Linux Device Driver Coding for Pseudo Device

Murali. B. A
Department of Electrical and Electronics Engineering, Anna University

Abstract
Device driver is the most important software of operating system to interact with hardware devices. As an essential part of operating system, device drivers must be reliable and efficient, because wrong operation can make a fatal system error and hardware performance depends on the device driver. Therefore, it must be developed carefully. In this paper, I present Linux Device Driver coding for Pseudo Device. The main goal of our work is reliable coding for Pseudo Device to improve quality.

Introduction
Device Drivers: The device drivers can be seen as a software layer that lies between the applications and the actual device. Device drivers are integral components of operating systems. The computational workloads imposed by device drivers tend to be aperiodic and unpredictable because they are triggered in response to events that occur in the device, and may arbitrarily block or preempt other time-critical tasks. Linux facilitates us to insert a piece of code along with the running modules into the kernel at run time which is called as Loadable Kernel Module (LKM). In order to develop Linux Device Drivers, it is necessary to have an understanding of the following:

C Programming: Some in-depth knowledge of C Programming is needed, like pointer usage ,bit manipulating functions, and so on.

Microprocessor Programming: It is necessary to know how microcomputers work internally: Memory Addressing Interrupts, and so on.

All of these concepts should be familiar to an assembly programmer.

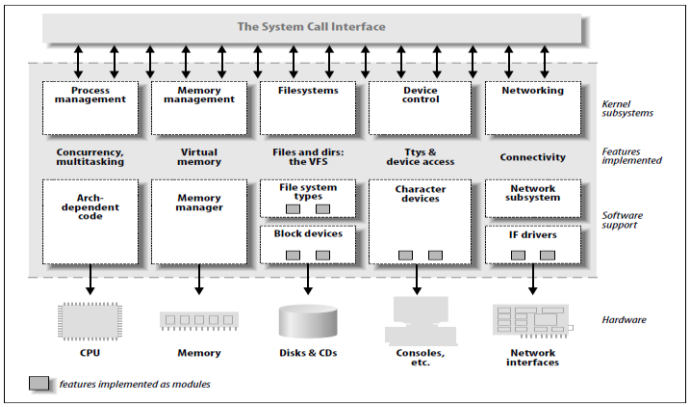
Literature Survey:

S.No.	Title	Remarks
1	A. Rubini, Linux Device Drivers, O'Reilly & Associates, Sebastopol, Calif., 1998	Learnt about basics of Linux Device driver Programming and syntax for Scull (Simple Character Utility for Loadable Localities) Device.
2	T. Burke, M.A. Parenti, A. Wojtas. Writing Device Drivers: Tutorial and Reference, Digital Press, Boston,1995	Learnt about Data Structure for Pseudo Device Driver.
3	Linux Operating System Documentation, http://www.sunsite.unc.edu/pub/Linux	Learnt about Linux Operating System Documentation of File Systems in Linux
4	Robert Love, Linux Kernel Development, Second Edition, 2005	Learnt about basic Commands in Linux for Device Driver.

5	A. Rubini, "Dynamic Kernels: Modularize Device Drivers," Linux J., Issue 23, Mar. 1996, http://www.ssc.com /lj/issue23/1219.html.	Learnt about inserting loadable Kernel Modules (LKM) dynamically in Linux OS.
6	ELF specifications be downloaded from ftp://sunsite.unc. edu/pub/Linux/GCC /ELF.doc.tar.gz.	Learnt about GNU C Compiler (GCC) & Executable and Link File System.
7	Chen lijun, "Understanding Linux kernel source code deeply"[M],Beijing: Posts & Telecom Press. 2002.	Learnt about Kernel source code for Device Driver.
8	Y. Zhou, M.S. Li, "Research and implementing of real-time support of Linux kernel", Journal of computer research and development,Vo1.39, No. 4, April 2002.	Learnt about importance of Linux OS in Real-time systems.
9	Linux Kernel http://www.kernel.org	Downloaded source code for Linux kernel
10	P. Mantegazza, E. Bianchi, L. Dozio, S. Papachar-alambous, RTAI: Real Time Application Interface, Linux Journal, April 2000.	Learnt about Application Interfaces availability for Linux Device Driver

Device Drivers:

A device driver is the set of kernel routines that makes a hardware device respond to the programming interface defined by the canonical set of VFS functions (open, read, lseek, ioctl, and so forth) that control a device. The actual implementation of all these functions is delegated to the device driver. Because each device has a different I/O controller, and thus different commands and different state information, most I/O devices have their own drivers.



