ECEN 749 Lab 6 Report

Tong Lu
UIN 621007982

October 19, 2018

TEXAS A&M
UNIVERSITY®

Introduction

In this lab, we created a device driver for embedded Linux system. We started with some kernel module examples, and multiply.c from lab 5, and created a complete character device driver.

Procedure

1. Examine my_chardev.c, my_charmem.c and their header files in '/home/faculty/shared/ECEN449/module examples.

2. Based on the example source code and multiply.c, create the device driver multiplier.c and multiplier.h (see Appendix)

3. Modify the makefile and cross compile (see Appendix):

```
make ARCH=arm CROSS COMPILE=arm-xilinx-linux-gnueabi-
```

4. Copy the generated multiplier.ko file to the SD card and mount the SD card on FPGA:

```
mount /dev/mmcb1k0p1 /mnt/
```

5. load the module into Linux kernel on ZYBO board:

```
insmod multiplier.ko
```

6. Check the output *printk* statement using *dmesg*:

```
dmesg
```

7. Complete devtest.c (see Appendix) and cross compile.

```
arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c
```

8. Copy the executable file to SD card, and execute the application:

```
mount /dev/mmcb1k0p1 /mnt/
insmod /mnt/multiplier.ko
```

9. Following the same step above, create a multipy.c (see Appendix), modify makefile (see Appendix) and cross-compile (see Appendix):

```
make ARCH=arm CROSS COMPILE=arm-xilinx-linux-gnueabi-
```

10. Copy the generated module multiply.ko on SD card, and load the module on Linux kernel on ZYBO:

```
mount /dev/mmcb1k0p1 /mnt/
insmod /mnt/multiply.ko
mknod /dev/multiplier c 245 0
```

11. run the devtest executable file and demo the result to TA:

```
/mnt/.devtest
```

