BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
K. K. BIRLA Goa Campus, II SEMESTER 2013-2014
Data Storage Technologies & Networks (CS C446/IS C446/CS F446)
Reading Material

By ***Shashank Gupta***

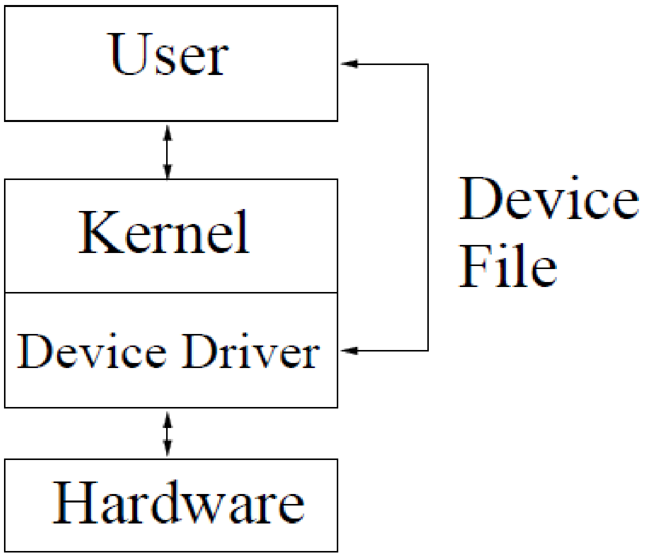
Writing a Linux Character Device Driver

Index ***Page***

Introduction

Device Driver

- Device driver is a computer program that allows higher-level computer programs to interact with a hardware device.
- Device driver communicates with the device through the computer bus or the communication subsystem to which the hardware connects.
- When a calling program invokes a routine in the driver, the driver issues commands to the device.
- Once the device sends data back to the driver, the driver may invoke routines in the original calling program.
- The device driver has three responsibilities:
  - ✓ Talks to the rest of the kernel
  - ✓ Talks to the hardware
  - ✓ Talks to the user



Device File

- The device file is an interface for a device driver that appears in a file system as if it were an ordinary file.

- The device file allows software to interact with a device driver using standard input/output system calls.
- This means that standard system calls like open, read, write, close etc can be used to read and write into the device.
- Every device file is associated with the device driver of that device, which is actually responsible for interacting with the device, on behalf of the user.
- So when you access a device file, the request is forwarded to the respective device driver which does the processing and returns the result.
- Device files can be of two types:
    - ✓ Character special files
    - ✓ Block special files

They differ in the manner in which data written to them and read from them.

Character Devices

- ✓ These are devices through which the system transmits data one character at a time.
- ✓ They are often used for stream communication with devices such as mice, keyboards, virtual terminals and serial modems.
- ✓ They usually do not support random read or write access.
- ✓ The system reads each character from the device immediately or writes each character to the device immediately.

Block Devices

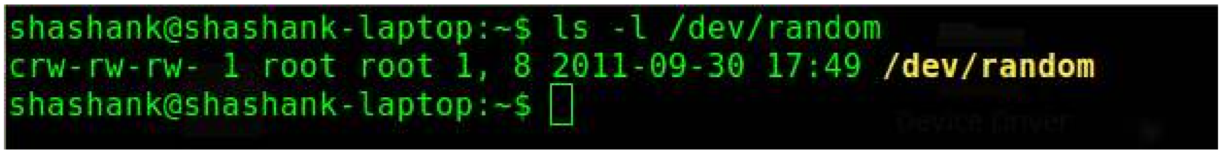
- ✓ These are devices through which the system moves data in the form of blocks.
- ✓ They often represent addressable devices such as hard disks, CD-ROM drives, or memory regions.
- ✓ They support random access and seeking.
- ✓ They use buffered input and output routines.
- ✓ The operating system allocates a data buffer to hold a single block each for input and output.
- ✓ The system stores each character of data in the appropriate buffer. When the buffer fills up, the data transfer takes place and the system clears the buffer.

