An Introduction to Device Drivers in the Linux Kernel

NARENDRA KANGRALKAR, OCTOBER 2, 2014 903 0 COMMENTS

In the article 'An Introduction to the Linux Kernel' in the August 2014 issue of OSFY, we wrote and compiled a kernel module. In the second article in this series, we move on to device drivers.
Have you ever wondered how a computer plays audio or shows video? The answer is: by using device drivers. A few years ago we would always install audio or video drivers after installing MS Windows XP. Only then we were able to listen the audio. Let us explore device drivers in this column.
A device driver (often referred to as 'driver') is a piece of software that controls a particular type of device which is connected to the computer system. It provides a software interface to the hardware device, and enables access to the operating system and other applications. There are various types of drivers present in GNU/Linux such as Character, Block, Network and USB drivers. In this column, we will explore only character drivers.
Character drivers are the most common drivers. They provide unbuffered, direct access to hardware devices. One can think of character drivers as a long sequence of bytes — same as regular files but can be accessed only in sequential order. Character drivers support at least the *open(), close(), read() and write()* operations. The text console, i.e., */dev/console*, serial consoles */dev/stty**, and audio/video drivers fall under this category.
To make a device usable there must be a driver present for it. So let us understand how an application accesses data from a device with the help of a driver. We will discuss the following four major entities.

- **User-space application**: This can be any simple utility like *echo,* or any complex application.
- **Device file:** This is a special file that provides an interface for the driver. It is present in the file system as an ordinary file. The application can perform all supported operation on it, just like for an ordinary file. It can *move, copy, delete, rename, read* and *write* these device files.
- **Device driver:** This is the software interface for the device and resides in the kernel space.
- **Device:** This can be the actual device present at the hardware level, or a pseudo device.

Let us take an example where a user-space application sends data to a character device. Instead of using an actual device we are going to use a pseudo device. As the name suggests, this device is not a physical device. In GNU/Linux */dev/null* is the most commonly used pseudo device. This device accepts any kind of data (i.e., input) and simply discards it. And it doesn't produce any output.
Let us send some data to the */dev/null* pseudo device:

```
[mickey]$ echo -n 'a' > /dev/null
```

In the above example, *echo* is a user-space application and *null* is a special file present in the */dev* directory. There is a null driver present in the kernel to control the pseudo device.
To send or receive data to and from the device or application, use the corresponding device file that is connected to the driver through the Virtual File System (VFS) layer. Whenever an application wants to perform any operation on the actual device, it performs this on the device file. The VFS layer redirects those operations to the appropriate functions that are implemented inside the driver. This means that whenever an application performs the *open()* operation on a device file, in reality the *open()* function from the driver is invoked, and the same concept applies to the other functions. The implementation of these operations is device-specific.

Major and minor numbers

We have seen that the *echo* command directly sends data to the device file. Hence, it is clear that to send or receive data to and from the device, the application uses special device files. But how does communication between the device file and the driver take place? It happens via a pair of numbers referred to as 'major' and 'minor' numbers.
The command below lists the major and minor numbers associated with a character device file:

```
[bash]$ ls -l /dev/null
crw-rw-rw- 1 root root 1, 3 Jul 11 20:47 /dev/null
```

In the above output there are two numbers separated by a comma (1 and 3). Here, '1' is the major and '3' is the minor number. The major number identifies the driver associated with the device, i.e., which driver is to be used. The minor number is used by the kernel to determine exactly which device is being referred to. For instance, a hard disk may have

three partitions. Each partition will have a separate minor number but only one major number, because the same storage driver is used for all the partitions.
Older kernels used to have a separate major number for each driver. But modern Linux kernels allow multiple drivers to share the same major number. For instance, */dev/full, /dev/null, /dev/random* and */dev/zero* use the same major number but different minor numbers. The output below illustrates this:

```
[bash]$ ls -l /dev/full /dev/null /dev/random /dev/zero
crw-rw-rw- 1 root root 1, 7 Jul 11 20:47 /dev/full
crw-rw-rw- 1 root root 1, 3 Jul 11 20:47 /dev/null
crw-rw-rw- 1 root root 1, 8 Jul 11 20:47 /dev/random
crw-rw-rw- 1 root root 1, 5 Jul 11 20:47 /dev/zero
```

