


Prentice Hall Open Source Software Development Series

# Embedded Linux<sup>®</sup> Primer



*A practical, real-world approach*

CHRISTOPHER HALLINAN

**Chapter 8. Device Driver Basics..... 1**

Section 8.1. Device Driver Concepts..... 2

Section 8.2. Module Utilities..... 11

Section 8.3. Driver Methods..... 17

Section 8.4. Bringing It All Together..... 21

Section 8.5. Device Drivers and the GPL..... 22

Section 8.6. Chapter Summary..... 22

# Chapter 8



## Device Driver Basics

### In this chapter

- Device Driver Concepts page 190
- Module Utilities page 199
- Driver Methods page 205
- Bringing It All Together page 209
- Device Drivers and the GPL page 211
- Chapter Summary page 211

One of the more challenging aspects of system design is partitioning functionality in a rational manner. The familiar device driver model found in UNIX and Linux provides a natural partitioning of functionality between your application code and hardware or kernel devices. In this chapter, we develop an understanding of this model and the basics of Linux device driver architecture. After reading this chapter, you will have a solid foundation for continuing your study of device drivers using one of the excellent texts listed at the end of this chapter.

This chapter begins by presenting Linux device driver concepts and the build system for drivers within the kernel source tree. We examine the Linux device driver architecture and present a simple working example driver. We introduce the user space utilities for loading and unloading kernel modules.<sup>1</sup> We present a simple application to illustrate the interface between applications and device drivers. We conclude this chapter with a discussion of device drivers and the GNU Public License.

## 8.1 Device Driver Concepts

Many experienced embedded developers struggle at first with the concepts of device drivers in a virtual memory operating system. This is because many popular legacy real-time operating systems do not have a similar architecture. The introduction of virtual memory and kernel space versus user space frequently introduces complexity that is not familiar to experienced embedded developers.

One of the fundamental purposes of a device driver is to isolate the user's programs from ready access to critical kernel data structures and hardware devices. Furthermore, a well-written device driver hides the complexity and variability of the hardware device from the user. For example, a program that wants to write data to the hard disk need not care if the disk drive uses 512-byte or 1024-byte sectors. The user simply opens a file and issues a write command. The device driver handles the details and isolates the user from the complexities and perils of hardware device programming. The device driver provides a consistent user interface to a large variety of hardware devices. It provides the basis for the familiar UNIX/Linux convention that everything must be represented as a file.

<sup>1</sup> The terms *module* and *device driver* are used here interchangeably.

### 8.1.1 Loadable Modules

Unlike some other operating systems, Linux has the capability to add and remove kernel components at runtime. Linux is structured as a monolithic kernel with a well-defined interface for adding and removing device driver modules dynamically after boot time. This feature not only adds flexibility to the user, but it has proven invaluable to the device driver development effort. Assuming that your device driver is reasonably well behaved, you can insert and remove the device driver from a running kernel at will during the development cycle instead of rebooting the kernel every time a change occurs.

Loadable modules have particular importance to embedded systems. Loadable modules enhance field upgrade capabilities; the module itself can be updated in a live system without the need for a reboot. Modules can be stored on media other than the root (boot) device, which can be space constrained.

Of course, device drivers can also be statically compiled into the kernel, and, for many drivers, this is completely appropriate. Consider, for example, a kernel configured to mount a root file system from a network-attached NFS server. In this scenario, you configure the network-related drivers (TCP/IP and the network interface card driver) to be compiled into the main kernel image so they are available during boot for mounting the remote root file system. You can use the initial ramdisk functionality as described in Chapter 6, “System Initialization,” as an alternative to having these drivers compiled statically as part of the kernel proper. In this case, the necessary modules and a script to load them would be included in the initial ramdisk image.

Loadable modules are installed after the kernel has booted. Startup scripts can load device driver modules, and modules can also be “demand loaded” when needed. The kernel has the capability to request a module when a service is requested that requires a particular module.

Terminology has never been standardized when discussing kernel modules. Many terms have been and continue to be used interchangeably when discussing loadable kernel modules. Throughout this and later chapters, the terms *device driver*, *loadable kernel module (LKM)*, *loadable module*, and *module* are all used to describe a loadable kernel device driver module.

### 8.1.2 Device Driver Architecture

The basic Linux device driver model is familiar to UNIX/Linux system developers. Although the device driver model continues to evolve, some fundamental constructs have remained nearly constant over the course of UNIX/Linux evolution. Device drivers are broadly classified into two basic categories: *character devices* and *block devices*. Character devices can be thought of as serial streams of sequential data. Examples of character devices include serial ports and keyboards. Block devices are characterized by the capability to read and write blocks of data to and from random locations on an addressable medium. Examples of block devices include hard drives and floppy disk drives.

