Kyle Loyka
Lab 6 Report

ECEN 449-503

Due: 7 March 2016

Introduction
The purpose of this lab was to write a device driver for a hardware multiplication device.

Procedure
In this lab, a `multiply` kernel module was created. This module simply multiplied the values "7" and "2". The purpose of this module was to demonstrate how to read and write to hardware, and how to load kernel modules into the kernel.

The second part of the lab involved creating the `multiplier` kernel module. This module was a character device driver, which allowed other programs running on the system to access the hardware device and perform read and write functions.

The `multiplier` kernel module was loaded into the XUP Linux kernel. Using the `devtest` program, integer values were written and read from the hardware device. The `devtest` program used the system calls `write()` and `read()` and `open()` and `close()`. These functions used the `multiplier` kernel module to "put" the C program integers into the hardware device. The program also output both the inputs and outputs to the hardware, and tested for correctness.

Results / Q&A
The project functioned as intended.

(A) The ioremap() command is necessary to map the physical address of the hardware into the virtual address space. It is important that the memory is mapped to the virtual address space; the kernel handles memory based on virtual addresses, and the processor reads or writes to memory using virtual addresses.

(B) The Lab 3 implementation would perform multiplication operations faster than the Lab 6 implementation. Part of this slowdown is due to translation times. The Lab 6 implementation does not have direct access to hardware. This means that virtual addresses must be translated to hardware addresses. The Lab 3 implementation deals directly with hardware read and write functions. The other part of this slowdown is due to process scheduling by the Linux kernel. When the `devtest` program was running in Lab 6, the Linux kernel had other processes running in the background. It is possible that this could have an impact on performance. In Lab 3, the test program was the only program running on the FPGA board.

Despite these considerations, on a large time scale (such as minutes or hours), there is little difference in the performance of the Lab 3 and Lab 6 implementations. A user probably wouldn't notice the difference.

(C) The benefit of the Lab 3 implementation is that the code is as close to the hardware as it can get. There is no operating system interference, i.e. there is nothing to manage write and read access, and there are no other processes running. The downside of this approach is that the developer doesn't have all of the "infrastructure" that an operating system provides. Instead, the developer is on there own to manage all system resources.

The benefit of the Lab 6 implementation is that reading and writing to the hardware is much easier (with the proper drivers installed). With driver support, the user doesn't have to worry about dealing with the hardware. This makes code easier to write and debug and also makes it more portable. The downside of this approach is that the direct access to hardware is lost.

(D) Device registration should be the last thing done in the initialization routine of a device driver. When a device is initialized you want to be sure that all necessary settings have been configured, such as memory allocation, before the device is registered and available for access by the user.

In this same way, a device should be un-registered before any other operations in the exit routine. Un-registering the device first ensures that the device cannot be accessed while it is being removed. During the exit and de-allocation process, the device may not have full functionality and is therefore unstable. The user should not interact with the device while it is in this state.

Conclusion

This lab gave great insight into how hardware and software interact. This lab highlighted how drivers are crucial for computing.

multiply.c

```
#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 */

/* from xparameters.h */
#define PHY_ADDR XPAR_MULTIPLY_0_BASEADDR // physical address of multiplier
/* size of physical adderss range for multiply */
#define MEMSIZE XPAR_MULTIPLY_0_HIGHADDR - XPAR_MULTIPLY_0_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 reserve resources used by the module. */
static int __init my_init(void) {
        /* Linux kernel's version of printf */
        printk(KERN_INFO "Mapping viritual address...\n");

        /* map virtual address to multiplier physical address */
        // use ioremap
        virt_addr = ioremap(PHY_ADDR, MEMSIZE);
        printk(KERN_INFO "Physical Address: %X", PHY_ADDR);
        printk(KERN_INFO "Virtual  Address: %X", virt_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");
        // use iowrite32
        iowrite32(2,virt_addr+4); // add 4 since byte addressing


        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 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 froa 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("Kyle Loyka");
MODULE_DESCRIPTION("Simnple multipler module");

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

multiplier.c

```
// multiplier.c - Character device module for multiplication 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
#include <asm/io.h>       //needed for IO read/write
#include <linux/moduleparam.h>//needed for module parameters
#include <linux/fs.h>     //file ops
#include <linux/sched.h>  //access to "current" processes structure
#include <asm/uaccess.h>  //utilites for userspace
#include "xparameters.h"  //physical multipler address
#include <linux/slab.h>


#define PHY_ADDR XPAR_MULTIPLY_0_BASEADDR
#define MEMSIZE XPAR_MULTIPLY_0_HIGHADDR - XPAR_MULTIPLY_0_BASEADDR + 1
#define DEVICE_NAME "multiplier"

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

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;
static int Major;

/*-------------------------------------------------------------------*/
/* Implementation */
/*-------------------------------------------------------------------*/

/* This structure defines the function pointers to our function for
   opening, closing, reading, and writing the device file. There are
   lots of other pointers in this strucutre 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 called when the module is loaded and
   registers a device for th driver to use */
static int __init my_init(void) {

      /* map virtual address to multiplier physical address */
      // use ioremap
```