Result

All the programs was finished and demonstrated to TA. The programs are working well and meet all the requirement on lab manual.

```
zynq> /mnt/devtest
Multiplier is opened
0 * 0 = 0
Result Correct!
0 * 1 = 0
Result Correct!
0 * 2 = 0
Result Correct!
0 * 3 = 0
Result Correct!
0 * 4 = 0
Result Correct!
0 * 5 = 0
Result Correct!
0 * 6 = 0
Result Correct!
0 * 7 = 0
Result Correct!
0 * 8 = 0
Result Correct!
0 * 9 = 0
Result Correct!
0 * 10 = 0
Result Correct!
0 * 11 = 0
Result Correct!
0 * 12 = 0
Result Correct!
0 * 13 = 0
Result Correct!
0 * 14 = 0
Result Correct!
0 * 15 = 0
Result Correct!
0 * 16 = 0
Result Correct!
1 * 0 = 0
Result Correct!
1 * 1 = 1
Result Correct!
```

```
1 * 2 = 2
Result Correct!
1 * 3 = 3
Result Correct!
1 * 4 = 4
Result Correct!
1 * 5 = 5
Result Correct!
1 * 6 = 6
Result Correct!
1 * 7 = 7
Result Correct!
1 * 8 = 8
Result Correct!
1 * 9 = 9
Result Correct!
1 * 10 = 10
Result Correct!
1 * 11 = 11
Result Correct!
1 * 12 = 12
Result Correct!
1 * 13 = 13
Result Correct!
1 * 14 = 14
Result Correct!
1 * 15 = 15
Result Correct!
1 * 16 = 16
Result Correct!
2 * 0 = 0
Result Correct!
2 * 1 = 2
Result Correct!
2 * 2 = 4
Result Correct!
2 * 3 = 6
Result Correct!
2 * 4 = 8
Result Correct!
2 * 5 = 10
Result Correct!
2 * 6 = 12
Result Correct!
2 * 7 = 14
Result Correct!
2 * 8 = 16
Result Correct!
2 * 9 = 18
Result Correct!
2 * 10 = 20
Result Correct!
2 * 11 = 22
Result Correct!
2 * 12 = 24
Result Correct!
2 * 13 = 26
Result Correct!
2 * 14 = 28
```

```
Result Correct!
2 * 15 = 30
Result Correct!
2 * 16 = 32
Result Correct!
3 * 0 = 0
Result Correct!
3 * 1 = 3
Result Correct!
3 * 2 = 6
Result Correct!
3 * 3 = 9
Result Correct!
3 * 4 = 12
Result Correct!
3 * 5 = 15
Result Correct!
3 * 6 = 18
Result Correct!
3 * 7 = 21
Result Correct!
3 * 8 = 24
Result Correct!
3 * 9 = 27
Result Correct!
3 * 10 = 30
Result Correct!
3 * 11 = 33
Result Correct!
3 * 12 = 36
Result Correct!
3 * 13 = 39
Result Correct!
3 * 14 = 42
Result Correct!
3 * 15 = 45
Result Correct!
3 * 16 = 48
Result Correct!
4 * 0 = 0
Result Correct!
4 * 1 = 4
Result Correct!
4 * 2 = 8
Result Correct!
4 * 3 = 12
Result Correct!
4 * 4 = 16
Result Correct!
4 * 5 = 20
Result Correct!
4 * 6 = 24
Result Correct!
4 * 7 = 28
Result Correct!
4 * 8 = 32
Result Correct!
4 * 9 = 36
Result Correct!
```

```
4 * 10 = 40
Result Correct!
4 * 11 = 44
Result Correct!
4 * 12 = 48
Result Correct!
4 * 13 = 52
Result Correct!
4 * 14 = 56
Result Correct!
4 * 15 = 60
Result Correct!
4 * 16 = 64
Result Correct!
5 * 0 = 0
Result Correct!
5 * 1 = 5
Result Correct!
5 * 2 = 10
Result Correct!
5 * 3 = 15
Result Correct!
5 * 4 = 20
Result Correct!
5 * 5 = 25
Result Correct!
5 * 6 = 30
Result Correct!
5 * 7 = 35
Result Correct!
5 * 8 = 40
Result Correct!
5 * 9 = 45
Result Correct!
5 * 10 = 50