The kernel uses the *dev_t* type to store major and minor numbers. *dev_t* type is defined in the <*linux/types.h*> header file. Given below is the representation of *dev_t* type from the header file:

```
#ifndef _LINUX_TYPES_H
#define _LINUX_TYPES_H

#define __EXPORTED_HEADERS__
#include <uapi/linux/types.h>

typedef __u32 __kernel_dev_t;

typedef __kernel_dev_t dev_t;
```

dev_t is an unsigned 32-bit integer, where 12 bits are used to store the major number and the remaining 20 bits are used to store the minor number. But don't try to extract the major and minor numbers directly. Instead, the kernel provides MAJOR and MINOR macros that can be used to extract the major and minor numbers. The definition of the MAJOR and MINOR macros from the <*linux/kdev_t.h*> header file is given below:

```
#ifndef _LINUX_KDEV_T_H
#define _LINUX_KDEV_T_H

#include <uapi/linux/kdev_t.h>

#define MINORBITS 20
#define MINORMASK ((1U << MINORBITS) - 1)

#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
```

If you have major and minor numbers and you want to convert them to the *dev_t* type, the MKDEV macro will do the needful. The definition of the MKDEV macro from the <*linux/kdev_t.h*> header file is given below:

```
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

We now know what major and minor numbers are and the role they play. Let us see how we can allocate major numbers. Here is the prototype of the *register_chrdev():*

```
int register_chrdev(unsigned int major, const char *name, struct file_operations *f
```

This function registers a major number for character devices. Arguments of this function are self-explanatory. The *major* argument implies the major number of interest, *name* is the name of the driver and appears in the */proc/devices* area and, finally, *fops* is the pointer to the *file_operations* structure.
Certain major numbers are reserved for special drivers; hence, one should exclude those and use dynamically allocated major numbers. To allocate a major number dynamically, provide the value zero to the first argument, i.e., *major == 0*. This function will dynamically allocate and return a major number.
To deallocate an allocated major number use the *unregister_chrdev()* function. The prototype is given below and the parameters of the function are self-explanatory:

```
void unregister_chrdev(unsigned int major, const char *name)
```

The values of the major and name parameters must be the same as those passed to the *register_chrdev()* function; otherwise, the call will fail.

File operations

So we know how to allocate/deallocate the major number, but we haven't yet connected any of our driver's operations to the major number. To set up a connection, we are going to use the file_operations structure. This structure is defined in the <*linux/fs.h*> header file.
Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations. The example given below illustrates that.

Without discussing lengthy theory, let us write our first 'null' driver, which mimics the functionality of a */dev/null* pseudo device. Given below is the complete working code for the 'null' driver.
Open a file using your favourite text editor and save the code given below as *null_driver.c:*

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

static int major;
static char *name = "null_driver";

static int null_open(struct inode *i, struct file *f)
{
printk(KERN_INFO "Calling: %s\n", __func__);
return 0;
}

static int null_release(struct inode *i, struct file *f)
{
printk(KERN_INFO "Calling: %s\n", __func__);
return 0;
}

static ssize_t null_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
printk(KERN_INFO "Calling: %s\n", __func__);
return 0;
}

static ssize_t null_write(struct file *f, const char __user *buf, size_t len, loff_
{
printk(KERN_INFO "Calling: %s\n", __func__);
return len;
}

static struct file_operations null_ops =
{
.owner = THIS_MODULE,
.open = null_open,
.release = null_release,
.read = null_read,
.write = null_write
};

static int __init null_init(void)
{
major = register_chrdev(0, name, &null_ops);
if (major < 0) {
printk(KERN_INFO "Failed to register driver.");
return -1;
}

printk(KERN_INFO "Device registered successfully.\n");
return 0;
}

static void __exit null_exit(void)
{
unregister_chrdev(major, name);
printk(KERN_INFO "Device unregistered successfully.\n");
}

module_init(null_init);
module_exit(null_exit);

MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Null driver");
```

Our driver code is ready. Let us compile and insert the module. In the article last month, we did learn how to write *Makefile* for kernel modules.

```
[mickey]$ make

[root]# insmod ./null_driver.ko
```

We are now going to create a device file for our driver. But for this we need a major number, and we know that our driver's *register_chrdev()* function will allocate the major number dynamically. Let us find out this dynamically allocated major number from */proc/devices*, which shows the currently loaded kernel modules:

```
[root]# grep "null_driver" /proc/devices
248 null_driver
```

From the above output, we are going to use '248' as a major number for our driver. We are only interested in the major number, and the minor number can be anything within a valid range. I'll use '0' as the minor number. To create the character device file, use the *mknod* utility. Please note that to create the device file you must have superuser privileges:

```
[root]# mknod /dev/null_driver c 248 0
```

