Device Driver/Kernel Interface

The programming interface between a device driver and the IRIX kernel is completely documented in the reference pages in volume "D." This chapter provides a survey and a summary of the API under the following headings:

- "Important Data Types" on page 180 describes the data types that are exchanged between the kernel and a driver.
- "Important Header Files" on page 185 summarizes the C header files that are frequently included in a driver source file.
- "Memory Allocation" on page 187 describes the kernel functions for general memory allocation, for allocation of objects of specific types, and for resource suballocation.
- "Transferring Data" on page 192 describes the functions for transferring data between a driver and a buffer in the address space of either the kernel or a user process.
- "Managing Virtual and Physical Addresses" on page 195 discusses the translation from virtual to physical storage locations for DMA.
- "User Process Administration" on page 202 discusses process signalling and authentication.
- "Waiting and Mutual Exclusion" on page 204 describes and summarizes a wide array of functions you can use for those purposes.

In addition to these topics, data types and functions specific to the following areas are in the chapters shown:

Debugging and logging	Chapter 11, “Testing and Debugging a Driver”
EISA Bus	Chapter 17, “EISA Device Drivers”
GIO Bus	Chapter 18, “GIO Device Drivers”
SCSI bus	Chapter 15, “SCSI Device Drivers”
STREAMS drivers	Chapter 19, “STREAMS Drivers”
VME bus	Chapter 14, “Services for VME Drivers”

Important Data Types

The Device Number Types

Two numbers are carried in the inode of a *device special file*: a *major device number* of up to 9 bits, and a *minor device number* of up to 18 bits. The numbers are assigned when the device special file is created, either by the */dev/MAKEDEV* script or by the system administrator. The contents and meaning of device numbers is discussed under “Device Representation” on page 33.

At almost every upper-half entry point, the first argument to a driver is a *dev_t* object, an unsigned integer containing the values of the major and minor numbers for the device that is to be used. The *dev_t* type is declared in *sys/types.h* along with types *major_t* and *minor_t*, which represent major and minor numbers as variables.

Use of the Device Numbers

You typically use the major device number to learn which device driver has been called. This is important only when a device driver supports multiple interfaces, for example when one driver represents both character and block access to the same hardware.

You use the minor device number to learn which hardware unit is being accessed. This is of interest only when a driver supports multiple units. In addition, device management options can be encoded into the minor number, as described under “Minor Device Number” on page 35.

Device Number Functions

The kernel provides several functions for manipulating device numbers, and these are summarized in Table 9-1.

Table 9-1 Functions to Manipulate Device Numbers

Function	Header Files	Can Sleep	Purpose
etoimajor(D3)	ddi.h	N	Convert external to internal major device number.
getemajor(D3)	ddi.h	N	Get external major device number.
geteminor(D3)	ddi.h	N	Get external minor device number.
getmajor(D3)	ddi.h	N	Get internal major device number.
getminor(D3)	ddi.h	N	Get internal minor device number.
itoemajor(D3)	ddi.h	N	Convert internal to external major device number.
makedevice(D3)	ddi.h	N	Make device number from major and minor numbers.

The most important of these are

- **getemajor()**, which extracts the major number from a *dev_t* and returns it as a *major_t*
- **geteminor()**, which extracts the minor number from a *dev_t* and returns it as a *minor_t*
- **makedevice()**, which combines a *major_t* and a *minor_t* to form a *dev_t*

External and Internal Numbers

The kernel uses the major device number as a subscript to index various tables. Some variants of UNIX, in order to avoid wasting space on sparse tables, translate the major device number to an internal code. Sometimes the minor number is translated too.

This internal encoding of the device number is of no interest in IRIX. If it is done, it is done only for the purpose of subscripting tables within the kernel that are not accessible to device drivers. Internal device numbers have no utility in IRIX. However, functions related to internal device numbers are included for compatibility with SVR4.

If you are writing a new device driver specifically for IRIX, use only external device numbers. If you are porting a device driver that uses the **getmajor()**, **getminor()**, **etoimajor()** and **etoiminor()** functions, you can leave these function calls unchanged. (But if the driver attempts to access the kernel switch tables, it is nonportable and should be changed.)

Structure uio_t

The *uio_t* structure describes data transfer for a character device:

- The *pfx***read()** and *pfx***write()** entry points receive a *uio_t* that describes the buffer of data.
- Within an *pfx***ioctl()** entry point, you might construct a *uio_t* to represent data transfer for control purposes.
- In a hybrid character/block driver, the **physiock()** function translates a *uio_t* into a *buf_t* for use by the *pfx***strategy()** entry point.

The fields and values in a *uio_t* are declared in *sys/uio.h*, which is included by *sys/ddi.h*. For a detailed discussion, see the uio(D4) reference page. Typically the contents of the *uio_t* reflect the buffer areas that were passed to a **read()**, **readv()**, **write()**, or **writev()** call (see the read(2) and write(2) reference pages).

Data Location and the iovec_t

One *uio_t* describes data transfer to or from a single address space, either the address space of a user process or the kernel address space. The address space is indicated by a flag value, either UIO_USERSPACE or UIO_SYSSPACE, in the *uio_segflg* field.

The total number of bytes remaining to be transferred is given in field *uio_resid*. Initially this is the total requested transfer size.

Although the transfer is to a single address space, it can be directed to multiple segments of data within the address space. Each segment of data is described by a structure of type *iovec_t*. An *iovec_t* contains the virtual address and length of one segment of memory.

The number of segments is given in field *uio_iovcnt*. The field *uio_iov* points to the first *iovec_t* in an array of *iovec_t* structures, each describing one segment. of data. The total size in *uio_resid* is the sum of the segment sizes.

For a simple data transfer, *uio_iovcnt* contains 1, and *uio_iov* points to a single *iovec_t* describing a buffer of 1 or more bytes. For a complicated transfer, the *uio_t* might describe a number of scattered segments of data. Such transfers can arise in a network driver where multiple layers of message header data are added to a message at different levels of the software.

Use of the uio_t

In the *pfx***read()** and *pfx***write()** entry points, you can test *uio_segflag* to see if the data is destined for user space or kernel space, and you can save the initial value of *uio_resid* as the requested length of the transfer.

In a character driver, you fetch or store data using functions that both use and modify the *uio_t*. These functions are listed under "Transferring Data Through a uio_t Object" on page 194. When data is not immediately available, you should test for the FNDELAY or FNONBLOCK flags in *uio_fmode*, and return when either is set rather than sleeping.

Structure buf_t

The *buf_t* structure describes a block data transfer. It is designed to represent the transfer (in or out) of a sequence of adjacent, fixed-size blocks from a random-access device to a block of contiguous memory. The size of one device block is NBPSCTR, declared in *sys/param.h*. For a detailed discussion of the *buf_t*, see the buf(D4) reference page.

The *buf_t* is used internally in IRIX by the paging I/O system to manage queues of physical pages, and by filesystems to manage queues of pages of file data. The paging system and filesystems are the primary clients of the *pfx***strategy()** entry point to a block device driver, so it is only natural that a *buf_t* pointer is the input argument to *pfx***strategy()**.

Tip: The idbg kernel debugging tool has several functions related to displaying the contents of *buf_t* objects. See "Commands to Display buf_t Objects" on page 268.

Fields of buf_t

The fields of the *buf_t* are declared in *sys/buf.h*, which is included by *sys/ddi.h*. This header file also declares the names of many kernel functions that operate on *buf_t* objects. (Many of those functions are not supported as part of the DDABI. You should only use kernel functions that have reference pages.)

Because *buf_t* is used by so many software components, it has many fields that are not relevant to device driver needs, as well as some fields that have multiple uses. The relevant fields include the following:

b_edev	*dev_t* giving device major and minor numbers.
b_flags	Operational flags; for a detailed list see buf(D4).
b_forw, *b_back*, *av_forw*, *av_back*	Queuing pointers, available for driver use within the *pfx***strategy()** routine.
b_un.b_addr	Sometimes the kernel virtual address of the buffer, depending on the *b_flags* setting BP_ISMAPPED.
b_bcount	Number of bytes to transfer.
b_blkno	Starting logical block number on device.
b_iodone	Address of a driver internal function to be called on I/O completion.
b_resid	Number of bytes not transferred, set at completion to 0 unless an error occurs.
b_error	Error code, set at completion of I/O.

Using the Logical Block Number

The logical block number is the number of the 512-byte block in the device. The "device" is encoded by the minor device number that you can extract from *b_edev*. It might be a complete device surface, or it might be a partition within a larger device (for example, the IRIX disk device drivers support different minor device numbers for different disk partitions).

