ECEN 449 - Lab Report

Lab Number: 6

Lab Title: An Introduction to Character Device Driver Development

Section Number: 511

Student's Name: Kris Gavvala

Student's UIN: 929000158

Date: 4/27/2025

TA: Gautham Nemani

Introduction:

This week's lab focused on creating device drivers. We extended the capabilities of our kernel module from lab 5 and created a complete device driver. We also created a Linux application to use the device driver and test its functionality.

Procedure:

- Create a petalinux project in the tmp folder of the linux machine
- Configure it with the multiply hardware xsa file created in lab 3
- Create the module multiplier in the petalinux project
- Write the code for multiplier.c
    - initialization routine
    - exit routine
    - open and close functions
    - read function
    - write function
- add xparameters.h and xparameters_ps.h files to the multiplier module
- edit multiplier.bb in the multiplier module
- write the user application devtest.c to test the diver functionality
    - write to and read from the device file '/dev/multiplier
    - print results of the multiplication to the terminal.
- Build the petalinux project
- Upon build completion, get the multiplier.ko file
- Compile the devtest.c and get the devtest exectutable
- Download the BOOT.bin, image.ub, boot.scr, devtest.exe, and multiplier.ko file onto a sd card and insert it into the ZYBO-z710 fpga
- After loading multiplier.ko onto the system, use picocom to view output from the fpga on the linux machine.
- Execute devtest and observe output on picocom

Results:

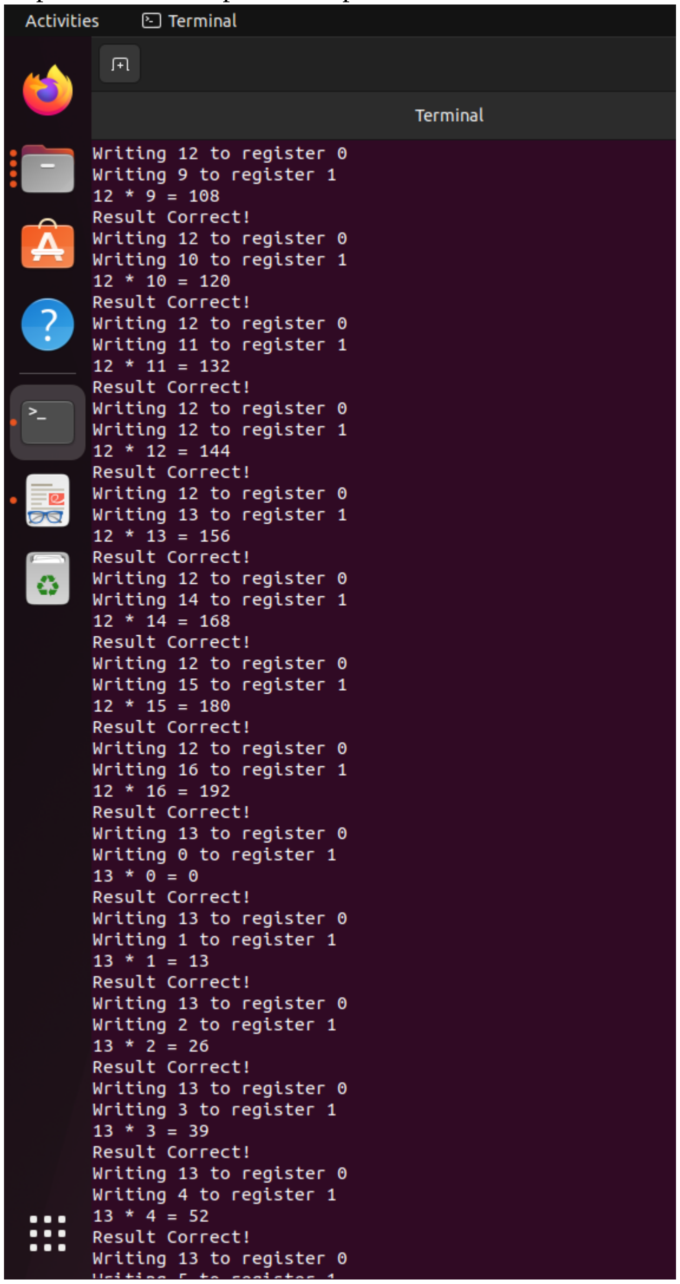
After executing devtest, the numbers 1-16 were multiplied by each other for a total of 256 operations, and each one was tested to see if the result from the hardware module got the same result as the operation in the devtest.c code. These results were printed out to the terminal.

