ECEN 749 LAB REPORT

Excercise #6: An Introduction to Linux Device Driver Development

Tanmay Verma

10.27.2017

INTRODUCTION

The purpose of this lab is to create device drivers in an embedded Linux environment.

PROCEDURE

The following steps were carried out:

1. The modules directory from the previous lab was copied in the project directory.
2. Created a new kernel module source file called ‘multiply.c’
3. The required header files ‘xparameters.h’ and ‘xparameters_ps.h’ were copied in the directory.
4. The multiply module was written and compiled to create a kernel object.
5. The module was loaded onto the linux running on Zybo board via SD card and its functioning was verified.
6. A character device driver, ‘multiplier.c’ was written following the guidelines in the manual and example codes
7. The driver was compiled to create a kernel object.
8. A user application program was written to verify the functioning of driver module written in previous step.
9. The kernel objects and binary of application were copied on the SD card and loaded on the system,
10. The execution verified with the expected outputs.

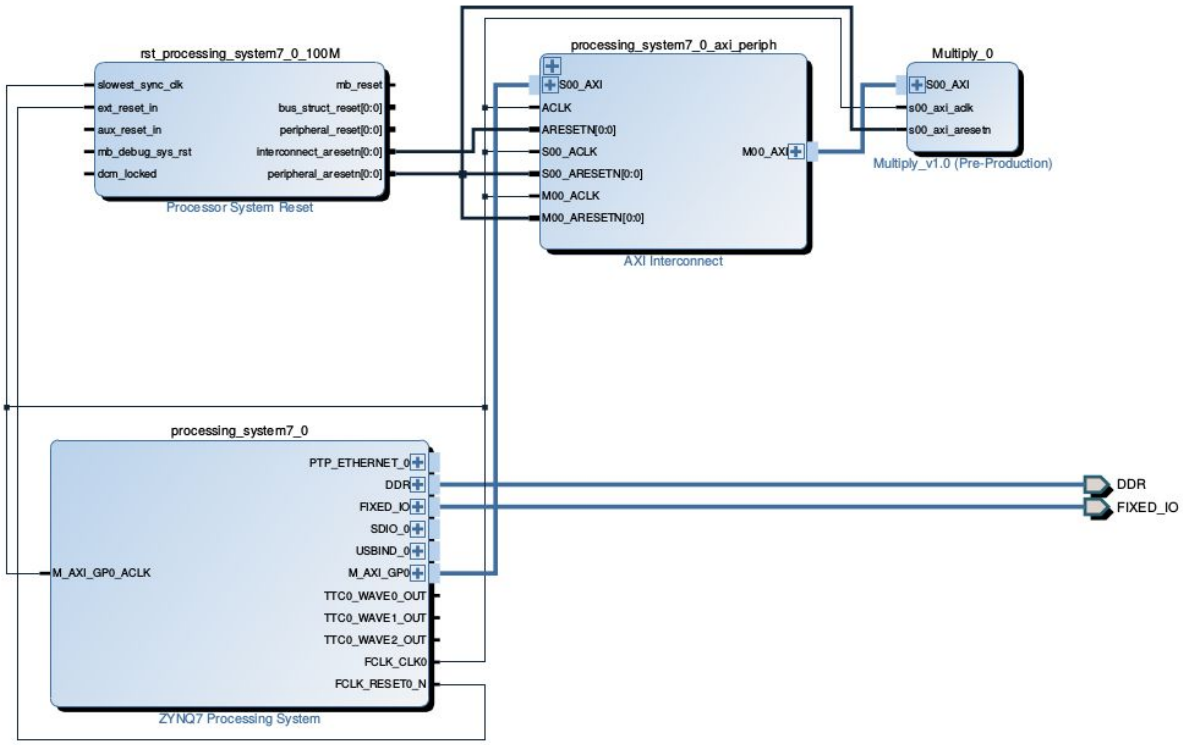
RESULTS

Through this exercise, module was successfully written and compiled on the centos machine and copied to the SD card to be integrated with the Linux Operating System running on Zybo board. The module successfully interacted with the Multiplier H/W unit. The driver designed in the next step, took the functionality even further allowing users level applications to read/write on h/w buffers.