The *pfx***strategy()** routine may have to translate the logical block number based on the driver's information about device partitioning and device geometry (sector size, sectors per track, tracks per cylinder).

Buffer Location and b_flags

The data buffer represented by a *buf_t* can be in one of two places, depending on bits in *b_flags*.

When the macro BP_ISMAPPED(*buf_t-address*) returns true, the buffer is in kernel virtual memory and its virtual address is in *b_un.b_addr*.

When BP_ISMAPPED(*buf_t-address*) returns false, the buffer is described by a chain of *pfdat* structures (declared in *sys/pfdat.h*, but containing no fields of any use to a device driver). In this case, *b_un.b_addr* contains only an offset into the first page frame of the chain. See “Managing Buffer Virtual Addresses” on page 199 for a method of mapping an unmapped buffer.

Lock and Semaphore Types

The header files *sys/sema.h* and *sys/types.h* declare the data types of locks of different types, including the following:

lock_t	Basic lock, or spin-lock, used with LOCK() and related functions
mutex_t	Sleeping lock, used for mutual exclusion between upper-half instances.
sema_t	Semaphore object, used for general locking.
mrlock_t	Reader-writer locks, used with RW_RDLOCK() and related functions.
sv_t	Synchronization variable, used with SV_WAIT and related functions

These lock types should be treated as opaque objects because their contents can change from release to release (and in fact their contents are different in IRIX 6.2 from previous releases).

The families of locking and synchronization functions contain functions for allocating, initializing, and freeing each type of lock. See “Waiting and Mutual Exclusion” on page 204.

Important Header Files

The header files that are frequently needed in device driver source modules are summarized in Table 9-2.

Table 9-2 Header Files Often Used in Device Drivers

Header File	Reason for Including
sys/buf.h	The *buf_t* structure and related constants and functions (included by *sys/ddi.h*).
sys/cmn_err.h	The **cmn_err()** function.

Table 9-2 (continued) Header Files Often Used in Device Drivers

Header File	Reason for Including
sys/conf.h	The constants used in the *pfx***devflags** global.
sys/ddi.h	Many kernel functions declared. Also includes *sys/types.h*, *sys/uio.h*, and *sys/buf.h*.
sys/debug.h	Defines the ASSERT macro and others.
sys/dmamap.h	Data types and kernel functions related to DMA mapping.
sys/edt.h	Declares the *edt_t* type passed to *pfx***edtinit()**.
sys/eisa.h	EISA-bus hardware constants and EISA kernel functions.
sys/errno.h	Names for all system error codes.
sys/file.h	Names for file mode flags passed to driver entry points.
sys/immu.h	Types and macros used to manage virtual memory and some kernel functions.
sys/kmem.h	Constants like KM_SLEEP used with some kernel functions.
sys/ksynch.h	Functions used for sleep-locks.
sys/log.h	Types and functions for using the system log.
sys/major.h	Names for assigned major device numbers.
sys/map.h	Types and functions used for suballocation using **rmalloc()**.
sys/mman.h	Constants and flags used with **mmap()** and the *pfx***mmap()** entry point.
sys/param.h	Constants like PZERO used with some kernel functions.
sys/pio.h	VME PIO functions.
sys/poll.h	Types and functions for pollhead allocation and poll callback.
sys/scsi.h	Types and functions used to call the inner SCSI driver.
sys/sema.h	Types and functions related to semaphores, mutex locks, and basic locks.
sys/stream.h	STREAMS standard functions and data types.
sys/strmp.h	STREAMS multiprocessor functions.
sys/sysmacros.h	Macros for conversion between bytes and pages, and similar values.

Table 9-2 (continued) Header Files Often Used in Device Drivers

Header File	Reason for Including
sys/systm.h	Kernel functions related to system operations.
sys/types.h	Common data types and types of system objects (included by *sys/ddi.h*).
sys/uio.h	The *uio_t* structure and related functions (included by *sys/ddi.h*).
sys/vmereg.h	VME bus hardware constants and VME-related functions.

Memory Allocation

A device or STREAMS driver can allocate memory statically, as global variables in the driver module, and this is a good way to allocate any object that is always needed and has a fixed size.

When the number or size of an object can vary, but can be determined at initialization time, the driver can allocate memory in the *pfx***init()**, *pfx***edtinit()**, or *pfx***start()** entry point.

You can allocate memory dynamically in an upper-half entry point. When this is necessary, it should be done in an entry point that is called infrequently, such as *pfx***open()**. The reason is that memory allocation is subject to unpredictable delays.

Memory allocation should never be attempted in an interrupt routine. The resources that might be needed at interrupt time should be obtained and set aside by an upper-half entry point before the interrupt is made possible.

General-Purpose Allocation

There are two groups of general-purpose functions used to allocate and release memory.

- **kmem_alloc()** and two associated functions supply a complete set of services for allocating kernel virtual memory.
- **kern_malloc()** and two associated functions are an obsolete mechanism for allocating kernel virtual memory.

The functions you can use to dynamically allocate kernel virtual memory are summarized in Table 9-3.

Table 9-3 Functions for Kernel Virtual Memory

Function Name	Header Files	Can Sleep?	Purpose
kmem_alloc(D3)	kmem.h & types.h	Y	Allocate space from kernel free memory.
kmem_free(D3)	kmem.h & types.h	N	Free previously allocated kernel memory.
kmem_zalloc(D3)	kmem.h & types.h	Y	Allocate and clear space from kernel free memory.
kern_calloc(D3)	systm.h & types.h	Y	Allocate space from kernel memory and clear it.
kern_free(D3)	systm.h & types.h	N	Free kernel memory space.
kern_malloc(D3)	systm.h & types.h	Y	Allocate kernel virtual memory.

The most important of these functions is **kmem_alloc()**. You use it to allocate blocks of virtual memory at any time. It offers these important options, controlled by a flag argument:

- Sleeping or not sleeping when space is not available. You specify not-sleeping when in a lower-half routine or when holding a basic lock, but then you must be prepared to deal with a return value of NULL.
- Physically-contiguous memory. The memory allocated is virtual, and when it spans multiple pages, the pages are not necessarily adjacent in physical memory. You need physically contiguous pages when doing DMA with a device that cannot do scatter/gather. However, contiguous memory is harder to get as the system runs, so it is best to obtain it in an initialization routine.
- Cache-aligned memory. By requesting memory that is a multiple of a cache line in size, and aligned on a cache-line boundary, you ensure that DMA operations will affect the fewest cache lines (see “Setting Up a DMA Transfer” on page 198).

The **kmem_zalloc()** function takes the same options, but offers the additional service of zero-filling the allocated memory.

Calls to the "kern" group of functions should be replaced as follows:

kern_malloc(*n*)	Change to **kmem_alloc**(*n*,KM_SLEEP).
kern_calloc(*n*,*s*)	Change to **kmem_zalloc**(*n***s*,KM_SLEEP)
kern_free(*p*)	Change to **kmem_free**(*p*)

Allocating Objects of Specific Kinds

The kernel provides a number of functions with the purpose of allocating and freeing objects of specific kinds. Many of these are variants of **kmem_alloc()** and **kmem_free()**, but others use special techniques suited to the type of object.

Allocating pollhead Objects

Table 9-4 summarizes the functions you use to allocate and free the *pollhead* structure that is used within the *pfx***poll()** entry point (see "Entry Point poll()" on page 156). Typically you would call **phalloc()** while initializing each minor device, and call **phfree()** in the *pfx***unload()** entry point.

Table 9-4 Functions for Allocating pollhead Structures

Function Name	Header Files	Can Sleep?	Purpose
phalloc(D3)	ddi.h & kmem.h & poll.h	Y	Allocate and initialize a pollhead structure.
phfree(D3)	ddi.h & poll.h	N	Free a pollhead structure.

Allocating Semaphores and Locks

There are symmetrical pairs of functions to allocate and free all types of lock and synchronization objects. These functions are summarized together with the other locking functions under "Waiting and Mutual Exclusion" on page 204.

Allocating buf_t Objects and Buffers

The argument to the *pfx***strategy()** entry point is a *buf_t* structure that describes a buffer (see “Entry Point strategy()” on page 154 and “Structure buf_t” on page 183).

Ordinarily, both the *buf_t* and the buffer are allocated and initialized by the kernel or the filesystem that calls *pfx***strategy()**. However, some drivers need to create a *buf_t* and associated buffer for special uses. The functions summarized in Table 9-5 are used for this.

