ECEN 749- Microprocessor System Design

Section 603

TA: Mr. Kunal Bharathi

LAB 6

An Introduction to Character Device Driver Development

Date of Performance: 10/10/2019

Student Name: Dhiraj Dinesh Kudva

UIN: 829009538

Introduction

The purpose of this lab to develop a character device driver in the embedded Linux environment. This device driver serves as the bridge between the user application and hardware device. This lab will also focus on creating a simple Linux application to test the device driver.

Design:

The below diagram represents a hardware and software system diagram. The multiplication peripheral is the hardware attached to the ARM processor. It communicates with the kernel module. The kernel module represents the character device driver executing in kernel space, while the user application represents the code executing in user space, reading and writing to the device file, '/dev/multiplier'. The device file is not a standard file in the sense that it does not reside on a disk, rather it provides an interface for user applications to interact with the kernel.

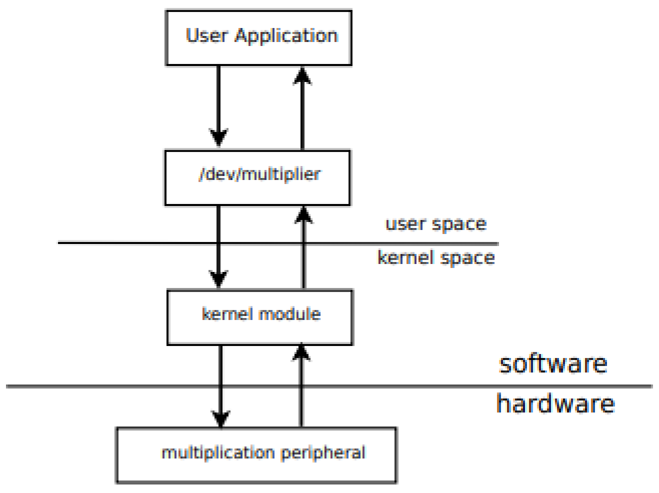


Figure 1: Hardware/Software System Diagram

Part 1: Creation of character device driver

1. In this lab, a character device driver called 'multiplier.c' is to be created.
2. The multiplier.c function will have the following functions: init, cleanup, device_open, device_release, device_read and device_write.
3. init is the initialization function. It maps the virtual address of the memory and also provide major and minor device numbers. Major device number identifies the driver associated with the device and minor number is used by the major number to differentiate between different processes.

4. In the initialization/ init routine, the device name must also be registered.
5. The major number is assigned by the Linux driver.
6. Handle the device registration error, if any, carefully.
7. In the exit routine, unregister the device driver before the virtual memory unmapping.
8. For the open and close functions, do nothing except print to the kernel message buffer informing the user when the device is opened and closed.
9. Put_user and get_user functions are written inside the device_read and device_write functions respectively. They used to get the data stored by the user application in the buffer and to write the data into the buffer respectively.
10. The makefile is modified to run the multiplier.c and run the following command.
    ```
    >make ARCH=arm CROSS COMPILE=arm-xilinx-linux-gnueabi
    ```
    This will create multiplier.ko module .
11. Load multiplier.ko on the ZYBO Z7-10 board.
12. Use 'dmesg' to view the output of the kernel module after it has been loaded. Follow the instructions provided within the provided example kernel modules to create the '/dev/multiplier 'device node.

<u>Part 2: Creating user application to test the device driver</u>.

1. To test the character device driver, a application file is written.
2. This user application reads and writes to the device file, '/dev/multiplier'.
3. Write the .c code as given below (Devtest.c)
4. Compile 'devtest.c' by executing the following command in the terminal window under the 'modules' directory:
   ```
   >arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c
   ```
   This will create an executable file named devtest.
5. Now copy this file in the SD card, mount it on the ZYBO board and then run
   . /devtest.
   This will produce the result. Compare the multiplication result and if the answer is correct, it will show as correct answer.

