Mauro Lopez

LAB 6 Report

ECEN-449

Section 502

October 18, 2018

Introduction:

In this lab we continue to learn more about embedding device drives into a Linux environment. We will be using our multiplier drive, and expand it on it to work with a program that we will be storing in our SD card.

Procedure:

For our program we will be reading and writing to the file "/dev/multiplier". As mention on the lab manual, this is not a standard file, but this allows for our application to interact with the kernel.

We begin by creating our multiplier.c file.

We are given some examples to follow for this lab. we can look at my_chardev.c and my_chardev_mem.c for some reference if we need to.

To create our multiplier we need to start of by using part of our code from our previous multiply.c.

We need to first define and include all of our libraries to make sure all of our functions are well defined, and we get no compile errors.

We are instructed to create our minor of 0 and let our linux system assign our major.

We also need to add error checking, to avoid any major crashes, since this will be embedded into software.

The exit should unmapped the memory and free any malloc that was called during the my_init().

For our open and close we just need to print out the statements of "open file" and "close file". While we are not directly using this functions, it's a good method for debugging.

Now we can move into our "Read" function. This function will allow us to read from our peripheral drives. In our case from our 3 registers.

First we need to allocate memory for us to transfer the data in our buffer into our local variables.

For our write functions, we do a similar process except this time we are grabbing information from our buffer into our periphial. We call this function from our program by calling the write command.