Conclusion:

This lab showed a basic example of creating a device driver for the linux OS. We learned how to create the device driver using the dynamic kernel module with the petalinux sdk. The module was built using the xsa file from a Verilog description of a simple multiplier that was run on the zybo-z7-10 fpga. We learned how to cross-compile the kernel module and test it with a user application written in C that wrote and read from the module using the device file.

Post-lab Deliverables:

1. [2 points.] The output of the picocom terminal



```
Writing 12 to register 0
Writing 9 to register 1
12 * 9 = 108
Result Correct!
Writing 12 to register 0
Writing 10 to register 1
12 * 10 = 120
Result Correct!
Writing 12 to register 0
Writing 11 to register 1
12 * 11 = 132
Result Correct!
Writing 12 to register 0
Writing 12 to register 1
12 * 12 = 144
Result Correct!
Writing 12 to register 0
Writing 13 to register 1
12 * 13 = 156
Result Correct!
Writing 12 to register 0
Writing 14 to register 1
12 * 14 = 168
Result Correct!
Writing 12 to register 0
Writing 15 to register 1
12 * 15 = 180
Result Correct!
Writing 12 to register 0
Writing 16 to register 1
12 * 16 = 192
Result Correct!
Writing 13 to register 0
Writing 0 to register 1
13 * 0 = 0
Result Correct!
Writing 13 to register 0
Writing 1 to register 1
13 * 1 = 13
Result Correct!
Writing 13 to register 0
Writing 2 to register 1
13 * 2 = 26
Result Correct!
Writing 13 to register 0
Writing 3 to register 1
13 * 3 = 39
Result Correct!
Writing 13 to register 0
Writing 4 to register 1
13 * 4 = 52
Result Correct!
Writing 13 to register 0
```

2. 4. [6 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? Answer: the operating system requires the use of virtual addresses. This helps the OS avoid addressing conflicts.
   (b) Do you think the overall wall-clock time required to perform a multiplication will be faster in this lab compared to the original Lab 3 implementation? Why? Answer: No, it won’t. Bare-metal implementation is faster since there is less abstraction. Less instructions means faster execution in most cases. This lab required the use of an operating system to manage read and write operations, while lab 3 was reading and writing directly to hardware.
   (c) Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach? Answer: speed is one benefit of not using an operating system. Operating systems require more memory, so using an OS might not be the best if you are memory constrained.
   (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. Answer: you register the device last because that signals to the OS that the driver is ready for read/write operations, and you don’t want to interface before it is ready. You unregister first because you don’t want any read/write to the driver while it is deallocating resources.

Appendix:

Devtest:

```
#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, i_read, j_read;
    int fd;   // file descriptor
    int i, j; // input variables
    char input = 0;
    unsigned int *inputs = (unsigned int *)malloc(2 * sizeof(int));
    char *outbuf = (char *)malloc(3 * sizeof(int));  //allocate dynamic memory
    int *charToint; // used to convert from char* to int* buffer
    // open device file for reading and writing
    fd = open("/dev/multiplier", O_RDWR);

    // error opening file descriptor
    if (fd == -1)
    {
        printf("Failed to open device file\n");
        return -1;
    }
```

```
while (input != 'q')
{ // continue until user enters 'q'
    for (i = 0; i <= 16; i++)
    {
        for (j = 0; j <= 16; j++)
        {
            inputs[0] = i;
            inputs[1] = j;
            char *buffer = (char *)inputs; // convert to char* buffer
            // write the values to the device file

            write(fd, buffer, 2 * sizeof(int));  //read to device file
            read(fd, outbuf, 3 * sizeof(int));
            // read the result from the device file
            // convert the char* buffer to int* buffer
            charToint = (int *)outbuf;
            i_read = charToint[0];
            j_read = charToint[1];
            result = charToint[2];

            printf("%u * %u = %u\n", i_read, j_read, result);
            if (result == (i * j))
            {
                printf("correct\n");
            }
            else
            {
                printf(“incorrect\n");
            }
        }
        }
        input = getchar();
    }
    close(fd);  //close device file
    free(inputs);  //free memory
    free(outbuf);
    return 0;
}
```

