Chapter 11

Kernel-level Dynamically Loadable Modules (DLMs)

This chapter describes how kernel modules can be loaded dynamically. It contains the following sections:

IRIX supports dynamic loading and unloading of modules into a running kernel. Kernel modules can be registered and then loaded automatically by the kernel when the corresponding device is opened, or they can be loaded using either the **lboot** or the **ml** command. Similarly, dynamically loaded modules can be unloaded from the kernel automatically or manually if the module includes an "unload" entry point. Loadable kernel modules that are supported include: character, block, and STREAMS device drivers; STREAMS modules; library modules; and the *idbg.o* module. This chapter describes how to configure and use dynamically loadable kernel modules.

Module Configuration

Each loadable module must contain the string shown below, where *M_VERSION* is defined in the *mload.h* header file that must be included by the loadable module:

```
char *drvmversion = M_VERSION;
```

A loadable module must be compiled and linked with the following **cc** options before it can be loaded into the kernel:[1]

```
% cc -non_shared -coff -Wx,-G0 -Wc,-pic0 -r -d -c -Wc,-jalr
```

-non_shared Produce a static executable. The output object created does not use any shared objects during execution.

-coff Produce a COFF object.

-Wx,-G0 Disable global pointer. Not supported for loadable modules.

-Wc,-pic0 Do not allocate extra stack space that is not necessary for *non_shared* coff objects.

-r Retain relocation entries in the output file.

-d Force definition of common storage and define loader-defined symbols. Without this option, space is not allocated in *bss* for common variables.

[1] No driver written for IRIX 6.0 should use the -coff flag.

-c Suppress the compilation loading phase and force an object file to be produced even if only one program is compiled.

-Wc,-jalr Force the compiler to produce **jalr** instructions rather than **jal** instructions. A **jal** instruction has a 26-bit target, so if a module is loaded into *k2seg*, for example, it could not call a kernel function in *k0seg*.

A loadable module must not be dependent on any loadable module, other than a library module. In order to load a module comprised of multiple object files, the object files must be linked into a single object file, using *ld* with the following options:[1]

```
% ld -non_shared -coff -G0 -r -d
```

For more information, see the **cc**(1) and **ld**(1) man page entries.

Using a Dynamically Loadable Kernel Module

You can use either **lboot** or the **ml** command to load, register, unload, unregister, and list loadable kernel modules. The **lboot** command parses module type, prefix, and major number information from the module's master file in the */var/sysgen/master.d* directory. The loadable object file is expected to be found in the */var/sysgen/boot* directory. The **ml** command also provides a means of loading, registering, and unloading loadable modules, without the having to create a master file or reconfigure the kernel.

Load

When a module is loaded, the following operations take place:

1. The object file's header is read.
2. Memory is allocated for the module's text, data, and bss.
3. The module's text and data are read.
4. The module's text and data are relocated, and unresolved references into the kernel are resolved.

5. A symbol table is created for the module; the module is added into the appropriate kernel switch table.
6. The module's *drv***init**() function 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.

A module can be loaded using the following **lboot** command:

```
% lboot -L master
```

A module can also be loaded using the following **ml** command:

```
% ml ld -[v] -[cbBfm] module.o -p prefix \
 [-s major major...] [-a modname]
```

If a major number is specified by the **ml** command, that major number must match the major number used to create the corresponding device in */dev*. If a major number is not specified, a device needs to be created in */dev* with the major number selected by the **ml** command. The major number selected by the **ml** command can be found by using the **ml list** command described below. For more information about creating devices, see the **mknod**(1M) man page entry.

If a module is loaded successfully by the **ml** command, an *id* number, which can be used to unload the module, is returned.

Register

A register command is used to register a module for loading when its corresponding device is opened. When a module is registered, a stub function is entered into the appropriate kernel switch table. When the corresponding device is opened, the stub function loads the module into the kernel.

A module can be registered with the following **lboot** command:

```
% lboot -R <master>
```

A module can also be registered with the following **ml** command:

```
% ml ld [-v] -[cbBfm] module.o -p prefix [-s major...] \
 [-a modname] [-t autounload_delay]
```

If a module is registered successfully with the *ml* command, an *id* number, which can be used to unregister the module, is returned.

Unload

A module can be unloaded only if it provides an "unload" entry point, described in "Module Entry Points." A module can be unloaded using the following **lboot** command:

```
% lboot -U <id>
```

A module can also be unloaded using the following **ml** command:

```
% ml unld id [id id ...]
```

Unregister

A module can be unregistered with the following **lboot** command:

```
% lboot -W id [id id ...]
```

A module can also be unregistered with the following **ml** command:

```
% ml unreg id [id id ...]
```

List

Loaded and/or registered modules can be listed with the following **lboot** command:

```
% lboot -V
```

Loaded and/or registered modules can also be listed with the following **lboot** command:

```
% ml list [-rlb]
```

For more information, see the **lboot**(1M) and **ml**(1M) man page entries.

