unmap()** entry is not called.

Entry Point unmap()

The kernel calls the *pfx***unmap()** entry point when a mapping is created using the *pfx***map()** entry point. This entry should be supplied, even if it is an empty function, when the *pfx***map()** entry point is supplied. If it is not supplied, the **munmap()** system function returns the ENODEV error.

The *pfx***unmap()** entry point is only called when the mapped region has been completely unmapped by all processes. For example, suppose a parent process calls **mmap()** to map a device. Then the parent creates one or more child processes using **sproc()**. Each child shares the address space, including the mapped segment. A process in the share group can terminate, or can explicitly **unmap()** the segment or part of the segment; these actions do not result in a call to *pfx***unmap()**. Only when the last process with access to the segment has fully unmapped the segment is *pfx***unmap()** called.

On entry, the kernel has completed unmapping the object from the user process address space. This entry point does not need to do anything to affect the user address space; it only needs to release any resources that were allocated to support the mapping.

The prototype is

```
int pfxunmap(dev_t dev, vhandl_t *vt);
```

The argument values are

dev A *dev_t* value from which you can extract both the major and minor device numbers.

vt The address of an opaque structure that describes the assigned address in the user process address space.

If the driver allocated no resources to support a mapping, no action is needed here; the entry point can consist of a “return 0” statement.

When the driver does allocate memory to support a mapping, and supports multiple mappings, the driver needs to identify the resource associated with this particular mapping in order to release it. The **vt_gethandle()** function returns a unique number based on the *vt* argument; this can be used to identify resources.

Interrupt Entry Point

When a hardware device presents an interrupt, the IRIX kernel locates a device driver for the device and calls its *pfx***intr()** entry point. The driver processes the interrupt as quickly as possible.

In principle an interrupt can happen at any time. Normally an interrupt occurs because at some previous time, the driver initiated a device operation. Some devices can interrupt without a preceding command.

Associating Interrupt to Driver

The association between an interrupt and the driver is established in different ways depending on the hardware.

- The VECTOR statement establishes the interrupt level and the associated driver for devices on the EISA and VME busses.
- For some VME devices, the interrupt level is established dynamically using **vme_ivec_set()** (see Chapter 14, “Services for VME Drivers”).

- The interrupt vector for a device on the GIO bus is set dynamically, by calling **setgiovector()** from the *pfx***init()** entry point (see Chapter 18, “GIO Device Drivers”).
- For devices on the SCSI bus, all interrupts are handled by a single, low-level driver which notifies a callback function (see Chapter 15, “SCSI Device Drivers”).

Entry Point intr()

The prototype of the *pfx***intr()** entry point is

```
void pfxintr(int ivec);
```

The *ivec* argument is a number that represents the interrupt, depending on the type of device, type of bus, and the way the interrupt was associated to the driver.

When an interrupt occurs, the system is in an unknown state. As a result, the interrupt handler can use only a restricted set of kernel services, and no services that can sleep. In general, the interrupt handler implements the following tasks.

- When the driver supports multiple logical units, use *ivec* to locate the data structure for the interrupting unit.
- Determine the reason for the interrupt by interrogating the device.
- When the interrupt is a response to a device operation, note the success or failure of the command.
- If the driver top half is waiting for the interrupt, waken it.
- If the driver supports polling, and the interrupt represents a pollable event, call **pollwakeup()**.
- If the device is not in an error state and another operation is waiting to be started, start it.

The details of each of these tasks depends on the hardware and on the design of the data structures used by the driver top half.

Mutual Exclusion

In a uniprocessor system, there is only one CPU and when it is executing the interrupt handler, nothing else is executing. An interrupt handler can only be preempted by an interrupt of higher priority—which would be an interrupt for a different driver, and so would have no conflicts with this driver over the use of data.

In a multiprocessor, an interrupt can be taken on any CPU, while other CPUs continue to execute kernel or user code. (In the Challenge and Onyx systems, interrupts are "sprayed" to CPUs in rotation to equalize the workload. Also, VME interrupts can be directed to specific CPUs for handling; see "Using the IPL Statement" on page 312.)

