



Drive Your Devices, Smoothly



Read about writing device drivers that have concurrent access to multiple instances of devices.



A device driver is a part of the kernel code that is capable of handling or interacting with the hardware. The development of device drivers is challenging, since it requires thorough understanding of both hardware and software.

Most of the device vendors or manufacturers provide device drivers to buyers. For writing software for a device driver, one should know what type of device it is and what operations can be done on it. Linux treats every device as a file. The development of a Linux device driver is more interesting, since Linux is published under the GPL. Since the device driver source code

is also available with Linux source code, we can modify and optimise it according to our requirements.

Application programmers make use of system calls such as *read*, *write*, etc to communicate with any device or file. In the Linux kernel, a system call interface layer understands these system calls. With the help of an inode table and switch table, corresponding code present in the device driver is invoked to serve the system call used in the application program.

In the Linux-2.6 kernel there are many changes compared to Linux-2.4. Because of its backward compatibility, many device driver programmers are continuing with the

old methods of device driver writing. But in the future, the support for old methods may vanish.

In this article we discuss writing a device driver for a simple-character pseudo device, which can handle multiple devices simultaneously. Here we strictly follow the methods to be used in the Linux-2.6 kernel. We have provided demo programs with suitable examples of code for each discussion, in order to understand the concepts clearly.

Necessary data structures

To write a device driver we need to know some of the important Linux kernel data structures. The data structures that are necessary to do file handling are found to be in `<linux/fs.h>`. The two major structures are *struct file* and *struct inode*.

```
struct file {
    ...
    const struct file_operations *f_op;
    atomic_t      f_count;
    unsigned int   f_flags;
    mode_t         f_mode;
    loff_t         f_pos;
    void           *private_data;
    ...
};
```

The *struct file_operations* contains array of function pointers, which define operations that can be done on any file, such as *open*, *release*, *read*, *write*, etc.

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t
    *);
    ssize_t (*write) (struct file *, const char __user *, size_t,
    loff_t *);
    int (*ioctl) (struct inode *, struct file *, unsigned int,
    unsigned long);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*flock) (struct file *, int, struct file_lock *);
    ...
};
```

The *private_data* refers to the allocated private data for that device. The *f_mode* refers to the opening mode of the file, such as O_RDONLY or O_RDWR, etc. The *f_pos* holds the current file pointer position. The *struct inode* holds information about *inode*. This contains information such as user ID, access time, size and more.

```
struct inode {
    ...
    unsigned long   i_ino;
    atomic_t        i_count;
```

```
umode_t          i_mode;
unsigned int      i_nlink;
uid_t            i_uid;
gid_t            i_gid;
dev_t            i_rdev;
loff_t           i_size;
struct timespec  i_atime;
struct timespec  i_mtime;
struct timespec  i_ctime;
...
};
```

Initialisation and removal of a module

Even though Linux is a monolithic kernel it allows dynamic module loading, i.e., a piece of code can be added to the running kernel. It may be a character, a block, a network driver or a file system.

To initialise a module, the *module_init()* function is used. In this module we register our character device driver. Once the driver is registered, it will reside in the system till we explicitly remove it or the system is restarted/halted. To remove the module, we can call the function *module_exit()*. An example of the *module_init/_exit* function is given below:

```
static int __init my_init(void)
{
    return 0;
}

static void __exit my_exit(void)
{
}

module_init(my_init);
module_exit(my_exit);
```

During the initialisation of a device driver, we typically do resource allocation or acquire the resource, such as device number, memory space, irq, port address, i/o memory map address, semaphore, spinlock, etc.

Allocating and de-allocating a device number

A special type of identification numbering system can identify any device. This is a combination of a *major* number and a *minor* number. The *major* number provides information like the type of device and identifies the device driver associated with it. The *minor* number provides the instance of the device class.

In Linux *dev_t* data type is used to hold the device number. In Linux it is a 32-bit quantity, in which 12 bits are for *major* number representation and the remaining 20 bits are for the *minor* number representation. In the 2.4 kernel, the range of the *major* and *minor* number is between 0-255 only, but in 2.6 it has been increased to

4095 major numbers and more than a million minor numbers. The `dev_t` is defined in

```
<linux/kdev_t.h>
```

A major number can be allocated dynamically or statically. To register the major number statically, we first look at the already allocated device numbers using the command “`$cat /proc/devices`”. The device number that has not been allocated already can be used.

For static device number allocation we use the following command:

```
int register_chrdev_region (dev_t first, unsigned int count,
char *name)
```

Here, the first argument contains the device number stored in the `dev_t` instance, the second argument indicates the total number of contiguous device numbers that are being requested, and the final argument is a device name. On success, it returns zero and on failure it returns a negative value, typically -1. The program for static device allocation based device drivers is shown in Figures 1 and 2.