- ✓ Both character devices and block devices are represented by their respective files in the /dev directory.
- ✓ For example, you could directly read or write the hard disk by accessing /dev/sd* file and RAM by reading /dev/mem. (May cause system damage if used improperly)
- ✓ A device file is a special file. It can't be created by just using cat or gedit or shell redirection.
- ✓ Device Files are created using the mknod system call.

mknod path type major minor

  - ✓ **Path** - Path where the file is to be created. It isn't necessary that device file needs to be created in the /dev directory; It is merely a convention. A device file can be created just about anywhere.
  - ✓ **Type** - 'c' or 'b' to indicate whether the device being represented is a character device or a block device.
  - ✓ **Major, Minor** - The major and minor number of the device.

- Each device and its device file have associated with it, a unique Major number and a Minor number.
- Major number identifies the device driver.
- When a device file is opened, Linux examines its major number and forwards the call to the driver registered for that device.
- Minor number is used to identify the specific device instance if the driver controls more than one device of a type.
- To know the major and minor number of a device, use the **ls – l** command as shown below.



```
shashank@shashank-laptop:~$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 2011-09-30 17:49 /dev/random
shashank@shashank-laptop:~$ 
```

The starting 'c' means it is a character device, 1 is the major number and 8 is the minor number.

Loadable Kernel Modules (.ko)

- It is an object file that contains code to extend the running kernel, or so-called base kernel of an operating system.
- Without loadable kernel modules, an operating system would have to have all possible anticipated functionality already compiled directly into the base kernel.
- Much of that functionality would reside in memory without being used, thus, wasting memory, and would require that users rebuild and reboot the base kernel every time new functionality is desired.
- lsmod is a command on Linux systems which prints the contents of the /proc/modules file. It shows which loadable kernel modules are currently loaded. [or use cat /proc/modules]
- Device drivers can be built either as part of the kernel or separately as loadable modules.
- Modules can be loaded using the insmod command

  ```
  insmod module_name
  ```

  [Where module_name is the object file (.ko) file to be loaded]

- Modules can be unloaded using the rmmod command

  ```
  rmmod module_name
  ```

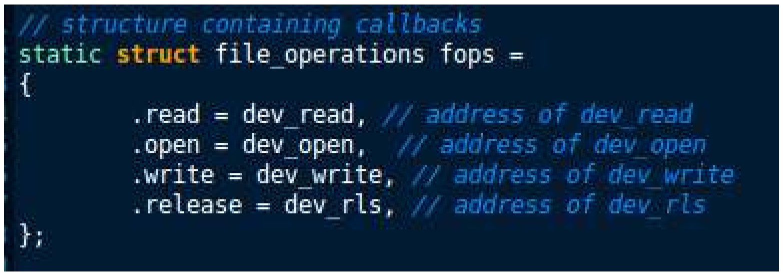
  [Where module_name is the name of the module to be unloaded]

- When the device driver module is loaded, the driver first registers itself as a driver for a particular device specifying a particular Major number.
- It uses the call **register_chrdev** function for registration. The call takes the Major number, Minor number, device name and an address of a structure of the type file_operations [discussed later] as argument.

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

- The register_chrdev call returns a non-negative number on success. If we specify the Major number as 0, the kernel returns a Major number unique at that instant which can be used to create a device file.
- A device file can be created either before the driver is loaded if we know the major and minor number beforehand, or it can be created later after letting the driver specify a major number for us.
- The driver is unregistered by calling the unregister_chrdev function.
- Since the device driver is a kernel module, it should implement init_module and cleanup_module functions. The register_chrdev call is done in the init_module function and unregister_chrdev call is done in the cleanup_module function.
- The driver must also define certain **callback functions** that would be invoked when file operations are done on the device file. Every driver must implement functions for processing the following requests :
    - ✓ Open
    - ✓ Close
    - ✓ Read
    - ✓ Write

- When register_chrdev call is done, the fourth argument is a structure that contains the addresses of these callback functions. The structure is of the type **file_operations** and has 4 main fields that should be set – **read, write, open and release**. Each field must be assigned an address of a function that would be called when open, read, write, close system calls are called respectively. For example



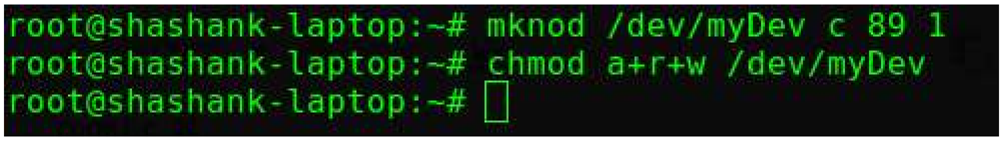
```
// structure containing callbacks
static struct file_operations fops =
{
        .read = dev_read, // address of dev_read
        .open = dev_open,  // address of dev_open
        .write = dev_write, // address of dev_write
        .release = dev_rls, // address of dev_rls
};
```

- These callback functions have a predefined prototype, although they can have any name.