### 8.1.3 Minimal Device Driver Example

Because Linux supports loadable device drivers, it is relatively easy to demonstrate a simple device driver skeleton. Listing 8-1 illustrates a loadable device driver module that contains the bare minimum structure to be loaded and unloaded by a running kernel.

#### Listing 8-1

##### Minimal Device Driver

---

```
/* Example Minimal Character Device Driver */
#include <linux/module.h>

static int __init hello_init(void)
{
    printk("Hello Example Init\n");

    return 0;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

---

The skeletal driver in Listing 8-1 contains enough structure for the kernel to load and unload the driver, and to invoke the initialization and exit routines. Let's look at how this is done because it illustrates some important high-level concepts that are useful for device driver development.

A device driver is a special kind of binary module. Unlike a stand-alone binary executable application, a device driver cannot be simply executed from a command prompt. The 2.6 kernel series requires that the binary be in a special “kernel object” format. When properly built, the device driver binary module contains a `.ko` suffix. The build steps and compiler options required to create the `.ko` module object can be quite complex. Here we outline a set of steps to harness the power of the Linux kernel build system without requiring you to become an expert in it, which is beyond the scope of this book.

#### 8.1.4 Module Build Infrastructure

A device driver must be compiled against the kernel on which it will execute. Although it is possible to load and execute kernel modules built against a different kernel version, it is risky to do so unless you are certain that the module does not rely on any features of your new kernel. The easiest way to do this is to build the module within the kernel's own source tree. This ensures that as the developer changes the kernel configuration, his custom driver is automatically rebuilt with the correct kernel configuration. It is certainly possible to build your drivers outside of the kernel source tree. However, in this case, you are responsible for making sure that your device driver build configuration stays in sync with the kernel you want to run your driver on. This typically includes compiler switches, location of kernel header files, and kernel configuration options.

For the example driver introduced in Listing 8-1, the following changes were made to the stock Linux kernel source tree to enable building this example driver. We explain each step in detail.

1. Starting from the top-level Linux source directory, create a directory under `.../drivers/char` called `examples`.
2. Add a menu item to the kernel configuration to enable building `examples` and to specify built-in or loadable kernel module.
3. Add the new `examples` subdirectory to the `.../drivers/char/Makefile` conditional on the menu item created in step 2.

4. Create a makefile for the new `examples` directory, and add the `hello1.o` module object to be compiled conditional on the menu item created in step 2.
5. Finally, create the driver `hello1.c` source file from Listing 8.1.

Adding the `examples` directory under the `.../drivers/char` subdirectory is self-explanatory. After this directory is created, two files are created in this directory: the module source file itself from Listing 8-1 and the makefile for the `examples` directory. The makefile for `examples` is quite trivial. It will contain this single line:

```
obj-$(CONFIG_EXAMPLES) += hello1.o
```

Adding the menu item to the kernel configuration utility is a little more involved. Listing 8-2 contains a patch that, when applied to the `.../drivers/char/Kconfig` file from a recent Linux release, adds the configuration menu item to enable our `examples` configuration option. For those readers not familiar with the `diff/patch` format, each line in Listing 8-1 preceded by a single plus (+) character is inserted in the file between the indicated lines (those without the leading + character).

## Listing 8-2

### Kconfig Patch for Examples

---

```
diff -u ~/base/linux-2.6.14/drivers/char/Kconfig
./drivers/char/Kconfig
--- ~/base/linux-2.6.14/drivers/char/Kconfig
+++ ./drivers/char/Kconfig
@@ -4,6 +4,12 @@

menu "Character devices"