Table 9-5 Functions for Allocating buf_t Objects and Buffers

Function Name	Header Files	Can Sleep?	Purpose
geteblk(D3)	ddi.h	Y	Allocate a *buf_t* and a buffer of 1024 bytes.
ngeteblk(D3)	ddi.h	Y	Allocate a *buf_t* and a buffer of specified size.
brelse(D3)	ddi.h	N	Return a buffer header and buffer to the system.
getrbuf(D3)	ddi.h	Y	Allocate a *buf_t* with no buffer.
freerbuf(D3)	ddi.h	N	Free a *buf_t* with no buffer.

To allocate a *buf_t* and its associated buffer in kernel virtual memory, use either **geteblk()** or **ngeteblk()**. Free this pair of objects using **brelse()**, or by calling **biodone()**.

You can allocate a *buf_t* to describe an existing buffer—one in user space, statically allocated in the driver, or allocated with **kmem_alloc()—**using **getrbuf()**. Free such a *buf_t* using **freerbuf()**.

Suballocation Functions

The functions summarized in Table 9-6 are used to manage suballocation of any resource.

Table 9-6 Functions for Suballocation

Function Name	Header Files	Can Sleep?	Purpose
rmalloc(D3)	map.h & types.h	N	Allocate space from a private space management map.
rmalloc_wait(D3)	map.h & types.h	Y	Allocate resources from a space management map.
rmallocmap(D3)	map.h & types.h	N	Allocate and initialize a private space management map.
rmfree(D3)	map.h & types.h	N	Release resources into a space management map.
rmfreemap(D3)	map.h & types.h	N	Free a private space management map.

You use these functions as a convenient, efficient set of subroutines for allocating some resource—for example, disk sectors—that you obtain by other means. The expected sequence of use is as follows.

1. During driver initialization, or possibly in *pfx***open()**, use **rmallocmap()** to allocate a map. A map is a data structure large enough to keep track of as many objects as you will create. Initially the map reflects no available resources.
2. Use **rmfree()** to release existing resources into the map. For example, while opening a disk drive, you could use **rmfree()** to release all unused sectors into a sector map.
3. When a resource is needed in an upper-half routine, use **rmalloc()** or **rmalloc_wait()** to acquire it. The index number of the first allocated item is returned.
4. When a resource is released in any entry point, use **rmfree()** to note the available items and to wake up any upper-half process waiting in **rmalloc_wait()**.
5. On device close or when the driver is unloaded, use **rmfreemap()** to release the map itself.

Transferring Data

The device driver executes in the kernel virtual address space, but it must transfer data to and from the address space of a user process. The kernel supplies two kinds of functions for this purpose:

- functions that transfer data between driver variables and the address space of the current process
- functions that transfer data between driver variables and the buffer described by a *uio_t* object

Warning: The use of an invalid address in kernel space with any of these functions causes a kernel panic.

All functions that reference an address in user process space can sleep, because the page of process space might not be resident in memory. As a result, such functions cannot be used in an interrupt handler, or while holding a basic lock.

General Data Transfer

The kernel supplies functions for clearing and copying memory within the kernel virtual address space, and between the kernel address space and the address space of the user process that is the current context. These general-purpose functions are summarized in Table 9-7.

Table 9-7 Functions for General Data Transfer

Function Name	Header Files	Can Sleep?	Purpose
bcopy(D3)	ddi.h	N	Copy data between address locations in the kernel.
bzero(D3)	ddi.h	N	Clear memory for a given number of bytes.
copyin(D3)	ddi.h	Y	Copy data from a user buffer to a driver buffer.
copyout(D3)	ddi.h	Y	Copy data from a driver buffer to a user buffer.
fubyte(D3)	systm.h & types.h	Y	Load a byte from user space.

Table 9-7 (continued) Functions for General Data Transfer

Function Name	Header Files	Can Sleep?	Purpose
fuword(D3)	systm.h & types.h	Y	Load a word from user space.
hwcpin(D3)	systm.h & types.h	N	Copy data from device registers to kernel memory.
hwcpout(D3)	systm.h & types.h	N	Copy data from kernel memory to device registers.
subyte(D3)	systm.h & types.h	Y	Store a byte to user space.
suword(D3)	systm.h & types.h	Y	Store a word to user space.

Block Copy Functions

The **bcopy()** and **bzero()** functions are used to copy and clear data areas within the kernel address space, for example driver buffers or work areas. These are optimized routines that take advantage of available hardware features.

The **bcopy()** function is not appropriate for copying data between a buffer and a device; that is, for copying between virtual memory and the physical memory addresses that represent a range of device registers (or indeed any uncached memory). The reason is that **bcopy()** uses doubleword moves and any other special hardware features available, and devices many not be able to accept data in these units. The **hwcpin()** and **hwcpout()** functions copy data in 16-bit units; use them to transfer bulk data between device space and memory. (Use simple assignment to move single words or bytes.)

The **copyin()** and **copyout()** functions take a kernel virtual address, a process virtual address, and a length. They copy the specified number of bytes between the kernel space and the user space. They select the best algorithm for copying, and take advantage of memory alignment and other hardware features.

If there is no current context, or if the address in user space is invalid, or if the address plus length is not contained in the user space, the functions return -1. This indicates an error in the request passed to the driver entry point, and the driver normally returns an EFAULT error.

Byte and Word Functions

The functions **fubyte()**, **subyte()**, **fuword()**, and **suword()** are used to move single items to or from user space. When only a single byte or word is needed, these functions have less overhead than the corresponding **copyin()** or **copyout()** call. For example you could use **fuword()** to pick up a parameter using an address passed to the *pfx***ioctl()** entry point. When transferring more than a few bytes, a block move is more efficient.

Transferring Data Through a uio_t Object

A *uio_t* object defines a list of one or more segments in the address space of the kernel or a user process (see "Structure uio_t" on page 182). The kernel supplies three functions for transferring data based on a *uio_t*, and these are summarized in Table 9-8.

Table 9-8 Functions Moving Data Using uio_t

Function	Header Files	Can Sleep?	Purpose
uiomove(D3)	ddi.h	Y	Copy data using *uio_t*.
ureadc(D3)	ddi.h	Y	Copy a character to space described by *uio_t*.
uwritec(D3)	ddi.h	Y	Return a character from space described by *uio_t*.

The **uiomove()** function moves multiple bytes between a buffer in kernel virtual space—typically, a buffer owned by the driver—and the space or spaces described by a *uio_t*. The function takes a byte count and a direction flag as arguments, and uses the most efficient mechanism for copying.

The **ureadc()** and **uwritec()** functions transfer only a single byte. You would use them when transferring data a byte at a time by PIO. When moving more than a few bytes, **uiomove()** is faster.

All of these functions modify the *uio_t* to reflect the transfer of data:

- *uio_resid* is decremented by the amount moved
- In the *iovec_t* for the current segment, *iov_base* is incremented and *iov_len* is decremented
- As segments are used up, *uio_iov* is incremented and *uio_iovcnt* is decremented

The result is that the state of the *uio_t* always reflects the number of bytes remaining to transfer. When the *pfx***read()** or *pfx***write()** entry point returns, the kernel uses the finsl value of *ui_resid* to compute the count returned to the **read()** or **write()** function call.

Managing Virtual and Physical Addresses

The kernel supplies functions for querying the address of hardware registers and for performing memory mapping.

Testing Device Physical Addresses

The functions **badaddr()** and **wbadaddr()**, summarized in Table 9-9, are used to test a physical address to find out if it represents a usable device register.

Table 9-9 Functions to Test Physical Addresses

Function Name	Header Files	Can Sleep?	Purpose
badaddr(D3)	systm.h	N	Test physical address for input.
badaddr_val(D3)	systm.h	N	Test physical address for input and return the input value received.
wbadaddr(D3)	systm.h	N	Test physical address for output.
wbadaddr_val(D3)	systm.h	N	Test physical address for output of specific value.

The functions return a nonzero value when the address is bad, that is, unusable. Related functions for the VME bus are covered in Chapter 14.

You normally use these functions in the *pfx***init()** entry point to verify that an expected device is in fact present. The functions can also be useful in the *pfx***edtinit()** entry point. However, that entry point is only called from a VECTOR statement, and the VECTOR statement can contain a PROBE argument that tests for valid hardware.

Note: These functions must not be called in an interrupt handler. Verify device addresses in the upper-half code, during initialization.

Managing Mapped Memory

The *pfx***map()** and *pfx***unmap()** entry points receive a *vhandl_t* object that describes the region of user process space to be mapped. The functions summarized in Table 9-10 are used to manipulate that object.