The three classes of devices are:

Character Devices:

A character (char) device is one that can be accessed as a stream of bytes (like a file); a char driver is in charge of implementing this behavior. Such a driver usually implements at least the open, close, read, and write system calls. Char devices are accessed by means of filesystem nodes, such as /dev/tty1 and /dev/lp0. The only relevant difference between a char device and a regular file is that you can always move back and forth in the regular file, whereas most char devices are just data channels, which you can only access sequentially.

Block Devices:

Like char devices, block devices are accessed by filesystem nodes in the /dev directory. A block device is a device (e.g., a disk) that can host a filesystem. Linux allows the application to read and write a block device like a char device—it permits the transfer of any number of bytes at a time. As a result, block and char devices differ only in the way data is managed internally by the kernel, and thus in the kernel/driver software interface.

Network Devices:

Any network transaction is made through an interface, that is, a device that is able to exchange data with other hosts. A network interface is in charge of sending and receiving data packets, driven by the network subsystem of the kernel, without knowing how individual transactions map to the actual packets being transmitted. Network devices are, usually, designed around the transmission and receipt of packets. A network driver knows nothing about individual connections, it only handles packets. Communication between the kernel and a network device driver is completely different from that used with char and block drivers. Instead of read and write, the kernel calls functions related to packet transmission.

Loading & Unloading Device Drivers:

Most modern kernels can dynamically load & unload some portions of the kernel code, which are usually called modules. These modules can be linked on demand.

To load & unload device drivers dynamically in the kernel, they are built in form of modules. Modules are stored in the file system as ELF object files. A user can link a module into the running kernel by executing the insmod utility. A module can be un-linked from kernel by executing the rmmod utility.

Kernel modules usually provide two functions:

An initialization function is called if the module is loaded. It usually scans for hardware, registers functions that are called later and allocates memory. Such functions can be callback functions at different subsystems like the USB subsystem or interrupt handlers. This function is required to load the kernel module.

A clean-up function usually reverts all the steps done in the initialization function in reverse order. It is possible for a module to provide only the initialization function; this means that it is impossible to unload the module.

Software Requirements & Specifications:

Description:
The project is the implementation of device driver for SCULL(Simple Character Utility for Loadable Localities) device using C programming language.

Requirements Analysis:

1. What are the complexities involved in making the project?

The project involves the implementation of generic and efficient device driver. Drivers reside in a layer of kernel space just above the Pseudo hardware i.e., Driver direct communicate with Pseudo hardware and system space, if something goes wrong with in programming it can crash the system and hardware also.

Functional Requirements:

System Requirements:

• Pentium machine or above.

• Linux-2.6.32.5 Environment

• 64 MB of RAM.