Now it's time for the action. Let us send some data to the pseudo device using the echo command and check the output of the *dmesg* command:

```
[root]# echo "Hello" > /dev/null_driver

[root]# dmesg
Device registered successfully.
Calling: null_open
Calling: null_write
Calling: null_release
```

Yes! We got the expected output. When *open, write, close* operations are performed on a device file, the appropriate functions from our driver's code get called. Let us perform the read operation and check the output of the *dmesg* command:

```
[root]# cat /dev/null_driver

[root]# dmesg
Calling: null_open
Calling: null_read
Calling: null_release
```

To make things simple I have used *printk()* statements in every function. If we remove these statements, then */dev/null_driver* will behave exactly the same as the */dev/null* pseudo device. Our code is working as expected. Let us understand the details of our character driver.
First, take a look at the driver's function. Given below are the prototypes of a few functions from the *file_operations* structure:

```
int (*open)(struct inode *i, struct file *f);
int (*release)(struct inode *i, struct file *f);
ssize_t (*read)(struct file *f, char __user *buf, size_t len, loff_t *off);
ssize_t (*write)(struct file *f, const char __user buf*, size_t len, loff_t *off);
```

The prototype of the *open()* and *release()* functions is exactly same. These functions accept two parameters—the first is the pointer to the *inode* structure. All file-related information such as size, owner, access permissions of the file, file creation timestamps, number of hard-links, etc, is represented by the *inode* structure. And each open file is represented internally by the *file* structure. The *open()* function is responsible for opening the device and allocation of required resources. The *release()* function does exactly the reverse job, which closes the device and deallocates the resources.
As the name suggests, the *read()* function reads data from the device and sends it to the application. The first parameter of this function is the pointer to the file structure. The second parameter is the user-space buffer. The third parameter is the size, which implies the number of bytes to be transferred to the user space buffer. And, finally, the fourth parameter is the file offset which updates the current file position. Whenever the *read()* operation is performed on a device file, the driver should copy len bytes of data from the device to the user-space buffer buf and update the file offset off accordingly. This function returns the number of bytes read successfully. Our null driver doesn't read anything; that is why the return value is always zero, i.e., EOF.
The driver's *write()* function accepts the data from the user-space application. The first parameter of this function is the pointer to the file structure. The second parameter is the user-space buffer, which holds the data received from the application. The third parameter is len which is the size of the data. The fourth parameter is the file offset. Whenever the *write()* operation is performed on a device file, the driver should transfer len bytes of data to the device and update the file offset off accordingly. Our *null* driver accepts input of any length; hence, return value is always len, i.e., all bytes are written successfully.
In the next step we have initialised the *file_operations* structure with the appropriate driver's function.
In *initialisation* function we have done a registration related job, and we are deregistering the character device in *cleanup* function.

Implementation of the full pseudo driver
Let us implement one more pseudo device, namely, *full*. Any write operation on this device fails and gives the 'ENOSPC'

error. This can be used to test how a program handles disk-full errors. Given below is the complete working code of the full driver:

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

static int major;
static char *name = "full_driver";

static int full_open(struct inode *i, struct file *f)
{
return 0;
}

static int full_release(struct inode *i, struct file *f)
{
return 0;
}

static ssize_t full_read(struct file *f, char __user *buf, size_t len, loff_t *off)
{
return 0;
}

static ssize_t full_write(struct file *f, const char __user *buf, size_t len, loff_
{
return -ENOSPC;
}

static struct file_operations full_ops =
{
.owner = THIS_MODULE,
.open = full_open,
.release = full_release,
.read = full_read,
.write = full_write
};

static int __init full_init(void)
{
major = register_chrdev(0, name, &full_ops);
if (major < 0) {
printk(KERN_INFO "Failed to register driver.");
return -1;
}

return 0;
}

static void __exit full_exit(void)
{
unregister_chrdev(major, name);
}

module_init(full_init);
module_exit(full_exit);

MODULE_AUTHOR("Narendra Kangralkar.");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Full driver");

Let us compile and insert the module.

[mickey]$ make

[root]# insmod ./full_driver.ko

[root]# grep "full_driver" /proc/devices
248 full_driver

[root]# mknod /dev/full_driver c 248 0

[root]# echo "Hello" > /dev/full_driver
-bash: echo: write error: No space left on device
```

If you want to learn more about GNU/Linux device drivers, the Linux kernel's source code is the best place to do so. You can browse the kernel's source code from *http://lxr.free-electrons.com/*. You can also download the latest source code from *https://www.kernel.org/*. Additionally, there are a few good books available in the market like 'Linux Kernel