Table 9-10 Functions to Manipulate a vhandl_t Object

Function Name	Header Files	Can Sleep?	Purpose
v_getaddr(D3)	region.h & types.h	N	Get the user virtual address associated with a *vhandl_t*.
v_gethandle(D3)	region.h & types.h	N	Get a unique identifier associated with a *vhandl_t*.
v_getlen(D3)	region.h & types.h	N	Get the length of user address space associated with a *vhandl_t*.
v_mapphys(D3)	region.h & types.h	N	Map kernel address space into user address space.

The **v_mapphys()** function actually performs a mapping between a kernel address and a segment described by a *vhandl_t* (see "Entry Point map()" on page 159).

The **v_getaddr()** function has hardly any use except for logging and debugging. The address in user space is normally undefined and unusable when the *pfx***map()** entry point is called, and mapped to kernel space when *pfx***unmap()** is called. The driver has no practical use for this value.

The **v_getlen()** function is useful only in the *pfx***unmap()** entry point—the *pfx***map()** entry point receives a length argument specifying the desired region size.

The **v_gethandle()** function returns a number that is unique to this mapping (actually, the address of a page table entry). You use this as a key to identify multiple mappings, so that the *pfx***unmap()** entry point can properly clean up.

Caution: Be careful when mapping device registers to a user process. Memory protection is available only on page boundaries, so configure the addresses of I/O cards so that each device is on a separate page or pages. When multiple devices are on the same page, a user process that maps one device can access all on that page. This can cause system security problems or other problems that are hard to diagnose.

Working With Page and Sector Units

In a 32-bit kernel, the page size for memory and I/O is 4 KB. In a 64-bit kernel, the memory page size is 16 KB, but because of hardware constraints such as the 4 KB span of DMA mapping registers in the Challenge and Onyx systems, a 4 KB page is used for I/O operations.

The header files *sys/immu.h* and *sys/sysmacros.h* contain constants and macros for working with page units. Some of the most useful are listed below:

NBPP	Number of bytes in a virtual memory page.
NBPSCTR	Number of bytes (512) in a standard disk "sector."
IO_NBPP	Number of bytes in an I/O page.
IO_PNUMSHFT	Number of bits to right-shift an address to get the I/O page number.
IO_POFFMASK	Mask to extract the I/O-page-offset value from an address.
btod**()**	Return number of 512-byte "sectors" in a byte count (rounded up)
btop(*x*)	Return number of I/O pages in a byte count (truncated)
io_pnum(*x*)	Return the I/O page number from an address *x*.
io_poff(*x*)	Return the I/O page offset from an address *x*.
io_numpages(*addr*, *len*)	Return the number of I/O pages that span a given address for a length.
io_ctob(*x*)	Return number of bytes in *x* I/O pages (rounded up).
io_ctobt(*x*)	Return number of bytes in *x* I/O pages (truncated).

The functions summarized in Table 9-11 are also provided as functions.

Table 9-11 Functions to Convert Bytes to Sectors or Pages

Function Name	Header Files	Can Sleep?	Purpose
btop(D3)	ddi.h	N	Return number of I/O pages in a byte count (truncate).
btopr(D3)	ddi.h	N	Return number of I/O pages in a byte count (round up).
ptob(D3)	ddi.h	N	Convert size in I/O pages to size in bytes.

Using these functions and macros, you can make your driver independent of the size of pages. When examining an existing driver, be alert for any assumption that a virtual memory page has a particular size, or that an I/O page is the same size as a memory page, and convert the code to use portable functions and macros.

Setting Up a DMA Transfer

There are two issues in preparing a DMA transfer:

- calculating physical addresses of the memory targets to be programmed into the device registers
- ensuring cache coherency in a uniprocessor

The functions you use to derive target addresses are different for different bus adapters and are discussed in the following chapters:

- The functions to set up DMA from a VME device are covered in Chapter 14, “Services for VME Drivers.”
- The functions to set up DMA from a SCSI device are covered in Chapter 15, “SCSI Device Drivers.”
- The functions to set up DMA from an EISA device are covered in Chapter 17, “EISA Device Drivers.”
- The functions to set up DMA from a GIO device are covered in Chapter 18, “GIO Device Drivers.”

Note: In addition, when designing a device driver for use in a Challenge or Onyx system, be sure to read the caution in Appendix B, “Challenge DMA with Multiple IO4 Boards.”

Converting Physical Addresses

General functions for calculating physical addresses are summarized in Table 9-12.

Table 9-12 Functions Related to Physical Memory

Function Name	Header Files	Can Sleep?	Purpose
kvtophys(D3)	ddi.h	N	Get physical address of kernel data
sgset(D3)	ddi.h & sg.h	N	Get physical addresses of a series of pages for simulated scatter/gather.

Managing Buffer Virtual Addresses

Functions to manipulate buffer page mappings are summarized in Table 9-13.

Table 9-13 Functions to Map Buffer Pages

Function Name	Header Files	Can Sleep?	Purpose
bp_mapin(D3)	buf.h	Y	Map buffer pages into kernel virtual address space.
bp_mapout(D3)	buf.h	N	Release mapping of buffer pages.
bptophys(D3)	ddi.h	N	Get physical address of buffer data.
clrbuf(D3)	buf.h	N	Clear the memory described by a mapped-in *buf_t*.
getnextpg(D3)	buf.h	N	Return *pfdat* structure for next page.
pptophys(D3)	buf.h	N	Return the physical address of a page described by a *pfdat* structure.

When a *pfx***strategy()** routine receives a *buf_t* that is not mapped into memory (see “Buffer Location and b_flags” on page 184), it must make sure that the pages of the buffer space are in memory, and it must obtain valid kernel virtual addresses to describe the pages. The simplest way is to apply the **bp_mapin()** function to the *buf_t*. This function allocates a contiguous range of page table entries in the kernel address space to describe the buffer, creating a mapping of the buffer pages to a contiguous range of kernel virtual addresses. It sets the virtual address of the first data byte in *b_un.b_addr*, and sets the flags so that BP_ISMAPPED**()** returns true—thus converting an unmapped buffer to a mapped case.

When the device does not handle scatter/gather DMA there is a disadvantage to using **bp_mapin()**. Without scatter/gather, each page’s worth of data must be set up as a unique I/O operation. There is no need to have all of a possibly large buffer locked into memory for the whole time. Using **getnextpg()**, a driver can step through the *pfdat* structures that describe the successive pages of the buffer, calling **pptophys()** to get the physical address of the page frame.

Managing Memory for Cache Coherency

Some kernel functions used for ensuring cache coherency are summarized in Table 9-14.

Table 9-14 Functions Related to Cache Coherency

Function Name	Header Files	Can Sleep?	Purpose
dki_dcache_inval(D3)	systm.h & types.h	N	Invalidate the data cache for a given range of virtual addresses.
dki_dcache_wb(D3)	systm.h & types.h	N	Write back the data cache for a given range of virtual addresses.
dki_dcache_wbinval(D3)	systm.h & types.h	N	Write back and invalidate the data cache for a given range of virtual addresses.
flushbus(D3)	systm.h & types.h	?	Make sure contents of the write buffer are flushed to the system bus

The functions for cache invalidation are essential when doing DMA on a uniprocessor. They cost very little to use in a multiprocessor, so it does no harm to call them in every system. You call them as follows:

- Call **dki_dcache_inval()** prior to doing DMA input. This ensures that when you refer to the received data, it will be loaded from real memory.
- Call **dki_dcache_wb()** prior to doing DMA output. This ensures that the latest contents of cache memory are in system memory for the device to load.
- Call **dki_dcache_wbinval()** prior to a device operation that samples memory and then stores new data.

The **flushbus()** function is needed because in some systems the hardware collects output data and writes it to the bus in blocks. When you write a small amount of data to a device through PIO, delay, then write again, the writes could be batched and sent to the device in quick succession. Use **flushbus()** after PIO output when it is followed by PIO input from the same device. Use it also between any two PIO outputs when the device is supposed to see a delay between outputs.

DMA Buffer Alignment

In some systems, the buffers used for DMA must be aligned on a boundary the size of a *cache line* in the current CPU. Although not all system architectures require cache alignment, it does no harm to use cache-aligned buffers in all cases. The size of a cache line varies among CPU models, but if you obtain a DMA buffer using the KMEM_CACHEALIGN flag of **kmem_alloc()**, the buffer is properly aligned. The buffer returned by **geteblk()** (see “Allocating buf_t Objects and Buffers” on page 190) is cache-aligned.

Why is cache alignment necessary? Suppose you have a variable, *X*, adjacent to a buffer you are going to use for DMA write. If you invalidate the buffer prior to the DMA write, but then reference the variable *X*, the resulting cache miss brings part of the buffer back into the cache. When the DMA write completes, the cache is stale with respect to memory. If, however, you invalidate the cache after the DMA write completes, you destroy the value of the variable *X*.

