# Part 7 – Linux Device Driver Tutorial Programming

[ <https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-tutorial-programming/> ]

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

# 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)

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. Now we are going to discuss more about file operations in 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 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 to 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 then we will call this function. Here we 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 following Makefile (sudo make)

obj-m += driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:

make -C $(KDIR) M=$(shell pwd) modules

clean:

make -C $(KDIR) M=$(shell pwd) clean

# User Space Application

This application will communicate with the device driver.

#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”.

**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 wont say why. You have to find the reason.

Answer :

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.