Our file should look similar to this:

```
1.  #include <linux/module.h> /* needed by all modules */
2.  #include <linux/kernel.h> /* needed for KERN_ and print k*/
3.  #include <linux/init.h>  /* needed for __init and __exit macros*/
4.  #include <asm/io.h> /* needed for IO reads and writes  */
5.
6.  #include "xparameters.h" /* needed for physical address of multiplier*/
7.
8.  #include <linux/moduleparam.h>
9.  #include <linux/sched.h>
10. #include <linux/fs.h>
11. #include <asm/uaccess.h>
12. #include <linux/slab.h>
13.
14. #define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR /* Physical address of multiplier */

15. #define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR+1
16. #define DEVICE_NAME "multiplier"
17. #define BUF_LEN 80
18.
19.   // declaring our global variables and define functions.
20.
21. static int Major;
22. static char *msg_bf_Ptr;
23.
24.
25. static int device_open(struct inode *, struct file *);
26. static int device_close(struct inode *, struct file *);
27. static ssize_t device_read (struct file *, char *, size_t, loff_t* );
28. static ssize_t device_write(struct file*,const char *, size_t, loff_t* );
29.
30.
31. ////////////////////////////////////////////////////////////////
32. ////////////////////////////////////////////////////////////////
33. static struct file_operations fops = {
34. .read = device_read,
35. .write = device_write,
36. .open = device_open,
37. .release = device_close
38. };
39.
40. void * virt_addr;
41.
42. ////////////////////////////////////////////////////////////////
43. ////////////////////////////////////////////////////////////////
```

```
44. static int __init my_init(void)
45. {
46. 
47.  /// our init will always execute first, need to map address and virtual address
48. printk(KERN_INFO "Mapping virtual address..\n");
49. virt_addr=ioremap(PHY_ADDR,MEMSIZE); // mapping our address.
50.  Major = register_chrdev(0,DEVICE_NAME, &fops);
51. 
52. if(Major < 0 ){
53. printk( KERN_ALERT " Registering multiplier device failed with %d\n",Major);
54. }
55.   // printing commands allows us to catch errors faster.
56. 
57. printk( KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);
58. 
59. printk( KERN_INFO "Create a device file for this device with this command: \n'mknod /de
    v/%s c %d 0'.\n",DEVICE_NAME,Major);
60. 
61. return Major;
62. }
63. ////////////////////////////////////////////////////////////////////
64. ////////////////////////////////////////////////////////////////////
65. ////////////////////////////////////////////////////////////////////
66. 
67. //writing in our registers
68. /*
69. printk(KERN_INFO "Writing a 7 to the register 0\n");
70. iowrite32( 7, virt_addr+0);
71. // writing into our registers
72. printk(KERN_INFO "Writing a 2 to the register 1\n");
73. iowrite32( 2, virt_addr+4);
74. //will print our the result of our multiplication
75. printk("read %d from register 0\n", ioread32(virt_addr+0));
76. printk("read %d from register 1\n", ioread32(virt_addr+4));
77. printk("read %d from register 2\n", ioread32(virt_addr+8));
78. */
79. //return 0;
80. //}
81. /*
82. this function run just prior to the modules removal from the system.
83.  * you hsould release _all_ resources used by your module
84.  * here (otherwise be prepared for a reboot) .
85. */
86. 
87. ////////////////////////////////////////////////////////////////////
88. ////////////////////////////////////////////////////////////////////
89. static void __exit my_exit(void)
90. {
91. 
92. unregister_chrdev(Major,DEVICE_NAME);
93. printk(KERN_ALERT "unmapping virtual addres space.... \n");
94. iounmap( (void*)virt_addr);
95. 
96. }
97. ////////////////////////////////////////////////////////////////////
98. ////////////////////////////////////////////////////////////////////
99. static int device_open(struct inode *inode, struct file *file){
100.       printk( KERN_ALERT "opeing device" );
101. 
102.       printk( KERN_ALERT "device open ...\n" );
103. 
```

```
104.        return 0;
105.        }
106.
107.        ////////////////////////////////////////////////////////////////
108.        ////////////////////////////////////////////////////////////////
109.        static int device_close(struct inode *inode, struct file *file){
110.        printk( KERN_ALERT "device close... \n");
111.        return 0;
112.        }
113.
114.        ////////////////////////////////////////////////////////////////
115.        ////////////////////////////////////////////////////////////////
116.
117.        static ssize_t device_read(struct file *filp, char *buffer, size_t length, loff_
    t * offset) {
118.
119.        int bytes_read =0;
120.
121.
122.        char * msg_bf_Ptr;
123.        msg_bf_Ptr =  (char *) kmalloc(12*sizeof(char),GFP_KERNEL);
124.
125.        int* int_msg= (int*) msg_bf_Ptr;
126.
127.        int_msg = ioread32(virt_addr);
128.        int_msg = ioread32(virt_addr+4);
129.        int_msg = ioread32(virt_addr+8);
130.
131.        char* ms_Ptr = (char*)KernelBuffer;
132.
133.        while (length && *msg_Ptr) {
134.
135.        put_user( * (msg_Ptr++), buffer++);
136.        length--;
137.        bytes_read++;
138.
139.        }
140.
141.        int * int_buffer = (int*) buffer;
142.
143.        return bytes_read;
144.        printk("read %d from register 0\n", ioread32(virt_addr+8));
145.        put_user( *
146.        return 0;
147.
148.
149.
150.        }
151.        ////////////////////////////////////////////////////////////////
152.        ////////////////////////////////////////////////////////////////
153.
154.        static ssize_t device_write( struct file *file, const char __user * buffer, size
    _t length, loff_t *offset) {
155.
156.        int i ;
157.        char *msg_Ptr;
158.        msg_bf_Ptr =  (char *) kmalloc(8*sizeof(char),GFP_KERNEL);
159.
160.        int * my_int_buff = (int*) msg_bf_Ptr;
161.
162.
```

```
163.        for(i =0; i<length; i++){
164.        get_user(msg_bf_Ptr[i], buffer +i );
165.        }
166.
167.
168.        msg_bf_Ptr[i]= '\0';
169.          /// we need to check each 4 bits of our buffer
170.
171.
172.
173.        iowrite32(my_int_buff[0], virt_addr+0);
174.        iowrite32(my_int_buff[1], virt_addr+4);
175.
176.        return i;
177.
178.        }
179.
180.        ////////////////////////////////////////////////////////////////////
181.        ////////////////////////////////////////////////////////////////////
182.
183.        // info that can be displayed by modinfo
184.        MODULE_LICENSE("GPL");
185.        MODULE_AUTHOR("ECEN449 Student (and others)");
186.        MODULE_DESCRIPTION("Simple multiplier module");
187.        /* Here we define which functions we want to use for initialization
188.         * and cleanup
189.        */
190.
191.        ////////////////////////////////////////////////////////////////////
192.        ////////////////////////////////////////////////////////////////////
193.
194.        module_init(my_init);
195.        module_exit(my_exit);
```

Once we have our multiplier.c, we use our cross compile command :

- Make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-

Make sure to have the setttings 64.