Sample Problem

We would be writing a Linux device driver for a hypothetical character device which reverses any string that is given to it i.e., if we write any string to the device file represented by the device and then read that file, we get the string written earlier in reverse. We will be simulating the functionality of the device in the driver itself.

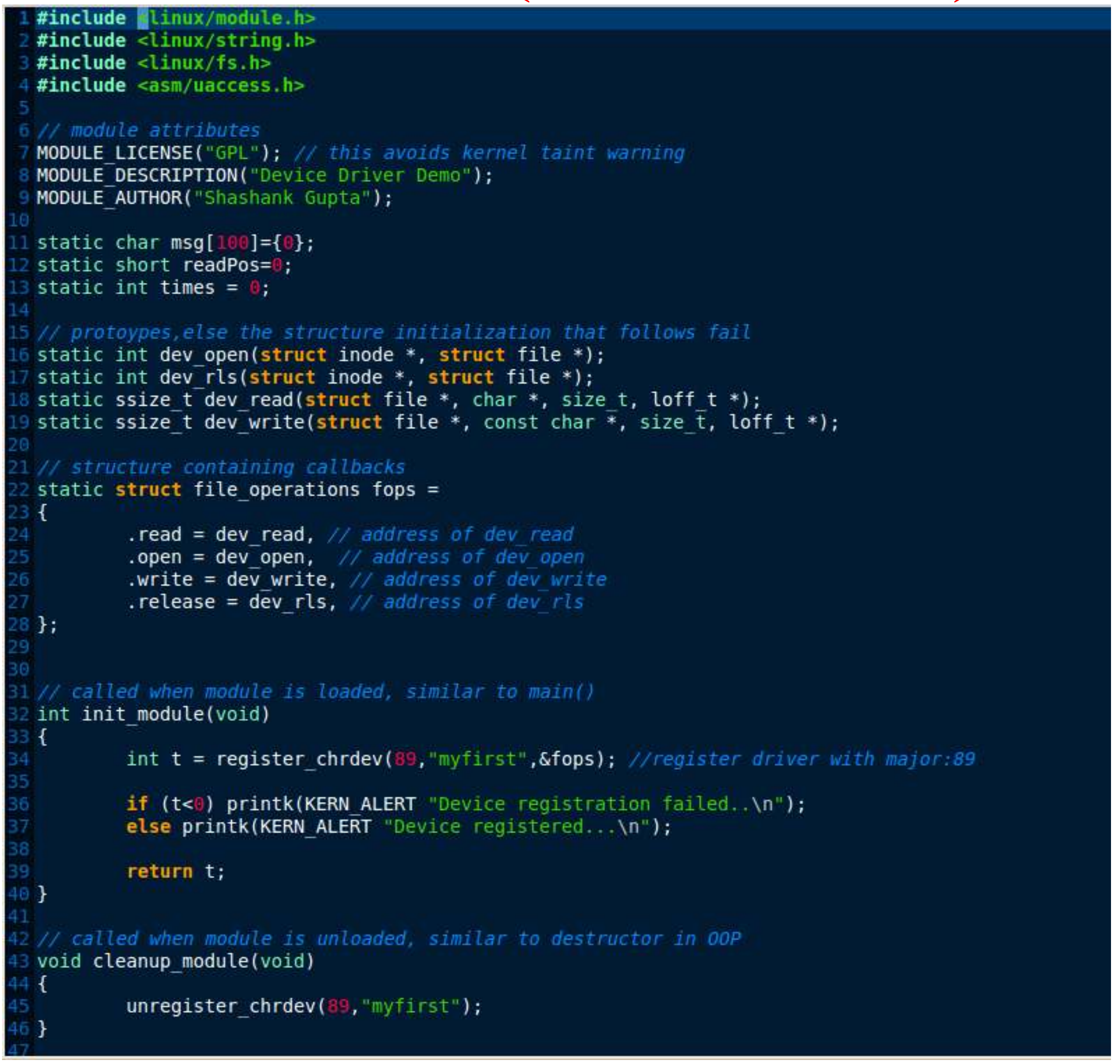
Creating a Device File (Device File name: myDev)



