Chapter 2

Writing a Device Driver

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

It contains the following sections:

Creating Device Drivers

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

Creating User-level Device Drivers

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

Creating Kernel-level Device Drivers

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

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

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

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

Device-special File

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

---

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

Creating Device-special Files

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

Synopsis

`mknod` *filename class major# minor#*

Arguments

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

Major and Minor Device Numbers

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

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

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

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

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

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

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

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

Device-special File Example

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

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

- 13: minor number of the device
- 2: major number of the device
- c: specifies a character device
- /dev/tty13: name of the device-special file

Configuration Files

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

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

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

STREAMS Name	X Device Name
sball	spaceball
calcomp	tablet

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

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

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

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

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

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

Including a Device Driver in the Kernel

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

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

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

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

mtune This directory contains information on the *external system tunable parameters* of the driver module, including default values and valid value ranges. For more information, see the **mtune**(4) man page.

system This directory contains files with directives that tell **lboot** whether to:

1. Include a driver module.
2. Conditionally include a driver module.
3. Exclude a driver module.

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

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

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

1. Become root.
2. Copy the current kernel to a safe place before rebooting.[1]

   ```
   # cp /unix /unix.orig
   ```

3. Create the new kernel, */unix.install*, by running:

   ```
   # /etc/autoconfig -f
   ```

   (Use the *-v* option during debugging.)

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

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

   ```
   # reboot
   ```

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

Driver Entry Points

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

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

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

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

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

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

Missing Driver Entry Points

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

Character and Block Entry Point Driver Routines

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

Table 2-1 Standard Entry Points

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

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

Table 2-2 Entry Point Driver Routines

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

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

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

open – Gain Access to a Device

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

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

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

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

Synopsis

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

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

Arguments

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

Returns

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

Notes

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

close – Relinquish Access to a Device

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

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

Synopsis

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

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

dev — Device major and minor numbers. Use **getemajor**() and **geteminor**() to get the major and minor device numbers from this parameter. The minor number helps you identify which device of a multidevice driver is being closed.

flag — A mode argument from the **close**() system call. Your code must check *flag* for *FREAD* and *FWRITE* bits. Typically, *flag* tells your code why the user wants to close the device.

otyp — A flag that tells your code the class of the device that it must close. This is useful if your driver must handle both character and block devices. For character devices, this flag is usually *OTYP_CHR*, but *OTYP_LYR* is also possible.

crp — A pointer to the user credential structure.

Returns

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

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

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

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

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

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

Synopsis

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

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

Arguments

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

Returns

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

ioctl – Control a Character Device

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

Synopsis

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

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

Arguments

dev — Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev*.

cmd — This parameter is useful when you have more than one “special routine.” The user cannot call these special routines directly. However, the user can call **ioctl**() with the appropriate value as its second parameter, and thus specify which special routine it wants. Within your code for the *drv***ioctl**() entry point, you must test the *cmd* parameter and take the appropriate action.

arg — This parameter can be used or ignored by your code as needed. Its type depends on the *cmd* argument. It can be either an integer value or a pointer to a device-specific data structure. (If it is a pointer, do not reference that address directly; instead, use **copyin**() or **copyout**() to retrieve the contents.)

Note: The size of int and pointer passed in can vary depending on the ABI outside a 64-bit kernel. See **userabi** and **userabi_t**. in “Device-special File” on page 22.

mode — The file modes set when the device was opened. Your driver can use this information to determine whether the device was opened for reading or writing.

crp — A pointer to the user credential structure.

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

Returns

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

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

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

Synopsis

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

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

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

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

    if ((events & (POLLOUT) && output not possible ) {
```

```
            *reventsp &= ~POLLOUT;
    }

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

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

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

Arguments

dev: Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev.*

events: A mask that indicates the events being polled. The significance of the bits of this value is defined in *sys/poll.h*. When the driver's **poll**() entry point is called, the driver must verify whether any of the events requested in *events* have occurred.

anyyet: A flag that indicates whether the driver must return a pointer to its *pollhead* structure to the caller.[1] If none of the events is pending, the driver must check the *anyyet* flag and, if it is zero, store the address of the device's *pollhead* structure in the pointer pointed to by *phpp*.

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

reventsp A pointer to a bitmask of the returned events satisfied. The driver must store the mask consisting of the subset of events that are pending in the short pointed to by *reventsp*. Note that this mask may be zero if none of the events is pending.

phpp A pointer to a pointer to a *pollhead* structure (defined in *sys/poll.h*).

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

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

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

Returns

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

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

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

Synopsis

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

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

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

Arguments

dev — Major and minor device numbers of the device it must handle. Use **getemajor**() and **geteminor**() to extract this information from *dev*.

vt — A handle to the virtual space in the calling process to which the device is mapped. (The structure for the handle is subject to change, so do not attempt to reference the members of the structure pointed to by the handle directly.)

off — An offset to an address within the device memory. This address is the start of the device memory that the user wants your code to map into user space. (The user may not want to map in all of the device memory.)

length — The number of bytes to map.

prot — A description of the protection to apply to the region it maps in. The values for this parameter can be found in *sys/man.h*.

devflag – driver flags

Synopsis

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

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

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

D_MP The driver is multithreaded (it handles its own locking and serialization).

D_WBACK The driver writes back cache before calling its *drv***strategy** routine.

D_OLD The driver uses the old-style interface (pre-5.0 release). This flag is not recommended for new work.

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

Writing Other Driver Routines

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

Table 2-3 Interrupt and Initialization Handling Routines

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

intr – Process a Device Interrupt

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

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

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

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

Arguments

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

strategy – Perform Block I/O

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

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

physiock() is a kernel routine that:

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

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

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

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

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

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

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

b_un.b_addr Contains the kernel virtual address from which information is read or to which information is written.

b_flags Contains a bit mask of flags that describe the transfer. *B_BUSY* is set to indicate that the buffer is in use for an I/O transfer. *B_READ* is set if the transfer is a read.

b_bcount Contains the number of bytes to be transferred.

b_edev Contains the major and minor device numbers.

b_blkno Contains the device block number to be transferred.

b_resid On completion, before calling **biodone**(), the driver must set this member to the number of bytes that were not transferred.

b_biodone If nonzero, this is taken as a function pointer, and the routine in question is called from **biodone**(); all normal **biodone**() processing is skipped. *b_biodone* may also be set by the user.

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

Synopsis

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

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

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

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

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

edtinit and init – Initialize a Device

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

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

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

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

Synopsis

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

halt – Shut Down the Driver When the System Shuts Down

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

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

Synopsis

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

Return Values

None

start – Initialize a Device at System Startup

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

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

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

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

Synopsis

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

Return Values

None

unload – Clean Up a Loadable Kernel Module

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

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

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

Synopsis

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

Return Values

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

Synchronization Constraints

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

STREAMS Driver Entry Points

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

Table 2-4 STREAMS Driver Entry Points

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

put – Coordinate Message Passing Between Queues in a Stream

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

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

Synopsis

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

Usage

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

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

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

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

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

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

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

Notes

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

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

Return Values

Ignored

Synchronization Constraints

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

srv – Service Routine

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

Synopsis

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

Usage

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

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

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

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

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

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

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

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

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

Notes

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

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

Return Values

Ignored

Synchronization Constraints

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