Maximum DMA Transfer Size

The maximum size for a single DMA transfer is set by the system tuning variable *maxdmasz*, settable with the *systune* command (see the systune(1) reference page). A single I/O operation larger than this produces the error ENOMEM.

The unit of measure for *maxdmasz* is the page, which varies with the kernel. Under IRIX 6.2, a 32-bit kernel uses 4 KB pages while a 64-bit kernel uses 16 KB pages. In both systems, *maxdmasz* is shipped with the value 1024 decimal, equivalent to 4 MB in a 32-bit kernel and 16 MB in a 64-bit kernel.

In Challenge and Onyx systems, *maxdmasz* can be set as high as 64 MB. However, it is not usually possible to allocate a DMA map for a single transfer that large—see “Mapping DMA Addresses” on page 330.

User Process Administration

The kernel supplies a small group of functions, summarized in Table 9-15, that help a driver upper-half routine learn about the current user process.

Table 9-15 Functions for User Process Management

Function Name	Header Files	Can Sleep?	Purpose
drv_getparm(D3)	ddi.h	N	Retrieve kernel state information.
drv_priv(D3)	ddi.h	N	Test for privileged user.
drv_setparm(D3)	ddi.h	N	Set kernel state information.
proc_ref(D3)	ddi.h	N	Obtain a reference to a process for signaling.
proc_signal(D3)	ddi.h & signal.h	N	Send a signal to a process.
proc_unref(D3)	ddi.h	N	Release a reference to a process.

Use **drv_getparm()** to retrieve certain miscellaneous bits of information including the process ID of the current process. In a character device driver, the current process is the user process that caused entry to the driver, for example by calling the **open()**, **ioctl()**, or **read()** system functions. In a block device driver, the current process has no direct relationship to any particular user; it is usually a daemon process of some kind.

The **drv_setparm()** function is primarily of use to terminal drivers.

The **drv_priv()** function tests a *cred_t* object to see if it represents a privileged user. A *cred_t* object is passed in to several driver entry points, and the address of the current one can be retrieved **drv_getparm()**.

Sending a Process Signal

In traditional UNIX kernels, a device driver identified the current user process by the address of the *proc_t* structure that the kernel uses to represent a process. Direct use of the *proc_t* is no longer supported by IRIX. The reason is that the contents of the *proc_t* change from release to release, and also differ between 64-bit and 32-bit kernels.

The most common use of the *proc_t* by a driver was to send a signal to the process. This capability is still supported. To do it, take three steps:

1. Call **proc_ref()** to get a process handle, a number unique to the current process. The returned value must be treated as an arbitrary number (in some releases of IRIX it was the *proc_t* address, but this is not the defined behavior of the function.)
2. Use the process handle as an argument to **proc_signal()**, sending the signal to the process.
3. Release the process handle by calling **proc_unref()**.

The third step is important. In order to keep the process handle valid, IRIX retains information about the process to which it is related. However, that process could terminate (possibly as a result of the signal the driver sends) but until the driver announces that it is done with the handle, the kernel must try to retain process information.

It is especially important to release a process handles before unloading a loadable driver (see “Entry Point unload()” on page 167).

Waiting and Mutual Exclusion

The kernel supplies a rich variety of functions for waiting and for mutual exclusion. In order to use these features well, you must understand the different purposes for which they are designed. In particular, you must clearly understand the distinction between *waiting* and *mutual exclusion* (or locking).

Note: These waiting and mutual exclusion functions have been expanded significantly in IRIX release 6.2.

Mutual Exclusion Compared to Waiting

Mutual exclusion allows one entity to have exclusive use of a global resource, temporarily denying use of the resource to other entities. When software is well-designed, mutual exclusion normally does not require waiting—the resource is normally free when it is requested. A driver that calls a mutual exclusion function *expects to proceed* without delay—although there is a chance that the resource is in use, and the driver will have to wait.

The kernel offers an array of functions for mutual exclusion, and the choice among them can be critical to performance. The functions are reviewed in the following topics:

- "Basic Locks" on page 205 covers basic locks and mutex locks, the best locks for multiprocessor use.
- "Long-Term Locks" on page 207 covers sleep locks, which can be held for longer periods.
- "Reader/Writer Locks" on page 211 covers a class of locks that allow multiple, concurrent, read-only access to resources that are infrequently changed.
- "Priority Level Functions" on page 213 covers the traditional UNIX method of mutual exclusion, the **splhi()** and **splx()** functions, which have many disadvantages.

Waiting allows a driver to coordinate its actions with a specific event or action that occurs asynchronously. A driver can wait for a specified amount of time to pass, wait for an I/O action to complete, and so on. Therefore, a driver that calls a waiting function *expects to wait* for something to happen—although there is a chance that the expected event has already happened, and the driver will be able to continue at once.

The kernel offers several functions that allow you to wait for specific events; and also offers functions for general synchronization. These are covered in the following topics:

- "Waiting for Time to Pass" on page 214 covers timer-related functions.
- "Waiting for Memory to Become Available" on page 216 covers memory allocation waits.
- "Waiting for Block I/O to Complete" on page 217 covers waits used in the *pfx***strategy()** entry point.
- "Waiting for a General Event" on page 218 covers the general-purpose functions that you can adapt to any synchronization problem.

The most general facility, the semaphore, can be used for synchronization and for locking. This topic is covered under "Semaphores" on page 222.

Basic Locks

IRIX supports basic locks using functions compatible with SVR4. These functions are summarized in Table 9-16.

Table 9-16 Functions for Basic Locks

Function Name	Header Files	Can Sleep?	Purpose
LOCK(D3)	ksynch.h & types.h	Y	Acquire a basic lock, waiting if necessary.
LOCK_ALLOC(D3)	ksynch.h,kmem.h & types.h	Y	Allocate and initialize a basic lock.
LOCK_DEALLOC(D3)	ksynch.h & types.h	N	Deallocate an instance of a basic lock.
LOCK_INIT(D3)	ksynch.h & types.h	N	Initialize a basic lock that was allocated statically, or reinitialize an allocated lock.
LOCK_DESTROY(D3)	ksynch.h & types.h	N	Uninitialize a basic lock that was allocated statically.

Table 9-16 (continued) Functions for Basic Locks

Function Name	Header Files	Can Sleep?	Purpose
TRYLOCK(D3)	types.h & ksynch.h	N	Try to acquire a basic lock, returning a code if the lock is not currently free.
UNLOCK(D3)	types.h & ksynch.h	N	Release a basic lock.

Basic locks are objects of type *lock_t*. Although functions are provided for allocating and freeing them, a basic lock is a very small object. Locks are typically allocated as global variables or as fields of structures.

Call LOCK**()** to seize a lock and gain possession of the resource for which it stands. Release the lock with UNLOCK**()**. These functions are optimized for mutual exclusion in the available hardware, and may be implemented differently in uniprocessors and multiprocessors. However, the programming and binary interface is the same in all systems.

The code in Example 9-1 illustrates the use of LOCK and UNLOCK in implementing a simple last-in-first-out (LIFO) queueing package. In these functions, the time between locking a queue head and releasing it is only a few microseconds.

Example 9-1 LIFO Queue Using Basic Locks

```
typedef struct qitem {
   qitem *next; ...other fields...
} qitem_t;
typedef struct lifo {
   qitem *latest;
   lock_t grab;
} lifo_t;
void putlifo(lifo_t *q, qitem_t *i)
{
   int lockpl = LOCK(&q->grab,plhi);
   i->next = q->latest;
   q->latest = i;
   UNLOCK(&q->grab,lockpl);
}
qitem_t *poplifo(lifo_t *q)
{
   int lockpl = LOCK(&q->grab,plhi);
```

```
    qitem_t *ret = q->latest;
    q->latest = ret->next;
    UNLOCK(&q->grab,lockpl);
    return ret;
}
```

This is a typical use of basic locks: to ensure that for a brief period only one process in the system can update a queue. Basic locks are optimized for such uses, but in order to get optimal performance they are restricted to these uses. In particular, if you seize a basic lock and hold it over a function call that can sleep, the system can deadlock.

Long-Term Locks

Sometimes you need a lock that can be held for a longer period, over a call to a function that can sleep. IRIX provides three types of such locks: mutex locks, sleep locks, and reader-writer locks.

Using Mutex Locks

Mutex locks are designed for mutual exclusion (as the name suggests). The IRIX implementation of mutex locks is compatible with the *kmutex_t* lock type of SunOS™, but optimized for use in Silicon Graphics hardware systems. The mutex functions are summarized in Table 9-17.