CONCLUSION

In conclusion, at the end of the exercise, I successfully compiled a multiply module and character driver for interacting with Multiply H/W.

Hardware Design(Same as Lab5)



Multiply.c Module

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_* and printk */
#include <linux/init.h> /* Needed for __init and __exit macros */
#include <asm/io.h> /* Needed for IO reads and writes */

//#include "xparameters.h" /* needed for physical address of multiplier */
#define XPAR_MULTIPLY_0_S00_AXI_BASEADDR        0x43C00000
#define XPAR_MULTIPLY_0_S00_AXI_HIGHADDR        0x43C0FFFF

/* from xparameters .h */
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR //physical address of multiplier
/* size of physical address range of multiply */
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR + 1

void* virt_addr; //virtual address pointing to multiplier

/*
 * This function is run upon module load. This is where you setup data structures
 * and reserves resources used by the modules.
 */
static int __init my_init(void)
{
        /* Linux kernel's version of printf */
        printk(KERN_INFO "Mapping virtual address\n");
```

```
	/* map virtual address to multiplier physical address */
	virt_addr = ioremap(PHY_ADDR, MEMSIZE);
	printk(KERN_INFO "VA %p gets mapped to PA %x", virt_addr, PHY_ADDR);
	/* write 7 to register 0 */
	printk(KERN_INFO "Writing a 7 to register 0 \n");
	iowrite32(7, virt_addr + 0 ); //base_address + offset
	/* write 2 to register 1 */
	printk(KERN_INFO "Writing a 2 to register 1 \n");
	iowrite32(2, virt_addr + 4); //base_address + offset

	printk("Read %d from register 0 \n", ioread32(virt_addr + 0));
	printk("Read %d from register 1 \n", ioread32(virt_addr + 4));
	printk("Read %d from register 2 \n", ioread32(virt_addr + 8));

	//A non 0 return mens the init_module failed: module can't be loaded.
	return 0;
}
/* This function is run just before the module removal. All resources must be released here which were held by the module other wise prepare for reboot.*/
static void __exit my_exit(void)
{
	printk(KERN_ALERT "unmapping virtual address space... \n");
	iounmap((void *)virt_addr);
}

/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN749 Student (and others)");
```

```
MODULE_DESCRIPTION("Simple Hello World Module");

/*
 * Here we define whihc functions we want to use for initialization
 * and cleanup
 */
module_init(my_init);
module_exit(my_exit);
```

Multiplier.c Character Driver

```
#include "multiplier.h"


/* This structure defines the function pointers to our functions for
 * opening, closing, reading and writing the device file.  There are
 * lots of other pointers in this structure which we are not using,
 * see the whole definition in linux/fs.h */
static struct file_operations fops = {
        .read = device_read,
        .write = device_write,
        .open = device_open,
        .release = device_release
};



static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
```

```
                char *buffer,           /* buffer to fill with data */
                size_t length,          /* length of the buffer         */
                loff_t * offset)
{
        /*
        * Number of bytes actually written to the buffer
        */
        int bytes_read = 0;
        int i;

        for(i = 0; i < 3; i++) {
        *((int *)msg_bf_Ptr + i) = ioread32(virt_addr + i * 4);
        printk("The read data is Got = %d  read = %d", *((int *)msg_bf_Ptr + i),
ioread32(virt_addr + i * 4));
        }
        /*
        * Actually put the data into the buffer
        */
        cur_Ptr = msg_bf_Ptr;

        while (length) {
        /*
        * The buffer is in the user data segment, not the kernel
        * segment so "*" assignment won't work.  We have to use
        * put_user which copies data from the kernel data segment to
        * the user data segment.
        */
        put_user(*(cur_Ptr++), buffer++); /* one char at a time... */
```

```
        length--;
        bytes_read++;
        }
        return bytes_read;
}


/*
 * Called when a process tries to open the device file, like "cat
 * dev/my_chardev_mem".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
{
        printk( KERN_INFO "The %s is opened", DEVICE_NAME);
        return 0;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
        printk( KERN_INFO "The %s is released", DEVICE_NAME);
        return 0;
}
```