```
	virt_addr = ioremap(PHY_ADDR, MEMSIZE);
	printk(KERN_INFO "Physical Address: %x", PHY_ADDR);
	printk(KERN_INFO "Virtual  Address: %x", virt_addr);

	/* 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 (i.e. buffers etc.) 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 and 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);
	    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
command:\n'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);

	return 0; // success
}


/* This function is called when the module is unloaded, it
   releases the device file */
static void __exit my_exit(void){
	/* Unregister the device */
	unregister_chrdev(Major, DEVICE_NAME);

	/* free memory */
	iounmap((void*)virt_addr);
}

/* Called when a process tries to open the device file, like
   "cat/dev/multiplier". Link to this function placed in file
   operations structure for our device file. */
static int device_open(struct inode *inode, struct file *file){

	/* In this case we are only allowing one process to hold
	   the device file open at a time */
	if (Device_Open)  /* Device_Open is the flag for the
	                     usage of the device file */
	      return -EBUSY;  /* Failure to open device is given
	                     back to userland program. */
	Device_Open++;          /* Keeps count of device opens */

	/* Create a string to output when the device is opened. This
```

```
            string is given to the user program in device_read. Note:
            we are using the "current" task structure which contains
            information about the process that opened the device file */

        try_module_get(THIS_MODULE); /* Incriment the module use count
                                  (make sure this is accurate or
                                  you won't be able to remove
                                  the module later */

        return 0;
}


/* Called when a process closes the device file */
static int device_release(struct inode *inode, struct file *file){
        Device_Open--;          /* We're now ready for our next caller */

        /* Decrement the usage count, or else once you opened the file,
           you'll never get rid of the module. */
        module_put(THIS_MODULE);

        return 0;
}


/* Called when a process, which already opened the dev file, attempts
   to read from it */
static ssize_t device_read(struct file *filep,  /* 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;

        /* Buffer protections: integer size is 4 bytes. Since our
           hardware has support for 3 integer registers, the total
           buffer memory size should range from 0*4 -to-  (3 registers) *(4
bytes) */
        if (length < 0 || length > 12) {
                printk(KERN_INFO "Invalid buffer length\n");
                return -1;
        }
        /* The buffer is in the user data segment, not the kernel segment
                so "*" assignment won't work. We ahve to use put_user which
                copies data from the kernel data segment to the user data
segment */
        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;
}


/* Called when a process writes to dev file */
static ssize_t device_write(struct file *filp, const char* buffer, size_t
length, loff_t* offset){


	char* char_buf = (char*)kmalloc(length*sizeof(char),GFP_KERNEL);

	int i;
	for(i=0; i<length; i++)
		get_user(char_buf[i],buffer+i);

	int* buf = (int*)char_buf;

	iowrite32(buf[0], virt_addr+0);
	iowrite32(buf[1], virt_addr+4);

	kfree(char_buf);

	return i;
}




/* These define info that can be displatyed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kyle Loyka");
MODULE_DESCRIPTION("Module which creates a character device and allows user
interaction with it");

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

Kermit output part 1

```
# insmod multiply.ko
Mapping viritual address...
Physical Address: C0A00000
Virtual  Address: D0040000
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
#
```

Kermit output part 2

```
# insmod multiplier.ko
Physical Address: c0a00000
Virtual  Address: d0040000
Registered a device with dynamic Major number of 254
Create a device file for this device with command:
'mknod /dev/multiplier c 254 0'.
# mknod /dev/multiplier c 254 0
# dmesg
Linux version 2.6.35.7 (kyleloyka@lin06-424cvlb.ece.tamu.edu) (gcc version 4.1.2) #3 Mon Feb 22 09:39:27 CST
2016
setup_cpuinfo: initialising
setup_cpuinfo: No PVR support. Using static CPU info from FDT
cache: wt_msr
setup_memory: max_mapnr: 0x10000
setup_memory: min_low_pfn: 0x90000
setup_memory: max_low_pfn: 0xa0000
On node 0 totalpages: 65536
free_area_init_node: node 0, pgdat c01f0518, node_mem_map c0e02000
  Normal zone: 512 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 65024 pages, LIFO batch:15
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 65024
Kernel command line: console=ttyUL0 root=/dev/ram
PID hash table entries: 1024 (order: 0, 4096 bytes)
Dentry cache hash table entries: 32768 (order: 5, 131072 bytes)
Inode-cache hash table entries: 16384 (order: 4, 65536 bytes)
Memory: 245292k/262144k available
Hierarchical RCU implementation.
	RCU-based detection of stalled CPUs is disabled.
	Verbose stalled-CPUs detection is disabled.
NR_IRQS:32
xlnx,xps-intc-1.00.a #0 at 0xd0000000, num_irq=3, edge=0x2
xlnx,xps-timer-1.00.a #0 at 0xd0004000, irq=0
microblaze_timer_set_mode: shutdown
microblaze_timer_set_mode: periodic
Calibrating delay loop... 36.45 BogoMIPS (lpj=182272)
pid_max: default: 4096 minimum: 301
Mount-cache hash table entries: 512
```

```
bio: create slab &lt;bio-0&gt; at 0
Switching to clocksource microblaze_clocksource
Skipping unavailable RESET gpio -2 (reset)
GPIO pin is already allocated
msgmni has been set to 479
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
84000000.serial: ttyUL0 at MMIO 0x84000000 (irq = 1) is a uartlite
console [ttyUL0] enabled
brd: module loaded
xsysace 83600000.sysace: Xilinx SystemACE revision 1.0.12
xsysace 83600000.sysace: capacity: 1957536 sectors
 xsa: xsa1
Xilinx SystemACE device driver, major=254
i2c /dev entries driver
Freeing unused kernel memory: 12187k freed
Physical Address: c0a00000
Virtual  Address: d0040000
Registered a device with dynamic Major number of 254
Create a device file for this device with command:
'mknod /dev/multiplier c 254 0'.
```