Table 9-17 Functions for Mutex Locks

Function Name	Header Files	Can Sleep?	Purpose
MUTEX_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a mutex lock.
MUTEX_INIT(D3)	types.h & ksynch.h	N	Initialize an existing mutex lock.
MUTEX_DESTROY(D3)	types.h & ksynch.h	N	Deinitialize a mutex lock.
MUTEX_DEALLOC(D3)	types.h & ksynch.h	N	Deinitialize and free a dynamically allocated mutex lock.

Table 9-17 (continued) Functions for Mutex Locks

Function Name	Header Files	Can Sleep?	Purpose
MUTEX_LOCK(D3)	types.h & kmem.h & ksynch.h	Y	Claim a mutex lock.
MUTEX_TRYLOCK(D3)	types.h & ksynch.h	N	Conditionally claim a mutex lock.
MUTEX_UNLOCK(D3)	types.h & ksynch.h	N	Release a mutex lock.
MUTEX_WAITQ(D3)	types.h & ksynch.h	N	Get the number of processes blocked by mutex lock.
MUTEX_ISLOCKED(D3)	types.h & ksynch.h	N	Test if a mutex lock is owned.
MUTEX_MINE(D3)	types.h & ksynch.h	N	Test if a mutex lock is owned by this process.

Although allocation and deallocation functions are supplied, a *mutex_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The MUTEX_INIT() operation prepares a statically-allocated *mutex_t* for use.

Once initialized, a mutex lock is used to gain exclusive use of the resource with which you have associated it. The mutex lock has the following important advantages over a basic lock:

- The mutex lock can safely be held over a call to a function that sleeps.
- The mutex lock supports the inquiry functions MUTEX_WAITQ, MUTEX_ISLOCKED, and MUTEX_MINE.
- When a debugging kernel is used (see "Including Lock Metering in the Kernel Image" on page 246) a mutex lock can be instrumented to keep statistics of its use.

The mutex lock implementation provides *priority inheritance*. When a low-priority process owns a mutex lock and a high-priority process attempts to seize the lock and is blocked, the process holding the lock is temporarily given the higher priority of the blocked process. This hastens the time when the lock can be released, so that a low-priority process does not needlessly impede a higher-priority process.

In order to implement priority inheritance and retain high performance, the mutex lock is subject to the following restrictions:

- A mutex lock can be locked and unlocked only by an upper-half driver routine; that is, from code that has a process context. A mutex lock cannot be locked or unlocked in an interrupt routine.
- A mutex lock must be unlocked by the same process that locked it. It cannot be locked in one process identity and unlocked in another.

Because of these restrictions, a mutex lock can only be used to mediate between upper-half driver entry points. It is very effective for this purpose; you can use mutex locks to coordinate the use of global variables between upper-half entry points of a driver in a multiprocessor design.

When you need mutual exclusion between an upper-half entry point and the interrupt handler, use a basic lock. Resources that are shared with an interrupt handler should never be in use for more than a brief period. When your design requires a lock to be seized by one process and released in another, use a sleep lock or semaphore.

Using Sleep Locks

IRIX supports sleep lock functions that are compatible with SVR4. These functions are summarized in Table 9-18.

Table 9-18 Functions for Sleep Locks

Function Name	Header Files	Can Sleep?	Purpose
SLEEP_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a sleep lock.
SLEEP_DEALLOC(D3)	types.h & ksynch.h	N	Deinitialize and deallocate a dynamically allocated sleep lock.
SLEEP_INIT(D3)	types.h & ksynch.h	N	Initialize an existing sleep lock.
SLEEP_DESTROY	types.h & ksynch.h	N	Deinitialize a sleep lock.

Table 9-18 (continued) Functions for Sleep Locks

Function Name	Header Files	Can Sleep?	Purpose
SLEEP_LOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a sleep lock, waiting if necessary until the lock is free.
SLEEP_LOCKAVAIL(D3)	types.h & ksynch.h	N	Query whether a sleep lock is available.
SLEEP_LOCK_SIG(D3)	types.h & ksynch.h & param.h	Y	Acquire a sleep lock, waiting if necessary until the lock is free or a signal is received.
SLEEP_TRYLOCK(D3)	types.h & ksynch.h	N	Try to acquire a sleep lock, returning a code if it is not free.
SLEEP_UNLOCK(D3)	types.h & ksynch.h	N	Release a sleep lock.

Although allocation and deallocation functions are supplied, a *sleep_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The SLEEP_INIT() operation prepares a statically-allocated *sleep_t* for use. (In IRIX 6.2, a *sleep_t* is identical to a *sema_t*, but this situation could change in a future release.)

A sleep lock is similar to a mutex lock in that it is used for mutual exclusion between processes, and can be held across a function call that sleeps. A sleep lock does not have either the advantages or the restrictions of a mutex lock:

- A sleep lock can be seized by one process and released by another.
- A sleep lock can be set in an upper-half entry point and released in an interrupt routine.
- A sleep lock does not provide priority inheritance. When a low-priority process holds a sleep lock, a higher-priority process can be blocked, causing a *priority inversion*.
- A sleep lock does not support the instrumentation or the query functions supported for mutex locks.

Reader/Writer Locks

Reader/writer locks are similar to sleep locks in that they are designed for mutually exclusive control of resources for relatively long periods of time. However, Reader/Writer locks are optimized for the case in which the resource is often used by processes that only interrogate it (readers), and only rarely used by processes that modify it (writers).

Reader/writer locks compatible with SVR4 are introduced in IRIX 6.2. The functions are summarized in Table 9-19.

Table 9-19 Functions for Reader/Writer Locks

Function Name	Header Files	Can Sleep?	Purpose
RW_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a reader/writer lock.
RW_DEALLOC(D3)	types.h & ksynch.h	N	Deallocate a reader/writer lock.
RW_INIT(D3)	types.h & ksynch.h	N	Initialize an existing reader/writer lock.
RW_DESTROY(D3)	types.h & ksynch.h	N	Deinitialize an existing reader/writer lock.
RW_RDLOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a reader/writer lock as reader, waiting if necessary.
RW_TRYRDLOCK(D3)	types.h & ksynch.h	N	Try to acquire a reader/writer lock as reader, returning a code if it is not free.
RW_TRYWRLOCK(D3)	types.h & ksynch.h	N	Try to acquire a reader/writer lock as writer, returning a code if it is not free.

Table 9-19 (continued) Functions for Reader/Writer Locks

Function Name	Header Files	Can Sleep?	Purpose
RW_UNLOCK(D3)	types.h & ksynch.h	N	Release a reader/writer lock as reader or writer.
RW_WRLOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a reader/writer lock as writer, waiting if necessary.

Although allocation and deallocation functions are supplied, a *mrlock_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The RW_INIT() operation prepares a statically-allocated *mrlock_t* for use.

A process that intends to modify a resource uses RW_WRLOCK to claim it. This process waits until the resource is not in use by any process, then it gains exclusive access. Only one process is allowed to hold a reader/writer lock as a writer. All other processes, readers or writers, wait until the writer releases the lock.

A process that intends only to interrogate a resource uses RW_RDLOCK to gain access. If a writer holds the lock, the process waits. When the lock is free, or is held only by other readers, the process continues. More than one reader can hold a reader/writer lock at one time. It is also valid for a reader to "double-trip" a reader/writer lock; that is, claim it two or more times. The reader must release the lock as many times as it claimed the lock.

A reader/writer lock serves the same basic purpose as a sleep lock, but it is more efficient in a multiprocessor when there are frequent, read-only uses of a resource.

Priority Level Functions

In traditional UNIX systems, one set of functions served all purposes of synchronization and locking: the set-priority-level, or **spl**, functions. These functions are still available in IRIX, and are summarized in Table 9-20.

Table 9-20 Functions to Set Interrupt Levels

Function Name	Header Files	Can Sleep?	Purpose
splbase(D3)	ddi.h	N	Block no interrupts.
spltimeout(D3)	ddi.h	N	Block only timeout interrupts.
spldisk(D3)	ddi.h	N	Block disk interrupts.
splstr(D3)	ddi.h	N	Block STREAMS interrupts.
spltty(D3)	ddi.h	N	Block disk, VME, serial interrupts.
splhi(D3)	ddi.h	N	Block all I/O interrupts.
spl0(D3)	ddi.h	N	Same as **splbase()**.
splx(D3)	ddi.h	N	Restore previous interrupt level.

These functions are commonly found in device drivers being ported from uniprocessors. Such drivers rely on the use of **splhi()** to gain exclusive use of a global resource.