Master File Configuration

If a dynamically loadable module has an associated master file, the master file must include a *d* in the *FLAG* field. The *d* flag indicates to **lboot** that the module is a dynamically loadable kernel module. If the *d* flag is present, **lboot** parses the module's master file but does not fill in the entry in the corresponding kernel switch table for the module. All global data defined in the master file are included in the *master.c* file generated by **lboot**. The kernel must be configured with master files that contain the *d* option for each module that will be a dynamically loadable module, if **lboot** is used to load, register, unload, unregister, or autoregister the module. If the **ml** command is used, then it is not necessary to create a master file for the module.

Auto Registration

Loadable modules can be registered by **lboot** automatically at system startup, when **autoconfig** is run. For a module to be auto-registered, its master file must contain an *R* in the *FLAG* field in addition to *d*, which indicates that the module is loadable. When **lboot** runs at system startup, it registers each module that contains an *R* in its master file.

Auto Unload

All registered modules that include an **unload** routine are automatically unloaded after last close unless they have been configured not to do so. By default, modules are unloaded five minutes after last close. You can change the default auto-unload delay by using **systune** to modify the *module_unld_delay* variable. For more information about **systune**, see the **systune** (1M) man page entry. To configure a particular module with a specific auto-unload delay, use the **ml** command. To configure a module so that it is not auto-unloaded, place an *N* in the flags filed of its *master.d* file, if it is registered using **lboot**; otherwise, use **ml** to register the module, then use the *-t* option.

Kernel Configuration

A kernel that supports loadable modules must be configured so that the kernel switch tables generated by **lboot** contain "extra" entries for the loadable modules. Extra entries are generated by **lboot** based on the values of the kernel-tunable parameters shown in Table 11-1.

Table 11-1 Kernel-tunable Parameters

Name	Default	Minimum	Maximum
bdevsw_extra	21	1	254
cdevsw_extra	23	3	254
fmodsw_extra	20	0	0
vfssw_extra	5	0	0

These tunable parameters are found in the */var/sysgen/mtune/kernel* kernel file and are set to the defaults listed above. It is not necessary to change the defaults unless you want the kernel to allow a greater or lesser number of loadable modules. For more information about changing tunable parameters, see the **mtune**(4) and **systune**(1M) manual entries or Chapter 11 of the *IRIX Advanced Site and Server Administration Guide.*

Module Entry Points

Loadable device drivers must conform to the UNIX System V Release 4 *DDI/DKI* standard. In addition to the entry points specified by the standard, if a loadable module is to be unloaded, the module needs to contain an unload entry point:

```
int drvunload (void)
```

An **unload**() routine should be treated as an interrupt routine. It should not call any routines that would cause it to sleep, such as **biowait**(), **sleep**(), **psema**(), or **delay**().

Module Initialization

After a module is loaded and linked into the kernel, and sanity checking is done, the module's initialization routines, *drv***init**(), *drv***edtinit**(), and *drv***start**(), are called, if they exist. For more information on these routines, refer to the *SVR4 DDI/DKI Reference Manual* and the *IRIX Device Driver Reference Pages*.

Edt Type Drivers

Drivers that have an *edt***init** entry point are passed a pointer to an *edt* structure. You must use **lboot** to load these drivers. Add a vector line to the *system* file for the driver. When the module is loaded using **lboot**, **lboot** parses the vector line from the *system* file to create an *edt* structure, which is passed through the kernel and to the driver's *edt***init** routine. The driver's *edt***init** routine is called after the driver is successfully loaded, after the driver's **init** routine is called. For more information, see the *system*(4) man page.

Library Modules

A library module is a module that contains a collection of functions and data that other loaded modules can link against. A library module that contains an **init** function calls it automatically after the module is loaded and linked into the kernel. To load a library module, use the **ml** command:

```
% ml ld [-v] -[l] library.o
```

A library module must be loaded before other modules that link against it are loaded. Library modules cannot be unloaded, registered, or unregistered. Only regular *COFF* object files are supported as loadable library modules.