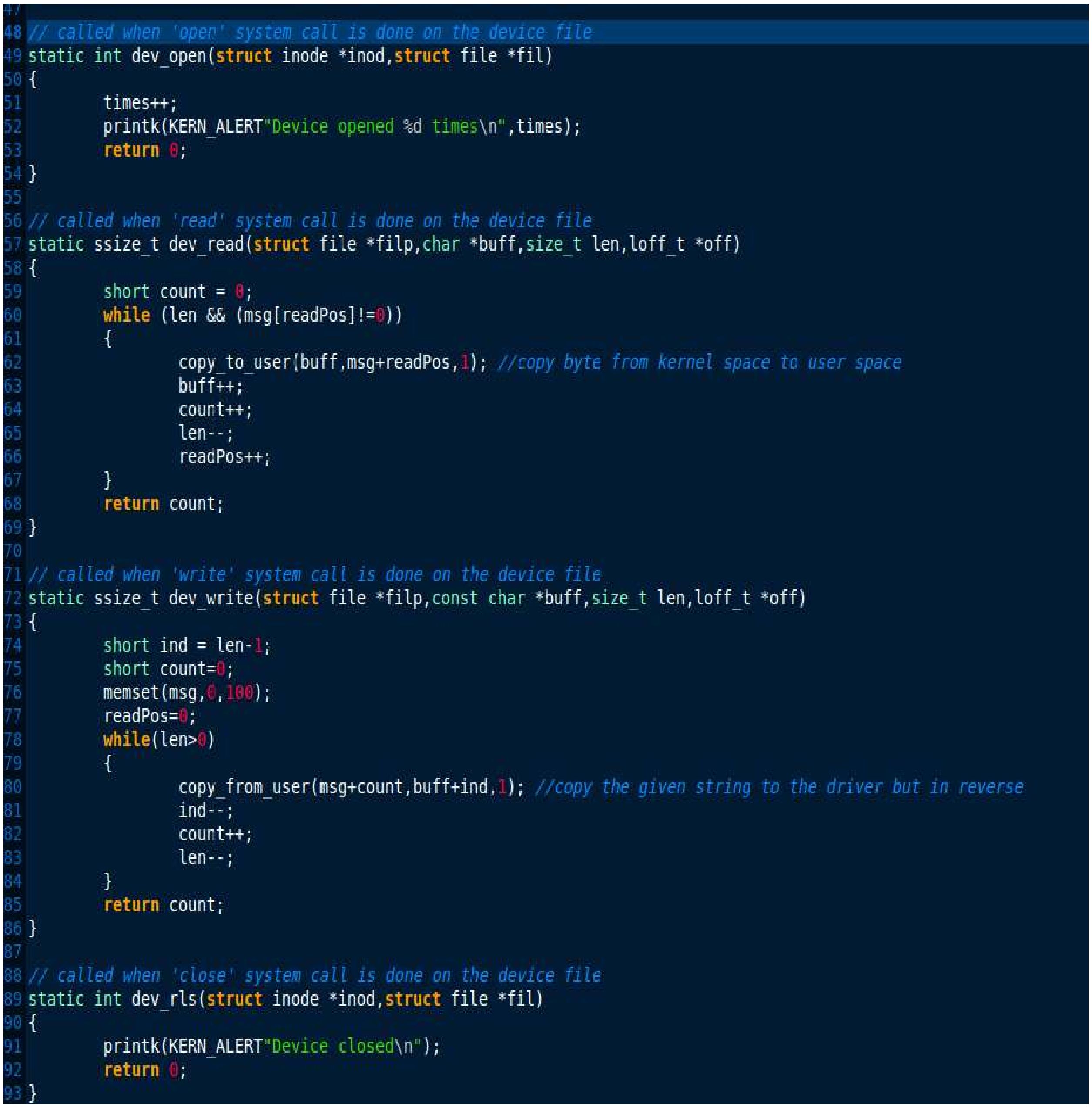
```
root@shashank-laptop:~# mknod /dev/myDev c 89 1
root@shashank-laptop:~# chmod a+r+w /dev/myDev
root@shashank-laptop:~# 
```

If we don't run **chmod**, only processes with root permission can read or write to our device file.