Now that we have our drive into our zybo board, we can begin to write our program that will call our functions.

We are giving some skeleton code and some comments as to, how to fill them.

We should get something similar to this.

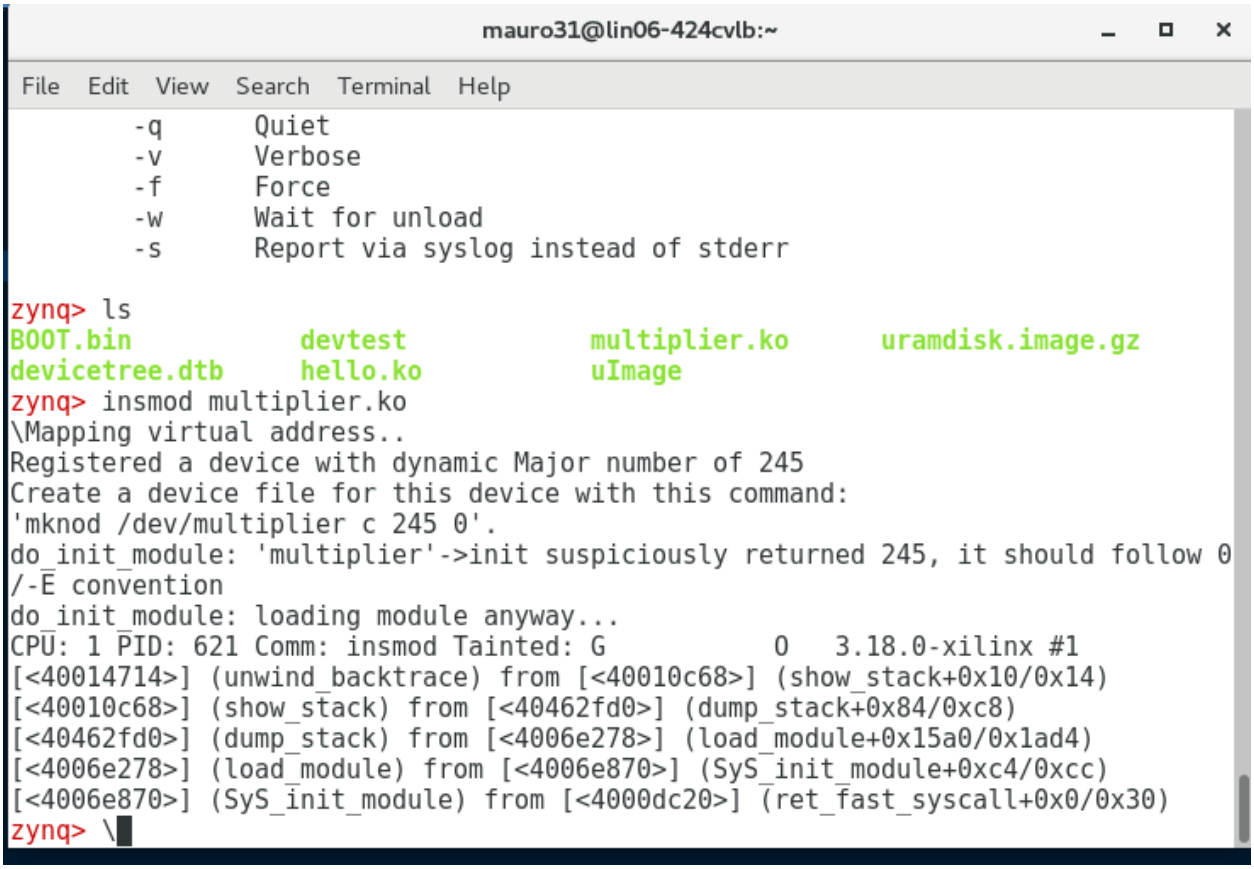
```
1.  #include <sys/types.h>
2.  #include <sys/stat.h>
3.  #include <fcntl.h>
4.  #include <stdio.h>
5.  #include <unistd.h>
6.  #include <stdlib.h>
7.
8.  int main() {
```