Program:

Multiplier.c

```
/*  my_chardev.c - Simple character device module
 *
 *  Demonstrates module creation of character device for user
 *  interaction.
 *
 *  (Adapted from various example modules including those found in the
 *  Linux Kernel Programming Guide, Linux Device Drivers book and
 *  FSM's device driver tutorial)
 */
```

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


/*
 * This function is called when the module is loaded and registers a
 * device for the driver to use.
 */
// 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

// virtual address pointing to multiplier
void* virt_addr;

int my_init(void) //when you call insmod
{
   // Linux kernel's version of printf
   printk(KERN_INFO "Mapping virtual address...\n");
   //virt_addr=PHY_ADDR;
   // map virtual address to multiplier physical address
   // use ioremap, print the physical and virtual address
   virt_addr=ioremap(PHY_ADDR, 4);
   printk("The address of virt is %p and of physical is %d.", virt_addr , PHY_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 (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);
    return Major;
  }

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

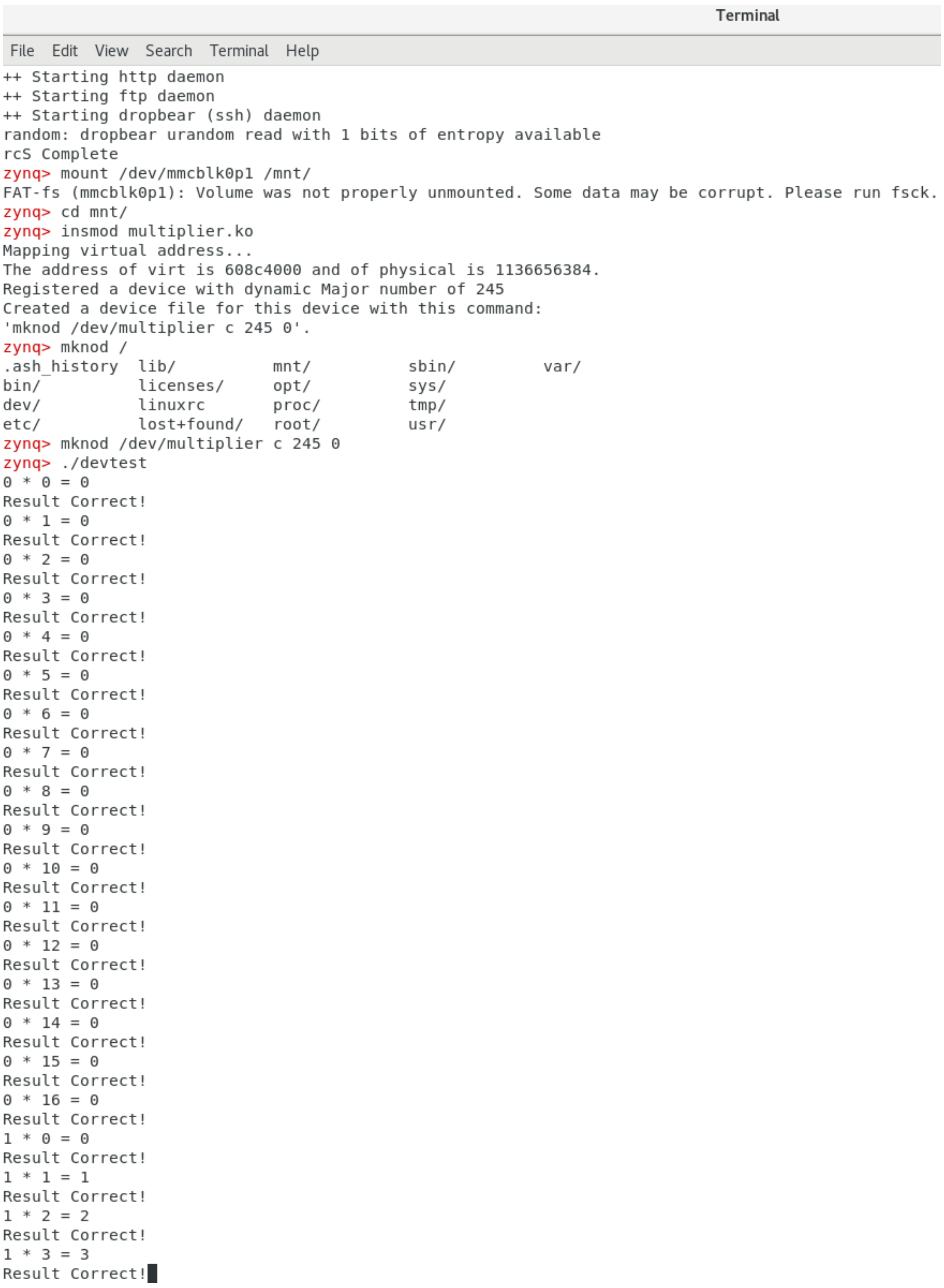
```
  printk(KERN_INFO "Created a device file for this device with this
command:\n'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);

  return 0;                  /* success */
}

/*
 * This function is called when the module is unloaded, it releases
 * the device file.
 */
void my_cleanup(void)
{
  /*
   * Unregister the device
   */
  unregister_chrdev(Major, DEVICE_NAME);
  printk("The device has be unregistered.");
}



/*
 * 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)
{

  static int counter = 0;           /* note this static variable only gets
                                       initialized the first time this
                                       function is called. */

  /* In these case we are only allowing one process to hold the device
     file open at a time. */
  if (Device_Open)                  /* Device_Open is my flag for the
                                       usage of the device file (definied
                                       in my_chardev.h)  */
    return -EBUSY;                  /* Failure to open device is given