The Driver Code (Filename: Code1.c)



```c
#include <linux/module.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

// module attributes
MODULE_LICENSE("GPL"); // this avoids kernel taint warning
MODULE_DESCRIPTION("Device Driver Demo");
MODULE_AUTHOR("Shashank Gupta");

static char msg[100]={0};
static short readPos=0;
static int times = 0;

// protoypes,else the structure initialization that follows fail
static int dev_open(struct inode *, struct file *);
static int dev_rls(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

// structure containing callbacks
static struct file_operations fops =
{
        .read = dev_read, // address of dev_read
        .open = dev_open,  // address of dev_open
        .write = dev_write, // address of dev_write
        .release = dev_rls, // address of dev_rls
};


// called when module is loaded, similar to main()
int init_module(void)
{
        int t = register_chrdev(89,"myfirst",&fops); //register driver with major:89

        if (t<0) printk(KERN_ALERT "Device registration failed..\n");
        else printk(KERN_ALERT "Device registered...\n");

        return t;
}

// called when module is unloaded, similar to destructor in OOP
void cleanup_module(void)
{
        unregister_chrdev(89,"myfirst");
}
```



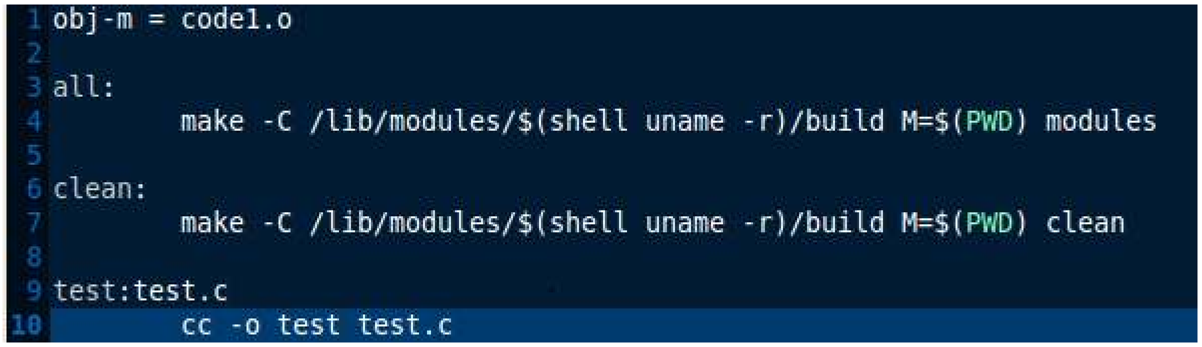
```
// called when 'open' system call is done on the device file
static int dev_open(struct inode *inod,struct file *fil)
{
        times++;
        printk(KERN_ALERT"Device opened %d times\n",times);
        return 0;
}

// called when 'read' system call is done on the device file
static ssize_t dev_read(struct file *filp,char *buff,size_t len,loff_t *off)
{
        short count = 0;
        while (len && (msg[readPos]!=0))
        {
                copy_to_user(buff,msg+readPos,1); //copy byte from kernel space to user space
                buff++;
                count++;
                len--;
                readPos++;
        }
        return count;
}

// called when 'write' system call is done on the device file
static ssize_t dev_write(struct file *filp,const char *buff,size_t len,loff_t *off)
{
        short ind = len-1;
        short count=0;
        memset(msg,0,100);
        readPos=0;
        while(len>0)
        {
                copy_from_user(msg+count,buff+ind,1); //copy the given string to the driver but in reverse
                ind--;
                count++;
                len--;
        }
        return count;
}

// called when 'close' system call is done on the device file
static int dev_rls(struct inode *inod,struct file *fil)
{
        printk(KERN_ALERT"Device closed\n");
        return 0;
}
```