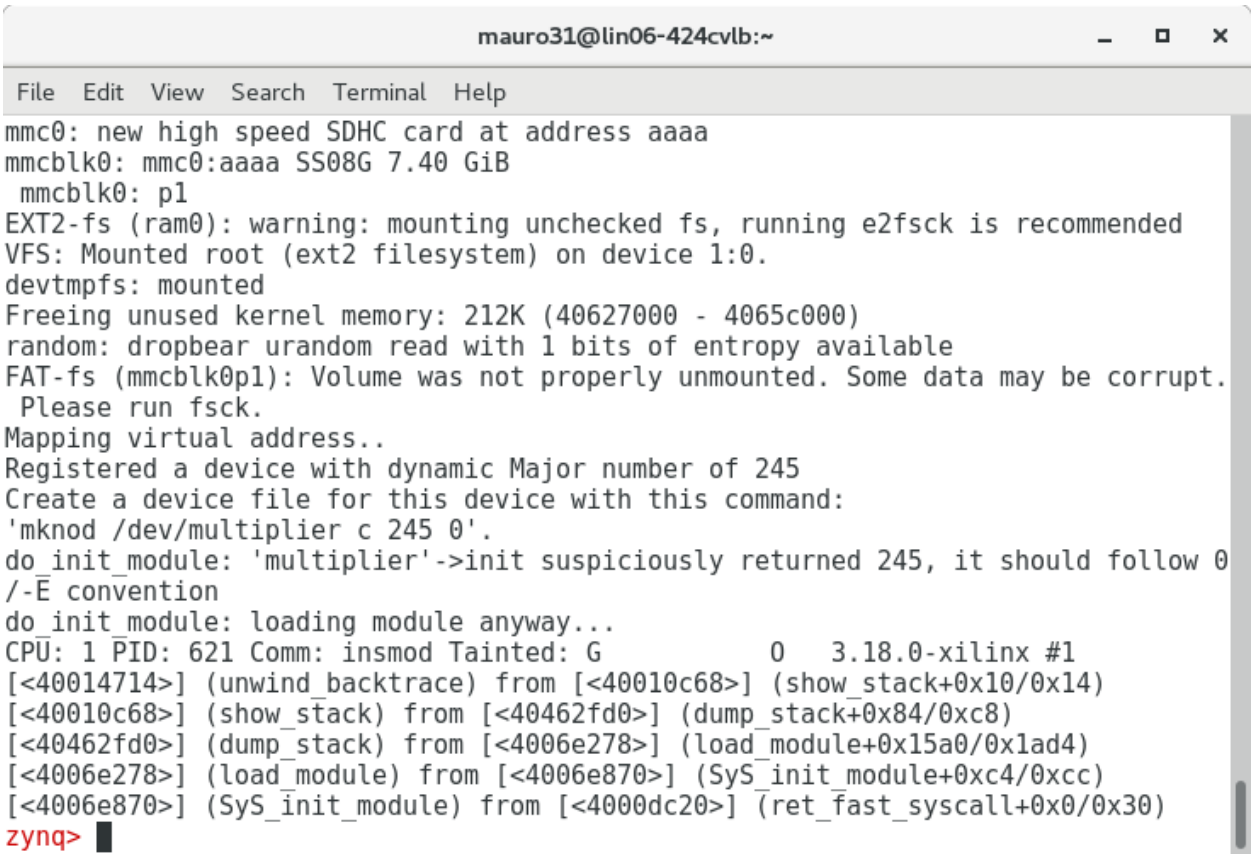
```
9.
10. unsigned int result;
11. int fd;
12. int i, j;
13.   /// declaring our global variables
14. char input =0;
15.
16. fd = open("/dev/multiplier",O_RDWR);
17.
18.
19.   // checking to see if we can open the file.
20. if(fd ==-1){
21. printf("Failed to open devide file ! \n");
22. return -1;
23. }
24.   /// we will compare the values from our register and see if they match.
25. while (input != 'q') {
26.
27. for( i =0; i<= 16; i++){
28.    for( j =0; j <= 16; j++){
29.
30.     if(result ==(i*j)){
31.     printf("    Result Correct !" );
32.     }
33.     else{
34.     printf("    Result Incorrect ! ");
35.     }
36.     input = getchar();
37.
38. } // for loop j
39. }//  for loop i
40. }//  while loop
41. close(fd);
42.
43.
44. return 0;
45. }
```