In a multiprocessor, when an interrupt must be handled by a driver that is not marked as multiprocessor-aware (see "Flag D_MP" on page 141), the interrupt may be received on some other CPU, but the driver interrupt entry point is always executed on CPU 0.

In a multiprocessor, when the driver is multiprocessor-aware, one or more other CPUs can execute in the driver's top-half entry points while another CPU executes the driver's interrupt entry point. An interrupt handler written for a multiprocessor must not assume that it has exclusive use of the driver's data (see "Planning for Multiprocessor Use" on page 171).

It is theoretically possible in a multiprocessor for a device to interrupt; for one CPU to enter the interrupt handler; and for the device to interrupt again, resulting in multiple concurrent entries to the same interrupt handler. However, IRIX prevents this. You can assume that your interrupt handler code is entered serially, and not used concurrently by multiple CPUs.

Performance and Latency

Speed in exiting the interrupt handler is critical to system performance. In a uniprocessor, the system is doing nothing else while it executes the handler, and it cannot respond to interrupts of a lower priority. In a multiprocessor, interrupts can be taken by different CPUs. While a CPU executes a handler, that CPU cannot respond to lower-priority interrupts, but other CPUs can be processing user-level code or responding to other interrupts.

Completing Block I/O

In a block device driver, an I/O operation is represented by the *buf_t* structure. The *pfx***strategy()** routine starts operations and waits for them to complete (see "Entry Point strategy()" on page 154).

The interrupt entry point sets the residual count in *b_resid*. It can post an error using **bioerror()**. It posts the operation complete and wakens the *pfx***strategy()** routine by calling **biodone()**. If the *pfx***strategy()** entry has set the address of a completion callback function in the *b_iodone* field of the *buf_t*, **biodone()** invokes it. (For more discussion, see "Waiting for Block I/O to Complete" on page 217.)

Completing Character I/O

In a character device driver, the driver top half typically awaits an interrupt by sleeping on a semaphore or synchronizing variable, and the interrupt routine posts the semaphore (see "Waiting for a General Event" on page 218). Error information must be passed in driver variables according to some local convention.

Calling pollwakeup()

When the interrupt represents an event that can be reported by the driver's *pfx***poll()** entry point (see "Entry Point poll()" on page 156), the interrupt handler must report the event to the kernel, in case some user process is waiting in a **poll()** call. Hypothetical code to do this is shown in Example 8-4.

Example 8-4 Hypothetical Call to pollwakeup()

```
hypo_intr(int ivec)
{
   struct hypo_dev_info *pinfo;
   if (! pinfo = find_dev_info(ivec))
      return; /* not our device */
   ...
   if (pinfo->have_data_flag)
      pollwakeup (pinfo->phead, POLLIN, POLLRDNORM);
   if (pinfo->output_ok_flag)
      pollwakeup (pinfo->phead, POLLOUT);
   ...
}
```

Support Entry Points

Certain driver entry points are used to support the operations of the kernel or the administration of the system.

Entry Point unload()

The *pfx***unload()** entry point is called when the kernel is about to dynamically remove a loadable driver from the running system. The prototype is

```
int pfxunload(void);
```

A driver can be unloaded either because all its devices are closed and a timeout has elapsed, or because the operator has used the *ml* command (see the ml(1) reference page). The kernel does not unload a driver unless the driver provides a *pfx***unload()** entry point. Without this entry point, the driver can be dynamically loaded, but then remains in memory.

It is not easy to retain state information about the device over the time when the driver is not in memory. The entire text and data of a loadable driver, including static variables, are removed and reloaded. Only global variables defined in the descriptive file (see "Describing the Driver in /var/sysgen/master.d" on page 233) remain in memory after the driver is unloaded. Be sure not to store any addresses of driver code or driver static variables in global variables, since these addresses will be different when the driver is reloaded.

