This is the second part of our char device driver development tutorial. In part1, we learnt how to write a simple char device driver that supports *read* or *write* functions. We dynamically assigned a *major* number for our driver and using the assigned *major*, we manually created a device file in */dev* directory.

In this tutorial we'll show how to create the */dev* file automatically. We'll also show how a user space program can communicate our char device driver using the *read* and *write* functions. Finally we'll be able to use our driver as a virtual device which reverse an input string.

We'll use the same code we developed in our previous first post of writing char device driver and will incrementally update it. Let's start by modifying the driver *init* function.

```
static int reverse_init(void){
        int ret = 0;
        printk("reverse_dev initializationn");

        majorNumber = register_chrdev(0, DEVICE_NAME, &reverseops);
        if (majorNumber < 0){
                printk("Device Registration Failedn");
                ret = majorNumber;
                goto register_unsuccessful;
        }
        else{
                printk("Device Registered With Major Number %dn",
majorNumber);
        }

        // register device class
        reversedevClass = class_create(THIS_MODULE, DEVICE_CLASS);
        if (IS_ERR(reversedevClass)){                // error: we've
to clean up allocated resource
                printk("Device Class Registration Unsuccessfuln");
                ret = PTR_ERR(reversedevClass);
                goto register_class_error;
        }

        // register the device, this will create the /dev file
        reverseDevice = device_create(reversedevClass,
NULL,
MKDEV(majorNumber, 0),
NULL,
DEVICE_NAME);

        if (IS_ERR(reverseDevice)){
                printk("device_create unsuccessfuln");
                ret = PTR_ERR(reverseDevice);;
                goto device_create_unsuccessful;
        }

        return ret;


device_create_unsuccessful:
        class_destroy(reversedevClass);

register_class_error:
        unregister_chrdev(majorNumber, DEVICE_NAME);

register_unsuccessful:
        return ret;
}
```

In line 15, we are creating a device class using the kernel API

struct class *class_create(struct module *owner, const
char* name)

. We pass this device class to the device_create API which will create the device and register it in file system. It'll create our device file in */dev* directory. It is important to check for error in these functions and release any allocated resource in case of error.

We also need to modify the *exit* function which will deallocate all allocated resource. It is very important as otherwise there'll be irregular behavior.

```
static void reverse_exit(void){
        printk("Reverse Device Exitn");
        device_destroy(reversedevClass, MKDEV(majorNumber, 0));
        class_unregister(reversedevClass);
        class_destroy(reversedevClass);
        unregister_chrdev(majorNumber, DEVICE_NAME);
}
```

We've to add the required headers at the beginning. The new header we need is *linux/devices.h*. Also don't forget to declare the DEVICE_NAME and DEVICE_CLASS constant and global structure variables.

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

#define DEVICE_NAME            "reversedev"
#define DEVICE_CLASS           "IsonProjects Char Device"


static int                                 majorNumber;
struct class*                              reversedevClass;
struct device*                              reverseDevice;
```

Now compile the driver using *Makefile* and install it using *insmod* command. A new entry */dev/reversedev* will be created on successful installation. Try

sudo cat /dev/reversedev

command and in the *dmesg* output you'll see the *read* function of our driver is being called.

Next we'll do the actual computation part which in this case is our *read* and *write* file operations. In our implementation we didn't actually use any real hardware. What we've is kind of virtual hardware. We mimic the behavior of a device which will reverse a string input. We pass data to the device using the *write* operation and get back the result using the *read* operation. We start by adding a buffer in our code. We'll also add some header file as we'll use some functions from them.

```
#include <linux/slab.h>
#include <asm/uaccess.h>
```

```
#define BUFFER_LEN      100


char    device_buffer[BUFFER_LEN];
```

This buffer is analogous to the buffer in a real hardware. If we had a real hardware, we would have sent the data to it instead of a local buffer. We modify our *write* file operation to store the written data in this buffer in reverse order.

```
static ssize_t reverse_write(struct file* filep, const char
*data_buffer, size_t len, loff_t *offset){
        int i = 0;
        int ret = 0;

        printk("reverse_dev write startn");

        if (len < BUFFER_LEN){
                char* temp_buffer = (char*)kmalloc(len, GFP_KERNEL);
                copy_from_user(temp_buffer, data_buffer, (int)len);

                // reverse the data and save to device_buffer
                for (i = 0; i<len; i++){
                        device_buffer[i] = temp_buffer[len - 1 - i];
                }
                kfree(temp_buffer);
                ret = len;
        }
        else{
                strcpy(device_buffer, "Error: Insufficient Spacen");
        }

        return ret;
}
```

During *read* operation, we just pass the data from our device buffer to the reader. Look at the code.

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

        if (len < BUFFER_LEN){
                copy_to_user(data_buffer, device_buffer, len);
                ret = len;
        }
        else{
                copy_to_user(data_buffer, device_buffer, BUFFER_LEN);
                ret = BUFFER_LEN;
        }

        return ret;
}
```

Both *read* and *write* function has an *offset* parameter. For this version of the code we don't need to manipulate this offset as we are reading from or writing to a local array starting from 0 index. But for a real device, where data buffer is limited, we might need to manipulate the offset accordingly so the hardware knows from which position to start read/write.

Another important thing is the use of *copy_from_user* and *copy_to_user* functions. Note that whenever we are copying some data from user buffer to kernel buffer or from kernel buffer to user buffer, we've to use this function. Using simply *strcpy* or *strncpy* won't work in that case. Use of these functions are mandatory for security purpose as they ensure that an user space program can't access a kernel space memory. Also if the addresses are inaccessible, these function will return an error instead of crashing the kernel.

So now our string reversal device driver is ready. We can install and start testing. To test open two command line. One terminal will write to the device and another will read from it. At first, from 1st terminal we initiate a write request using the

```
echo "abc" &gt; /dev/reversedev
```

command. Here we are asking our driver to write 3 bytes to our virtual device.

Next, from the 2nd terminal, read 3 bytes from our virtual device using

```
head -c 3 /dev/reversedev
```

command. If you’ve done everything correctly, this command should print “*cba*” in the terminal window which is the reverse of what we’ve written.

If the above commands throw any “**Permission Denied”** issue, then you might have to change the permission of the */dev/reversedev* file. You can use either *chmod* or *chown* command for doing that. I’ll not discuss details of that in this tutorial.

We’re done with writing our first char device driver. I hope it’ll help you to get some basic idea of device driver development.