- For debugging purposes, to see the printk messages while the driver is in action, do dmesg|tail

Name of the device is myfirst

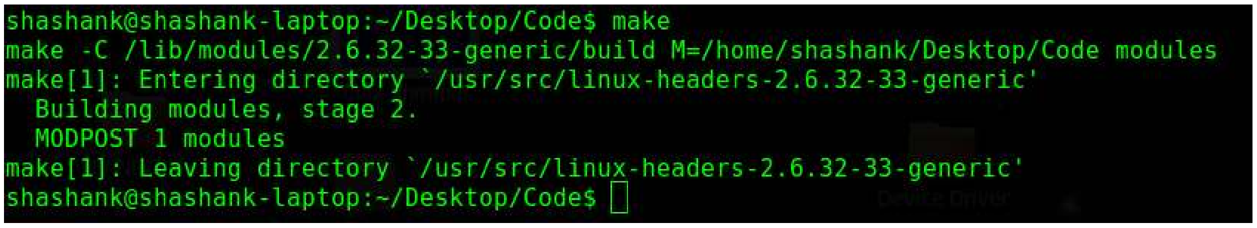
Compiling the Driver

- A Linux module cannot simply be compiled like an ordinary C file.
- Compiling a Linux module is a separate process of its own. We use the help of kernel Makefile for compilation.
- The makefile we'll be using is as follows:



```
obj-m = code1.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

test:test.c
	cc -o test test.c
```

- Here, we are making use of the **kbuild** mechanism used for compiling the kernel.
- Use make command now.

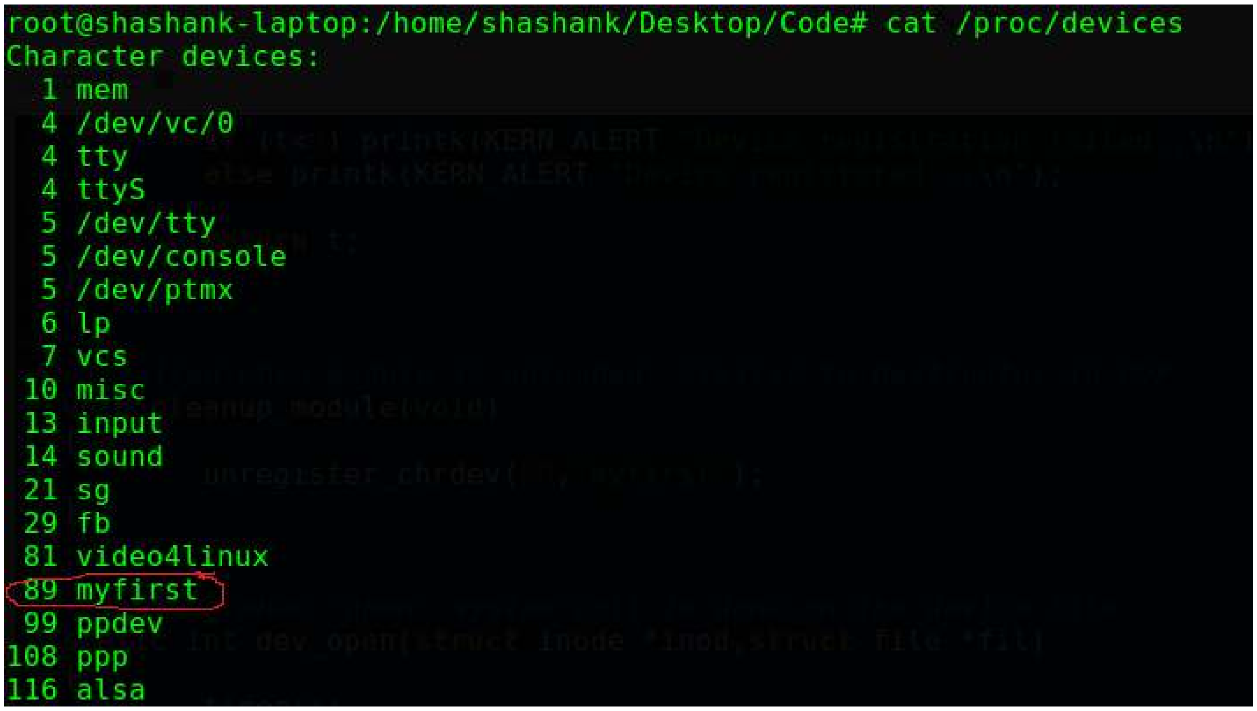


```
shashank@shashank-laptop:~/Desktop/Code$ make
make -C /lib/modules/2.6.32-33-generic/build M=/home/shashank/Desktop/Code modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.32-33-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-2.6.32-33-generic'
shashank@shashank-laptop:~/Desktop/Code$ 
```