The driver may have allocated dynamic memory. This should be released, because the addresses of allocated memory will be lost when the driver is unloaded, and more will be allocated if the driver is reloaded. For example, the driver should use **phfree()** to release a pollhead structure allocated by **phalloc()** (see "Use and Operation of poll(2)" on page 155, and the phalloc(D3) and phfree(D3) reference pages). It is also the time to release any PIO maps using **pio_mapfree()** (see "Mapping PIO Addresses" on page 325), and to release any process handles (see "Sending a Process Signal" on page 203).

The driver is not required to unload. If the driver should not be unloaded at this time, it returns a nonzero return code to the call, and the kernel does not unload it. There are several reasons why a driver should not be unloaded.

The kernel calls *pfx***unload()** only when no device special files managed by the driver are open. If any device had been opened, the *pfx***close()** entry has been called. However, if any device was mapped through the *pfx***map()** entry, the mapping could still exist. If the driver has any resources tied up in association with a memory mapping, it should return a nonzero value to the *pfx***unload** call.

A driver should never permit unloading when there is any kind of pointer to the driver held in any kernel data structure. It is a frequent design error to unload when there is a live pointer to the driver. Unpredictable kernel panics often result.

One example of a live pointer to a driver is a pending callback function. Any pending **itimeout()** or **bufcall()** timers should be cancelled before returning 0 from *pfx***unload()**. A driver for the EISA or GIO bus can register an interrupt handler. There is no way to retract the registration of an EISA interrupt handler (see “Allocating and Programming an IRQ” on page 442), so an EISA driver should never permit unloading once an IRQ has been programmed. A GIO driver is able to unregister an interrupt handler (see “GIO-Specific Kernel Functions” on page 515), and must do so before it permits unloading.

Entry Point halt()

The kernel calls the *pfx***halt()** entry point, if one exists, while performing an orderly system shutdown (see the halt(1) reference page). No other driver entry points are called after this one. The prototype is simply

```
void pfxhalt(void);
```

Since the system is shutting down, there is no point in returning allocated memory. The only purpose this entry point can serve is to leave the device in a safe and stable condition. For example, this is the place at which a disk driver could command the heads of the drive to move to a safe zone for power off.

The driver cannot assume that interrupts are disabled or enabled. The driver cannot block waiting for device actions, so whatever commands it issues to the device must take effect immediately.

Entry Point size()

The *pfx***size()** entry point is required of block device drivers. It reports the size of the device in "sector" units, where a "sector" size is declared as NBPSCTR in *sys/param.h* (currently 512). The prototype is

```
int pfxsize(dev_t dev);
```

The device major and minor numbers can be extracted from the dev argument. The entry point is not called until *pfx***open()** has been called. Typically the driver will calculate the size of the medium during *pfx***open()**.

Since the *int* return value is 32 bits in all systems, the largest possible block device is 1,024 gigabytes ($(2^{31}$*512)/1,024^3).

Entry Point print()

The *pfx***print()** entry point is called from the kernel to display a diagnostic message when an error is detected on a block device. The prototype and the complete logic of the entry point is shown in Example 8-5.

Example 8-5 Entry Point pfxprint()

```
#include <sys/cmn_err.h>
#include <sys/ddi.h>
int hypo_print(dev_t dev, char *str)
{
   cmn_err(CE_NOTE,"Error on dev %d: %s\n",geteminor(dev),str);
   return 0;
}
```

Handling 32-Bit and 64-Bit Execution Models

The *pfx***ioctl()** entry point can be passed a data structure from the user process address space; that is, the *arg* value can be a pointer to a structure or an array of data. In order to interpret such a structure, the driver has to know the execution model for which the user process was compiled.

The execution model is specified when code is compiled. The 32-bit model (compiler option -32 or -n32) uses 32-bit address values and a *long int* contains 32 bits. The 64-bit model (compiler option -64) uses 64-bit address values and a *long int* contains 64 bits. (The size of an unqualified *int* is 32 bits in both models.) The execution model is sometimes casually called the "ABI" (Authorized Binary Interface), but this is an improper use of that term—an ABI comprises calling conventions, public names, and structure definitions, as well as the execution model.