```
dev_t mydev;
mydev = MKDEV(253, 0);
register_chrdev_region(dev, MAX_DEVS, "pgm");
```

For dynamic device number allocation, we use the following:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
unsigned int count, char *name)
```

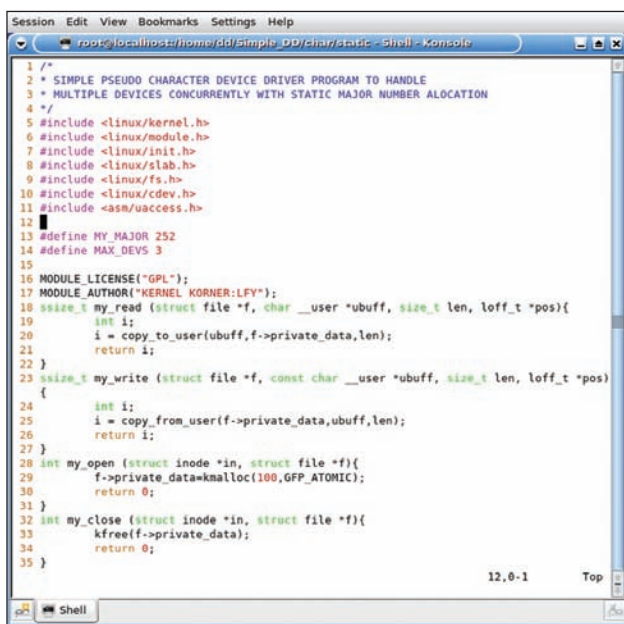


Figure 1: Static device number allocation based device driver program snapshot1

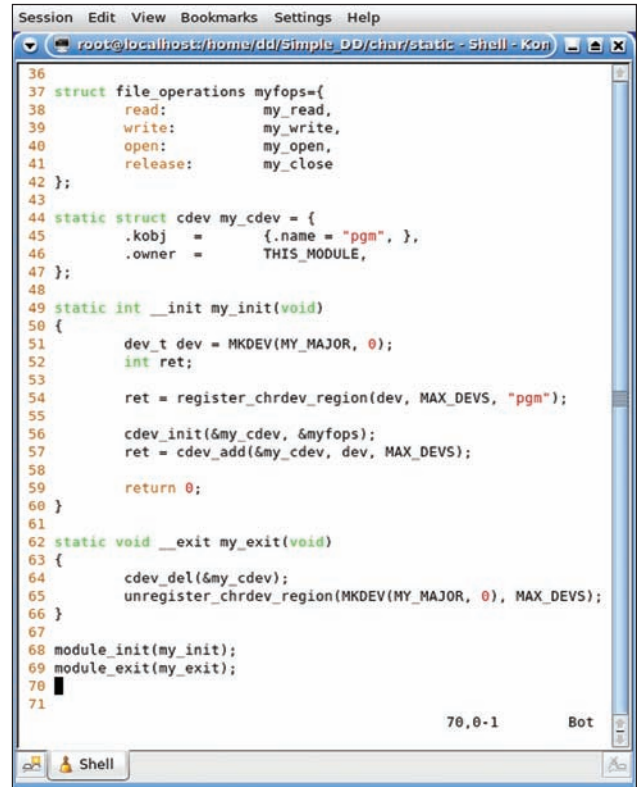


Figure 2: Static device number allocation based device driver program snapshot2

On the successful execution of this, the first argument holds the allocated device number. The second argument is the first minor number that can be initialised to zero. The last two arguments are just as in static allocation.

```
dev_t mydev
alloc_chrdev_region(&mydev, 0, MAX_DEVS, "pgm");
```

The program for a dynamic device number allocation based device driver is as shown in Figures 3 and 4.

