Chapter 10

Building and Installing a Driver

After a kernel-level driver has been designed and coded, it must be compiled, linked, and installed. The topics in this chapter describe the major steps of this process, as follows:

- "Defining Device Numbers" on page 226 covers the choice of major and minor device numbers.
- "Defining Device Special Files" on page 227 describes options for creating the file or files controlled by the driver.
- "Compiling and Linking" on page 228 covers the compiler and linker options used for driver modules.
- "Configuring a Nonloadable Driver" on page 232 describes the configuration files used to set up a driver loaded at boot time.
- "Configuring a Loadable Driver" on page 237 describes the additional configuration needed for a loadable driver.

Defining Device Numbers

The topics “Major Device Number” on page 34 and “Minor Device Number” on page 35 cover the purpose and use of the device numbers, which can be summarized as follows:

- Both numbers are encoded in the inode of a device special file in */dev.*
- The major number selects the device driver.
- The minor number specifies the logical unit, and can encode device features.
- Both numbers are passed as a parameter to driver entry points.

An important part of creating and installing a device driver is the selection of device numbers and the definition of device special files.

Selecting a Major Number

You must select a major number to stand for your driver. The numbers that already exist are listed in *sys/major.h*. However, the major number should not be coded into the driver. Typically the driver code does not need to know its major number, and if it does, the driver should discover its major number dynamically. A method of doing this is discussed under “Variables Section” on page 235.

A driver is associated with its major number in the *master.d* configuration file. When the driver discovers this number dynamically, the system administrator is free to change major numbers in */var/sysgen/master.d* files to correct conflicts between one product and another.

It is possible to let the *lboot* command choose a major number dynamically. This is discussed under “Configuring for a Dynamic Major Number” on page 241.

Selecting Minor Numbers

Each device minor number comprises 18 bits of information that are passed to driver entry points. The format of minor numbers can be coded into the driver. You design the content of these numbers to give your driver the information it needs about each device special file.

Examine the */dev/MAKEDEV* script to see some techniques for assembling minor numbers dynamically based on the hardware inventory and other commands.

Defining Device Special Files

As described under “Device Special Files” on page 32, the association between a device and a driver is established by encoding the major device number in the *inode* of a device special file in the */dev* directory. Without at least one device special file, a device can never be opened.

Static Definition of Device Special Files

The system administrator can create device special files using *mknod* or *install* (see “Making Device Files” on page 37). This can be done manually, or through an installation script, or as an exit operation of the software manager program. The device special files can be created at any time—whether or not the device driver is installed, and whether or not the hardware exists. The special device files continue to exist until the administrator removes them.

Dynamic Definition of Device Special Files

In a more sophisticated installation you might want to have the device special files created, or recreated, dynamically each time the system boots. This is the purpose of */dev/MAKEDEV* (see “The Script MAKEDEV” on page 36); it removes and redefines device special files based on information in the hardware inventory.

Much of the logic in */dev/MAKEDEV* depends on information reported by the *hinv* command. Through IRIX 6.2, there is no OEM interface for drivers to the hardware inventory (see “Hardware Inventory” on page 29), so at this time there is no opportunity for your driver to pass information through the inventory to */dev/MAKEDEV*.

However, there are other possibilities. For example, if your driver supports a control unit with an unknown number of minor devices, you could statically create a single device special file to represent the control unit. You could write the driver so that an **ioctl()** function directed to this file returns the count of actual minor devices.

Compiling and Linking

You compile a kernel device driver to an ELF binary (in IRIX 5.3 and before, the COFF binary format was used) not using shared libraries. The compile options differ between 32-bit and 64-bit modules.

Using /var/sysgen/Makefile.kernio

The file */var/sysgen/Makefile.kernio* is a sample Makefile for compiling kernel modules. You can include it from a Makefile of your own to set the definitions of compiler variables and compiler options appropriately for different CPUs and module types.

The *Makefile.kernio* file tests the following environment variables:

CPUBOARD	Set to the type of CPU used in the target system, for example IP19 or IP22.
COMPILATION_MODEL	Set to 64 for a 64-bit kernel module, or to 32 for a 32-bit kernel module.

The purpose of the rules in *Makefile.kernio* is to set compiler variables and compiler options into a Make variable CFLAGS. Other Make variables would be set by your own Makefile.