Loadable Modules and Hardware Inventory

Many device drivers add to the hardware inventory in their **init** or *edt***init** routines. If a driver is a dynamically loadable driver and is auto-registered, it will not show up in the hardware inventory until the driver has been loaded on the first open of the corresponding device. If a clean install or a diskless install is done, a */dev* entry will not be created by **MAKEDEV** for such a driver, since it does not appear in the hardware inventory. If this situation arises, the *D master.d* flag can be used to indicate that the driver should be loaded, then unloaded, by **autoconfig**. If the *R master.d* flag, which indicates that the driver should be auto-registered, is also used, then the driver will be auto-registered as usual. A startup script can then be added to run **MAKEDEV** after **autoconfig**, if necessary. For an example, see the startup script in */etc/init.d/chkdev.*

Run-time Symbol Table

lboot creates a run-time symbol table from the tables in *master.d/rtsymtab* and *master.d/*.exports*. The run-time symbol table contains kernel routines and global data that modules can link against. Only routines and globals that are always present in the kernel should be added to *master.d/rtsymtab*.

If a module contains a routine or global that could be configured out of the kernel, add it to an *xxx.exports* file that will not be included in the kernel if the module is configured out. See *master.d/idev.exports* for an example.

If a loadable module contains globals in its *master.d* file, you must create an *xxx.exports* file to include those globals so that they can be added to the run-time symbol table. For more information, see *master.d/rtsymtab*.

Debugging Loadable Kernel Modules

symmon supports debugging of loadable kernel modules. **symmon** commands that do a symbol table lookup, such as *brk, lkup, lkaddr, hx*, and *nm*, also search the symbol tables created for loadable modules. The **msyms** command can also be used to list the symbols for a particular loaded module:

```
% msyms id
```

Use the *mlist* command to list correctly loaded and registered modules:

```
% mlist
```

For more information about using **symmon**, refer to Chapter 10, "Driver Installation and Testing".

The **ml** command contains a debug option that can be used to turn verbose error reporting on or off; it may also be used to disable the loading and registering of modules:

```
% ml debug [-vns]
```

-v the *verbose* option turns verbose error reporting on.

-n *no load or register* disallows the loading or registering of modules.

-s *silence* silences verbose error reporting and enables loading a registering of modules.

Load/Register Failures

If a registered module fails to load, unregister and reload it with **ml**, **ld**, or **lboot -L** to get a more detailed error message about the failure. The kernel will fail to load or register a module for any of the following reasons:

- The major number specified either in the master file or by the *ml* command is already in use.
- The object file is not compiled with the correct options.
- The module is an "old style" driver, with either *xxxdevflag* set to *D_OLD*, or if no *xxxdevflag* exists in the driver.
- If the object file is corrupted, it may cause "invalid JMPADDR" errors from the relocation code in the kernel.
- The kernel did not resolve all of the module's symbols.
- All major numbers are in use.
- The switch table is full.
- Required entry points for the particular type of module are not found in the loaded object file.

Example 1

The following example lists the steps necessary to build a kernel and load a character device driver, called *dlkm*, using **lboot**:

1. Add *d* to the *dlkm* master file:

*Flag	Prefix	Soft	#Dev	Dependencies
cd	dlkm	38	2	

2. Make sure that the *cdevsw_extra* kernel tuneable parameter allows for extra entries in the *cdevsw* table. The default settings in */var/sysgen/mtune/kernel* are:

cdevsw_extra	23	3	254

   The **systune** command also lists the current values of all of the tunable parameters. If the kernel is not configured to allow extra entries in the *cdevsw* table, use the **systune** command to change the *cdevsw_extra* parameter:

```
# systune -i
systune-> cdevsw_extra 3
systune-> quit >
```

3. Build a new kernel and boot the target system with the new kernel.
4. Compile the *dlkm.c* driver:

```
# cc -non_shared -coff -G0 -r -d -Wc,-jalr -c dlkm.c
```

5. Copy *dlkm.o* to */var/sysgen/boot*.
6. Load the driver into the kernel:

```
# lboot -L dlkm
```

7. List the currently loaded modules to verify that the module is loaded:

```
# lboot -V
```

Example 2

The following example lists the steps necessary to load a character device driver called *dlkm*, using the **ml** command:

1. Make sure that the *cdevsw_extra* kernel tunable parameter allows for extra entries in the *cdevsw* table. The default settings in */var/sysgen/mtune/kernel* are:

cdevsw_extra	23	3	254

   The **systune** command also lists the current values of all of the tunable parameters. If the kernel is not configured to allow extra entries in the *cdevsw* table, use the **systune** command to change the *cdevsw_extra* parameter:

   ```
   # systune -i
   systune-> cdevsw_extra 3
   systune-> quit >
   ```

2. Compile the *dlkm.c* driver:

   ```
   # cc -non_shared -coff -G0 -r -d -Wc,-jalr -c dlkm.c
   ```

3. Load the driver into the kernel:

   ```
   # ml ld -c dlkm.o -p dlkm -s38
   ```

4. List the currently loaded modules to verify that the module was loaded:

   ```
   # ml list
   ```