Result Correct!
5 * 11 = 55
Result Correct!
5 * 12 = 60
Result Correct!
5 * 13 = 65
Result Correct!
5 * 14 = 70
Result Correct!
5 * 15 = 75
Result Correct!
5 * 16 = 80
Result Correct!
6 * 0 = 0
Result Correct!
6 * 1 = 6
Result Correct!
6 * 2 = 12
Result Correct!
6 * 3 = 18
Result Correct!
6 * 4 = 24
Result Correct!
6 * 5 = 30
```

```
Result Correct!
6 * 6 = 36
Result Correct!
6 * 7 = 42
Result Correct!
6 * 8 = 48
Result Correct!
6 * 9 = 54
Result Correct!
6 * 10 = 60
Result Correct!
6 * 11 = 66
Result Correct!
6 * 12 = 72
Result Correct!
6 * 13 = 78
Result Correct!
6 * 14 = 84
Result Correct!
6 * 15 = 90
Result Correct!
6 * 16 = 96
Result Correct!
7 * 0 = 0
Result Correct!
7 * 1 = 7
Result Correct!
7 * 2 = 14
Result Correct!
7 * 3 = 21
Result Correct!
7 * 4 = 28
Result Correct!
7 * 5 = 35
Result Correct!
7 * 6 = 42
Result Correct!
7 * 7 = 49
Result Correct!
7 * 8 = 56
Result Correct!
7 * 9 = 63
Result Correct!
7 * 10 = 70
Result Correct!
7 * 11 = 77
Result Correct!
7 * 12 = 84
Result Correct!
7 * 13 = 91
Result Correct!
7 * 14 = 98
Result Correct!
7 * 15 = 105
Result Correct!
7 * 16 = 112
Result Correct!
8 * 0 = 0
Result Correct!
```

```
8 * 1 = 8
Result Correct!
8 * 2 = 16
Result Correct!
8 * 3 = 24
Result Correct!
8 * 4 = 32
Result Correct!
8 * 5 = 40
Result Correct!
8 * 6 = 48
Result Correct!
8 * 7 = 56
Result Correct!
8 * 8 = 64
Result Correct!
8 * 9 = 72
Result Correct!
8 * 10 = 80
Result Correct!
8 * 11 = 88
Result Correct!
8 * 12 = 96
Result Correct!
8 * 13 = 104
Result Correct!
8 * 14 = 112
Result Correct!
8 * 15 = 120
Result Correct!
8 * 16 = 128
Result Correct!
9 * 0 = 0
Result Correct!
9 * 1 = 9
Result Correct!
9 * 2 = 18
Result Correct!
9 * 3 = 27
Result Correct!
9 * 4 = 36
Result Correct!
9 * 5 = 45
Result Correct!
9 * 6 = 54
Result Correct!
9 * 7 = 63
Result Correct!
9 * 8 = 72
Result Correct!
9 * 9 = 81
Result Correct!
9 * 10 = 90
Result Correct!
9 * 11 = 99
Result Correct!
9 * 12 = 108
Result Correct!
9 * 13 = 117
```

```
Result Correct!
9 * 14 = 126
Result Correct!
9 * 15 = 135
Result Correct!
9 * 16 = 144
Result Correct!
10 * 0 = 0
Result Correct!
10 * 1 = 10
Result Correct!
10 * 2 = 20
Result Correct!
10 * 3 = 30
Result Correct!
10 * 4 = 40
Result Correct!
10 * 5 = 50
Result Correct!
10 * 6 = 60
Result Correct!
10 * 7 = 70
Result Correct!
10 * 8 = 80
Result Correct!
10 * 9 = 90
Result Correct!
10 * 10 = 100
Result Correct!
10 * 11 = 110
Result Correct!
10 * 12 = 120
Result Correct!
10 * 13 = 130
Result Correct!
10 * 14 = 140
Result Correct!
10 * 15 = 150
Result Correct!
10 * 16 = 160
Result Correct!
11 * 0 = 0
Result Correct!
11 * 1 = 11
Result Correct!
11 * 2 = 22
Result Correct!
11 * 3 = 33
Result Correct!
11 * 4 = 44
Result Correct!
11 * 5 = 55
Result Correct!
11 * 6 = 66
Result Correct!
11 * 7 = 77
Result Correct!