For de-allocating the device number, we use `unregister_chrdev_region (dev_t first, unsigned int count)`. The first argument has the major number and the last argument contains the total number of devices.

```
unregister_chrdev_region(mydev, MAX_DEVS);
```

Registering and unregistering the character device

The Linux kernel uses `struct cdev` to represent character devices internally. The member of `struct cdev`, namely `ops`, must point to our file operation structure. To do this, we have two methods. In the first method, we allocate memory for `cdev` at runtime using `cdev_alloc ()` and referring to `cdev->ops=&my_dev`. The second method is by using `cdev_init`, the syntax for which is shown below:

```
void cdev_init (struct cdev *cdev, struct file_operations *fops).
```



```

1 /*
2  * SIMPLE PSEUDO CHARACTER DEVICE DRIVER PROGRAM TO HANDLE MULTIPLE
3  * DEVICES CONCURRENTLY WITH DYNAMIC DEVICE NUMBER ALLOCATION
4  */
5 #include <linux/kernel.h>
6 #include <linux/module.h>
7 #include <linux/init.h>
8 #include <linux/slab.h>
9 #include <linux/fs.h>
10 #include <linux/cdev.h>
11 #include <asm/uaccess.h>
12
13 #define MAX_DEVS 3
14
15 MODULE_LICENSE("GPL");
16 MODULE_AUTHOR("KERNEL KORNER:LFY");
17
18 dev_t mydev;
19 ssize_t my_read(struct file *f, char __user *ubuff, size_t len, loff_t *pos){
20     int i;
21     i = copy_to_user(ubuff,f->private_data,len);
22     return i;
23 }
24 ssize_t my_write(struct file *f, const char __user *ubuff, size_t len, loff_t *pos){
25     int i;
26     i = copy_from_user(f->private_data,ubuff,len);
27     return i;
28 }
29 int my_open(struct inode *in, struct file *f){
30     f->private_data=kmalloc(100,GFP_ATOMIC);
31     return 0;
32 }
33 int my_close(struct inode *in, struct file *f){
34     kfree(f->private_data);
35     return 0;
36 }
37
38 static struct cdev my_cdev = {
39     .kobj = { .name = "pgm", },
40     .owner = THIS_MODULE,
41 };
42
43 static int __init my_init(void)
44 {
45     int ret;
46     ret = alloc_chrdev_region(&mydev,0,MAX_DEVS, "pgm");
47     printk("MAJOR IS %d\n",MAJOR(mydev));
48     cdev_init(&my_cdev, &myfops);
49     ret = cdev_add(&my_cdev, mydev, MAX_DEVS);
50     return 0;
51 }
52
53 static void __exit my_exit(void)
54 {
55     cdev_del(&my_cdev);
56     unregister_chrdev_region(mydev, MAX_DEVS);
57 }
58
59 module_init(my_init);
60 module_exit(my_exit);

```

Figure 3: Dynamic device number allocation based device driver program snapshot1

To use this method we must initialise *cdev* instant statically. The *cdev_init* contains two data structures that are already defined and initialised. The structure *struct cdev *cdev* must be initialised and its owner field has to be set to *THIS_MODULE*.

```

static struct cdev my_cdev = {
    .kobj = { .name = "pgm", },
    .owner = THIS_MODULE,
};

```

After the *cdev* structure initialisation, the kernel should call *cdev_add*.

*int cdev_add(struct cdev *cdev, dev_t num, unsigned int count)*

The first argument is the *cdev* structure instance, the second argument contains the device number, and the final argument has the number of devices. If *cdev_add* returns a negative value, that means the device hasn't been added to the system. The example of code for *cdev_init/add* is given below:

```

cdev_init(&my_cdev, &myfops);
cdev_add(&my_cdev, mydev, MAX_DEVS);

```

For removal of the character device from the system, use the following:

```

void cdev_del(struct cdev *dev) is used. These are defined in
<linux/cdev.h>.

```

System call entry points

An application program has only one entry point, which is

main(). But a device driver may contain many entry points. Each entry point does a particular job on behalf of a system call used in the application program. Here, in our example, we mainly concentrate on file handling entry points such as *open*, *read*, *write* and *release*. It is a device driver developer's responsibility to handle the entry points. These entry points are defined inside the *struct file_operations* structure.

```

struct file_operations myfops = {
    read:      my_read,
    write:     my_write,
    open:      my_open,
    release:   my_close
};

```

The function pointers *my_read*, *my_write*, *my_open*, and *my_release* used in the above declaration are assigned to the corresponding *file_operations* device entry points.