The **spl** functions are supported by IRIX and they are effective in a uniprocessor driver. However, in a multiprocessor, the functions affect only the interrupt handling of the current CPU. Other CPUs in the system continue to handle interrupts, including interrupts initiated by the driver that called **splhi()**.

A driver that is not multiprocessor-aware (one that does not have D_MP in its *pfx***devflag** constant; see “Driver Flag Constant” on page 140) runs only in CPU 0 of a multiprocessor, so in this case the **spl** functions are still effective. Since they set the interrupt level on CPU 0 where the driver runs, and since the driver’s interrupts can only be handled on CPU 0, the use of **splhi()** gives the driver exclusive use of its resources.

A driver that is multiprocessor-aware uses basic locks, synchronization variables, and other tools to control access to resources, and never uses an **spl** function. This improves performance in a multiprocessor, does not harm performance in a uniprocessor, and reduces the latency of all interrupts.

Waiting for Time to Pass

The kernel offers functions for timed delays, as summarized in Table 9-21.

Table 9-21 Functions for Timed Delays

Function Name	Header Files	Can Sleep?	Purpose
delay(D3)	ddi.h	Y	Delay for a specified number of clock ticks.
drv_hztousec(D3)	ddi.h	N	Convert clock ticks to microseconds
drv_usectohz(D3)	ddi.h	N	Convert microseconds to clock ticks.
drv_usecwait(D3)	ddi.h	N	Busy-wait for a specified interval.
dtimeout(D3)	ddi.h & ksynch.h	N	Schedule a function execute on a specified processor after a specified length of time.
itimeout(D3)	ddi.h & ksynch.h	N	Schedule a function to be executed after a specified number of clock ticks.
fast_itimeout(D3)	ddi.h & ksynch.h	N	Same as **itimeout()** but takes an interval in “fast ticks.”
fasthzto(D3)	types.h & time.h	N	Returns the value of a *struct timeval* as a count of “fast ticks.”
timeout(D3)	ddi.h & ksynch.h	N	Schedule a function to be executed after a specified number of clock ticks.
untimeout(D3)	ddi.h	N	Cancel a previous itimeout or fast_itimeout request.
untimeout_func(D3)	ddi.h	N	Cancel a previous itimeout or fast_itimeout request by function name.

Time Units

The basic time unit is the “tick.” Its value can differ between hardware platforms and between versions of IRIX. The **drvhztousec()** and **drvusectohz()** functions convert between ticks and microseconds in the current system. Use them in order to schedule a delay in a portable mannter. (However, the timer function precision is the tick, not the microsecond.)

The “fast tick” is a fraction of a tick. Like the tick, the fast tick’s value can differ between systems. Use **fasthzto()** to convert from microseconds to fast ticks.

Timer Support

Timer support is based on the idea of a “callback” function. You specify the following to **dtimeout()**, **itimeout()**, **timeout()** or **fast_itimeout()**:

- an interval in clock ticks or fast ticks
- a function to be called at the expiration of the interval
- one or more arguments to be passed to the function
- a priority (interrupt) level at which the function should run

After a delay of at least the length requested, the function is called. The function is entered asynchronously. On a uniprocessor, it can interrupt execution of an upper-half routine. On a multiprocessor, it can execute concurrently with an upper-half routine or with an interrupt handler. You should not rely on the priority level of the function for mutual exclusion (see “Priority Level Functions” on page 213 for an explanation).

The difference between **itimeout()** and **timeout()** is that the latter takes no argument values to be passed to the function when it is called. In order to get a repeated series of timer events, start a new timeout from the callback function.

The **untimeout()** and **untimeout_func()** functions cancel a pending timeout. In a loadable driver that has an *pfx***unload()** entry point, cancel any pending timeouts before unloading.

The STREAMS_TIMOUT macro supplies similar timeout capability for a STREAMS driver (see “Special Considerations for Multiprocessing” on page 547).

Short-Term Delay Support

In rare circumstances, a driver needs to pause briefly between two hardware operations. For example, the Silicon Graphics support for external interrupts in the Challenge and Onyx computers sometimes needs to set a high output level, wait for a brief, precise interval, then set a low output level.

The **drv_usecwait()** function supports this type of very short, precisely-timed delay. It "spins" for a specified number of microseconds, then returns to the caller. The CPU does nothing else during this period, so clearly a delay of more than a few microseconds can interfere with other work. Furthermore, if interrupts are disabled during the wait, the response to another interrupt is delayed also—the delay contributes directly to the "latency" of interrupt handling.

Waiting for Memory to Become Available

Whenever you request memory of any kind, you must allow for the possibility that the memory will not be available. When you allocate memory in bulk (see "General-Purpose Allocation" on page 187) using **kmem_alloc()** you have the option of receiving a null response, or of waiting for the memory to be available.

When you request memory for specific object types (see "Allocating Objects of Specific Kinds" on page 189) there is usually no choice; the functions sleep until they can acquire an object of the requested type.

Within a STREAMS driver you have the ability to schedule a callback function to be entered when memory for a message buffer becomes available (see the bufcall(D3) reference page).

Waiting for Block I/O to Complete

The *pfx***strategy()** routine initiates the I/O operation to fill a buffer based on a *buf_t* structure. Then it has to wait for the I/O to complete. The functions for managing this synchronization are summarized in Table 9-22.

Table 9-22 Functions for Synchronizing Block I/O

Function Name	Header Files	Can Sleep?	Purpose
biodone(D3)	ddi.h	N	Release buffer after I/O and wake up waiting process.
bioerror(D3)	ddi.h	N	Manipulate error fields in a *buf_t*.
biowait(D3)	ddi.h	Y	Suspend process pending completion of I/O.
geterror(D3)	ddi.h	N	Retrieve error number from a *buf_t*.
physiock(D3)	ddi.h	Y	Validate a raw I/O request and pass to a strategy function.
uiophysio(D3)	ddi.h	Y	Validate a raw I/O request and pass to a strategy function.
undma(D3)	ddi.h	?	Unlock physical memory after I/O complete
userdma(D3)	ddi.h	?	Lock physical memory in user space.small number of

How the strategy() Entry Point Is Called

The *pfx***strategy()** entry point is called directly from the filesystem or virtual memory management, or it can be called indirectly from a *pfx***read()** or *pfx***write()** entry point (see “Calling Entry Point strategy() From Entry Point read() or write()” on page 153).

Strategies of the strategy() Entry Point

Typically the *pfx***strategy()** routine must interact with its interrupt handler. The *pfx***strategy()** routine can be designed in either of two ways, synchronous or asynchronous.

The synchronous *pfx***strategy()** routine initiates every I/O operation. Its interrupt handler is responsible only for detecting and signalling the completion of one I/O. The *pfx***strategy()** routine proceeds as follows:

1. Lock the data buffer in memory using **userdma()**.
2. Place the address of the *buf_t* where the *pfx***intr()** entry point can find it.
3. Program the device (see “Setting Up a DMA Transfer” on page 198) and initiate the I/O activity.
4. Call **biowait()**.

When the interrupt handler is entered, the handler uses **bioerror()** if necessary, and **biodone()** to signal the completion of the I/O. Then it exits. The strategy code, which is waiting in the call to **biowait()**, regains control following the call to **biodone()**, and can use **geterror()** to check the results.

The asynchronous *pfx***strategy()** routine only initiates the first I/O operation of a series, and never waits. It proceeds as follows:

1. Lock the data buffer in memory using **userdma()**.
2. Append the address of the *buf_t* to a queue shared with the interrupt handler.
3. If the queue was empty, no I/O is in progress. Call a subroutine that programs the device and initiates the I/O.
4. Return to the caller. The caller (a filesystem or paging system or **uiophysio()**) waits using **biowait()**.

When the interrupt occurs, the handler proceeds as follows:

1. The first queued *buf_t* has completed. Remove it from the queue.
2. Apply **bioerror()** if necessary, and **biodone()** to the *buf_t*. This releases the caller of the strategy routine from **biowait()**.
3. If any operations remain in the queue, call a subroutine to program and initiate the next one.

Waiting for a General Event

There are causes for synchronization other than time, block I/O, and memory allocation. For example, there is no defined interface comparable to **biowait()**/**biodone()** to mediate between an interrupt handler and the *pfx***read()** or *pfx***write()** entry points. You must design a mechanism of your own, using either a synchronization variable or the **sleep()**/**wakeup()** function pair.

Using sleep() and wakeup()

The **sleep()** and **wakeup()** function pair are the simplest, oldest, and least efficient of the general synchronization mechanisms. They are summarized in Table 9-23.