+config EXAMPLES
+    tristate "Enable Examples"
+    default M
+    ---help---
+    Enable compilation option for driver examples
+
config VT
    bool "Virtual terminal" if EMBEDDED
    select INPUT
```

---

When applied to `Kconfig` in the `.../drivers/char` subdirectory of a recent Linux kernel, this patch results in a new kernel configuration option called



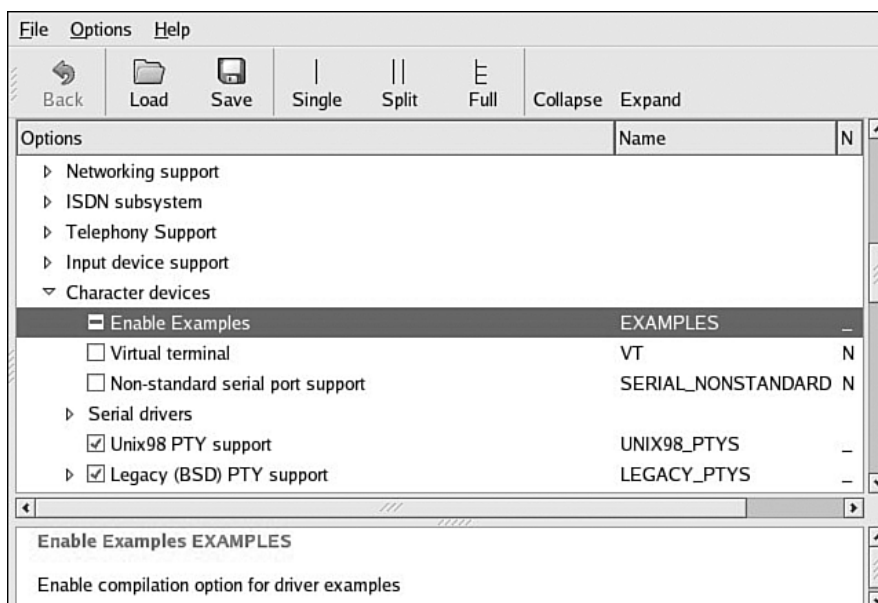
CONFIG\_EXAMPLES. As a reminder from our discussion on building the Linux kernel in Chapter 4, “The Linux Kernel—A Different Perspective,” the configuration utility is invoked as follows (this example assumes the ARM architecture):

```
$ make ARCH=ARM CROSS_COMPILE=xscale_be- gconfig
```

After the configuration utility is invoked using a command similar to the previous one, our new Enable Examples configuration option appears under the Character devices menu, as indicated in the patch. Because it is defined as type tristate, the kernel developer can choose from three choices:

- (N) No. Do not compile examples.
- (Y) Yes. Compile examples and link with final kernel image.
- (M) Module. Compile examples as dynamically loadable module.

Figure 8-1 shows the resulting gconfig screen with the new configuration option added. The dash (-) in the check box selects (M)odule, as indicated in the M column on the right. A check mark in the check box selects (Y)es, indicating that the driver module should be compiled as part of the kernel proper. An empty check box indicates that the option is not selected.



**FIGURE 8-1**  
Kernel configuration with Examples module

Now that we have added the configuration option to enable compiling our examples device driver module, we need to modify the makefile in `.../drivers/char` to instruct the build system to descend into our new `examples` subdirectory if the configuration option `CONFIG_EXAMPLES` is present in our configuration. Listing 8-3 contains the patch for this against the makefile in a recent Linux release.

### Listing 8-3

#### Makefile Patch for Examples

---

```
diff -u ~/base/linux-2.6.14/drivers/char/Makefile
./drivers/char/Makefile
--- ~/base/linux-2.6.14/drivers/char/Makefile
+++ ./drivers/char/Makefile
@@ -88,6 +88,7 @@
obj-$(CONFIG_DRM) += drm/
obj-$(CONFIG_PCMCIA) += pcmcia/
obj-$(CONFIG_IPMI_HANDLER) += ipmi/
+obj-$(CONFIG_EXAMPLES) += examples/

obj-$(CONFIG_HANGCHECK_TIMER) += hangcheck-timer.o
```

---

The patch in Listing 8-3 adds the single line (preceded by the `+` character) to the makefile found in `.../drivers/char`. The additional lines of context are there so that the patch utility can determine where to insert the new line. Our new `examples` directory was added to the end of the list of directories already being searched in this makefile, which seemed like a logical place to put it. Other than for consistency and readability, the location is irrelevant.

Having completed the steps in this section, the infrastructure is now in place to build the example device driver. The beauty of this approach is that the driver is built automatically whenever a kernel build is invoked. As long as the configuration option defined in Listing 8-3 is selected (either `M` or `Y`), the driver module is included in the build.

Building for an arbitrary ARM system, the command line for building modules might look like this:

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- modules
```

Listing 8-4 shows the build output after a typical editing session on the module (all other modules have already been built in this kernel source tree.)