Open: The *open* function contains *inode* structure and *file* structure as its arguments. When successful, it returns the file descriptor number.

```

int my_open(struct inode *in, struct file *f){
    f->private_data=kmalloc(100,GFP_ATOMIC);
    return 0;
}

```

The explanation for memory allocation inside the open entry point is discussed in the later part of this article.

```

37
38 struct file_operations myfops={
39     read:      my_read,
40     write:     my_write,
41     open:      my_open,
42     release:   my_close
43 };
44 static struct cdev my_cdev = {
45     .kobj = { .name = "pgm", },
46     .owner = THIS_MODULE,
47 };
48
49 static int __init my_init(void)
50 {
51     int ret;
52     ret = alloc_chrdev_region(&mydev,0,MAX_DEVS, "pgm");
53     printk("MAJOR IS %d\n",MAJOR(mydev));
54     cdev_init(&my_cdev, &myfops);
55     ret = cdev_add(&my_cdev, mydev, MAX_DEVS);
56     return 0;
57 }
58
59 static void __exit my_exit(void)
60 {
61     cdev_del(&my_cdev);
62     unregister_chrdev_region(mydev, MAX_DEVS);
63 }
64
65 module_init(my_init);
66 module_exit(my_exit);

```

Figure 4: Dynamic device number allocation based device driver program snapshot2

Release: The release function has a similar argument as *open*.

```
int my_close (struct inode *in, struct file *f){
    kfree(f->private_data);
    return 0;
}
```

The explanation for memory freeing is discussed in the later part of this article.

Read: The *my_read* function has mainly four arguments – the first is the pointer to a file structure, the second is a user buffer in which the *read* data is copied, the third argument contains the size of the data to be read, and the final argument refers to the file position. The return value of type *ssize_t* holds the number of bytes read from the file.

```
ssize_t my_read (struct file *f, char __user *ubuff, size_t
len, loff_t *pos){
    int i;
    i = copy_to_user(ubuff, f->private_data, len);
    return i;
}
```

In the read function we copy the data from the kernel space to the user space. For this purpose we use the *copy_to_user* function, which is defined in the *<asm/uaccess.h>*. *copy_to_user(void __user *ubuff, const void *kbuff, unsigned long n)*, in which the first argument contains the destination address that is in user space, the second argument refers to the source address in kernel space and the final argument refers to the number of bytes to copy.

Write: The write function has mainly four arguments – the first is the pointer to a file structure, the second is the user buffer in which the input data that has to be written is present, the third argument contains the size of the data to be written, and the final argument refers to the file position. The return value of type *ssize_t* holds the number of bytes of data written into the file.

```
ssize_t my_write (struct file *f, const char __user *ubuff,
size_t len, loff_t *pos){
    int i;
    i = copy_from_user(f->private_data, ubuff, len);
    return i;
}
```

In the write function we copy the data from user space to the kernel space. For this purpose we make use of the *copy_from_user* function, which is defined below:

<asm/uaccess.h>.

In *copy_from_user(void *kbuff, const void __user*

**ubuff, unsigned long n)*, the first argument refers to the destination address in the kernel space, the second argument refers to the source address in user space and the final argument contains the number of bytes to copy.

Handling multiple devices concurrently

For handling multiple devices, the kernel buffer should be private or local for every device. Hence the *void *private_data* member present in the structure *struct file *fp* is used as the kernel buffer to store data copied from the user space or user application. Here we allocate memory in the open entry point. For memory allocation we use *kmalloc(size_t, gfp_t)*.

The first argument refers to the bytes of memory required, and the second argument (flag) refers to the type of memory to allocate. This may be one of *GFP_USER* (allocates memory on behalf of user, which may sleep), *GFP_KERNEL* (allocates normal kernel memory space, which may sleep) or *GFP_ATOMIC* (allocation will not sleep and is mostly used in interrupt handlers).

In the release, we free the acquired memory. To do that, we make use of *void kfree (const void *)*. The argument is the address of allocated memory. These are defined in *<linux/slab.h>*.

So for every opening of the device, memory allocation is done for that particular instance, as we can see in the device driver open entry point.

Compiling the device driver program

The device driver development and testing is carried out on the latest version of the Linux kernel, i.e., Linux-2.6.19. The system is installed with Fedora Core Linux and the Intel Centrino duo processor. We have downloaded the latest Linux-2.6.19 and rebuilt the kernel.

We booted the system with the new rebuilt kernel. To compile the demo device driver programs we have used a simple *make* file. The content of *Makefile* is shown in Figure 5.

Here, *KDIRS* is the kernel source code directory path and *PWD* is the present working directory in which we can find our device driver program.