```
static ssize_t device_write(struct file *file, const char __user * buffer, size_t length, loff_t *
offset)
{
        int i;
        /* printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, (int)length); */
        /* get_user pulls message from userspace into kernel space */
        for (i = 0; i < length; i++)
        get_user(msg_bf_Ptr[i], buffer + i);
        /* write data1 to register 0 */
        printk(KERN_INFO "Writing %d to register 0 \n", *((int *)msg_bf_Ptr));
        iowrite32(*((int *)msg_bf_Ptr), virt_addr + 0 ); //base_address + offset
        /* write data2 to register 1 */
        printk(KERN_INFO "Writing %d to register 1 \n", *((int *)msg_bf_Ptr + 1));
        iowrite32(*((int *)msg_bf_Ptr + 1), virt_addr + 4); //base_address + offset

        /*
        * Again, return the number of input characters used
        */
        return i;
}

/*
 * This function is run upon module load. This is where you setup data structures
 * and reserves resources used by the modules.
 */
static int __init my_init(void)
{
        /* We need to allocate the memspace _BEFORE_ registering the device
```

```
* to avoid any race conditions */
msg_bf_Ptr = (char *)kmalloc(BUF_LEN*sizeof(char), GFP_KERNEL);

/* Note: kmalloc can fail, even on a non-borked kernel, always exit
* gracefully.  In the event of a failure pointer will be set to
* NULL. */
if (msg_bf_Ptr == NULL) {
/* Failed to get memory, exit gracefully */
printk(KERN_ALERT "Unable to allocate needed memory\n");
return 10;                  /* Defining error code of 10 for
                               "Unable to allocate memory" */
}
cur_Ptr = msg_bf_Ptr;



/* Linux kernel's version of printf */
printk(KERN_INFO "Mapping virtual address to the multiplier\n");
/* map virtual address to multiplier physical address */
virt_addr = ioremap(PHY_ADDR, MEMSIZE);
printk(KERN_INFO "VA %p gets mapped to PA %x", virt_addr, PHY_ADDR);
/* write 7 to register 0 */



/* This function call registers a device and returns a major number
* associated with it.  Be wary, the device file could be accessed
* as soon as you register it, make sure anything you need (ie
```

```
        * buffers ect) are setup _BEFORE_ you register the device.*/
        Major = register_chrdev(0, DEVICE_NAME, &fops);
        /* Negative values indicate a problem */
        if (Major < 0) {
        /* Make sure you release any other resources you've already
        * grabbed if you get here so you don't leave the kernel in a
        * broken state. */
        printk(KERN_ALERT "Registering char device failed with %d\n", Major);
        /* We won't need our memory so make sure to free it here... */
        kfree(msg_bf_Ptr);
        return Major;
        }

        printk(KERN_INFO "Registered a device with dynamic Major number of %d\n",
Major);

        printk(KERN_INFO "Create a device file for this device with this
command:\n'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);

        return 0;       /* success */
}

/* This function is run just before the module removal. All resources must be released
here which were held by the module other wise prepare for reboot.*/
static void __exit my_exit(void)
{
        printk(KERN_ALERT "removing the driver... \n");
        /*
        * Unregister the device
```

```
        */
        unregister_chrdev(Major, DEVICE_NAME);

        kfree(msg_bf_Ptr);              /* free our memory (note the ordering
                        here) */
        iounmap((void *)virt_addr);
}

/* ehese define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN749 Student (and others)");
MODULE_DESCRIPTION("Simple Hello World Module");

/*
 * Here we define whihc functions we want to use for initialization
 * and cleanup
 */
module_init(my_init);
module_exit(my_exit);
```

Multiplier.h header file for Character Driver

```
/* All of our linux kernel includes. */
#include <linux/module.h>  /* Needed by all modules */
#include <linux/moduleparam.h>  /* Needed for module parameters */
#include <linux/kernel.h>  /* Needed for printk and KERN_* */
#include <linux/init.h>        /* Need for __init macros  */
```

```
#include <linux/fs.h>            /* Provides file ops structure */
#include <linux/sched.h>   /* Provides access to the "current" process
                              task structure */
#include <linux/slab.h>
#include <asm/uaccess.h>   /* Provides utilities to bring user space
                              data into kernel space.  Note, it is
                              processor arch specific. */
#include <asm/io.h> /* Needed for IO reads and writes */


