BY: DR B. THANGARAJU

Kernel Corner: Starting with Linux Device Drivers

What is the Linux device driver? How are the application program and kernel modules different? Learn here about the role of major and minor numbers, how to register a character device, and loading and unloading a simple module.

The Linux source code is freely available under GPL (General Public License) and it has a facility to build a custom kernel for specific applications. Such applications may include some devices. Hence, device drivers to interface these devices are mandatory. The device driver is a set of functions used to control access to a device. The device drivers for keyboard, monitor, mouse and Ethernet card are usually inbuilt with the Linux kernel. But when you need to access new devices for a particular application, then you have to write device drivers to access them. Also, if you want to modify the default actions of the existing devices, you need to rewrite the corresponding device drivers.

A deep knowledge of writing the device driver is, in their best interest, mandatory for Linux kernel programmers, particularly for those who work in

the embedded and realtime arena. Here we discuss the basics of Linux device drivers, the difference between application program and kernel modules, the role of major and minor numbers, how to register a character device, and loading and unloading a simple module.

EVERYTHING IS A FILE

Linux (like Unix) treats everything as a file. Any device is also treated as a file. Linux file types can be seen in Figure 1.

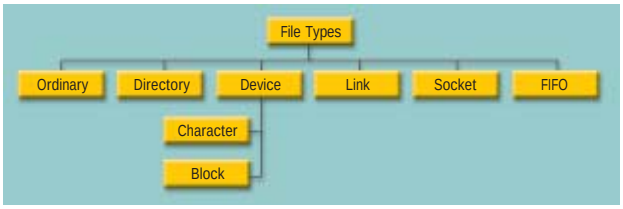

Figure 1

The following is the long list of various files in the system.

-rw-r—r—	1 root	root	2771	Mar 31 13:14	ordinary
drwxr-xr-x	2 root	root	4096	Mar 31 13:08	directory
brw-r—r—	1 root	root	252, 1	Mar 31 13:12	block
crw-r—r—	1 root	root	251, 1	Mar 31 13:11	character
lrwxrwxrwx	1 root	root	8	Mar 31 13:15	link -> ordinary
srwxrwxr-x	1 root	root	0	Mar 31 13:18	socket
prw-r—r—	1 root	root	0	Mar 31 13:09	fifo

The first two lines are ordinary and directory files, while the last two are socket and fifo special files respectively. Socket and fifo files are communication mechanisms used to communicate between processes. Successful creation of socket and fifo special files will return the socket and pipe descriptors, which are used to write or read data. A fifo descriptor is used to communicate between different processes, but only within a machine and in half-duplex mode. A socket descriptor is used to communicate between different machines and in full-duplex mode. The fifth line in the list shows the link file detail, which points to its target file. The third and fourth lines, which are of interest in this article, are the device files. The first column in the above list is the type of file and permission bits, the second column is the link count, third and fourth columns specify the user and group id, the fifth column indicates the file size, the sixth column is the date of creation or modification and the last column is the file name. Linux files are arranged in an inverted tree structure (starting from root) to achieve effective maintenance. System device files are stored in /dev directory.

MAJOR AND MINOR NUMBERS

In this article, we are only interested in device files and, hence, an explanation of other files is out of scope. Devices are classified into two categories, namely, character and block, depending on the access to the device by the system. A character device is accessed as a sequence of bytes, for example, mouse, keyboard, monitor and modem. Otherwise, if a device is accessed as a block of data, it is called a block device—for example, floppy, hard disk and CD-ROM. In the fifth column of the device files, two numbers can be seen instead of the file size. The numbers are major and minor numbers. The major number is used to identify the type of the device, such as floppy or hard disk. The minor number is used to identify the instance of the device—for example, in a multi-partition hard disk, if the major number identifies the hard disk, the minor number directs which partition of the device is going to be accessed. The rule for allocating major and minor numbers will be examine shortly.