Note: *Makefile.kernio* is designed for nonloadable drivers. In particular it sets the compiler option *-G8*, which is valid for nonloadable drivers. A loadable driver must use *-G0*.

Compiler Variables

The compiler variables listed in Table 10-1 are tested in system header files. They are usually defined on the compiler command line. The rules in */var/sysgen/Makefile.kernio* set definitions of the variables appropriately for different CPU types.

Table 10-1 Compiler Variables Tested by System Header Files

Variable	Meaning
_K32U32	Compile for a 32-bit kernel running (only) 32-bit user programs.
_K32U64	Compile for a 32-bit kernel running 32-bit or 64-bit user programs (not supported in IRIX 6.2).
_K64U64	Compile for a 64-bit kernel running 32-bit or 64-bit user programs.
_KERNEL	Compile for a kernel module, not a user program.
STATIC=static	Use of pseudo-declarator STATIC is converted to real static.
JUMP_WAR	Compile workaround code for R4000 branch on end of page bug.
PROBE_WAR	Compile workaround code for R4000 bug which requires TLBprobe instruction to be performed in uncached mode.
BADVA_WAR	Compile workaround code for R4000 badvaddr bug.
TFP	Target machine is the R8000.
R4000	Target machine is the R4000.
R3000	Target machine is the R3000 (not supported after IRIX 5.3).
IP*nn*	Target machine uses the IP*nn* CPU module: IP17, IP19, IP20, IP22, IP26 are currently supported.
_PAGESZ=16384	Compile for a kernel using 16K memory pages (with IRIX 6.2 this implies _K64U64 also defined).
_PAGESZ=4096	Compile for a kernel using 4K memory pages (with IRIX 6.2 this implies _K32U32 also defined).
_MIPS3_ADDRSPACE	Kernel for a MIPS III (R8000) machine.
EVEREST	Compile for a Challenge or Onyx system.
MP	Compile for a multiprocessor.

Compile Options, 32-Bit Kernel

The *cc* and *ld* options shown in Table 10-2 are needed to compile and link a kernel-level driver for a 32-bit kernel in IRIX 6.2.

Table 10-2 Compiler Options for 32-Bit Kernel Modules

Option	Purpose
-non_shared	Do not compile for shared libraries (dynamic linking).
-elf	Compile and link an ELF binary.
-32 -mips2	Create a 32-bit executable for the MIPS II architecture.
-G 8	In a nonloadable driver, use the global table for objects up to 8 bytes.
-G 0	In a loadable driver, do not use the global table. Refer to the gp_overflow(5) reference page for a discussion of the global table.
-O2	Maximum recommended optimization level.
-r	Linker to retain symbols (needed by loadable drivers only).
-d	Force definition of common storage even though *-r* used.
-Wc,-pic0	Do not allocate stack space used by shared objects.
-jalr	Generate **jalr** instructions for subroutine calls rather than **jal**, which allows only a 26-bit target and so cannot address all kernel virtual storage.

Compile Options, 64-Bit Kernel

The *cc* and *ld* options listed in Table 10-3 are needed to compile and link a kernel-level driver for a 64-bit kernel in IRIX 6.2.

Table 10-3 Compiler Options for 64-Bit Kernel Modules

Option	**Purpose**
-non_shared	Do not compile for shared libraries (dynamic linking).
-elf	Compile and link an ELF binary.
-64 -mips3	Create a 32-bit executable for the MIPS III architecture. You can use *-mips4* instead, but only in a system based on the R10000 processor.
-G 8	In a nonloadable driver, use the global table for objects up to 8 bytes.
-G 0	In a loadable driver, do not use the global table. Refer to the gp_overflow(5) reference page for a discussion of the global table.
-O2	Maximum recommended optimization level.
-TENV:kernel *-TENV:misalignment=1*	Execution environment options for 64-bit compiler.
-OPT:space *-OPT:quad_align_branch_targets=FALSE* *-OPT:quad_align_with_memops=FALSE* *-OPT:unroll_times=0*	Specific optimization constraints for 64-bit compiler.

Configuring a Nonloadable Driver

When the driver is not loadable, it is linked as part of the IRIX kernel. The following steps are needed to make the driver usable:

1. Place the driver executable file in */var/sysgen/boot*.
2. Place a descriptive file in */var/sysgen/master.d*.
3. Place a directive file in */var/sysgen/system* (or simply add a line to */var/sysgen/system/irix.sm*).
4. Run *autoconfig* to generate a new kernel.
5. Reboot the system.