### Listing 8-4

#### Module Build Output

---

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- modules
CHK      include/linux/version.h
make[1]: 'arch/arm/kernel/asm-offsets.s' is up to date.
make[1]: 'include/asm-arm/mach-types.h' is up to date.
CC [M]   drivers/char/examples/hello1.o
Building modules, stage 2.
MODPOST
LD [M]   drivers/char/examples/hello1.ko
```

---

#### 8.1.5 Installing Your Device Driver

Now that this driver is built, we can load and unload it on a running kernel to observe its behavior. Before we can load the module, we need to copy it to an appropriate location on our target system. Although we could put it anywhere we want, a convention is in place for kernel modules and where they are populated on a running Linux system. As with module compilation, it is easiest to let the kernel build system do that for us. The makefile target `modules_install` automatically places modules in the system in a logical layout. You simply need to supply the desired location as a prefix to the default path.

In a standard Linux workstation installation, you might already know that the device driver modules live in `/lib/modules/<kernel-version>/...` ordered in a manner similar to the device driver directory hierarchy in the Linux kernel tree.<sup>2</sup> The `<kernel-version>` string is produced by executing the command `uname -r` on your target Linux system. If you do not provide an installation prefix to the kernel build system, by default, your modules are installed in your own workstation's `/lib/modules/...` directory. This is probably not what you had intended. You can point to a temporary location in your home directory and manually copy the modules to your target's file system. Alternatively, if your target embedded system uses NFS root mount to a directory on your local development

---

<sup>2</sup> This path is used by Red Hat and Fedora distributions, and is also required by the File System Hierarchy Standard referenced at the end of this chapter. Other distributions might use different locations in the file system for kernel modules.

workstation, you can install the modules directly to the target file system. The following example assumes the latter.

---

```
$ make ARCH=arm CROSS_COMPILE=xscale_be- \
  INSTALL_MOD_PATH=/home/chris/sandbox/coyote-target \
  modules_install
```

---

This places all your modules in the directory `coyote-target`, which on this example system is exported via NFS and mounted as `root` on the target system.<sup>3</sup>

### 8.1.6 Loading Your Module

Having completed all the steps necessary, we are now in a position to load and test the device driver module. Listing 8-5 shows the output resulting from loading and subsequently unloading the device driver on the embedded system.

#### Listing 8-5

##### Loading and Unloading a Module

---

```
$ modprobe hello1          <<< Load the driver
Hello Example Init
$ modprobe -r hello1       <<< Unload the driver
Hello Example Exit
$
```

---

You should be able to correlate the output with our device driver source code found in Listing 8-1. The module does no work other than printing messages to the kernel log system via `printk()`, which we see on our console.<sup>4</sup> When the module is loaded, the module-initialization function is called. We specify the initialization function that will be executed on module insertion using the `module_init()` macro. We declared it as follows:

```
module_init(hello_init);
```

In our initialization function, we simply print the obligatory hello message and return. In a real device driver, this is where you would perform any initial resource allocation and hardware device initialization. In a similar fashion, when we unload the module (using the `modprobe -r` command), our module exit routine is called.

---

<sup>3</sup> Hosting a target board and NFS root mount are covered in detail in Chapter 12, “Embedded Development Environment”.

<sup>4</sup> If you don’t see the messages on the console, either disable your `syslogd` logger or lower the console loglevel. We describe how to do this in Chapter 14, “Kernel Debugging Techniques”.

As shown in Listing 8-1, the exit routine is specified using the `module_exit()` macro.

That's all there is to a skeletal device driver capable of live insertion in an actual kernel. In the sections to follow, we introduce additional functionality to our loadable device driver module that illustrates how a user space program would interact with a device driver module.

## 8.2 Module Utilities

We had a brief introduction to module utilities in Listing 8-5. There we used the module utility `modprobe` to insert and remove a device driver module from a Linux kernel. A number of small utilities are used to manage device driver modules. This section introduces them. You are encouraged to refer to the man page for each utility, for complete details. In fact, those interested in a greater knowledge of Linux loadable modules should consult the source code for these utilities. Section 8.6.1, “Suggestions for Additional Reading” at the end of this chapter contains a reference for where they can be found.

### 8.2.1 insmod

The `insmod` utility is the simplest way to insert a module into a running kernel. You supply a complete pathname, and `insmod` does the work. For example:

---

```
$ insmod /lib/modules/2.6.14/kernel/drivers/char/examples/hello1.ko
```

---

This loads the module `hello1.ko` into the kernel. The output would be the same as shown in Listing 8-5—namely, the `Hello` message. The `insmod` utility is a simple program that does not require or accept any options. It requires a full pathname because it has no logic for searching for the module. Most often, you will use `modprobe`, described shortly, because it has many more features and capabilities.

### 8.2.2 Module Parameters

Many device driver modules can accept parameters to modify their behavior. Examples include enabling debug mode, setting verbose reporting, or specifying module-specific options. The `insmod` utility accepts parameters (also called *options* in some contexts) by specifying them after the module name. Listing 8-6 shows our

modified `hello1.c` example, adding a single module parameter to enable debug mode.

## Listing 8-6

### Example Driver with Parameter

---

```
/* Example Minimal Character Device Driver */
#include <linux/module.h>

static int debug_enable = 0;          /* Added driver parameter */
module_param(debug_enable, int, 0); /* and these 2 lines */
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");

static int __init hello_init(void)
{
    /* Now print value of new module parameter */
    printk("Hello Example Init - debug mode is %s\n",
           debug_enable ? "enabled" : "disabled")

    return 0;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

---

Three lines have been added to our example device driver module. The first declares a static integer to hold our debug flag. The second line is a macro defined in `.../include/linux/moduleparam.h` that registers the module parameter with the kernel module subsystem. The third new line is a macro that registers a string description associated with the parameter with the kernel module subsystem. The purpose of this will become clear when we examine the `modinfo` command later in this chapter.

