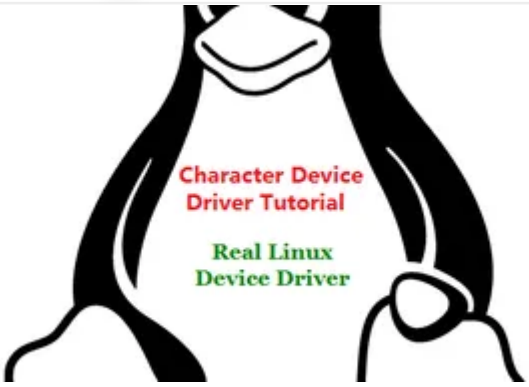
EmbeTronicX
Embedded Tutorials Zone

Sidebar

Device Drivers



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

9

This article is a continuation of the Series on Linux Device Driver and carries on the discussion on character drivers and their implementation. This is Part 7 of the Linux device driver tutorial. In this tutorial, we will discuss the Linux Device Driver Tutorial Programming.

Apple Music Turns 5 as It Continues Rivalry With Spotify

From our previous tutorials, we know about major, minor numbers, device file, and file operations of device drivers using the dummy drivers. But today we are going to write a real driver without hardware.

**Post Contents** [hide]

## Linux Device Driver Tutorial Programming Introduction

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)

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

## Kernel Space Program (Device Driver)

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

1. Open driver
2. Write Driver
3. Read Driver
4. 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 our next tutorials. 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 userspace. 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 the userspace. 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 the kernel space

`from` – The source address in the 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 userspace (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 the user space

`from` – The source address in the 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.

```c
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 writing 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.

```c
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_
{
        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 the userspace application. In userspace application, you can use **read()** system call to read the data from the device file.

```c
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

You can download the all codes Here.

```
#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 * of
static ssize_t etx_write(struct file *filp, const char *buf, size_t len, loff_t * o

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 *of
{
        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, lof
{
        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;
```

```c
}

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.

```c
#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 the below line in the 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.

***Note:*** *Instead of using user space application, you can use* `echo`*and* `cat`*command. But one condition. If you are going to use* `echo`*and* `cat`*command, please allocate the kernel space memory in init function instead of open() function. I won't say why. You have to find the reason. If you found the reason please comment below. You can use* `dmesg`*it to see the kernel log.* 😆

In our next tutorial, we will discuss IOCTL in the Linux device driver.

5

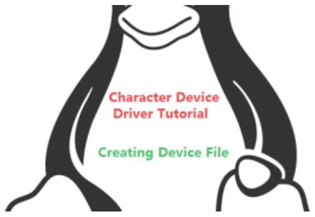
Article Rating

★★★★★

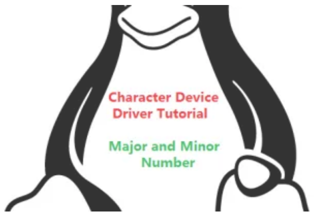
**Like this:**

Loading...

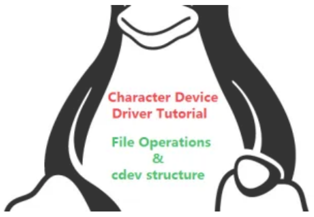
**Related**



Linux Device Driver Tutorial Part 5 – Device File Creation
In "Device Drivers"



Linux Device Driver Tutorial Part 4 – Character Device Driver
In "Device Drivers"



Linux Device Driver Tutorial Part 6 – Cdev structure and File Operations
In "Device Drivers"

9

Subscribe

Connect with | Login

*Join the discussion*

B *I* U ~~S~~ {} [+]

This site uses Akismet to reduce spam. Learn how your comment data is processed.

**9 COMMENTS** Oldest

9

monnoliv

September 14, 2017 8:31 AM

Hi,
Very good tutorial, that’s clear and easy to learn.
But ok, now I can read/write a kernel space memory (that’s a software side). What’s up with a real hardware (homemade PCIExpress card, custom USB profil, hardware IO,…), how to do the link between kernel space memory and the hardware (hardware side code).
I don’t know if this is more complicated or not.
What do you think?

Loading...

0 Reply

owl Author

*Reply to monnoliv* September 15, 2017 12:15 AM

Hi Monnoliv,

Thanks for your appreciation. Hardware side, That is different story. We need to register the device driver to the subsystem. In our future tutorials we will cover those topics.

Loading...

0 Reply

9

Mahesh Babu

October 29, 2018 6:29 AM

Can you please explain why "If you are going to use echo and cat command, please allocate the kernel space memory in init function instead of open() function"

Loading...

0 Reply

EmbeTronicx India

*Reply to Mahesh Babu* October 30, 2018 4:31 AM

Hi Mahesh Babu,

Have you see the dmesg when you are doing echo and cat? If you saw that, you will get some clue.

Loading...

0 Reply

Saikiran Muppidi

*Reply to EmbeTronicx India* December 13, 2018 12:03 AM

In both the case open call in driver invoked then whats the difference if something else please let me know

Loading...

0 Reply

9

EmbeTronicx India

*Reply to* *Saikiran Muppidi* December 17, 2018 9:01 PM

Yes you are correct. See our application. We are opening the driver. We are doing our operations. Once our job done, then only we are closing the driver. Until that we don't close the driver. That's why we are allocating memory in open call. In that "echo" and "cat" case, each command will open and close immediately. So each command will create a new buffer. To avoid that if we allocate memory at init time, then each command will access the same memory.

Loading...

0 Reply

Abhinav Asati

*Reply to* *EmbeTronicx India* June 1, 2019 11:45 PM

First of all, great content... it's easy to understand and to the point.

Still I have little doubt. What is the problem if each command creates a new buffer ? Anyway, we are calling close in each echo and cat commandthat releases the buffer. Please let me know if my understanding is corrrect

Loading...

0 Reply

9

EmbeTronicx India

Reply to *Abhinav Asati* June 3, 2019 6:32 AM

Hi Abhinav, Yes you are correct. When we use echo and cat command, it will open and close. But our aim is to check that read and write is working or not. How will you check? That's why we thought of creating one variable and then keep on reading and writing the same variable.When you do echo, it will open, write and finally close. When you do cat, it will open, read and finally close. So in this case, if you do echo / cat command, it will create buffer in open, then write/read and finally release that buffer in… Read more »

0 Reply

Ajay Kumar

May 12, 2020 4:00 PM

Good Explanation sir.Easy understanding of Drivers.

Loading...

1 Reply

9

report this ad

9