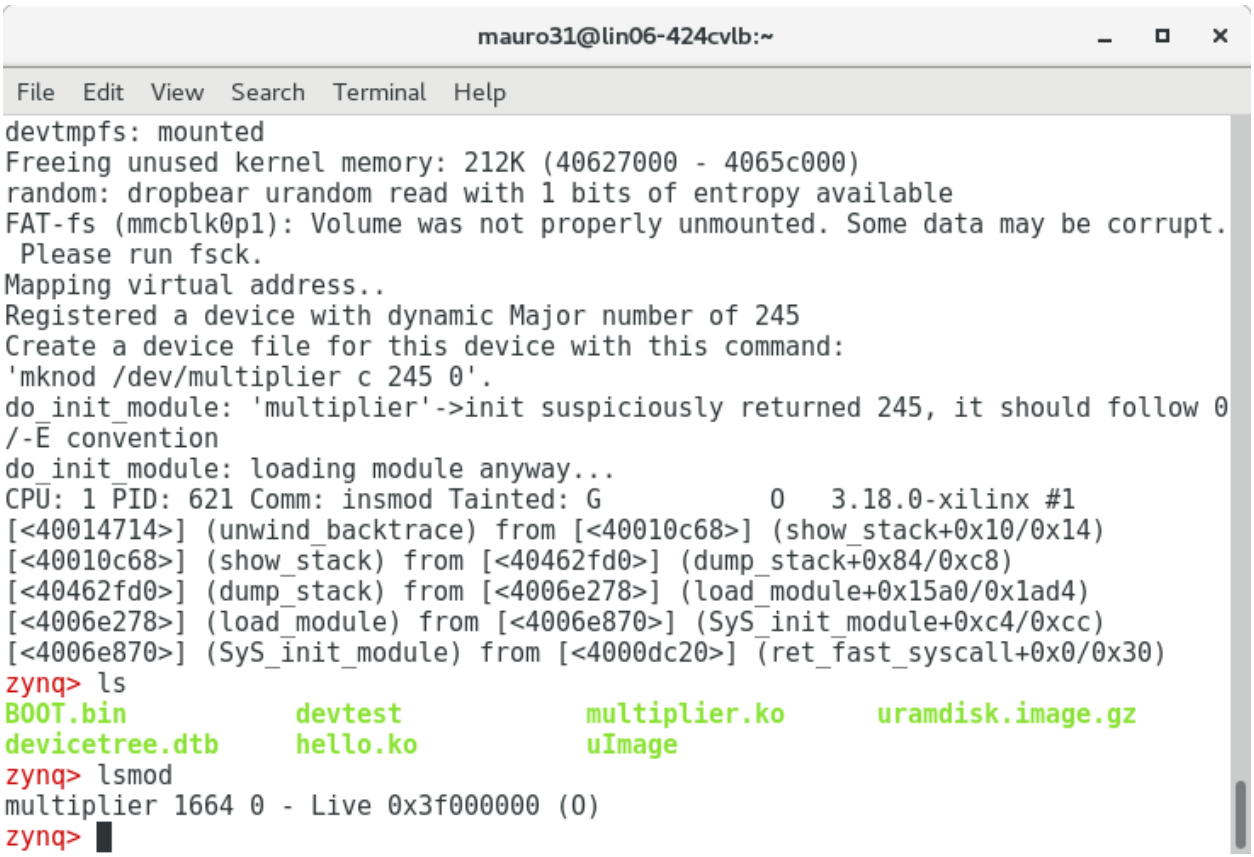
Once we have compile both of our c codes, we can upload them into our SD card and use the mount command in our Zybo boards.