Some of these steps are elaborated in the following topics.

How Names Are Used in Configuration

The naming of a kernel-level driver begins in a file in */var/sysgen/system*, such as */var/sysgen/system/irix.sm*. Names are used as follows:

- A USE, INCLUDE, or VECTOR statement in */var/sysgen/system* specifies a name, for example

  ```
  USE hypothetical
  ```

- This statement directs *lboot* to read a file of the same name in */var/sysgen/master.d*, for example, */var/sysgen/master.d/hypothetical*.
- The file in */var/sysgen/master.d* specifies the prefix for driver entry points, for example *hypo_*.
- The same name with the suffix *.o*, is searched for in */var/sysgen/boot*—for example, */var/sysgen/boot/hypothetical.o*. This object file is linked with the kernel.
- The public symbols in the object file are searched for names that start with the prefix, for example **hypo_edtinit()**. These are noted in the kernel switch table so the driver can be called as needed.

Placing the Object File in /var/sysgen/boot

The */var/sysgen/boot* directory, where the kernel object modules reside, is actually a symbolic link to one of the directories */usr/cpu/sysgen/IP*nn*boot*, where *nn* is the number of one of the CPU modules supported by the current release of IRIX (see "CPU Modules" on page 5). When you place the object file of a driver in */var/sysgen/boot*, you actually place it in the CPU directory for the system in use.

Describing the Driver in /var/sysgen/master.d

You describe your driver in a file with the name of the driver in */var/sysgen/master.d*. The format of these files is described in two places: the master(4) reference page, and in */var/sysgen/master.d/README*. In addition, you can examine the many examples in the distributed system.

Descriptive Line

The first noncomment line of the master file contains a series of fields, delimited by white space, to describe the driver. These fields are:

Flags	Described in the text.
Prefix	The string of 1-14 characters that identify the public symbols of driver entry points.
Major number	The major number, or a comma-separated list of major numbers, found in device special files that are managed by this driver.
Number of sub-devices	Size of the driver's static arrays of device information, or given as a hyphen "-" when not known.
Dependencies	A list of other modules that must be in the kernel for this driver to work—usually omitted except for SCSI drivers.

Of the many flags that can be specified in the first field, the following are most frequently used:

b or *c*	Block (b) or character (c) device. One or the other is essential for any device driver.
f or *m*	STREAMS driver (f) or module (m).
n or *p*	Multiprocessor-aware (n) or uniprocessor-only (p) driver. This flag must correspond to the presence or absence of D_MP in the *pfx***devflag** global.
s	Software driver, either a pseudo-device or a SCSI driver.

The *s* (software-only) flag tells lboot not to attempt to probe for hardware. This is the case with software-only (pseudo-device) drivers, and with SCSI drivers. If *lboot* tries to probe for a SCSI device, it fails, and assumes that the device is not present, and does not include your SCSI device driver.

Additional flags (d, R, N, D) for loadable drivers are discussed later under "Configuring a Loadable Driver" on page 237.

Listing Dependencies

The descriptive line ends with a comma-separated list of other loadable kernel modules on which this driver depends. The *lboot* command makes sure that, if it fails to load a dependent module, it also will not load this module.

In most cases, an OEM driver does not have any dependencies. However, a SCSI driver (see Chapter 15, "SCSI Device Drivers") should list the name *scsi*, since it depends on the inner SCSI driver. A STREAMS driver might list the name of a STREAMS support module such as clone (see "Support for CLONE Drivers" on page 552).

It is possible for you to design a driver in the form of multiple, loadable modules. In that case, you would name your support modules in this field.

Stubs Section

Noncomment lines that follow the descriptive line and precede a line beginning "$" are used by library modules—not by device drivers or STREAMS drivers. Each such line specifies an entry point that this module provides, and which is used by the kernel or some other loadable module. These lines instruct *lboot* in how to create a harmless "stub" function in the event that this driver is not included in the kernel—for example, because it is specified by an EXCLUDE line in the system file. The format is discussed in the master(4) reference page.

Since a device or STREAMS driver provides only standard entry points that are accessed via the switch tables rather than by linking, drivers do not need to define any stubs.

