Linux Device Driver Tutorial Part 7 – Linux Device Driver Tutorial Programming

We already know that in Linux everything is a File. So in this tutorial we are going to develop two applications.

1. User Space application (User program)
2. Kernel Space program (Driver)

User Program will communicate with the kernel space program using device file. Lets Start.

Kernel Space Program (Device Driver)

We already know about major, minor number, device file and file operations of device driver. If you don't know please visit our previous tutorials. Now we are going to discuss more about file operations in device driver. Basically there are four functions in device driver.

Open driver
Write Driver
Read Driver
Close Driver

Now we will see one by one of this functions. Before that i will explain the concept of this driver.

Concept

Using this driver we can send string or data to the kernel device driver using write function. It will store those string in kernel space. Then when i read the device file, it will send the data which is written by write by function.

Functions used in this driver

- **kmalloc()**
- **kfree()**
- **copy_from_user()**
- **copy_to_user()**
-

kmalloc()

We will see the memory allocation methods available in kernel, in future tutorial. But now we will use only kmalloc in this tutorial.

kmalloc function is used to allocate the memory in kernel space. This is like a malloc() function in user space. The function is fast (unless it blocks) and doesn't

clear the memory it obtains. The allocated region still holds its previous content. The allocated region is also contiguous in physical memory.

```
#include <linux/slab.h>
void *kmalloc(size_t size, gfp_t flags);
```

Arguments
size_t size– how many bytes of memory are required.
gfp_t flags– the type of memory to allocate.
The *flags* argument may be one of:
GFP_USER – Allocate memory on behalf of user. May sleep.
GFP_KERNEL – Allocate normal kernel ram. May sleep.
GFP_ATOMIC – Allocation will not sleep. May use emergency pools. For example, use this inside interrupt handlers.
GFP_HIGHUSER – Allocate pages from high memory.
GFP_NOIO – Do not do any I/O at all while trying to get memory.
GFP_NOFS – Do not make any fs calls while trying to get memory.
GFP_NOWAIT – Allocation will not sleep.
__GFP_THISNODE – Allocate node-local memory only.
GFP_DMA – Allocation suitable for DMA. Should only be used for kmalloc caches. Otherwise, use a slab created with SLAB_DMA.
Also it is possible to set different flags by OR'ing in one or more of the following additional *flags*:
__GFP_COLD – Request cache-cold pages instead of trying to return cache-warm pages.
__GFP_HIGH – This allocation has high priority and may use emergency pools.
__GFP_NOFAIL – Indicate that this allocation is in no way allowed to fail (think twice before using).
__GFP_NORETRY – If memory is not immediately available, then give up at once.
__GFP_NOWARN – If allocation fails, don't issue any warnings.
__GFP_REPEAT – If allocation fails initially, try once more before failing.
There are other flags available as well, but these are not intended for general use, and so are not documented here. For a full list of potential flags, always refer to linux/gfp.h.

kfree()

This is like a free() function in user space. This is used to free the previously allocated memory.

```
void kfree(const void *objp)
```

Arguments
*objp – pointer returned by kmalloc

copy_from_user()

This function is used to Copy a block of data from user space (Copy data from user space to kernel space).

```
unsigned long copy_from_user(void *to, const void
__user *from, unsigned long  n);
```

Arguments

`to` – Destination address, in kernel space
`from` – Source address in user space
`n` – Number of bytes to copy
Returns number of bytes that could not be copied. On success, this will be zero.

copy_to_user()

This function is used to Copy a block of data into user space (Copy data from kernel space to user space).

```
unsigned long copy_to_user(const void __user *to, const
void *from, unsigned long  n);
```

Arguments

`to` – Destination address, in user space
`from` – Source address in kernel space
`n` – Number of bytes to copy
Returns number of bytes that could not be copied. On success, this will be zero.

Open()

This function is called first, whenever we are opening the device file. In this function i am going to allocate the memory using `kmalloc`. In user space application you can use open() system call to open the device file.

```
static int etx_open(struct inode *inode, struct file *file)
{
    /*Creating Physical memory*/
    if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
       printk(KERN_INFO "Cannot allocate memory in kernel\n");
       return -1;
    }
    printk(KERN_INFO "Device File Opened...!!!\n");
```

```
    return 0;
}
```

write()

When write the data to the device file it will call this write function. Here i will copy the data from user space to kernel space using copy_from_user() function. In user space application you can use write() system call to write any data the device file.

```
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len,
loff_t *off)
{
    copy_from_user(kernel_buffer, buf, len);
    printk(KERN_INFO "Data Write : Done!\n");
    return len;
}
```

read()