An IRIX kernel compiled to the 32-bit model contains 32-bit drivers and supports only 32-bit user processes. A kernel compiled to the 64-bit model contains 64-bit drivers, but it supports user processes compiled to *either* 32-bit or 64-bit models. Therefore, in a 64-bit kernel, a driver can be asked to interpret data produced by a 32-bit program.

This is true only of the *pfx***ioctl()** and *pfx***poll()** entry points. Other driver entry points move data to and from user space as streams or blocks of bytes—not as a structure with fields to be interpreted.

Since in other respects it is easy to make your driver portable between 64-bit and 32-bit systems, you should design your driver so that it can handle the case of operating in a 64-bit kernel, receiving **ioctl()** requests alternately from 32-bit and 64-bit programs.

The simplest way to do this is to define the arguments passed to the entry points in such a way that they have the same precision in either system. However, this is not always possible. To handle the general case, the driver must know to which model the user process was compiled.

You find this out by calling the **userabi()** kernel function (for which, unfortunately, there is no reference page available).

The prototype of **userabi()** (declared in *sys/ddi.h*) is

```
int userabi(__userabi_t *);
```

If there is no user process context, **userabi()** returns ESRCH. Otherwise it fills out a *__userabi_t* structure and returns 0. The structure of type *__userabi_t* (declared in *sys/types.h*) contains the fields listed below:

uabi_szint	Size of a user int (4).
uabi_szlong	Size of a user long (4 or 8).
uabi_szptr	Size of a user address (4 or 8).
uabi_szlonglong	Size of a user long long (8).

Store the value of *uabi_szptr* when opening a device. Then you can use it to choose between 32-bit and 64-bit declarations of a structure passed to *pfx***ioctl()** or an address passed to *pfx***poll()**.

Planning for Multiprocessor Use

Multiprocessor computers are a central part of the Silicon Graphics product line and will become increasingly common in the future. A device driver that is not multiprocessor-ready can be used in a multiprocessor, but it is likely to cause a performance bottleneck. A multiprocessor-ready driver, on the other hand, works well in a uniprocessor with little if any loss of speed.

The Multiprocessor Environment

A multiprocessor has two or more CPU modules, all of the same type. The CPUs execute independently, but all share the same main memory. Any CPU can execute the code of the IRIX kernel, and it is common for two or more CPUs to be executing kernel code, including driver code, simultaneously.

Uniprocessor Assumptions

The original UNIX architecture assumed a uniprocessor hardware environment with a hierarchy of interrupt levels. Ordinary code could be preempted by an interrupt, but an interrupt handler could only be preempted by an interrupt at a higher level.

This assumed hardware environment was reflected in the design of device drivers and kernel support functions.

- In a uniprocessor, an upper-half driver entry point such as *pfx***open()** cannot be preempted except by an interrupt. It has exclusive access to driver variables except for those changed by the interrupt handler.
- Once in an interrupt handler, no other code can possibly execute except an interrupt of a higher hardware level. The interrupt handler has exclusive access to driver variables.
- The interrupt handler can use kernel functions such as **splhi()** to set the hardware interrupt mask, blocking interrupts of all kinds, and thus getting exclusive access to all memory including kernel data structures.

All of these assumptions fail in a multiprocessor.

- Upper-half entry points can be entered concurrently on multiple CPUs. For example, one CPU can be executing *pfx***open()** while another CPU is in *pfx***strategy()**. Exclusive use of driver variables cannot be assumed.
- An interrupt can be taken on one CPU while upper-half routines or a timeout function execute concurrently on other CPUs. The interrupt routine cannot assume exclusive use of driver variables.
- Interrupt-level functions such as **splhi()** are meaningless, since at best they set the interrupt mask on the current CPU only. Other CPUs can accept interrupts at all levels. The interrupt handler can never gain exclusive access to kernel data.