11 * 8 = 88
Result Correct!
```

```
11 * 9 = 99
Result Correct!
11 * 10 = 110
Result Correct!
11 * 11 = 121
Result Correct!
11 * 12 = 132
Result Correct!
11 * 13 = 143
Result Correct!
11 * 14 = 154
Result Correct!
11 * 15 = 165
Result Correct!
11 * 16 = 176
Result Correct!
12 * 0 = 0
Result Correct!
12 * 1 = 12
Result Correct!
12 * 2 = 24
Result Correct!
12 * 3 = 36
Result Correct!
12 * 4 = 48
Result Correct!
12 * 5 = 60
Result Correct!
12 * 6 = 72
Result Correct!
12 * 7 = 84
Result Correct!
12 * 8 = 96
Result Correct!
12 * 9 = 108
Result Correct!
12 * 10 = 120
Result Correct!
12 * 11 = 132
Result Correct!
12 * 12 = 144
Result Correct!
12 * 13 = 156
Result Correct!
12 * 14 = 168
Result Correct!
12 * 15 = 180
Result Correct!
12 * 16 = 192
Result Correct!
13 * 0 = 0
Result Correct!
13 * 1 = 13
Result Correct!
13 * 2 = 26
Result Correct!
13 * 3 = 39
Result Correct!
13 * 4 = 52
```

```
Result Correct!
13 * 5 = 65
Result Correct!
13 * 6 = 78
Result Correct!
13 * 7 = 91
Result Correct!
13 * 8 = 104
Result Correct!
13 * 9 = 117
Result Correct!
13 * 10 = 130
Result Correct!
13 * 11 = 143
Result Correct!
13 * 12 = 156
Result Correct!
13 * 13 = 169
Result Correct!
13 * 14 = 182
Result Correct!
13 * 15 = 195
Result Correct!
13 * 16 = 208
Result Correct!
14 * 0 = 0
Result Correct!
14 * 1 = 14
Result Correct!
14 * 2 = 28
Result Correct!
14 * 3 = 42
Result Correct!
14 * 4 = 56
Result Correct!
14 * 5 = 70
Result Correct!
14 * 6 = 84
Result Correct!
14 * 7 = 98
Result Correct!
14 * 8 = 112
Result Correct!
14 * 9 = 126
Result Correct!
14 * 10 = 140
Result Correct!
14 * 11 = 154
Result Correct!
14 * 12 = 168
Result Correct!
14 * 13 = 182
Result Correct!
14 * 14 = 196
Result Correct!
14 * 15 = 210
Result Correct!
14 * 16 = 224
Result Correct!
```

```
15 * 0 = 0
Result Correct!
15 * 1 = 15
Result Correct!
15 * 2 = 30
Result Correct!
15 * 3 = 45
Result Correct!
15 * 4 = 60
Result Correct!
15 * 5 = 75
Result Correct!
15 * 6 = 90
Result Correct!
15 * 7 = 105
Result Correct!
15 * 8 = 120
Result Correct!
15 * 9 = 135
Result Correct!
15 * 10 = 150
Result Correct!
15 * 11 = 165
Result Correct!
15 * 12 = 180
Result Correct!
15 * 13 = 195
Result Correct!
15 * 14 = 210
Result Correct!
15 * 15 = 225
Result Correct!
15 * 16 = 240
Result Correct!
16 * 0 = 0
Result Correct!
16 * 1 = 16
Result Correct!
16 * 2 = 32
Result Correct!
16 * 3 = 48
Result Correct!
16 * 4 = 64
Result Correct!
16 * 5 = 80
Result Correct!
16 * 6 = 96
Result Correct!
16 * 7 = 112
Result Correct!
16 * 8 = 128
Result Correct!
16 * 9 = 144
Result Correct!
16 * 10 = 160
Result Correct!
16 * 11 = 176
Result Correct!
16 * 12 = 192
```

```
Result Correct!
16 * 13 = 208
Result Correct!
16 * 14 = 224
Result Correct!
16 * 15 = 240
Result Correct!
16 * 16 = 256
Result Correct!
Multiplier is closed
```