If we now use `insmod` to insert our example module, and add the `debug_enable` option, we should see the resulting output, based on our modified `hello1.c` module in Listing 8-6.

---

```
$ insmod /lib/modules/.../examples/hello1.ko debug_enable=1
Hello Example Init - debug mode is enabled
```

---

Or, if we omit the optional module parameter:

---

```
$ insmod /lib/modules/.../examples/hello1.ko
Hello Example Init - debug mode is disabled
```

---

### 8.2.3 lsmod

The `lsmod` utility is also quite trivial. It simply displays a formatted list of the modules that are inserted into the kernel. Recent versions take no parameters and simply format the output of `/proc/modules`.<sup>5</sup> Listing 8-7 is an example of the output from `lsmod`.

#### Listing 8-7

##### lsmod Example Output Format

---

```
$ lsmod
Module                Size  Used by
ext3                  121096  0
jbd                   49656   1 ext3
loop                  12712   0
hello1                 1412   0
$
```

---

Notice the rightmost column labeled `Used by`. This column indicates that the device driver module is in use and shows the dependency chain. In this example, the `jbd` module (journaling routines for journaling file systems) is being used by the `ext3` module, the default journaling file system for many popular Linux desktop distributions. This means that the `ext3` device driver depends on the presence of `jbd`.

### 8.2.4 modprobe

This is where the cleverness of `modprobe` comes into play. In Listing 8-7, we see the relationship between the `ext3` and `jbd` modules. The `ext3` module depends on the

---

<sup>5</sup> `/proc/modules` is part of the `proc` file system, which is introduced in Chapter 9, “File Systems”.

jbd module. The modprobe utility can discover this relationship and load the dependent modules in the proper order. The following command loads both the jbd.ko and ext3.ko driver modules:

```
$ modprobe ext3
```

The modprobe utility has several command line options that control its behavior. As we saw earlier, modprobe can be used to remove modules, including the modules upon which a given module depends. Here is an example of module removal that removes both jbd.ko and ext3.ko:

```
$ modprobe -r ext3
```

The modprobe utility is driven by a configuration file called modprobe.conf. This enables a system developer to associate devices with device drivers. For a simple embedded system, modprobe.conf might be empty or might contain very few lines. The modprobe utility is compiled with a set of default rules that establish the defaults in the absence of a valid modprobe.conf. Invoking modprobe with only the -c option displays the set of default rules used by modprobe.

Listing 8-8 represents a typical modprobe.conf, which might be found on a system containing two Ethernet interfaces; one is a wireless adapter based on the Prism2 chipset, and the other is a typical PCI Ethernet card. This system also contains a sound subsystem based on an integrated Intel sound chipset.

## Listing 8-8

### Typical modprobe.conf File

---

```
$ cat /etc/modprobe.conf
alias eth1 orinoci_pci
options eth1 orinoco_debug=9
alias eth0 e100
alias snd-card-0 snd-intel8x0
options snd-card-0 index=0
$
```

---

When the kernel boots and discovers the wireless chipset, this configuration file instructs modprobe to load the orinoco\_pci device driver, bound to kernel device eth1, and pass the optional module parameter orinoco\_debug=9 to the device driver. The same action is taken upon discovery of the sound card hardware. Notice the optional parameters associated with the sound driver snd-intel8x0.



### 8.2.5 depmod

How does modprobe know about the dependencies of a given module? The depmod utility plays a key role in this process. When modprobe is executed, it searches for a file called `modules.dep` in the same location where the modules are installed. The depmod utility creates this module-dependency file.

This file contains a list of all the modules that the kernel build system is configured for, along with dependency information for each. It is a simple file format: Each device driver module occupies one line in the file. If the module has dependencies, they are listed in order following the module name. For example, from Listing 8-7, we saw that the `ext3` module had a dependency on the `jbd` module. The dependency line in `modules.dep` would look like this:

```
ext3.ko: jbd.ko
```

In actual practice, each module name is preceded by its absolute path in the file system, to avoid ambiguity. We have omitted the path information for readability. A more complicated dependency chain, such as sound drivers, might look like this:

---

```
snd-intel8x0.ko: snd-ac97-codec.ko snd-pcm.ko snd-timer.ko \
snd.ko soundcore.ko snd-page-alloc.ko
```

---

Again, we have removed the leading path components for readability. Each module filename in the `modules.dep` file is an absolute filename, with complete path information, and exists on a single line. The previous example has been truncated to two lines, to fit in the space on this page.