#include "xparameters.h"


/* Some defines */
#define DEVICE_NAME "multiplier"
#define BUF_LEN 12
/* from xparameters .h */
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR //physical address of
multiplier
/* size of physical address range of multiply */
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR -
XPAR_MULTIPLY_0_S00_AXI_BASEADDR + 1


/* Function prototypes, so we can setup the function pointers for dev
   file access correctly. */
int init_module(void);
```

```
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);

/*
 * Global variables are declared as static, so are global but only
 * accessible within the file.
 */
static int Major;                /* Major number assigned to our device driver */
static char *msg_bf_Ptr;         /* This time we'll use kmalloc and
                                    kfree to handle the memory */
static char *cur_Ptr;            /* current position within buffer */
void* virt_addr; //virtual address pointing to multiplier
```

DEVTEST.C: The C application to test out driver

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

```
int main() {
        unsigned int result;
        int fd;
        int i,j;
        int in_buf[2];
        int out_buf[3];

        char input = 0;
        fd = open("/dev/multiplier", O_RDWR);
        if(fd == -1) {
        printf("Failed to open device file!\n");
        return -1;
        }

        while(input != 'q') {
        for(i = 0; i <= 16; i++) {
                for(j = 0; j <= 16; j++) {
                        in_buf[0] = i;
                        in_buf[1] = j;
                        write(fd, (char *)in_buf, 8);
                        printf(" written %u * %u \n", in_buf[0], in_buf[1]);
                        read(fd, (char *)out_buf, 12);
```

```
                        printf("%u * %u = %u \n", out_buf[0], out_buf[1], out_buf[2]);
                        if( out_buf[2] == (i*j))
                        printf("Results Correct ! \n");
                        else
                        printf("Results Incorrect ! \n");
                        input = getchar();
                }
        }
        }
        close(fd);
        return 0;
}
```

Makefile

```
obj-m += hello.o

all:
        make -C /home/grads/t/tanmay2592/ECEN749/lab_5/linux-3.14 M=$(PWD) modules

clean:
        make -C /home/grads/t/tanmay2592/ECEN749/lab_5/linux-3.14 M=$(PWD) clean
```

PICOCOM OUTPUT(The Third Part)

The First Part :
zynq> insmod multiply.ko
Mapping virtual address
VA 60900000 gets mapped to PA 43c00000
Writing a 7 to register 0
Writing a 2 to register 1
Read 7 from register 0
Read 2 from register 1
Read 14 from register 2
zynq> rmmod multiply
unmapping virtual address space...
zynq>

QUESTIONS

5. [4 points.] Answers to the following questions:

(a) Given that the multiplier hardware uses memory mapped I/O (the processor communicates with it through explicitly mapped physical addresses), why is the ioremap command required?

The ioremap command is required to obtain where exactly the memory of the device got mapped into the virtual address space used by the system. The memory mapped i/o too is a part of virtual memory setup, where a section of **virtual** addresses maps the physical registers of Multiplier H/W.

(b) Do you expect that the overall (wall clock) time to perform a multiplication would be better in part 3 of this lab or in the original Lab 3 implementation? Why?

No. Because in lab 3 the C code directly interacted with the registers on multiplier, but in the present case, there is a time overhead involved in the communication

through drivers which are responsible for moving the data from user space to the device registers,

(c) Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach?

In lab 3 the user program had a direct access to the device's registers, whereas here the access is moderated via driver interface to the device. Lab3 implementation was fast but couldn't handle multiple such execution requests coming simultaneously. Whereas in this lab, the operating system takes care of the device scheduling at expense of an added time overhead.

(d) Explain why it is important that the device registration is the last thing that is done in the initialization routine of a device driver. Likewise, explain why unregistering a device must happen first in the exit routine of a device driver.

For registering the device in the kernel data structure, the working memory for the device should be known from before. Hence, the registration should be the last thing in initialization when all allocations have been made.

In the same way, when freeing memory is done before unregistering the device, the system might misbehave as the kernel has already registered it in its data structures. Hence, the unregistering device should be the first thing done for the device at exit.