System Design And Source Code:

```
/* SAMPLE CODE */
#include<linux/init.h>
#include<linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
        printk(KERN_ALERT "HELLO, WORLD \n");

return 0;
}
static void hello_exit(void)
{
        printk(KERN_ALERT "GOOD BYE WORLD \n");
}
module_init(hello_init);
module_exit(hello_exit);
```

```
/*  DEVICE REGISTRATION */
#include<linux/init.h>
#include<linux/module.h>
#include<linux/types.h>
#include<linux/fs.h>
#include<linux/kdev_t.h>
```

```
#include<linux/moduleparam.h>
#include<linux/cdev.h>
MODULE_LICENSE("Dual BSD/GPL");
static int scull_major=0;
static int scull_minor = 0;
static int scull_result=0;
static int scull_res=0;
static int scull_num_dev=0;
dev_t dev;
module_param(scull_major,int, S_IRUGO);
module_param(scull_num_dev,int,S_IRUGO);
struct file_operations scull_fops
={
        .owner= THIS_MODULE,
        //.open = scull_open,
        //.read = scull_read,
        //.write = scull_write,
        //.release = scull_release,
};
struct scull_dev
{
        struct cdev cdev;
};
struct scull_dev scull_devices;
static void scull_setup_cdev(struct scull_dev *dev,int index)
{
        int err, devno = MKDEV(scull_major, scull_minor + index);
        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add (&dev->cdev, devno,1)
              if(err)
               printk(KERN_NOTICE "Error %d adding scull%d", err, index);
               else
               printk(KERN_ALERT "The number with device number %d is initialized\n",index);
}
static int scull_init(void)
```

```
{
	dev=MKDEV(scull_major,scull_minor);
	scull_result = register_chrdev_region(dev,1,"hello");
	if(scull_result < 0 )
	scull_res = alloc_chrdev_region(&dev,0,1,"hello");
	scull_major = MAJOR(dev);
	printk("major number required is allocate %d",scull_major);
	scull_setup_cdev(&scull_devices, 0);
	return 0;
}
static void scull_exit(void)
{
		cdev_del(&scull_devices.cdev);
		printk(KERN_ALERT "Device deleted\n");
		unregister_chrdev_region(dev,1);
		printk(KERN_ALERT "Device unregistered\n");
		printk("\nModule unloaded\n");
}
	module_init(scull_init);
	module_exit(scull_exit);

/* KERNEL MODULE LOADING AND UNLOADING */
/* STATIC AND DYNAMIC ALLOCATION OF MAJOR NUMBERS */
#include<linux/init.h>
#include<linux/module.h>
#include<linux/types.h>
#include<linux/fs.h>
#include<linux/kdev_t.h>
#include<linux/moduleparam.h>
MODULE_LICENSE("Dual BSD/GPL");
static int major=0;
static int minor = 0;
static int result=0;
static int res=0;
dev_t dev;
module_param(major,int, S_IRUGO);
```

```
static int static1_init(void)
{
        dev=MKDEV(major,minor);
        result = register_chrdev_region(dev,1,"hello");
        if(result < 0 )
        //result = alloc_chrdev_region(&dev,0,1,"hello")        //printk(KERN_ALERT "required major number is not available\n");
        res = alloc_chrdev_region(&dev,0,1,"hello");
        major = MAJOR(dev);
        printk("major number required is allocate %d",major);
        return 0;
}
static void static1_exit(void)
{
        unregister_chrdev_region(dev, 1);
}
module_init(static1_init);
module_exit(static1_exit);

/* ARRAY OF SCULL DEVICES INSERTION */
#include<linux/init.h>
#include<linux/module.h>
#include<linux/types.h>
#include<linux/fs.h>
#include<linux/kdev_t.h>
#include<linux/moduleparam.h>
#include<linux/cdev.h>
MODULE_LICENSE("Dual BSD/GPL");
static int scull_major=0;
static int scull_minor = 0;
static int scull_result=0;
static int scull_res=0;
static int scull_num_dev=0;
static int i;
static dev_t dev;
module_param(scull_major,int, S_IRUGO);
module_param(scull_num_dev,int,S_IRUGO);
```

```
struct file_operations scull_fops
{
        .owner= THIS_MODULE,
        .open = scull_open,
        .read = scull_read,
        .write = scull_write,
        .release = scull_release,
};
struct scull_dev
{
        struct cdev cdev;
};
 struct scull_dev scull_devices[5];
static void scull_setup_cdev(struct scull_dev *dev,int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno,1);
        if(err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
        else
        printk(KERN_ALERT "The number with device number %d is initialized\n",index);
}
static int scull_init(void)
{
        dev=MKDEV(scull_major,scull_minor);
        scull_result = register_chrdev_region(dev,1,"hello");
        if(scull_result < 0 )
        scull_res = alloc_chrdev_region(&dev,0,1,"hello");
        scull_major = MAJOR(dev);
        printk("major number required is allocate %d",scull_major);

for(i=0;i<scull_num_dev;i++)
                scull_setup_cdev(&scull_devices[i], i);
                return 0;
```

```
}

static void scull_exit(void)
{
        for(i=0;i>scull_num_dev;i++)
        {
                cdev_del(&scull_devices[i].cdev);
                printk("Device no : %d deleted\n",i);
        }
    unregister_chrdev_region(dev,1);
        printk("\nModule unloaded\n");
}
        module_init(scull_init);
        module_exit(scull_exit);

/* API LEVEL CODE  FILE OPEN */
#include<stdio.h>
#include<stdlib.h>
main()
{
        int f;
        f=open("./text","r");
        if(f<0)
        {
                printf("module busy not inserted\n");
                exit(0);
        }
        printf("Module inserted");
      close(f);
 }
/* API LEVEL CODE  FILE READ */
#include<stdio.h>
#include<stdlib.h>
#define MAX 1024
main()
{
        int f,n;
```

```
        char buf[MAX];
        f=open("./scull","r");

if(f<0)
        {
                printf("module busy not inserted\n");
                exit(0);
        }
        printf("Module inserted\n");
        n=read(f,buf,MAX);

if(n<0)
        {
                printf("Reading Error");
                exit(0);
        }
        printf("Data Read : %s\n",buf);
        close(f);
}
/* API LEVEL CODE  FILE WRITE */
#include<stdio.h>
#include<stdlib.h>
#define MAX 1024
main()
{
    int f1,n1=0;
    char buf[MAX];
    f1=open("./scull","r");
    if(f1<0)
    {
        printf("Module is busy,Not inserted");
        exit(0);
    }

  printf("Module is inserted");
        printf("\nEnter the data:\n");
        scanf("%s",buf);
        n1=write(f1,buf,5);
        if(n1<0)
```

```
        {
            printf("\nWriting  error\n");
            exit(0);
        }
        printf("\nData Written :%s\n",buf);
        close(f1);
}
```

