In this post we'll take our 1st step of writing a char device driver for Linux. In our previous post, which I would recommend you to read unless you already did, we created a very simple kernel module. Other than dumping some log message, it didn't do anything fancy. In this post we'll create a char device driver which creates a device file in */dev* directory. We'll also show how a user space program can read from or write to this device file. And finally we'll modify it to reverse a string. We'll name our device driver as "***reversedev***". I also assume, by now you know how to compile and install a kernel module and how to test kernel log using *dmesg* command. If you don't you definitely need to read this.

We'll go step by step and will make sure at each step our code compiles and works as a functional kernel module. So let start with following simple kernel module code. We name our source file as "***reverse_dev.c***".

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

#define DEVICE_NAME    "reversedev"

static int reverse_init(void){
        printk("Reverse Device Initializationn");

        return 0;
}

static void reverse_exit(void){
        printk("Reverse Device Exitn");
}


module_init(reverse_init);
module_exit(reverse_exit);


MODULE_LICENSE("GPL");
MODULE_AUTHOR("IsonProjects");
MODULE_DESCRIPTION("Virtual Device For String Reversal");
```

The above code can work as a standalone kernel module. But to make it work as a char device, we've to provide implementation for *open*, *close*, *read* and *write* functions.

Otherwise no userspace application can access our device as a file. We use the *file_operations* structure which works as a placeholder for these functions.

The *file_operations* structure is a collection of function pointers and defined in <*linux/fs.h*>. The following code block shows our function implementation and how they are assigned to *file_operations* structure. Add following code block just after the header files of the above code.

```
static int reverse_open(struct inode*, struct file*);
static ssize_t reverse_read(struct file*, char*, size_t, loff_t*);
static ssize_t reverse_write(struct file*, const char*, size_t,
loff_t*);
static int reverse_release(struct inode*, struct file*);


static struct file_operations reverseops =
{
        .open = reverse_open,
        .read = reverse_read,
        .write = reverse_write,
        .release = reverse_release,
};


static int reverse_open(struct inode* inodep, struct file* filep){
        printk("reverse_dev openedn");
        return 0;
}

static int reverse_release(struct inode* inodep, struct file* filep){
        printk("reverse_dev releasedn");
        return 0;
}

static ssize_t reverse_read(struct file* filep, char *data_buffer,
size_t len, loff_t *offset){
        printk("reverse_dev read startn");
        return 0;
```

```
}

static ssize_t reverse_write(struct file* filep, const char
*data_buffer, size_t len, loff_t *offset){
        printk("reverse_dev write startn");
        return len;
}
```

The next step is to register our char device driver with the system. We do it using *register_chrdev* API. This API can dynamically allocate a *major* number for our driver and return it if the registration process is successful, otherwise return a negative number. Registration process should start during driver initialization. Also an allocated *major* should be released using *unregister_chrdev* API during driver exit. So replace the *reverse_init* and *reverse_exit* function with the following code.

```
static int reverse_init(void){
        printk("reverse_dev initializationn");

        // dynamically allocate a major number and assing
file_operations
        majorNumber = register_chrdev(0, DEVICE_NAME, &reverseops);
// 0 means dynamic allocation
        if (majorNumber < 0){
                printk("Device Registration Failedn");
                goto register_unsuccessful;
        }
        else{
                printk("Device Registered With Major Number %dn",
majorNumber);
        }

        return 0;

register_unsuccessful:
        return majorNumber;
}
```

```
static void reverse_exit(void){
        printk("Reverse Device Exitn");
        unregister_chrdev(majorNumber, DEVICE_NAME);
}
```

In the above code portion, we're asking the system to dynamically allocate a *major* number for our device. We assign the returned number to a global variable. Add the global variable at the beginning of the source file just after the header files.

```
static int majorNumber;
```

All the codes we've written so far assigns a *major* for our device and assign some file operation to that major. But it didn't create any device file in the */dev* directory. For this tutorial we'll create the device file manually and on our next tutorial we'll learn how to create the device file dynamically.

Compile and install our char driver and note the *major* number from the output of *dmesg* command. We'll need this *major* for creating device file entry. The device file is created using the *mknod* command. This command needs the assigned *major* number for our device. In my system, the *major* number assigned to me was 250. So I used the following command to create the device file.

```
sudo mknod /dev/reversedev c 250 0
```

Now our char device driver is ready to test. We can use some user mode application for testing the file operations on our driver. I'll use the *cat* command for reading from the device and *echo* command for writing to the device.

Read from Char Device Test:

Open a terminal window and run

cat /dev/reversedev

command. If everything is okay you should be able to see following text in *dmesg* output.

```
reverse_dev opened
reverse_dev read start
```

```
reverse_dev released
```

Write to Char Device Test:

Open a terminal window and run

sudo echo "abc" /dev/reversedev

command. If everything is okay you should be able to see following text in *dmesg* output.

```
reverse_dev opened
reverse_dev write start
reverse_dev released
```

In both cases compare the *dmesg* output with the *printk* function calls used inside the functions used for *file_operations* structure to get an idea which function gets called at which time.

This tutorial only demonstrated how to create a char device driver. We've installed the driver, create device file for our driver and showed how a userspace program can *read* from or *write* to the device file. But so far our driver doesn't do anything else other than printing some messages in the kernel log. In my next tutorial I'll show how to modify the *read* and *write* function to do something useful.