Multiplier.c

```
#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>        //needed for kmalloc() and kfree()
#include <asm/io.h>            //needed for IO reads and writes
#include <asm/uaccess.h>       // Provides utilities to bring user space

#include "xparameters.h" //needed for physical address of the multiplier

#define DEVICE_NAME "multiplier"
#define BUF_LEN 80
```

```
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR // physical address of
multiplier

// size of physical address range for 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 *);


static int Major;

/ Major number assigned to our device driver * /

   static int Device_Open = 0; // Flag to signify open device


void *virt_addr; // virtual address pointing to multiplier


static struct file_operations fops = {

   .read = device_read,

   .write = device_write,

   .open = device_open,

   .release = device_release};


// initialize the module

// called when the module is loaded

static int __init my_init(void)

{
   printk(KERN_INFO "Mapping virutal address...\n");
   // map virtual address to multiplier physical address//use ioremap
   virt_addr = ioremap(PHY_ADDR, MEMSIZE);
   printk("Physical Address: 0x%x\n", PHY_ADDR);
   printk("Virtual Address: 0x%x\n", virt_addr);

   Major = register_chrdev(0, DEVICE_NAME, &fops); // dynamic allocation

   /* Negative values indicate a problem */
   if (Major < 0)
   {
       printk(KERN_ALERT "Registering char device failed with %d\n", Major);
       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 */
}

// called when the module is unloaded
// called when the module is removed
static void __exit my_cleanup(void)
{
   // Unregister the device
   unregister_chrdev(Major, DEVICE_NAME);
   printk(KERN_ALERT "unmapping virtual address space...\n");
   iounmap((void *)virt_addr);
```

```
}

// called when a process opens the device file
static int device_open(struct inode *inode, struct file *file)
{
   printk(KERN_ALERT "Device has been 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 "Device has been closed\n");
   return 0;
}

// called when a process reads from the device file
static ssize_t device_read(struct file *filp,
                           char *buffer,
                           size_t length,
                           loff_t *offset)
{

   // bytes read from the buffer
   int bytes_read = 0;
   // allocating kernel buffer
   int *kBuff = (int *)kmalloc(length * sizeof(int), GFP_KERNEL);
   kBuff[0] = ioread32(virt_addr);
   kBuff[1] = ioread32(virt_addr + 4);
   kBuff[2] = ioread32(virt_addr + 8);
```

```
    char *kBuffer = (char *)kBuff; // bytes written one at a time

    int i;
    for (i = 0; i < length; i++)
    {                                   // read the buffer one byte at a time
        put_user(*(kBuffer++), buffer++); // char is one byte
        // put_user copies the data from kernel space to user space
        // buffer is the user space buffer
        // kBuffer is the kernel space buffer

        bytes_read++;
    }

    kfree(kBuff);

    // printk("bytes_read: %d\n", bytes_read);
    return bytes_read;
}

static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t *off)
{
    char *kBuff = (char *)kmalloc((len + 1) * sizeof(char), GFP_KERNEL);
    /* use kBuff to write one
        byte at a time from user buffer*/
    int i;
    for (i = 0; i < len; i++)
    {
        get_user(kBuff[i], buff++);
    }
    kBuff[len] = '\0';
    /* Convert kBuff to int* to write to
       multiply*/
    int *writeBuffer = (int *)kBuff;

    // write to register 0
    printk(KERN_INFO "Writing %d to register 0\n", writeBuffer[0]);
    iowrite32(writeBuffer[0], virt_addr + 0); // base address + offset

    // write to register 1
    printk(KERN_INFO "Writing %d to register 1\n", writeBuffer[1]);
    iowrite32(writeBuffer[1], virt_addr + 4);

    kfree(writeBuffer);
    return i; // number of bytes written
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kris Gavvala");
MODULE_DESCRIPTION("Multiplier Device Driver");

module_init(my_init);
module_exit(my_cleanup);
```

Multiplier.bb

```
SUMMARY = "Recipe for  build an external multiplier Linux kernel module"
```

```
SECTION = "PETALINUX/modules"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

INHIBIT_PACKAGE_STRIP = "1"

SRC_URI = "file://Makefile \
           file://multiplier.c \
              file://COPYING \
              file://xparameters.h \
              file://xparameters_ps.h \
          "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.
```