Variables Section

Following the descriptive line (and the stubs section, if any), you can place a line that begins with "$" and, following this, you can write definitions of global variables using C syntax. This text is compiled into a module linked with the kernel. You refer to these variables as *extern* in the driver source code.

The advantage of defining global variables in the master file is that the initializing expressions for these variables can include the major device number and the number of supported sub-devices, as specified in the descriptive line of the file. This is how you make these items available to your driver without coding them into the driver. In the source code of the driver you can write

```
extern major_t myMajNum;
extern int myDevLimit;
```

In the master file you can write

```
$$$
major_t myMajNum = ##E;
int myDevLimit = ##C;
```

The global *myDevLimit* is compiled with the number of devices from the descriptive line. The global *myMajNum* is compiled with the major number. (In a loadable driver this technique requires one additional step; see "Master File for Loadable Drivers" on page 238.)

Configuring a Kernel

Once you have placed the binary in */var/sysgen/boot* and the master file in */var/sysgen/master.d*, you can configure and generate a new kernel. This is done using the *autoconfig* command, which in turn calls *lboot* to actually create a new bootable file.

The *lboot* program only loads modules that are specified in a file in */var/sysgen/system*. One command is required to specify the new driver; the command is one of the following:

VECTOR — To specify hardware details, to request a hardware probe at boot time, to load the driver and invoke *pfx***edtinit()**.

INCLUDE — To load the driver and invoke *pfx***init()**.

USE — To load the driver and invoke *pfx***init()** only if the master file exists in *master.d*.

The form of these commands is detailed in the system(4) reference page. In addition, you should examine the distributed files in */var/sysgen/system*, especial *irix.sm*, which contains many comments on the meaning and use of different entries. Specific uses of the VECTOR statement are discussed in the following topics:

- The form of VECTOR lines for VME devices is discussed under "Configuring the System Files" on page 311.
- The form of VECTOR lines for EISA devices is discussed under "Configuring IRIX" on page 436.
- The form of VECTOR lines for GIO devices is discussed under "Configuring a GIO Device" on page 514.

You could place the VECTOR, USE, or INCLUDE line for your driver in *irix.sm*. However, since *lboot* treats all files in */var/sysgen/system* as a single file, you can create a small file unique to your driver. The name of this file is not significant, but a good name is the driver name with the suffix *.sm*.

Generating a Kernel

The *autoconfig* script invokes *lboot* to read the system files, read the master files, and link all the kernel executables. Provided there are no errors, the output is a new file */unix.install*. At boot time this file is moved to the name */unix* and used as the kernel.

During the testing period you may want to keep the original kernel file available as */unix.original*. A simple way to retain this file is to create a link to it using the *ln* command.

Configuring a Loadable Driver

You compile and configure a loadable driver very much as you would a nonloadable driver. The differences are as follows:

- You provide an additional global variable with the public name *pfx***mversion**.
- You use a few different compile options.
- You decide when the driver should be loaded, and use appropriate flags in the descriptive line in the master file.

For more background on loadable modules, see the mload(4) and ml(1) reference pages.

Public Global Variables

To be loadable, a driver must specify a *pfx***devflag** entry point, and it may not contain the D_OLD flag (see “Driver Flag Constant” on page 140).

Any loadable module must define a public name *pfx***mversion**, declared as follows:

```
#include <sys/mload.h>
char *pfxmversion = M_VERSION;
```

Note the exact spelling of the variable; it is easy to overlook the letter “m” after the prefix. If the variable does not exist or is incorrectly spelled, an attempt to load the driver will fail.

Compile Options for Loadable Drivers

Use the *-G 0* option when compiling and linking a loadable driver, since the global option table is not available to a loadable driver (see “Compile Options, 32-Bit Kernel” on page 230 and “Compile Options, 64-Bit Kernel” on page 231).

In a loadable driver, link using the *-r* and *-d* options to retain the symbol table yet generate a bss segment.

Master File for Loadable Drivers

The file in */var/sysgen/master.d* for a loadable driver has different flags.

Descriptive Line

In the flags field of the descriptive line of the master file (see “Descriptive Line” on page 233), you specify that the driver is loadable, and when it should be loaded. The possible flags are as follows:

d Specifies that this is a loadable driver.

R Auto-register the module (discussed in text).

D Load, then unload, at boot time, in order to let the driver initialize the hardware inventory.