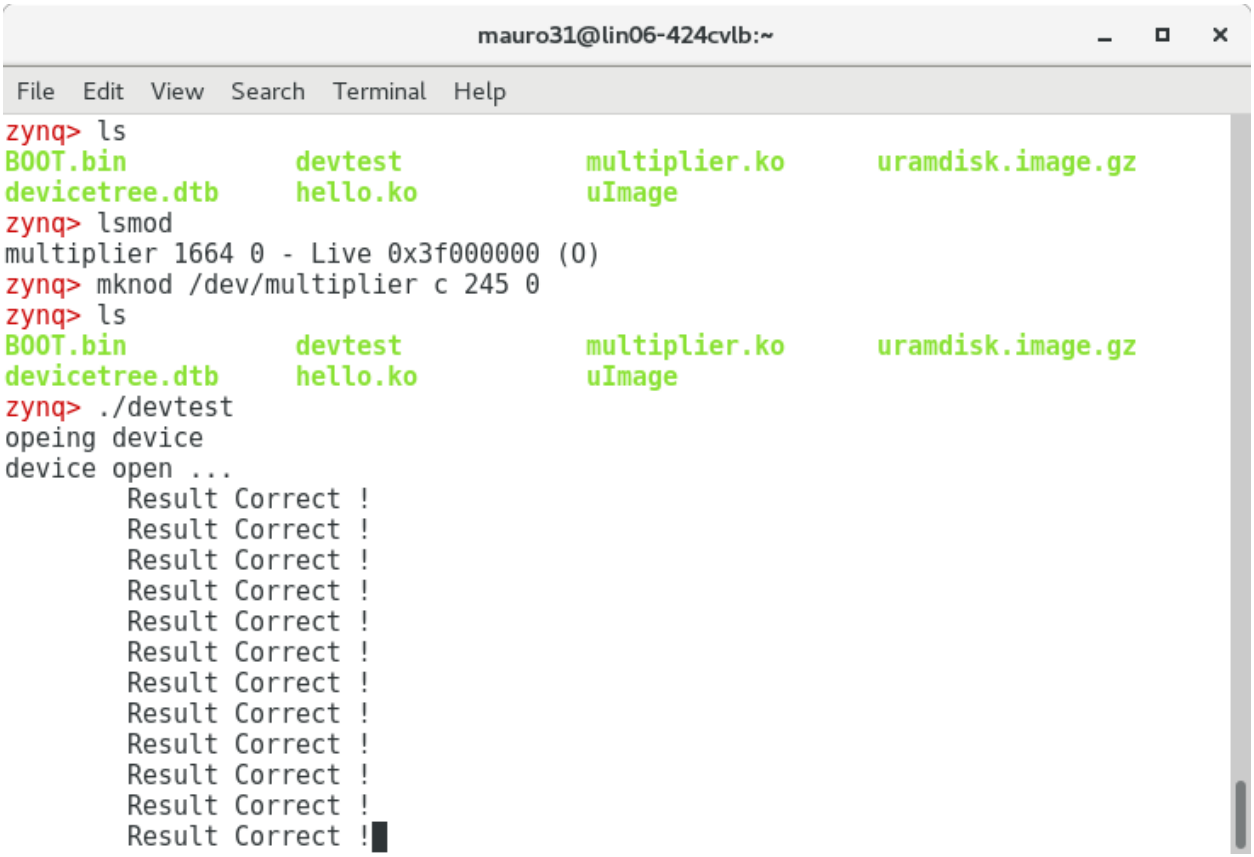
Result:

Although I wasn't able to completely finish the lab, I was able to compile some of my code and got stuck on the debugging phase. Part of me not finishing was on my very limited knowledge of C. The examples we were given were great, but still lack some information. With that being said I was able to produce these results:


mauro31@lin06-424cvlb:~
File Edit View Search Terminal Help
-q Quiet
-v Verbose
-f Force
-w Wait for unload
-s Report via syslog instead of stderr
zynq> ls
BOOT.bin devtest multiplier.ko uramdisk.image.gz
devicetree.dtb hello.ko uImage
zynq> insmod multiplier.ko
\Mapping virtual address..
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'.
do_init_module: 'multiplier'->init suspiciously returned 245, it should follow 0
/-E convention
do_init_module: loading module anyway...
CPU: 1 PID: 621 Comm: insmod Tainted: G O 3.18.0-xilinx #1
[<40014714>] (unwind_backtrace) from [<40010c68>] (show_stack+0x10/0x14)
[<40010c68>] (show_stack) from [<40462fd0>] (dump_stack+0x84/0xc8)
[<40462fd0>] (dump_stack) from [<4006e278>] (load_module+0x15a0/0x1ad4)
[<4006e278>] (load_module) from [<4006e870>] (SyS_init_module+0xc4/0xcc)
[<4006e870>] (SyS_init_module) from [<4000dc20>] (ret_fast_syscall+0x0/0x30)
zynq> \