Normally, depmod is run automatically during a kernel build. However, in a cross-development environment, you must have a cross-version of depmod that knows how to read the modules that are compiled in the native format of your target architecture. Alternatively, most embedded distributions have a method and `init` script entries to run depmod on each boot, to guarantee that the module dependencies are kept up-to-date.

### 8.2.6 rmmod

This utility is also quite trivial. It simply removes a module from a running kernel. Pass it the module name as a parameter. There is no need to include a pathname or file extension. For example:

---

```
$ rmmod hello1
Hello Example Exit
```

---

The only interesting point to understand here is that when you use `rmmod`, it executes the module's `*_exit()` function, as shown in the previous example, from our `hello1.c` example of Listings 8-1 and 8-6.

It should be noted that, unlike `modprobe`, `rmmod` does not remove dependent modules. Use `modprobe -r` for this.

### 8.2.7 modinfo

You might have noticed the last three lines of the skeletal driver in Listing 8-1, and later in Listing 8-6. These macros are there to place tags in the binary module to facilitate their administration and management. Listing 8-9 is the result of `modinfo` executed on our `hello1.ko` module.

#### Listing 8-9

##### modinfo Output

---

```
$ modinfo hello1
filename:          /lib/modules/.../char/examples/hello1.ko
author:            Chris Hallinan
description:       Hello World Example
license:           GPL
vermagic:          2.6.14 ARMv5 gcc-3.3
depends:
parm:              debug_enable:Enable module debug mode. (int)
$
```

---

The first field is obvious: It is the full filename of the device driver module. For readability in this listing, we have truncated the path again. The next lines are a direct result of the descriptive macros found at the end of Listing 8-6—namely, the filename, author, and license information. These are simply tags for use by the module utilities and do not affect the behavior of the device driver itself. You can learn more about `modinfo` from its `man` page and the `modinfo` source itself.

One very useful feature of `modinfo` is to learn what parameters the module supports. From Listing 8-9, you can see that this module supports just one parameter. This was the one we added in Listing 8-6, `debug_enable`. The listing gives the name, type (in this case, an `int`), and descriptive text field we entered with the `MODULE_PARM_DESC()` macro. This can be very handy, especially for modules in which you might not have easy access to the source code.

## 8.3 Driver Methods

We've covered much ground in our short treatment of module utilities. In the remaining sections of this chapter, we describe the basic mechanism for communicating with a device driver from a user space program (your application code).

We have introduced the two fundamental methods responsible for one-time initialization and exit processing of the module. Recall from Listing 8-1 that these are `module_init()` and `module_exit()`. We discovered that these routines are invoked at the time the module is inserted into or removed from a running kernel. Now we need some methods to interface with our device driver from our application program. After all, two of the more important reasons we use device drivers are to isolate the user from the perils of writing code in kernel space and to present a unified method to communicate with hardware or kernel-level devices.

### 8.3.1 Driver File System Operations

After the device driver is loaded into a live kernel, the first action we must take is to prepare the driver for subsequent operations. The `open()` method is used for this purpose. After the driver has been opened, we need routines for reading and writing to the driver. A `release()` routine is provided to clean up after operations when complete (basically, a close call). Finally, a special system call is provided for nonstandard communication to the driver. This is called `ioctl()`. Listing 8-10 adds this infrastructure to our example device driver.

#### Listing 8-10

##### Adding File System Ops to `Hello.c`

---

```
#include <linux/module.h>
#include <linux/fs.h>

#define HELLO_MAJOR 234

static int debug_enable = 0;
module_param(debug_enable, int, 0);
MODULE_PARM_DESC(debug_enable, "Enable module debug mode.");

struct file_operations hello_fops;

static int hello_open(struct inode *inode, struct file *file)
{
    printk("hello_open: successful\n");
    return 0;
}
```

```

static int hello_release(struct inode *inode, struct file *file)
{
    printk("hello_release: successful\n");
    return 0;
}

static ssize_t hello_read(struct file *file, char *buf, size_t count,
                          loff_t *ptr)
{
    printk("hello_read: returning zero bytes\n");
    return 0;
}

static ssize_t hello_write(struct file *file, const char *buf,
                           size_t count, loff_t * ppos)
{
    printk("hello_read: accepting zero bytes\n");
    return 0;
}

static int hello_ioctl(struct inode *inode, struct file *file,
                       unsigned int cmd, unsigned long arg)
{
    printk("hello_ioctl: cmd=%ld, arg=%ld\n", cmd, arg);
    return 0;
}

static int __init hello_init(void)
{
    int ret;
    printk("Hello Example Init - debug mode is %s\n",
           debug_enable ? "enabled" : "disabled");
    ret = register_chrdev(HELLO_MAJOR, "hello1", &hello_fops);
    if (ret < 0) {
        printk("Error registering hello device\n");
        goto hello_fail1;
    }
    printk("Hello: registered module successfully!\n");

    /* Init processing here... */

    return 0;

hello_fail1:
    return ret;
}

static void __exit hello_exit(void)
{
    printk("Hello Example Exit\n");
}