                                       back to the userland program. */

  Device_Open++;                    /* Keeping the count of the 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.*/
  sprintf(msg, "\"%s\" (pid %i): This device has been accessed %d times\n", current-
>comm, current->pid, counter++);
  msg_Ptr = msg;
  try_module_get(THIS_MODULE);     /* increment 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 get rid of the module.
   */
  module_put(THIS_MODULE);
  printk("Device has been closed");
  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 buffer_read[3];
 buffer_read[0]=ioread32(virt_addr+0);
 buffer_read[1]=ioread32(virt_addr+4);
 buffer_read[2]=ioread32(virt_addr+8);

 char* msg_Ptr_read = (char *)buffer_read;
 /*
  * If we're at the end of the message, return 0 signifying end of
  * file
  */
 //if (*msg_Ptr_read == 0)
  //return 0;

 /*
  * Actually put the data into the buffer
  */

 while (length>0) {

  /*
   * 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(*(msg_Ptr_read++), buffer++); /* one char at a time... */

  length--;
  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: echo "hi" > /dev/hello
 * Next time we'll make this one do something interesting.
 */
static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{
  char x;
  int i=0;
  int j=0;
  char msg_Ptr_storage[8];
  while (len>0 ) {

    /*
     * 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.
     */
    get_user(msg_Ptr_storage[i], buff++); /* one char at a time... */
    i++;
    len--;
  }
  int* reg_values_use =  (int*) msg_Ptr_storage;

  iowrite32(reg_values_use[0], virt_addr + 0);     // base address + offset
    // write 2 to register 1
    //printk(KERN_INFO "Value in reg1 %d\n", reg_values_use[0]);
    iowrite32(reg_values_use[1] , virt_addr + 4);
    //printk(KERN_INFO "Value in reg2 %d\n", reg_values_use[1]);
    //printk(KERN_INFO "Multiplied %d\n", ioread32(virt_addr+8));
  /* not allowing writes for now, just printing a message in the
     kernel logs. */
  //printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
  //return -EINVAL;                  /* Fail */
}
/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Paul V. Gratz (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);
```