Table 9-23 Functions for Synchronization: sleep/wakeup

Function Name	Header Files	Can Sleep?	Purpose
sleep(D3)	ddi.h & param.h	Y	Suspend execution pending an event.
wakeup(D3)	ddi.h	N	Waken a process waiting for an event.

Used carefully, these functions are suitable for simple character device drivers. However, when you are writing new code or converting a driver to multiprocessing you should avoid them and use synchronization variables instead (see "Using Synchronization Variables" on page 220).

The basic concept is that the upper-layer routine calls **sleep**(*n*) in order to wait for an event that is keyed to an arbitrary address *n*. Typically *n* is a pointer to a data structure related to an I/O operation. The interrupt handler executes **wakeup**(*n*) to cause the sleeping process to resume execution.

The main reason to avoid **sleep()** is that, in a multiprocessor system, it is hard to ensure that sleeping always begins before **wakeup()** is called. The usual intended sequence of events is as follows:

1. Upper-half routine initiates a device operation that will lead to an interrupt.
2. Upper-half routine executes **sleep**(*n*).
3. Interrupt occurs, and handler executes **wakeup**(*n*).

In a multiprocessor-aware driver (one with D_MP in its *pfx***devflag** constant; see "Driver Flag Constant" on page 140), there is a small chance that the interrupt can occur, calling **wakeup**(*n*), before the **sleep**(*n*) call has been completed. Because **sleep()** has not been called, the **wakeup()** is lost. When the **sleep()** call completes, the process sleeps forever. Synchronization variables are designed to handle this case.

Using Synchronization Variables

Synchronization variables, a feature of UNIX SVR4, are supported by IRIX beginning with release 6.2. These functions are summarized in Table 9-24.

Table 9-24 Functions for Synchronization: Synchronization Variables

Function Name	Header Files	Can Sleep?	Purpose
SV_ALLOC(D3)	types.h & sema.h	Y	Allocate and initialize a synchronization variable.
SV_DEALLOC(D3)	types.h & sema.h	N	Deinitialize and deallocate a synchronization variable.
SV_INIT	types.h & sema.h	N	Initialize an existing synchronization variable.
SV_DESTROY	types.h & sema.h		Deinitialize a synchronization variable.
SV_BROADCAST(D3)	types.h & sema.h	N	Wake all processes sleeping on a synchronization variable.
SV_SIGNAL(D3)	types.h & sema.h	N	Wake one process sleeping on a synchronization variable.
SV_WAIT(D3)	types.h & sema.h	Y	Sleep until a synchronization variable is signalled.
SV_WAIT_SIG(D3)	types.h & sema.h	Y	Sleep until a synchronization variable is signalled or a signal is received.

A synchronization variable is a memory object of type *sv_t*, representing the occurrence of an event. You can allocate objects of this type dynamically, or declare them as static variables or as fields of structures.

One or more processes may wait for an event using SV_WAIT(). An interrupt handler or timer callback function can signal the occurrence of an event using SV_SIGNAL (to wake up only one waiting process) or SV_BROADCAST (to wake up all of them).

SV_WAIT is specifically designed to handle the difficult case that arises when the driver needs to initiate an I/O operation and then sleep, and do these things in such a way that it always begins to sleep before the SV_SIGNAL can possibly be issued. The procedure is done as follows:

1. The driver seizes a basic lock (see "Basic Locks" on page 205) or a mutex lock (see "Using Mutex Locks" on page 207)that is also used by the interrupt handler.

   A LOCK() call returns an integer that is needed later.
2. The driver initiates an I/O operation that can lead to an interrupt.
3. The driver calls SV_WAIT, passing the lock it holds and an integer, either the value returned by LOCK() or a zero if the lock is a mutex lock.
4. In one indivisible operation, SV_WAIT releases the lock and begins waiting on the synchronization variable.
5. The interrupt handler or other process is entered, and seizes the lock.

   This step ensures that, if the interrupt handler or other process is entered preceding the SV_WAIT call, it will not proceed until SV_WAIT has completed.
6. The interrupt handler or other process does its work and calls SV_SIGNAL to release the waiting driver.

This process is sketched in Example 9-2.

Example 9-2 Skeleton Code for Use of SV_WAIT

```
lock_t seize_it;
sv_t wait_on_it;
initiator(...)
{
   int lock_cookie;
   for( as often as necessary )
   {
      lock_cookie = LOCK(&seize_it,PL_ZERO);
      [do something that causes a later interrupt]
      SV_WAIT(&wait_on_it, 0, &seize_it, lock_cookie);
      [interrupt has been handled]
   }
}

void handler(...)
{
   int lock_cookie = LOCK(&seize_it,PL_ZERO);
```

```
    [handle the interrupt]
    SV_SIGNAL(&seize_it);
    UNLOCK(&seize_it);
}
```

If it is necessary to use a semaphore as the lock, the header file sys/sema.h declares versions of SV_WAIT that accept a semaphore and a synchronization variable. The combination of a mutual exclusion object and a synchronization variable ensures that even in a multiprocessor, the interrupt handler cannot exit before the driver has entered a predictable wait state.

Tip: When a debugging kernel is used, you can display statistics about the use of a given synchronization variable. See "Including Lock Metering in the Kernel Image" on page 246.

Semaphores

The *semaphore* is a generalized tool that can be used for both mutual exclusion and for waiting. The IRIX kernel support for semaphores is summarized in Table 9-25.

Table 9-25 Functions for Semaphores

Function Name	Header Files	Can Sleep?	Purpose
cpsema(D3)	sema.h & types.h	N	Conditionally perform a "P" or wait semaphore operation.
cvsema(D3)	sema.h & types.h	N	Conditionally perform a "V" or release semaphore operation.
freesema(D3)	sema.h & types.h	N	Free the resources associated with a semaphore.
initnsema(D3)	sema.h & types.h	N	Initialize a semaphore to a given value.
initnsema_mutex(D3)	sema.h & types.h	N	Initialize a semaphore to a value of 1.
psema(D3)	sema.h & types.h & param.h	Y	Perform a "P" or wait semaphore operation.

Table 9-25 (continued) Functions for Semaphores

Function Name	Header Files	Can Sleep?	Purpose
valusema(D3)	sema.h & types.h	N	Return the value associated with a semaphore.
vsema(D3)	sema.h & types.h	N	Perform a "V" or signal semaphore operation.

Conceptually, a semaphore contains an integer. The "P" operation claims the semaphore, decrementing its count by 1 (mnemonic: dePlete). If the count is 0 or less, the process waits until the count is greater than 0 before it decrements the semaphore and returns.

The "V" operation increments the semaphore count (mnemonic: reViVe) and wakens any process that is waiting.

Tip: When a debugging kernel is used, you can display statistics about the use of a given semaphore. See "Including Lock Metering in the Kernel Image" on page 246.

Note: In releases before IRIX 6.2, **initnsema_mutex()** was used to initialize a semaphore in a special way that got the performance of a basic lock in a multiprocessor. Since IRIX 6.2, this function is simply a macro that initializes the semaphore to a count of 1.

Using a Semaphore for Mutual Exclusion

To use a semaphore for locking, initialize it to 1. (This reflects the idea that a process calling a locking function expects to continue.) When you require exclusive use of the associated resource, call **psema()**. Typically this finds a semaphore count of 1, reduces it to 0, and returns.

When you are finished with the resource, call **vsema()** to increment the semaphore count, and release any process that is blocked in a **psema()** call for the same semaphore.

For locking, a semaphore is comparable to a sleep lock. In some systems, the performance of semaphore operations may not be as good as the performance of a mutex lock. In other systems, mutex locks may be implemented using semaphores.

Using a Semaphore for Waiting

To use a semaphore for waiting, initialize it to 0. Then call **psema()**. Because the semaphore count is 0, the process waits. When the desired event occurs, typically in the interrupt handler, call **vsema()** to release the waiting process.

This synchronization method is as reliable as a synchronization variable, but it has slightly different behavior. When a synchronization variable is used correctly (see "Using Synchronization Variables" on page 220), if the interrupt handler is entered before the SV_WAIT call completes, the interrupt handler waits on a LOCK call.

When a semaphore is used, if the interrupt handler is entered before the **psema()** call completes, the **vsema()** operation is done immediately and the interrupt handler continues without waiting. The fact that **vsema()** was called is stored as a count within the semaphore, where **psema()** will find it. Because the semaphore can contain this state information, the interrupt handler does not have to be synchronized in time using a lock.

Note: In releases before IRIX 6.2, the **vpsema()** function was used in a way similar to synchronization variables are used: to release one semaphore and wait on another in an atomic operation. This function is no longer supported; replace it with syncronization variable.