mauro31@lin06-424cvlb:~
File Edit View Search Terminal Help
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SS08G 7.40 GiB
mmcblk0: p1
EXT2-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
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..
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'.
do_init_module: 'multiplier'->init suspiciously returned 245, it should follow 0
/-E convention
do_init_module: loading module anyway...
CPU: 1 PID: 621 Comm: insmod Tainted: G O 3.18.0-xilinx #1
[<40014714>] (unwind_backtrace) from [<40010c68>] (show_stack+0x10/0x14)
[<40010c68>] (show_stack) from [<40462fd0>] (dump_stack+0x84/0xc8)
[<40462fd0>] (dump_stack) from [<4006e278>] (load_module+0x15a0/0x1ad4)
[<4006e278>] (load_module) from [<4006e870>] (SyS_init_module+0xc4/0xcc)
[<4006e870>] (SyS_init_module) from [<4000dc20>] (ret_fast_syscall+0x0/0x30)
zynq>


mauro31@lin06-424cvlb:~
File Edit View Search Terminal Help
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
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..
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'.
do_init_module: 'multiplier'->init suspiciously returned 245, it should follow 0
/-E convention
do_init_module: loading module anyway...
CPU: 1 PID: 621 Comm: insmod Tainted: G O 3.18.0-xilinx #1
[<40014714>] (unwind_backtrace) from [<40010c68>] (show_stack+0x10/0x14)
[<40010c68>] (show_stack) from [<40462fd0>] (dump_stack+0x84/0xc8)
[<40462fd0>] (dump_stack) from [<4006e278>] (load_module+0x15a0/0x1ad4)
[<4006e278>] (load_module) from [<4006e870>] (SyS_init_module+0xc4/0xcc)
[<4006e870>] (SyS_init_module) from [<4000dc20>] (ret_fast_syscall+0x0/0x30)
zynq> ls
BOOT.bin devtest multiplier.ko uramdisk.image.gz
devicetree.dtb hello.ko uImage
zynq> lsmod
multiplier 1664 0 - Live 0x3f000000 (O)
zynq>


mauro31@lin06-424cvlb:~
File Edit View Search Terminal Help
zynq> ls
BOOT.bin devtest multiplier.ko uramdisk.image.gz
devicetree.dtb hello.ko uImage
zynq> lsmod
multiplier 1664 0 - Live 0x3f000000 (O)
zynq> mknod /dev/multiplier c 245 0
zynq> ls
BOOT.bin devtest multiplier.ko uramdisk.image.gz
devicetree.dtb hello.ko uImage
zynq> ./devtest
opeing device
device open ...
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !
Result Correct !

Conclusion: Over all I think this was a great lab, it is Definity a learning experience when I have to work with new software. In my case I am not too familiar with C. Some of my confusion came from using kmalloc(). I was confused in the sense, that I didn't understand how the casting working, and how to use the pointers for this lab. Despite the confusion, this lab provides good hands on work experience. Up to this point, I have learned more about embedding software in many ways, and I think that is a very important thing.

Questions:

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?

In our case we are using both hardware memory address, and virtual address. Ioremap allows us to map both the physical memory to our virtual memory, this prevents our program from having memory leaks, or misplacing our information in a different memory address.

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?

I think our hardware will work better simply because they are all directly written and accessed. Looking at my code I saw that the program had to go through lots of functions calls. While it might not seem much of a difference our runtime will be slower compare to our lab 3 implementation.

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

When I look at this lab, I see more software and less hardware interaction. This is good in the sense that, debugging code is easier. You are able to create your drivers more freely in the sense that you have options.

Compare to our lab 3 where we had to look at a lower lever and focus more on hardware. While our program will run faster, it will also be harder to debug and change. To change the code, we would have to back to our IP and change the Verilog code. Compare this to lab 6, where we can change how we assign and read the values from our registers.

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

Registering our device in our case allows us to set up the device with its right settings. For instance in this lab, once our registration is done, we print out the statement on how to make the new directory. We let the system take care of this, to avoid any human mistake. Similar to using the function "free()" in

C, we need to unregister our device, to release all of the used memory, and prevent our device from interfering when we call rmmod command.