When we read the device file it will call this function. In this function i used copy_to_user(). This function is used to copy the data to user space application. In user space application you can use read() system call to read the data from the device file.

```
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
    copy_to_user(buf, kernel_buffer, mem_size);
    printk(KERN_INFO "Data Read : Done!\n");
    return mem_size;
}
```

close()

When we close the device file that will call this function. Here i will free the memory that is allocated by kmalloc using kfree(). In user space application you can use close() system call to close the device file

```
static int etx_release(struct inode *inode, struct file *file)
{
    kfree(kernel_buffer);
    printk(KERN_INFO "Device File Closed...!!!\n");
    return 0;
```

```
        }
```

Full Driver Code

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()


#define mem_size        1024

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
uint8_t *kernel_buffer;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);
static int etx_open(struct inode *inode, struct file *file);
static int etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .release        = etx_release,
};

static int etx_open(struct inode *inode, struct file *file)
{
        /*Creating Physical memory*/
        if((kernel_buffer = kmalloc(mem_size , GFP_KERNEL)) == 0){
            printk(KERN_INFO "Cannot allocate memory in kernel\n");
            return -1;
        }
        printk(KERN_INFO "Device File Opened...!!!\n");
```

```
        return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
        kfree(kernel_buffer);
        printk(KERN_INFO "Device File Closed...!!!\n");
        return 0;
}

static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        copy_to_user(buf, kernel_buffer, mem_size);
        printk(KERN_INFO "Data Read : Done!\n");
        return mem_size;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        copy_from_user(kernel_buffer, buf, len);
        printk(KERN_INFO "Data Write : Done!\n");
        return len;
}

static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                printk(KERN_INFO "Cannot allocate major number\n");
                return -1;
        }
        printk(KERN_INFO "Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            printk(KERN_INFO "Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
            printk(KERN_INFO "Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
```

```
        if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
            printk(KERN_INFO "Cannot create the Device 1\n");
            goto r_device;
        }
        printk(KERN_INFO "Device Driver Insert...Done!!!\n");
    return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
    printk(KERN_INFO "Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com or admin@embetronicx.com>");
MODULE_DESCRIPTION("A simple device driver");
MODULE_VERSION("1.4");
```

Building the Device Driver

1. Build the driver by using Makefile (*sudo make*) You can download the Makefile Here.

User Space Application

This application will communicate with the device driver. You can download the all codes (driver, Makefile and application) Here.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int8_t write_buf[1024];
int8_t read_buf[1024];
int main()
{
	int fd;
	char option;
	printf("*********************************\n");
	printf("*******WWW.EmbeTronicX.com*******\n");

	fd = open("/dev/etx_device", O_RDWR);
	if(fd < 0) {
		printf("Cannot open device file...\n");
		return 0;
	}

	while(1) {
		printf("****Please Enter the Option******\n");
		printf("	1. Write		\n");
		printf("	2. Read		\n");
		printf("	3. Exit		\n");
		printf("*********************************\n");
		scanf(" %c", &option);
		printf("Your Option = %c\n", option);

		switch(option) {
			case '1':
				printf("Enter the string to write into driver :");
				scanf("  %[^\t\n]s", write_buf);
				printf("Data Writing ...");
				write(fd, write_buf, strlen(write_buf)+1);
				printf("Done!\n");
				break;
			case '2':
				printf("Data Reading ...");
				read(fd, read_buf, 1024);
				printf("Done!\n\n");
				printf("Data = %s\n\n", read_buf);
				break;
			case '3':
				close(fd);
```

```
                exit(1);
                break;
            default:
                printf("Enter Valid option = %c\n",option);
                break;
        }
    }
    close(fd);
}
```

Compile the User Space Application

Use below line in terminal to compile the user space application.

```
gcc -o test_app test_app.c
```

Execution (Output)

As of now, we have driver.ko and test_app. Now we will see the output.

- Load the driver using `sudo insmod driver.ko`
- Run the application (`sudo ./test_app`)

```
*********************************
*******WWW.EmbeTronicX.com*******
****Please Enter the Option******
1. Write
2. Read
3. Exit
*********************************
```

- Select option 1 to write data to driver and write the string (In this case i'm going to write "embetronicx" to driver.

```
1
Your Option = 1
Enter the string to write into driver :embetronicx
Data Writing ...Done!
****Please Enter the Option******
1. Write
2. Read
3. Exit
*********************************
```

- That "embetronicx" string got passed to the driver. And driver stored that string in the kernel space. That kernel space was allocated by `kmalloc`.
- Now select the option 2 to read the data from the device driver.

```
2
Your Option = 2
Data Reading ...Done!
```

```
Data = embetronicx
```

- See now, we got that string “embetronicx”.

Just see the below image for your clarification.