Devtest.c

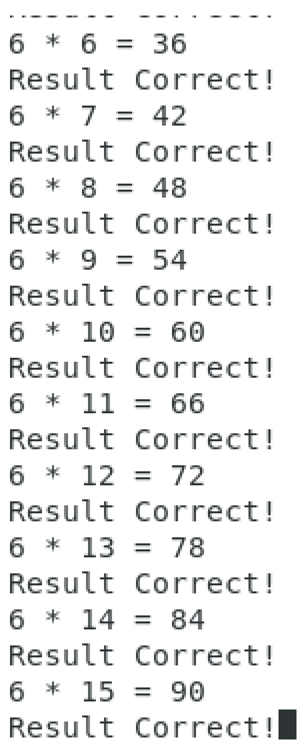
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
   unsigned int result;
   int fd;     // File descriptor
   int i, j;   // Loop variables
   int buff[2];
   int read_buff[3];
   char input = 0;

   // 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
         // Read i, j, and result using char dev
         // Use read to read from peripheral
         // print unsigned ints to screen
             buff[0]=i;
             buff[1]=j;
         write(fd,buff,8);
             read(fd,read_buff,12);
         printf("%u * %u = %u\n\r", read_buff[0], read_buff[1], read_buff[2]);

         // Validate result
         if(read_buff[2] == (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);
            return 0;
}
```

Output:



```
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
zynq> cd mnt/
zynq> insmod multiplier.ko
Mapping virtual address...
The address of virt is 608c4000 and of physical is 1136656384.
Registered a device with dynamic Major number of 245
Created a device file for this device with this command:
'mknod /dev/multiplier c 245 0'.
zynq> mknod /
.ash_history  lib/          mnt/          sbin/         var/
bin/          licenses/     opt/          sys/
dev/          linuxrc       proc/         tmp/
etc/          lost+found/   root/         usr/
zynq> mknod /dev/multiplier c 245 0
zynq> ./devtest
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!
```



```
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!
```

<u>**Result**</u>:

In this lab, character device driver was created and tested using a devtest application. The multiplication was carried out on the hardware developed in the lab 3 and checked. If the error in output occurs, then the main reason would be inefficient parsing of the data between the user application and device driver. Put_user and get_user functions generally provide output in char format. So the typecasting must be done carefully using as the hardware takes integer inputs.

<u>**Conclusion:**</u>

This lab helped in understanding the functioning of character device driver and using user application to perform the necessary functions on the hardware with the help of character device driver.

<u>**Questions:**</u>

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??

Ans. The multiplier block is interfaced as an I/O device to the ARM processor and the code required to interact with this I/O device is running on ARM processor in virtual memory space. A successful call to ioremap() returns a kernel virtual address corresponding to start of the requested physical address range. As the I/O memory is accessed through the page tables, the kernel must run the ioremap which provides the entries of physical address of the device in the page table, through which the device is accessed.

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?

Ans. No. I expect the multiplication would be faster in the original Lab 3 implementation, as the C code is directly interfacing with the hardware. In this lab, however, the C code has to pass from the device driver to reach the hardware. Also there might be an additional delay if the OS is busy with some operation.

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

Ans. Benefits of this lab over lab3 are as follows:

i. Higher level of abstraction: User needed be worried about the underlying hardware and the ways to interact with the register. Just need to create an application file, pass inputs and receive outputs from the same. Implementation is completely masked.

Costs or disadvantage of this lab:

i. Increase in the overall time.
ii. System complexity.
iii. In the 3rd lab, the bitstream must be generated and loaded on the FPGA. Execution is then done using .elf file. Hence knowledge about register space and FPGA is needed.

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.

Ans. The device registration is done last in the initialization routine because as the device is registered, the kernel can make calls to the module even before the initialization is finished. For example if the call is made before any allocation of buffer, then this might cause unexpected behaviour. Also the device is unregistered in the exit routine first, so that the kernel cannot make calls to the module to access the device once it is the exit routine.