src/printout

.

Conclusion

In this lab, I learned to build a simple char device driver based on a simple Linux kernel module. This lab helped me understand the mechanism of kernel module and Linux device drive, and these knowledge will be very benefit in the future lab.

Answer to 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?** When running a program, the processor always deals with physical address, while the kernel uses virtual address. For processor and operating system to understand each other, the ioremap is required: it will map the physical address of the hardware into the virtual address space.

(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. The overall time to perform a multiplication in lab 6 would be slower than lab 3. There are two reasons.
First, translation of address space takes time. The program in lab 3 runs directly on top of the arm processor (which uses hardware address), while the program in lab 6 runs on top on a Linux system (which apparently uses virtual address). The memory access in devtest has to be translated to physical address in operating system so that the processor can understand it. Lab 3 doesn't need this step, so it is faster.
Secondly, process scheduling takes time. In lab 3, the hello.c was the only program running on the procressor at the time; however, in lab 6 the operating has multiple processes running in background while running the devtest. Switching between processes takes time, too.
However, since the arm processor very powerful and running at a very fast clock rate, I don't think the user will notice a difference on running time between lab 3 and lab 6. However, when the program gets bigger, the overall running time difference will be very noticeable.

(c) **Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach?** The comparison between lab 3 an lab 6 is a perfect example of the trade-off between performance and development time.
The benefits of lab 3: great performance, fast running time, low memory cost, small size... However, the program in lab 3 requires a deep understanding of the hardware platform, and it's hard to develop and maintain.
The benefits of lab 6: easy to develop and maintain (the developer just need to understand the Linux system and C), easy to port to other hardware platform (as far as the hardware is able to run Linux OS). However, the cost of this approach is total running time: the performance and size of the program in lab 6 can never catch its counterparts in lab 3.

(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.** Device registration should be the last thing that is done in the initialization routine of a device driver. This is because we should prepare everything for the device before its registration, as memory allocation, ioremap, etc. In this way, the program won't be able access the device during the initialization step, and this can prevent a lot of potential bugs.

On the other hand, device-unregistering should be happen first in the exit routine. We want to unregister the device first, and then finish other clean-ups. In this way, the program won't be able to access the device during the clean-up, and this can prevent a lot of potential bugs.

Appendix

```c
/* Moved all prototypes and includes into the headerfile */
#include "multiplier.h"


/*
 * This function is called when the module is loaded and registers a
 * device for the driver to use.
 */
int my_init(void) {

    virt_addr = (void*)ioremap(PHY_ADDR, MEMSIZE); // map physical address with virtual address
    bf_Ptr = (int*)kmalloc(INT_BUF_LEN*sizeof(int), GFP_KERNEL); // allocate the buffer for reading/writing data
    char_bf_Ptr = (char*)bf_Ptr; // cast the pointer to char type
    /* 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); // register device

      /* Negative values indicate a problem */
    if (Major < 0) { // handle error
        printk(KERN_ALERT "Registering multiply 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 */
}

/*
 * This function is called when the module is unloaded, it releases
 * the device file.
 */
void my_cleanup(void)
{
    /*
     * Unregister the device
     */
    //print out a message for cleaning up
    printk(KERN_ALERT "unmapping virtual address space...\n");
    unregister_chrdev(Major, DEVICE_NAME); // unregister the char device
    iounmap((void*)virt_addr); // unmap the virtual aaddress
    kfree(bf_Ptr); // free the buffer memory

}

/*
 * Do nothing except print to the kernel message buffer informing
 * the user when the device is opened.
 */
```

```c
static int device_open(struct inode *inode, struct file *file) {
    // print out a message when multiplier is opened
    printk(KERN_INFO "Multiplier is opened\n");
    return 0;
}

/*
 * Do nothing except print to the kernel message buffer informing
 * the user when the device is closed.
 */
static int device_release(struct inode *inode, struct file *file) {
    // print out a message when multiplier is closed
    printk(KERN_INFO "Multiplier 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)
{
    int i, bytes_read; // declear increament and byte read varible
    /*
     * Valid value for length parameter include 0 through 12.
     */
    // printk(KERN_INFO "Reading multiplier...");
    if (length > 12){ // doesn't allow reading more than 12 bytes
        printk(KERN_ALERT "Sorry, reading more than 12 bytes isn't supported.\n");
        return -EINVAL;
    }

    /*
     * Number of bytes actually written to the buffer
     */
    //int bytes_read = 0;

    for (i = 0; i < 3; i++) // read three bytes from multiplier
        bf_Ptr[i] = ioread32(virt_addr + i*sizeof(int));

    //printk(KERN_INFO "READ: bf_Ptr[0] = %d, bf_Ptr[1] = %d, bf_Ptr[2] = %d\n",
    bf_Ptr[0], bf_Ptr[1], bf_Ptr[2]);
    char_bf_Ptr = (char*)bf_Ptr; // cast the buffer pointer to char type


    /*
     * Actually put the data into the buffer
     */
    bytes_read = 0; // start from zero
    while (bytes_read < 12) { // read 12 bytes

        /*
         * 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.
         */
        //printk(KERN_INFO "put_user: char = %d\n", char_bf_Ptr[0]);
        put_user(*(char_bf_Ptr++), buffer++); /* one char at a time... */

        bytes_read++; // increment
    }

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


static ssize_t
device_write(struct file *filp,
             const char *buffer,
             size_t length,
             loff_t * offset)
{
    int i; // declearation for incrementor
    // printk(KERN_INFO "Writing multiplier...");
    if (length > 8){ // doesn't support more than 8 bytes
        printk(KERN_ALERT "Sorry, writing more than 8 bytes isn't supported.\n");
        return -EINVAL; // throw an error message
    }

    for (i = 0; i < length; i++) // 12 chars
        get_user(char_bf_Ptr[i], buffer++); // write to kernel
    bf_Ptr = (int*) char_bf_Ptr; // cast to int type

    for (i = 0; i < 2; i++) // 3 ints
        iowrite32(bf_Ptr[i], virt_addr + i * sizeof(int)); // write to register

    //printk(KERN_INFO "WRITE: bf_Ptr[0] = %d, bf_Ptr[1] = %d\n", bf_Ptr[0], bf_Ptr[1]);

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


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

src/multiplier.c