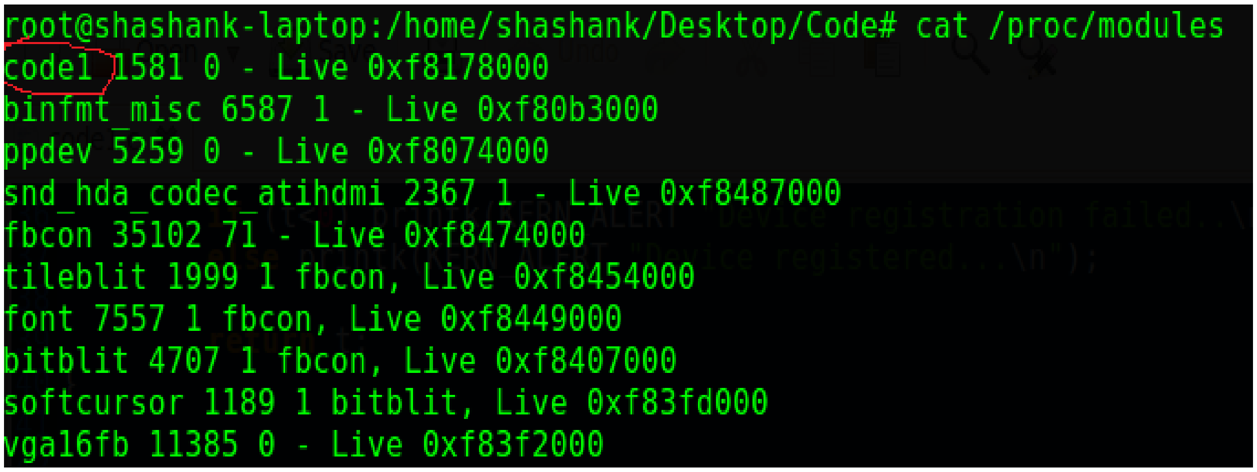
- The result of the compilation is a **ko** file (kernel object) which can then be loaded dynamically when required.

Loading and Unloading the Driver

- Once the compilation is complete, we can use the **insmod** command to load the module. (**insmod code1.ko**)
- To test if the driver has been loaded successfully, do **cat /proc/modules** and **cat /proc/devices**. We should see our module name in the first case, and the device name in the second.



```
root@shashank-laptop:/home/shashank/Desktop/Code# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 81 video4linux
 89 myfirst
 99 ppdev
108 ppp
116 alsa
```