N Prevent this module from being automatically unloaded even when it has a *pfx***unload()** entry point.

When the *d* flag is given for an included module, *lboot* parses the master file for the driver. Global variables defined in the variables section of the master file (see “Variables Section” on page 235) are defined and included in the kernel. However, object code of the driver is not included in the kernel, and the names of its entry points are not entered into the kernel switch table.

Such a driver has to be manually loaded with the *ml* or *lboot* command before it can be used; and it cannot be used from the miniroot.

Registration

A loadable module is "registered" by having a stub entry placed in the *pfx***open()** column of a kernel switch table, indicating it exists but is not loaded. Registration can be done manually, after bootstrap, or automatically during bootstrap.

Registration is done automatically by for each master descriptive file that contains the *d* and *R* flags. Autoregistration is done at bootstrap phase 2. It is initiated by the script */etc/rc2/S23autoconfig*. Registration can be done manually at any time after bootstrap by using the *ml* or *lboot* command with the *reg* option (see the ml(1M) and lboot(1M) reference pages).

Once it has been registered, a driver is loaded automatically the first time a process attempts to open a device special file with the major device number of this driver. You can also load a registered driver in advance of any use with the *ml* or *lboot* command—loading implies registration.

Loading

A registered, loadable driver can be loaded by the *lboot* and *ml* commands. When a driver is loaded, the following steps are taken:

1. The object file header is read.
2. Memory is allocated for the module's text, data, and bss segments.
3. The module's text and data are read.
4. The module's text and data are relocated. References to kernel names and to global variables named in the master file are resolved.
5. The module entry points are noted in the appropriate kernel switch table.
6. The module's *pfx***init()** or *pfx***edtinit()** entry point is called, depending on whether the module is specified by an INCLUDE or a VECTOR statement in the system file (see "Initialization Entry Points" on page 143).
7. The module's *pfx***start()** entry point, if any, is called.

Space allocated for the module's text, data, and bss is located in either *k0seg* or *k2seg*. Static buffers in loadable modules are not necessarily physically contiguous in memory.

When the *D* flag is included in the descriptive line in the descriptive file, the module is loaded at boot time and then unloaded. This gives the module a chance to update the hardware inventory.

If errors occur when a module is loaded, see the mload(4) reference page for a list of possible causes.

Unloading

A module can be unloaded only when it provides an *pfx***unload()** entry point (see "Entry Point unload()" on page 167). The *N* flag can be specified in the master file to prevent automatic unloading in any case.

A loaded module is automatically unloaded following a delay after the last close of a device it supports. The delay is configurable using *systune*, as the *module_unld_delay* variable (see the systune(1) reference page). You can use *ml* to specify an unloading delay for a particular module.

The *lboot* or *ml* command can be used to unload a module before the delay expires, or when the *N* flag prevents automatic unloading.

Just before unloading, the kernel invokes the driver's *pfx***unload()** entry point. Then the module is removed from memory, and returned to registered status. It is up to the *pfx***unload()** entry point to decide whether unloading can be permitted. Experience has shown that most of the problems with loadable drivers arise from unloading and reloading. The precautions to take are described under "Entry Point unload()" on page 167.

Configuring for a Dynamic Major Number

You can place a hyphen (“-”) in the third field of the descriptive line; that is, the field for the major device number (see “Descriptive Line” on page 233). When the line also contains the *b* or *c* flag, indicating a device driver, a hyphen in the third field means that *lboot* is to assign some unused major device number dynamically when the module is registered.

Since a device driver can only be called by opening a device special file, and since a device special file has to be defined based on a major device number, how can the device special files be created?

The assigned major number can be discovered using the *ml* command. The command

```
ml list -r
```

writes a list of all registered modules, including their major numbers. The following procedure can be used to make the assignment of the major device number completely dynamic.

1. Make the device driver loadable, specifying the *d*, *R*, and *D* flags.
2. Specify a hyphen for the major number.
3. In the driver, get the major number dynamically, if indeed the driver needs to know its major number at all.
4. In a script executed from *rc2.d* (just as the */dev/MAKEDEV* script is executed), call *ml* to list registered drivers.
5. Parse the output of *ml* using UNIX utilities to extract the major device number from the line describing your driver.
6. Execute *mknod* or *install* commands to create special device files using the discovered major number.