```
/* 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/ioport.h>   // Used for io memory allocation
#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"    // Needed for IO reads and writes
#include "xparameters_ps.h"    // Needed for IO reads and writes



/* Some defines */
#define DEVICE_NAME "multiplier"

// From xparameters.h, physical address of multiplier
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR
// Size of physical address range for multiply
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR +
    1


// Integer Buffer length
#define INT_BUF_LEN 3
/* 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 void* virt_addr;
static int* bf_Ptr;
static char* char_bf_Ptr;
/* 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
};
```

src/multiplier.h

```
obj-m += multiplier.o

all:
    make -C /home/grads/l/lvtongtom305/ecen749/lab5/linux-3.14 M=$(PWD) modules

clean:
    make -C /home/grads/l/lvtongtom305/ecen749/lab5/linux-3.14 M=$(PWD) clean
```

src/Makefile

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

int main() {
    unsigned int read_i, read_j, result;
    int fd;     // File descriptor
    int i, j;   // Loop variables
    // allocate read buffer
    char* rd_buf = (char*) malloc(3 * sizeof(int));
    // allocate write buffer
    int* wr_buf = (int*) malloc(2 * sizeof(int));
    unsigned int* int_rd_buf; // int pointer for read buffer
    char input = 0; // input from user

    // Open device file for reading and writing
    // Use 'open' to open '/dev/multiplier'
    fd = open("/dev/multiplier", O_RDWR);
    // Handle error opening file
    if(fd == -1) {
        printf("Failed to open device file!\n");
        return -1;
    }

    for(i = 0; i <= 16; i++) {
        for(j = 0; j <= 16; j++) {
            // Write value to registers using char dev
            // Use write to write i and j to peripheral
            wr_buf[0] = i; // assign to write buffer
            wr_buf[1] = j; // assign to write buffer
            write(fd, (char*)wr_buf, 2 * sizeof(int)); // write to device
            // Read i, j, and result using char dev
            // Use read to read from peripheral
            read(fd, rd_buf, 3 * sizeof(int)); // read from device
            int_rd_buf = (unsigned int*) rd_buf; // cast to unsigned int
            read_i = int_rd_buf[0]; // assign to read buffer
            read_j = int_rd_buf[1]; // assign to read buffer
            result = int_rd_buf[2]; // assign to read buffer
            // print unsigned ints to screen
            printf("%u * %u = %u\n\r", read_i, read_j, result);

            // Validate result
```

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

            // Read from terminal
            input = getchar();
            // Continue unless user entered 'q'
            if(input == 'q') {
                close(fd);
                return 0;
            }
        }
    }
    close(fd); // close device
    free(wr_buf); // free write buffer
    free(rd_buf); // free read buffer
    return 0;
}
```

src/devtest.c