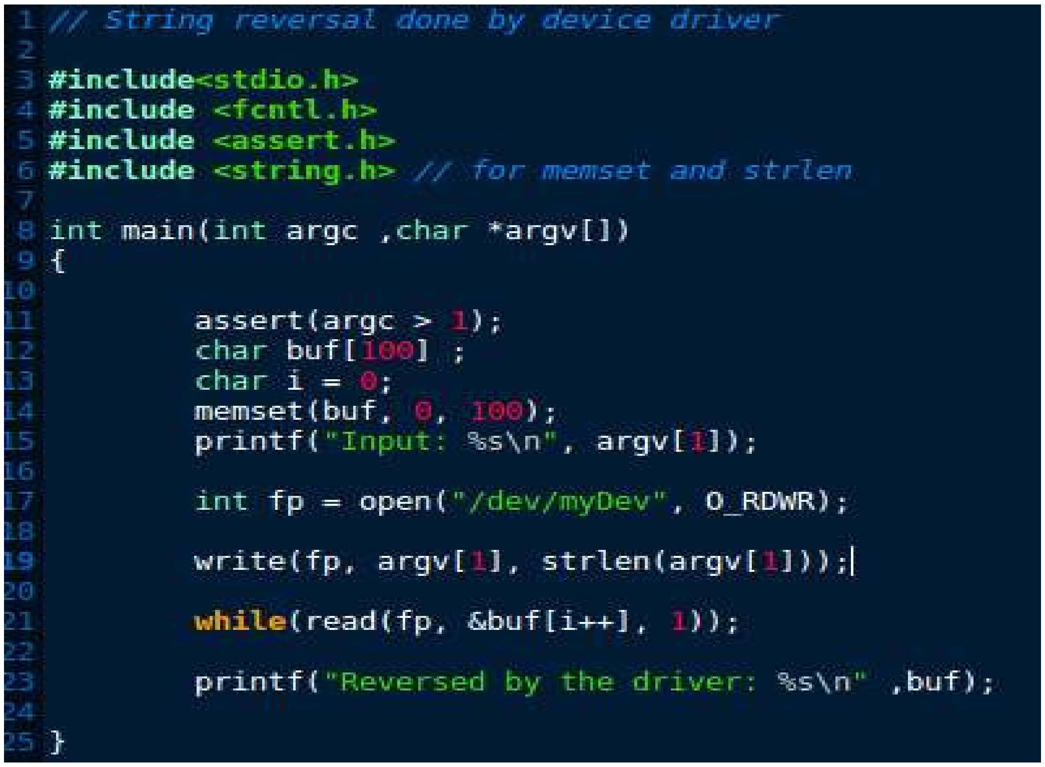


```
root@shashank-laptop:/home/shashank/Desktop/Code# cat /proc/modules
code1 1581 0 - Live 0xf8178000
binfmt_misc 6587 1 - Live 0xf80b3000
ppdev 5259 0 - Live 0xf8074000
snd_hda_codec_atihdmi 2367 1 - Live 0xf8487000
fbcon 35102 71 - Live 0xf8474000
tileblit 1999 1 fbcon, Live 0xf8454000
font 7557 1 fbcon, Live 0xf8449000
bitblit 4707 1 fbcon, Live 0xf8407000
softcursor 1189 1 bitblit, Live 0xf83fd000
vga16fb 11385 0 - Live 0xf83f2000
```

- To unload the driver, use **rmmod** command. (**rmmod code1**)

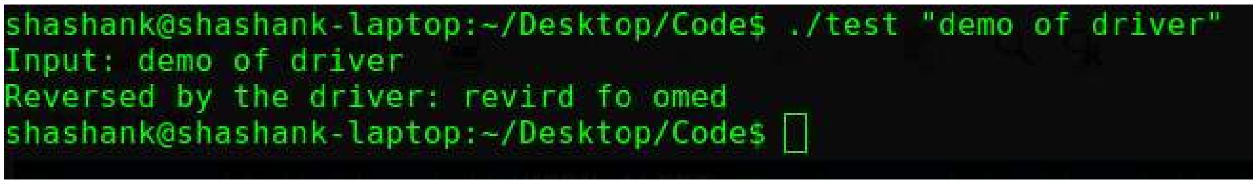
Testing the driver

- To test the driver, we try writing something to the device file and then reading it.



```c
// String reversal done by device driver

#include<stdio.h>
#include <fcntl.h>
#include <assert.h>
#include <string.h> // for memset and strlen

int main(int argc ,char *argv[])
{

        assert(argc > 1);
        char buf[100] ;
        char i = 0;
        memset(buf, 0, 100);
        printf("Input: %s\n", argv[1]);

        int fp = open("/dev/myDev", O_RDWR);

        write(fp, argv[1], strlen(argv[1]));

        while(read(fp, &buf[i++], 1));

        printf("Reversed by the driver: %s\n" ,buf);

}
```

- Compile it normally and run **./a.out some_string** and see the output.



```
shashank@shashank-laptop:~/Desktop/Code$ ./test "demo of driver"
Input: demo of driver
Reversed by the driver: revird fo omed
shashank@shashank-laptop:~/Desktop/Code$ 
```

Note: You need to be root to compile, load and unload the module.