ELECTRICAL & COMPUTER ENGINEERING
TEXAS A&M UNIVERSITY

ECEN 449: Microprocessor System Design

Lab 6: An Introduction to Character Device Driver Development

Kylan Lewis

UIN: 719001131

ECEN 449 -504

TA: Ashwin Ashokan

Date: 11/13/20

Introduction:

The purpose of this lab is to understand the process of creating and working with the device driver in Linux. For this lab, we made a driver called multiplier and saw how to tell the kernel and userspace how to use it and then we made a devtest.c file to test this device.

Procedure:

Part 1:

1. Create a new lab 6 directory with the previous Lab5b contents.
2. Create a new file called multiplier.c, that functions as a multiplier. Similar to the previous lab multiply.c file.
3. Edit the Makefile and compile multiply.c to generate multiply.ko.

Part 2:

1. Finish writing the devtest.c that will test the functionality of the multiply.c file

2. Compile this with, arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c

Part 3:

1. Copy and paste the output files from both multiply.c and devtest.c to an SD Card and insert it into the ZYBO board.

2. Mount the SD card and make a device from multiplier.ko after loading it.

3. Run the executable ./devtest, and examine the output as you scroll.

Results:

multiplier registered:

```
zynq> ls
console             ram6                tty38
cpu_dma_latency     ram7                tty39
full                ram8                tty4
i2c                 ram9                tty40
iio:device0         random              tty41
input               root                tty42
kmsg                snd                 tty43
loop-control        timer               tty44
loop0               tty                 tty45
loop1               tty0                tty46
loop2               tty1                tty47
loop3               tty10               tty48
loop4               tty11               tty49
loop5               tty12               tty5
loop6               tty13               tty50
loop7               tty14               tty51
mem                 tty15               tty52
memory_bandwidth    tty16               tty53
mice                tty17               tty54
mmcblk0             tty18               tty55
mmcblk0p1           tty19               tty56
multiplier          tty2                tty57
network_latency     tty20               tty58
network_throughput  tty21               tty59
null                tty22               tty6
port                tty23               tty60
psaux               tty24               tty61
ptmx                tty25               tty62
pts                 tty26               tty63
ram0                tty27               tty7
ram1                tty28               tty8
ram10               tty29               tty9
ram11               tty3                ttyPS0
ram12               tty30               urandom
ram13               tty31               vcs
ram14               tty32               vcs1
ram15               tty33               vcsa
ram2                tty34               vcsa1
ram3                tty35               vga_arbiter
ram4                tty36               xdevcfg
ram5                tty37               zero
zynq> cd /mnt/modules/
zynq> ls
Makefile          modules.order     multiplier.o      multiply.o
Module.symvers    multiplier.c      multiply.c        xparameters.h
devtest           multiplier.ko     multiply.ko       xparameters_ps.h
```

Major number of multiplier and execution of devtest.

```
zynq> insmod multiplier.ko
insmod: can't insert 'multiplier.ko': File exists
zynq> dmesg | tail
random: dropbear urandom read with 1 bits of entropy available
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt.
 Please run fsck.
Mapping virtual address...
Physical Address: 43c00000
Virtual Address: 608e0000
Registered a device with dynamic Major number of 245
Create a device file for this device with this command:
'mknod /dev/multiplier c 245 0'.
This device is opened
This device is closed
zynq> ./devtest
This device is opened
0 * 0 = 0 Result Correct!
0 * 1 = 0 Result Correct!
0 * 2 = 0 Result Correct!
0 * 3 = 0 Result Correct!
```

Post-Lab 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?

Ioremap was required because we needed to map the physical address of the multiplication peripheral to a virtual address. Virtual Memory isn't required because in this it can be assumed that the peripheral address is located contiguously in memory.

(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?

Since there is no OS in the way the lab 3 implementation might perform faster because the hardware in lab 3 was directly mapped to the ARM processor.

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

Lab 3 might be harder to interact with because Linux device drivers make it easy for user interaction with the hardware. So, while it might be a little slower it is much more appealing from the user aspect. In lab 6, it is easier to read and write. The only bad thing is it uses Linux as a sort of middle man making it a bit slower.

(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 un-registering a device must happen first in the exit routine of a device driver.

It is important that device registration is the last thing done because we need to set up before initializing the device driver. We don't want to perform initialization first before mapping the virtual address space because we are missing a component that allows for the device driver to work properly. In the exit routine of a device driver we unregister our first device so that it can stay uninterrupted by the unmapping and cleaning of resources while the device is connected because the device might be using components of memory allocation.

Code:

multiplier.c

```
#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 <linux/moduleparam.h>  /* Needed for module parameters */
#include <linux/fs.h>        /* Provides file ops structure */
#include <linux/sched.h>   /* Provides access to the "current" process task structure */
#include <asm/uaccess.h>   /* Provides utilities to bring user space */
#include "xparameters.h" /* Needed for physical address of multiplier */

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

/* 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 *);
```

```
void* virt_addr; //virtual address pointing to multiplier
static int Major; /* Major number assigned to our device driver */

/* 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
};



/* This function is run upon module load. This is where you setup data structures and reserve resources
used by the module. */

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*/
	//use ioremap
	virt_addr = ioremap(PHY_ADDR, MEMSIZE);
	printk("Physical Address: %x\n", PHY_ADDR); //Print physical address
	printk("Virtual Address: %x\n", virt_addr); //Print virtual address

	/* 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);
		iounmap((void*)virt_addr);
		return Major;
	} else {
		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);
	}

	//a non 0 return means init_module failed; module can't be loaded.
```

```
        return 0;
}
/* This function is run just prior to the module's removal from the system. You should release _ALL_
resources used by your module here (otherwise be prepared for a reboot). */
static void __exit my_exit(void) {
        printk(KERN_ALERT "unmapping virtual address space...\n");
        unregister_chrdev(Major, DEVICE_NAME);
        iounmap((void*)virt_addr);
}

/*
 * Called when a process tries to open the device file, like "cat
 * /dev/my_chardev".  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_ALERT "This device is opened\n");
  return 0;
}



/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
  printk(KERN_ALERT "This device is closed\n");
  return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts
 * to read from it.
 */
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<length; i++) {
                put_user(ioread8(virt_addr+i), buffer+i);
```

```
		bytes_read++;
	}

	/*
	* Most read functions return the number of bytes put into the
	* buffer
	*/
	return bytes_read;
}

/*
 * This function is called when somebody tries to write into our
 * device file.
 */
static ssize_t device_write(struct file *file, const char __user * buffer, size_t length, loff_t * offset)
{
	int i;
	char message;

	/* get_user pulls message from userspace into kernel space */
	for(i=0; i<length; i++) {
		get_user(message, buffer+i);
		iowrite8(message, virt_addr+i);
	}



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

/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN449 Kylan Lewis");
MODULE_DESCRIPTION("Simple multiplier module");

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

devtest.c

```
#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=open("/dev/multiplier",O_RDWR);
        int i,j;
 unsigned int read_i;
 unsigned int read_j;
 int buffer[3];

        char input = 0;

        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++)
                        {
    buffer[0]=i;
    buffer[1]=j;
    write(fd,(char*)&buffer,8);
    read(fd,(char*)buffer,12);


    read_i=buffer[0];
    read_j=buffer[1];
    result=buffer[2];

                                printf("%u * %u = %u ",read_i,read_j,result);

                                if(result==(i*j))
                                        printf("Result Correct!");
                                else
                                        printf("Result Incorrect!");
```

```
                        input = getchar();
                    }

                }
            }
            close(fd);
            return 0;
}
```