Linux Device Driver Tutorial Part 8 – I/O Control in Linux IOCTL()

Introduction

Operating system segregates virtual memory into kernel space and user space. Kernel space is strictly reserved for running the kernel, kernel extensions, and most device drivers. In contrast, user space is the memory area where all user mode applications work and this memory can be swapped out when necessary.

There are many ways to Communicate between the User space and Kernel Space, they are:

- IOCTL
- Procfs
- Sysfs
- Configfs
- Debugfs
- Sysctl
- UDP Sockets
- Netlink Sockets

In this tutorial we will see IOCTL.

IOCTL Tutorial in Linux

IOCTL

IOCTL is referred as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories. The major use of this is in case of

handling some specific operations of a device for which the kernel does not have a system call by default.

Some real time applications of ioctl is Ejecting the media from a “cd” drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have write and read function in our device driver. But it is not enough for all cases.

Steps involved in IOCTL

There are some steps involved to use IOCTL.

- Create IOCTL command in driver
- Write IOCTL function in driver
- Create IOCTL command in User space application
- Use IOCTL system call in User space

Create IOCTL Command in Driver

To implement a new ioctl command we need to follow the following steps.

1. Define the ioctl code

```
#define "ioctlname" __IOX("magic number","command number","argument type")
```

where *IOX* can be :
“IO”: an ioctl with no parameters
“IOW”: an ioctl with write parameters (copy_from_user)
“IOR”: an ioctl with read parameters (copy_to_user)
“IOWR”: an ioctl with both write and read parameters

- The Magic Number is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. some times the major number for the device is used here.
- Command Number is the number that is assigned to the ioctl .This is used to differentiate the commands from one another.
- The last is the type of data.

2. Add the header file linux/ioctl.h to make use of the above mentioned calls.

Example

```
#include <linux/ioctl.h>
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)
```

Write IOCTL function in driver

The next step is to implement the ioctl call we defined in to the corresponding driver. We need to add the ioctl function which has the prototype.
Where
<file> : is the file pointer to the file that was passed by the application.
<cmd> : is the ioctl command that was called from the user space.
<arg> : are the arguments passed from the user space.
With in the function "ioctl" we need to implement all the commands that we defined above. We need to use the same commands in switch statement which is defined above.

Then need to inform the kernel that the ioctl calls are implemented in the function "etx_ioctl". This is done by making the fops pointer "unlocked_ioctl" to point to "etx_ioctl" as shown below.

Example:

```
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        copy_from_user(&value ,(int32_t*) arg, sizeof(value));
                        printk(KERN_INFO "Value = %d\n", value);
                        break;
                case RD_VALUE:
                        copy_to_user((int32_t*) arg, &value, sizeof(value));
                        break;
        }
        return 0;
}


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

Now we need to call the new ioctl command from a user application.

Create IOCTL command in User space application

Just define the ioctl command like how we defined in driver.

Example:

```
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)
```

Use IOCTL system call in User space
Include the header file `<sys/ioctl.h>`.Now we need to call the new ioctl command from a user application.

```
long ioctl( "file descriptor","ioctl command","Arguments");
```

`<file descriptor>`: This the open file on which the ioctl command needs to be executed, which would generally be device files.
`<ioctl command>`: ioctl command which is implemented to achieve the desired functionality
`<arguments>`: The arguments that needs to be passed to the ioctl command.

Example:

```
ioctl(fd, WR_VALUE, (int32_t*) &number);
ioctl(fd, RD_VALUE, (int32_t*) &value);
```

Now we will see the complete driver and application.

Device Driver Source Code

In this example we only implemented IOCTL. In this driver, i defined one variable (int32_t value). Using ioctl command we can read or change the variable. So other functions like open, close, read, write, We simply left empty. Just go through the code below.

driver.c

```
#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()
#include <linux/ioctl.h>


#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int32_t value = 0;

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

```
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 long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

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

static int etx_open(struct inode *inode, struct file *file)
{
      printk(KERN_INFO "Device File Opened...!!!\n");
      return 0;
}

static int etx_release(struct inode *inode, struct file *file)
{
      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)
{
      printk(KERN_INFO "Read Function\n");
      return 0;
}
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
      printk(KERN_INFO "Write function\n");
      return 0;
}

static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
       switch(cmd) {
            case WR_VALUE:
                  copy_from_user(&value ,(int32_t*) arg, sizeof(value));
                  printk(KERN_INFO "Value = %d\n", value);
```

```
                break;
            case RD_VALUE:
                copy_to_user((int32_t*) arg, &value, sizeof(value));
                break;
        }
        return 0;
}


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("1.5");
```

Makefile:

```
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
```

Application Source Code

This application is used to write the value to the driver. Then read the value again

test_app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include<sys/ioctl.h>

#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int main()
```

```
{
        int fd;
        int32_t value, number;
        printf("*********************************\n");
        printf("*******WWW.EmbeTronicX.com*******\n");

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

        printf("Enter the Value to send\n");
        scanf("%d",&number);
        printf("Writing Value to Driver\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Value is %d\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

Building Driver and Application

- Build the driver by using Makefile (*sudo make*)
- 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*******
Opening Driver
Enter the Value to send
```

- Enter the value to pass

```
23456
Writing Value to Driver
Reading Value from Driver
Value is 23456
Closing Driver
```

- Now check the value using dmesg

```
Device File Opened...!!!
Value = 23456
Device File Closed...!!!
```

- Our value 23456 was passed to the kernel and it was updated.

This is the simple example using ioctl in driver. If you want to send multiple arguments, put those variables into structure and pass the structure.