```

```

struct file_operations hello_fops = {
    owner:    THIS_MODULE,
    read:     hello_read,
    write:    hello_write,
    ioctl:    hello_ioctl,
    open:     hello_open,
    release:  hello_release,
};

module_init(hello_init);
module_exit(hello_exit);

MODULE_AUTHOR("Chris Hallinan");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");

```

This expanded device driver example includes many new lines. From the top, we've had to add a new kernel header file to get the definitions for the file system operations. We've also defined a major number for our device driver. (*Note to device driver authors:* This is not the proper way to allocate a device driver major number. Refer to the Linux kernel documentation (.../Documentation/devices.txt) or one of the excellent texts on device drivers for guidance on the allocation of major device numbers. For this simple example, we simply choose one that we know isn't in use on our system.)

Next we see definitions for four new functions, our open, close, read, and write methods. In keeping with good coding practices, we've adopted a consistent naming scheme that will not collide with any other subsystems in the kernel. Our new methods are called `hello_open()`, `hello_release()`, `hello_read()`, and `hello_write()`, respectively. For purposes of this simple exercise, they are do-nothing functions that simply print a message to the kernel log subsystem.

Notice that we've also added a new function call to our `hello_init()` routine. This line registers our device driver with the kernel. With that registration call, we pass a structure containing pointers to the required methods. The kernel uses this structure, of type `struct file_operations`, to bind our specific device functions with the appropriate requests from the file system. When an application opens a device represented by our device driver and requests a `read()` operation, the file system associates that generic `read()` request with our module's `hello_read()` function. The following sections examine this process in detail.

### 8.3.2 Device Nodes and `mknod`

To understand how an application binds its requests to a specific device represented by our device driver, we must understand the concept of a device node. A device node is a special file type in Linux that represents a device. Virtually all Linux distributions keep device nodes in a common location (specified by the Filesystem Hierarchy Standard<sup>6</sup>), in a directory called `/dev`. A dedicated utility is used to create a device node on a file system. This utility is called `mknod`.

An example of node creation is the best way to illustrate its functionality and the information it conveys. In keeping with our simple device driver example, let's create the proper device node to exercise it:

```
$ mknod /dev/hello1 c 234 0
```

After executing this command on our target embedded system, we end up with a new file called `/dev/hello1` that represents our device driver module. If we list this file to the console, it looks like this:

---

```
$ ls -l /dev/hello1
crw-r--r--  1 root  root  234, 0 Jul 14 2005 /dev/hello1
```

---

The parameters we passed to `mknod` include the name, type, and major and minor numbers for our device driver. The name we chose, of course, was `hello1`. Because we are demonstrating the use of a character driver, we use `c` to indicate that. The major number is 234, the number we chose for this example, and the minor number is 0.

By itself, the device node is just another file on our file system. However, because of its special status as a device node, we use it to bind to an installed device driver. If an application process issues an `open()` system call with our device node as the path parameter, the kernel searches for a valid device driver registered with a major number that matches the device node—in our case, 234. This is the mechanism by which the kernel associates our particular device to the device node.

As most C programmers know, the `open()` system call, or any of its variants, returns a reference (file descriptor) that our applications use to issue subsequent file system operations, such as read, write, and close. This reference is then passed to the various file system operations, such as read, write, or their variants.

---

<sup>6</sup> Reference to this standard is found in the “Suggestions for Additional Reading,” at the end of this chapter.

For those curious about the purpose of the minor number, it is a mechanism for handling multiple devices or subdevices with a single device driver. It is not used by the operating system; it is simply passed to the device driver. The device driver can use the minor number in any way it sees fit. As an example, with a multiport serial card, the major number would specify the driver. The minor number might specify one of the multiple ports handled by the same driver on the multiport card. Interested readers are encouraged to consult one of the excellent texts on device drivers for further details.

## 8.4 Bringing It All Together

Now that we have a skeletal device driver, we can load it and exercise it. Listing 8-11 is a simple user space application that exercises our device driver. We've already seen how to load the driver. Simply compile it and issue the `make modules_install` command to place it on your file system, as previously described.