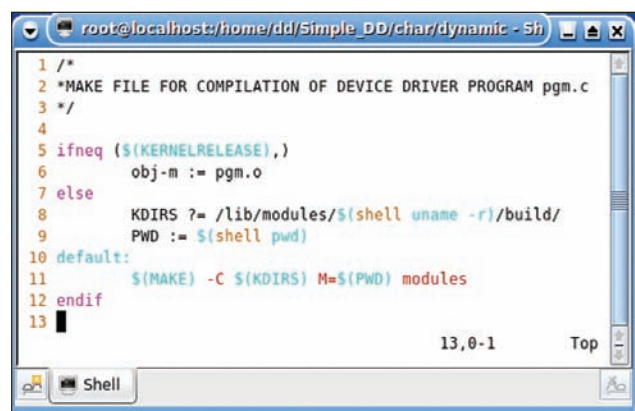


Figure 5: Makefile content

```

Session Edit View Bookmarks Settings Help
root@localhost:/home/dd/Simple_DD/char/dynamic - Shell - Konsole

[root@localhost dynamic]# make
make -C /lib/modules/2.6.19/build/ M=/home/dd/Simple_DD/char/dynamic module
make[1]: Entering directory '/usr/src/linux-2.6.19'
CC [M] /home/dd/Simple_DD/char/dynamic/pgm.o
Building modules, stage 2.
MODPOST 1 modules
CC      /home/dd/Simple_DD/char/dynamic/pgm.mod.o
LD [M]  /home/dd/Simple_DD/char/dynamic/pgm.ko
make[1]: Leaving directory '/usr/src/linux-2.6.19'
[root@localhost dynamic]#

```

Figure 6: Makefile execution result

```

Session Edit View Bookmarks Settings Help
root@localhost:/home/dd/Simple_DD/char/c

1 /*
2 *Application Program to test device driver
3 */
4 #include <fcntl.h>
5 #include <unistd.h>
6 #include <stdio.h>
7
8 int main(void)
9 {
10     int fd1,fd2,fd3;
11     char buff[5];
12
13     //open the pseudo character devices
14     fd1=open("mydev1",O_RDWR);
15     fd2=open("mydev2",O_RDWR);
16     fd3=open("mydev3",O_RDWR);
17
18     //write data into the character devices
19     write(fd1,"fedora\n",7);
20     write(fd2,"core\n",5);
21     write(fd3,"linux\n",6);
22
23     //read data & display into the console
24     read(fd1,buff,7);
25     write(1,buff,7);
26     read(fd2,buff,5);
27     write(1,buff,5);
28     read(fd3,buff,6);
29     write(1,buff,6);
30
31     //close the devices
32     close(fd1);
33     close(fd2);
34     close(fd3);
35     return 0;
36 }

```

Figure 7: Application program

To get the kernel module, the command “*\$make*” is used. The output of the device driver program compilation, i.e., the result of executing makefile, is shown in Figure 6.

To insert a module, “*\$insmod <pgm.ko>*” is used. In the Linux-2.6 kernel *.ko* is the extension for a kernel module. The command “*\$lsmod*” is used to see the inserted module. The command “*\$cat /proc/devices*” provides information about registered devices. For the removal of the module, “*\$rmmod pgm*” is used.

Testing the device driver program

If we are using the dynamic device number allocation, we first need to know the major number allocated. Then we create pseudo character devices with that major number and different minor numbers. For example:

```

$mknod mydev1 c 253 0
$mknod mydev2 c 253 1
$mknod mydev3 c 253 2

```

The number 253 is the major number allocated dynamically by *alloc_chrdev_region*. Here *mydev1*, *mydev2* and *mydev3* are the names of pseudo character devices with minor numbers 0, 1 and 2, respectively, and ‘c’ represents the device class, i.e., the character device family. If we are using the static allocation method, we must know the major number of the device.


The application code is given in Figure 7.

The output of the program gives the following lines:

```

fedora
core
linux

```

Thus, in this article, we studied how to write a simple-character pseudo device driver to handle multiple devices. In the next article we will discuss interrupt handling like sharing IRQ, bottom halves, Soft IRQ, Tasklet and work queues. **END** 

By: Venkatesha Sarpangala and Dr B. Thangaraju, who are working with Talent Transformation, Wipro Technologies, Bangalore. They can be reached at: sarpangala.venkatesha@wipro.com and balat.raju@wipro.com.