APPLICATION PROGRAM VS MODULE

The device driver is a set of functions and is written in C. The device driver program is then compiled into an object (*.o) file. The object file (generally called module) is then inserted into the existing kernel dynamically. After loading the module, it becomes part of the kernel. This module will persist until the system is rebooted or explicitly removed. The device drivers for keyboard, monitor and mouse are in-built with the operating system and loaded statically when the system boots itself. The differences between application program and a module are:

- An application program often does a specific task and then exits, but a module is event driven and is always waiting for a future request or interrupt.
- In a module, there is no single entry point since it doesn't have main ( ). It has many entry points, depending on the request the corresponding entry point has invoked.
- Including standard headers is not possible in a module since it won't link them. So use of library functions, for example, printf, scanf and strcpy should be avoided. Instead of library functions, functions known to the kernel can be used; for example, printk is used instead of printf.
- An erroneous application program will affect only a particular process, but a flawed module will crash the entire system. So care should be taken when we write a device driver.

WRITING A SIMPLE MODULE

The two functions, namely, *init_module* and *cleanup_module*, are important to load and unload a device driver. *int init_module (void)* function is called when a module is inserted. This is the place where we can register a device name, allocate the major number (either dynamically or statically), assign resources for a device, and initialise such synchronising tools as the semaphore and spinlock. This

function accepts no argument and returns to an integer. *printk* function is used inside a driver module for debugging and for printing variable values.

To register a character device, use register_chrdev functions. The syntax of the function is follows:

```
int register_chrdev (unsigned int Major_number, const char
*device_name, struct file_operations *fos);
```

The function accepts three arguments and returns an integer. The first argument is an integer data type used to allocate major numbers to the device. If you choose dynamic major number allocation for a device, enter zero. Then the kernel will allocate the unused major number to the device. If you want to allocate a major number statically, then specify the number. But before choosing an arbitrary number, you should check the existing major number from the */proc/devices* file, which will list the allocated major number with the corresponding device. Some of the numbers, such as 60 – 63, 120 – 127 and 240 – 254, are not used for the standard devices and are used only for experimental purposes. More detailed information about the major number associated with the devices can be read in */usr/src/linux-2.4.18-3/Documentation/Devices.txt*. The second argument of *register_chrdev* function is the device name, and the third argument is the address of the file_operations structure. This structure is used to specify the list of function pointers (entry points), which will be used by this device. The *register_chrdev* function returns a major number if you choose dynamic allocation, zero for successful static allocation and -1 for error in both the cases.

void cleanup_module (void) is called when removing the module. Freeing the major number and removing the device name are done by calling the *unregister_chrdev* function.

*void unregister_chrdev (unsigned int Major_number, const char *device_name)*—the meaning of the arguments are the same as in the *init_module*. The following simple module (say *module.c*) registers a device and allocates the dynamic major number.

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
static int mno;

struct file_operations fos;

int init_module (void) {
        mno = register_chrdev(0, "my_device", &fos);
        if (mno == -1)
           return -1;
        else
           printk (" The major number is %d\n", mno);
        return 0;
}

void cleanup_module (void) {
            unregister_chrdev(mno, "my_device");
            printk (" Module is unloaded successfully and major
                                                    number %d is
released\n", mno);

}
```

Now, compile this module as *gcc –c module.c* and create *module.o* file. Insert module.o to the kernel by executing *insmod ./module.o*. Please note that only root can insert and remove a module. If the module is loaded successfully, the printk function output is seen on the monitor or it will be sent to */var/log/messages*. The *dmesg* command is also used to view the printk output. The module entry can be seen in */proc/modules* file or executing *lsmod* command and the device entry can be seen in */proc/devices* file.

rmmod command is used to remove the module and the corresponding printk output can be seen. Also, you can verify the absence of the module and device entry in the */proc/modules* and */proc/device* files respectively.

After experimenting with this module, I recommend you to read the articles lilsted in the box 'More Info', to know the basics of I/O port and I/O memory allocation for a device. In the next article, we use this module to include some of the entry points, such as open, close, read and write. Then we write an application program to access the device, such as moving data to and from the device. LFY

The author is a manager—Talent Transformation, at Wipro Technologies, Bangalore. He has been working in the fields of Linux Internals, Linux Kernel, Linux Device Drivers, Embedded and Real Time Linux for the past few years.

More Info

1. 'Kernel Corner: Fail Safe Port Allocation for Linux Device Drivers' by B. Thangaraju, Linux Focus Magazine, November/December 2002. http://www.linuxfocus.org/English/November2002/article264.shtml

2. 'Risk-Free Resource Allocation for I/O Memory-Mapped Device Drivers' by B. Thangaraju, Linux Gazette, Issue 83, October 2002. http://www.linuxgazette.com/issue83/thangaraju.html