### Listing 8-11

#### Exercising Our Device Driver

---

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    /* Our file descriptor */
    int fd;
    int rc = 0;
    char *rd_buf[16];

    printf("%s: entered\n", argv[0]);

    /* Open the device */
    fd = open("/dev/hello1", O_RDWR);
    if ( fd == -1 ) {
        perror("open failed");
        rc = fd;
        exit(-1);
    }
    printf("%s: open: successful\n", argv[0]);
```

## 8.5 Device Drivers and the GPL

Much discussion and debate surrounds the issue of device drivers and how the terms of the GNU Public License apply to device drivers. The first test is well understood: If your device driver (or any software, for that matter) is based, even in part, on existing GPL software, it is called a *derived work*. For example, if you start with a current Linux device driver and modify it to suit your needs, this is certainly considered a derived work, and you are obligated to license this modified device driver under the terms of the GPL, observing all its requirements.

This is where the debate comes in. First, the disclaimer. This is not a legal opinion, and the author is not a lawyer. Some of these concepts have not been tested in court as of this writing. The prevailing opinion of the legal and open source communities is that if a work can be proven<sup>8</sup> to be independently derived, and a given device driver does not assume “intimate knowledge” of the Linux kernel, the developers are free to license it in any way they see fit. If modifications are made to the kernel to accommodate a special need of the driver, it is considered a derived work and, therefore, is subject to the GPL.

A large and growing body of information exists in the open source community regarding these issues. It seems likely that, at some point in the future, these concepts will be tested in a court of law and precedent will be established. How long that might take is anyone’s guess. If you are interested in gaining a better understanding of the legal issues surrounding Linux and open source, you might enjoy [www.open-bar.org](http://www.open-bar.org).

## 8.6 Chapter Summary

This chapter presented a high-level overview of device driver basics and how they fit into the architecture of a Linux system. Armed with the basics, readers new to device drivers can jump into one of the excellent texts devoted to device driver writers. Consult Section 8.6.1 for references.

- Device drivers enforce a rational separation between unprivileged user applications and critical kernel resources such as hardware and other devices, and present a well-known unified interface to applications.

---

<sup>8</sup> This practice is not unique to open source. Copyright and patent infringement is an ongoing concern for all developers.



```

/* Issue a read */
rc = read(fd, rd_buf, 0);
if ( rc == -1 ) {
    perror("read failed");
    close(fd);
    exit(-1);
}
printf("%s: read: returning %d bytes!\n", argv[0], rc);

close(fd);
return 0;
}

```

---

This simple file, compiled on an ARM XScale system, demonstrates the binding of application to device driver, through the device node. Like the device driver, it doesn't do any useful work, but it does demonstrate the concepts as it exercises some of the methods we introduced in the device driver of Listing 8-10.

First we issue an `open()` system call<sup>7</sup> on our device node created earlier. If the `open` succeeds, we indicate that with a message to the console. Next we issue a `read()` command and again print a message to the console on success. Notice that a read of 0 bytes is perfectly acceptable as far as the kernel is concerned and, in actual practice, indicates an end-of-file or out-of-data condition. Your device driver defines that special condition. When complete, we simply close the file and exit. Listing 8-12 captures the output of running this example application on an ARM XScale target:

## Listing 8-12

### Using the Example Driver

---

```

$ modprobe hello1
Hello Example Init - debug mode is disabled
Hello: registered module successfully!
$ ./use-hello
./use-hello: entered
./use-hello: open: successful
./use-hello: read: returning zero bytes!
$

```

---

<sup>7</sup> Actually, the `open()` call is a C library wrapper function around the Linux `sys_open()` system call.

- The minimum infrastructure to load a device driver is only a few lines of code. We presented this minimum infrastructure and built on the concepts to a simple shell of a driver module.
- Device drivers configured as loadable modules can be inserted into and removed from a running kernel after kernel boot.
- Module utilities are used to manage the insertion, removal, and listing of device driver modules. We covered the details of the module utilities used for these functions.
- Device nodes on your file system provide the glue between your userspace application and the device driver.
- Driver methods implement the familiar open, read, write, and close functionality commonly found in UNIX/Linux device drivers. This mechanism was explained by example, including a simple user application to exercise these driver methods.
- We concluded this chapter with an introduction to the relationship between kernel device drivers and the Open Source GNU Public License.

### 8.6.1 Suggestions for Additional Reading

*Linux Device Drivers*, 3rd Edition

Alessandro Rubini and Jonathan Corbet

O'Reilly Publishing, 2005

Filesystem Hierarchy Standard

Edited by Rusty Russel, Daniel Quinlan, and Christopher Yeoh

The File Systems Hierarchy Standards Group

[www.pathname.com/fhs/](http://www.pathname.com/fhs/)

Rusty's Linux Kernel Page

Module Utilities for 2.6

Rusty Russell

<http://kernel.org/pub/linux/kernel/people/rusty/>