Implementation :

Layout of the Driver

As this driver is to be added to the running kernel dynamically, it has to be designed in form of a module. After compilation a kernel object will be created (.ko file) which can be added using **i***nsmod* utility.

Insmod:

In this routine, the driver structure is allocated using usb_dev..

- Initialization of the driver is done here.
- List of USB devices is scanned. (using usb_get_device)
- If the desired device is found, it is added to the driver's list of devices.
- Then the device is enabled.
- IO port is allocated for the specified device (using usb_request_regions)
- Required information is read from configuration space of USB device and then stored into the private object of the device.
- Here we perform three steps for initialization of a module:

They are ,

1.Sys_create_module():

The functionality of this system call is that it requests to allocate a memory in the kernel space using vmalloc().

If the memory is not available it returns a status of -1.

If the memory is available it returns a status of 0.

2.get_kern_syms():

- The functionality of this function is that it resolves all the variables,symbols and functions inserted into the symbol table with respect to kernel,it gets all declarations and definitions.

3.module_init():

- The functionality of this is like it initializes

The module.

rmmod **:**

This routine is the counterpart of the init routine.

- Whatever allocation was done in init is de-allocated here.
- First driver's every device structure is removed from the memory.
- Then the driver structure is de-allocated.

For removing a module we use two functions .They are:

- 1.Sys_delete_module():

The functionality of this module is like it checks whether the module is in use or not.

- The other thing is like dereferencing the symbols,variables,definitions declared in symbol table.
- 2.module_cleanup(): The functionality of this is to free the memory.

Let us consider we are having one folder named hello in which we have tho following files

1 hello.c ------- API

2 Makefile

3 scull.c ------- scull with major and minor numbers

STEPS TO INSERT A MODULE:

1 make

it will create .ko,.o files

2 insmod scull.ko major_num=252 num=2(here num represents the number of minor numbers)

if 252 is busy , dynamically it will allocate the major number fot the module

3 lsmod | head 5

to check the module was inserted ot not

4 dmesg | tail -10

to get the major and minor numbers for the inserted module

5 compile the API

gcc -o hello hello.c

we will get hello binary

6 We have to create the device special files

mknod scull0 c major_num minor_num

example: mknod scull0 c 252 0(for 1st device)

depending on the minor numbers we have to create those many device special files

7 Now run the API binary means

./hello

it will ask the path of device special file . Give the path of scull0

Steps To Remove A Module:

8 remove the module

rmmod scull.ko

Conclusions

The project proposal was aimed to develop a Linux based Device Driver Coding for “Simple Character Utility for Loadable Localities(Scull)” using C language .

References

1. A. Rubini, Linux Device Drivers, O'Reilly & Associates,Sebastopol, Calif., 1998
2. T. Burke, M.A. Parenti, A. Wojtas. Writing Device Drivers: Tutorial and Reference, Digital Press, Boston,1995.
3. Linux Operating System Documentation,http://www.sunsite.unc.edu/pub/Linux
4. Robert Love, Linux Kernel Development, Second Edition, 2005
5. A. Rubini, “Dynamic Kernels: Modularize Device Drivers,” Linux J.,Issue 23, Mar. 1996, http://www.ssc.com/lj/issue23/1219.html.

6. Y. Zhou, M.S. Li, "Research and implementing of real-time support of Linux kernel", Journal of computer research and development,Vo1.39, No. 4, April 2002.
7. P. Mantegazza, E. Bianchi, L. Dozio, S. Papachar-alambous, RTAI: Real Time Application Interface,Linux Journal, April 2000.
8. Chen lijun, "Understanding Linux kernel source code deeply"[M],Beijing: Posts & Telecom Press. 2002.
9 Linux Kernel http://www.kernel.org.
10. ELF specifications be downloaded from ftp://sunsite.unc.edu/pub/Linux/GCC/ELF.doc.tar.gz.