The process of making a driver multiprocessor-ready consists of changing all code whose correctness depends on uniprocessor assumptions.

Protecting Common Data

Whenever a common resource can be updated by two processes concurrently, the resource must be protected by a *lock* that represents the exclusive right to update the resource. Before changing the resource, the software acquires the lock, claiming exclusive access. After changing the resource, the software releases the lock.

The IRIX kernel provides a set of functions for creating and using locks. It provides another set of functions for creating and using *semaphore* objects, which are like locks but sometimes more flexible. Both sets of functions are discussed under “Waiting and Mutual Exclusion” on page 204.

Sleeping and Waking

Sometimes the lock is not available—some other process executing in another CPU has acquired the lock. When this happens, the requesting process is delayed in the lock function until the lock is free. To delay, or *sleep*, is allowed for upper-half entry points, because they execute (in effect) as subroutines of user processes.

Interrupt handlers and timeout functions are not permitted to sleep. They have no process identity and so there is no mechanism for saving and restoring their state. An interrupt handler can test a lock, and can claim the lock conditionally, but if a lock is already held, the handler must have some alternate way of storing data.

Synchronizing Within Upper-Half Functions

When designing an upper-half entry point, keep in mind that it could be executed concurrently with any other upper-half entry point, and that the one entry point could even be executed concurrently by multiple CPUs. Only a few entry points are immune:

- The *pfx*__init()__, *pfx*__edtinit()__, and *pfx*__start()__ entry points cannot be entered concurrently with each other or any other entry point (*pfx*__start()__ could be entered concurrently with the interrupt handler).
- The *pfx*__unload()__ and *pfx*__halt()__ entry points cannot be entered concurrently with any other entry point except for stray interrupts.
- Certain entry points have no cause to use shared data; for example, *pfx*__size()__ and *pfx*__print()__ normally do not need to take any precautions.
- Other upper-half entry points, and all STREAMS entry points, can be entered concurrently by multiple CPUs, when the driver is multiprocessor-aware.

You can deal with concurrency at different levels of sophistication.

Running on CPU 0

If you do not set the D_MP flag in a character driver or STREAMS driver (see "Flag D_MP" on page 141), the driver is executed only on CPU 0. As a result, upper-half entry points cannot execute concurrently, and the interrupt handler cannot run in true concurrency with an upper-half routine (although it can preempt an upper-half routine as it does in a uniprocessor).

The result is that user processes are serialized for the use of the driver for any purpose. Since CPU 0 is often busy with other housekeeping activities, access to the driver can have a latency that is long and variable.

Serializing on a Single Lock

You can create a single lock for upper-half serialization. Each upper-half function begins with read-only operations such as extracting the device minor number and testing and validating arguments. You allow these to execute concurrently on any CPU (the D_MP flag is set.)

In each enry point, when the preliminaries are complete, you acquire the single lock, and release it just before returning. The result is that processes are serialized for I/O through the driver. If the driver supports only a single device, processes would be serialized in any case, waiting for the device to operate. Since the upper half can execute on any CPU, latency is more predictable.

Serializing on a Lock Per Device

When the driver supports multiple minor devices, you will normally have a data structure per device, indexed by the device minor number. Typically an upper-half routine is concerned only with one minor device. You can define a lock in the data structure for the minor device, and acquire that lock as soon as the device number is known.

This permits concurrent execution of upper-half requests for different minor devices, while serializing access to any one device.

Coordinating Upper-Half and Interrupt Entry Points

Upper-half entry points prepare work for the device to do, and the interrupt routine reports the completion of the device action. In a block device driver, this communication is relatively simple. In a character driver, you have more design options. The kernel functions mentioned in the following topics are covered under "Waiting and Mutual Exclusion" on page 204.