Kermit output part 3

.... program started with 0*0,0*1, etc, but the console history doesn't go back that far.

```
Read 12 bytes
13 * 7 = 91  Result Correct   **
Read 12 bytes
13 * 8 = 104  Result Correct   **
Read 12 bytes
13 * 9 = 117  Result Correct   **
Read 12 bytes
13 * 10 = 130  Result Correct   **
Read 12 bytes
13 * 11 = 143  Result Correct   **
Read 12 bytes
13 * 12 = 156  Result Correct   **
Read 12 bytes
13 * 13 = 169  Result Correct   **
Read 12 bytes
13 * 14 = 182  Result Correct   **
Read 12 bytes
13 * 15 = 195  Result Correct   **
Read 12 bytes
13 * 16 = 208  Result Correct   **
Read 12 bytes
14 * 0 = 0  Result Correct   **
Read 12 bytes
```

14 * 1 = 14 Result Correct   **
Read 12 bytes
14 * 2 = 28 Result Correct   **
Read 12 bytes
14 * 3 = 42 Result Correct   **
Read 12 bytes
14 * 4 = 56 Result Correct   **
Read 12 bytes
14 * 5 = 70 Result Correct   **
Read 12 bytes
14 * 6 = 84 Result Correct   **
Read 12 bytes
14 * 7 = 98 Result Correct   **
Read 12 bytes
14 * 8 = 112 Result Correct   **
Read 12 bytes
14 * 9 = 126 Result Correct   **
Read 12 bytes
14 * 10 = 140 Result Correct   **
Read 12 bytes
14 * 11 = 154 Result Correct   **
Read 12 bytes
14 * 12 = 168 Result Correct   **
Read 12 bytes
14 * 13 = 182 Result Correct   **
Read 12 bytes
14 * 14 = 196 Result Correct   **
Read 12 bytes
14 * 15 = 210 Result Correct   **
Read 12 bytes
14 * 16 = 224 Result Correct   **
Read 12 bytes
15 * 0 = 0 Result Correct   **
Read 12 bytes
15 * 1 = 15 Result Correct   **
Read 12 bytes
15 * 2 = 30 Result Correct   **
Read 12 bytes
15 * 3 = 45 Result Correct   **
Read 12 bytes
15 * 4 = 60 Result Correct   **
Read 12 bytes
15 * 5 = 75 Result Correct   **
Read 12 bytes
15 * 6 = 90 Result Correct   **
Read 12 bytes
15 * 7 = 105 Result Correct   **
Read 12 bytes
15 * 8 = 120 Result Correct   **
Read 12 bytes
15 * 9 = 135 Result Correct   **
Read 12 bytes
15 * 10 = 150 Result Correct   **
Read 12 bytes
15 * 11 = 165 Result Correct   **
Read 12 bytes

```
15 * 12 = 180  Result Correct   **
Read 12 bytes
15 * 13 = 195  Result Correct   **
Read 12 bytes
15 * 14 = 210  Result Correct   **
Read 12 bytes
15 * 15 = 225  Result Correct   **
Read 12 bytes
15 * 16 = 240  Result Correct   **
Read 12 bytes
16 * 0 = 0  Result Correct   **
Read 12 bytes
16 * 1 = 16  Result Correct   **
Read 12 bytes
16 * 2 = 32  Result Correct   **
Read 12 bytes
16 * 3 = 48  Result Correct   **
Read 12 bytes
16 * 4 = 64  Result Correct   **
Read 12 bytes
16 * 5 = 80  Result Correct   **
Read 12 bytes
16 * 6 = 96  Result Correct   **
Read 12 bytes
16 * 7 = 112  Result Correct   **
Read 12 bytes
16 * 8 = 128  Result Correct   **
Read 12 bytes
16 * 9 = 144  Result Correct   **
Read 12 bytes
16 * 10 = 160  Result Correct   **
Read 12 bytes
16 * 11 = 176  Result Correct   **
Read 12 bytes
16 * 12 = 192  Result Correct   **
Read 12 bytes
16 * 13 = 208  Result Correct   **
Read 12 bytes
16 * 14 = 224  Result Correct   **
Read 12 bytes
16 * 15 = 240  Result Correct   **
Read 12 bytes
16 * 16 = 256  Result Correct   **
q
#
```