Coordinating Through the buf_t

In a block device driver, the *pfx***strategy()** routine initiates a read or a write based on a *buf_t* structure (see "Entry Point strategy()" on page 154), and leaves the address of the *buf_t* where the interrupt routine can find it. Then *pfx***strategy()** calls the **biowait()** kernel function to wait for completion.

The *pfx***intr()** entry point updates the *buf_t* (using *pfx***bioerror()** if necessary) and then uses **biodone()** to mark the *buf_t* as complete. This ends the wait for *pfx***strategy()**. These kernel functions are multiprocessor-aware.

Coordination in a Character Driver

In a character driver that supports interrupts, you design your own coordination mechanism. The simplest (and not recommended) would be based on using the kernel function **sleep()** in the upper half, and **wakeup()** in the interrupt routine. You can also use a semaphore and use **psema()** in the upper half and **vsema()** in the interrupt handler.

If you need to allow for timeouts, you have to deal with the complication that the timeout function can be called concurrently with an interrupt. When you use a semaphore, the interrupt routine can use **vsema()** to post completion, and the timeout function can use **cvsema()** to post it only if it has not already been posted.

Converting a Uniprocessor Driver

As a general approach, you can convert a uniprocessor driver to make it multiprocessor-safe in the following steps:

1. If it currently uses the D_OLD flag (or has no *pfx***devflag** constant), convert it to use the current interface, with a *pfx***devflag** of 0x00.
2. Make sure it works in the original uniprocessor at the current release of IRIX.
3. Test it in a multiprocessor running in CPU 0.
4. Begin adding semaphores, locks, and other exclusion and synchronization tools. Since the driver still runs serially on CPU 0, it will never wait for a lock, but the coordination between upper half and interrupt handler should work.
5. Add the D_MP flag and test on a multiprocessor.

In performing the conversion, you can use calls to **spl..()** functions as signs that work is needed. These functions are used for mutual exclusion in a uniprocessor, and they are all ineffective or unnecessary in a multiprocessor-safe driver.

Example Conversion Problem

The code in Example 8-6 shows typical logic in a uniprocessor character driver.

Example 8-6 Uniprocessor Upper-Half Wait Logic

```
s = splvme();
flag |= WAITING;
while (flag & WAITING) {
   sleep(&flag, PZERO);
}
splx(s);
```

The upper half calls the **splvme()** function with the intention of blocking interrupts, and thus preventing execution of this driver's interrupt handler while the *flag* variable is updated. In a multiprocessor this is ineffective because at best it sets the interrupt level on the current CPU. The interrupt handler can execute on another CPU and change the variable.

The corresponding interrupt handler is sketched in Example 8-7.

Example 8-7 Uniprocessor Interrupt Logic

```
if (flag & WAITING) {
   wakeup(&flag);
   flag &= ~WAITING;
}
```

The interrupt handler could execute on another CPU, and test the flag after the upper half has called **splvme()** and before it has set WAITING in *flag*. The interrupt is effectively lost. This would happen rarely and would be hard to repeat, but it would happen and would be hard to trace.

A more reliable, and simpler, technique is to use a semaphore. The driver defines a global semaphore:

```
static sema_t sleeper;
```

A driver with multiple devices would have a semaphore per device, perhaps as an array of *sema_t* items indexed by device minor number.

The semaphore (or array) would be initialized to a starting value of 1 in the *pfx***init()** or *pfx***start()** entry:

```
void hypo_start()
{
...
   initnsema(&sleeper,1,"sleeper");
}
```

After the upper half started a device operation, it would await the interrupt using **psema()**:

```
psema(sleeper,PZERO);
```

The PZERO argument makes the wait immune to signals. If the driver should wake up when a signal is sent to the calling process (such as SIGINT or SIGTERM), the second argument can be PCATCH. A return value of -1 indicates the semaphore was posted by a signal, not by a **vsema()** call.

The interrupt handler would use **vsema()** or **cvsema()** to post the semaphore. The use of **cvsema()** ensures that the semaphore is not incremented past 1, in the event that it is posted from more than one location (as from a timeout or a signal handler).