Description: Description: Description: Description: Description: Description: Description: Description: Description: Description: Texas A&M University Logo - aTm

Lab #6:

**Device Driver**

Khanh Nguyen

UIN# 525000335

ECEN 449– 503

Date: March 2, 2018

**INTRODUCTION:**

The purpose of this lab is to practice creating device drivers in an Linux environment. Device driver serves as a bridge between user application and hardware. In this lab, we created a driver for multiplier and a device test to test its functionality.

**PROCEDURE:**

1/ Create a new folder ‘modules’ under lab5b (another copy of lab5), then create a file called ‘multiplier.c’

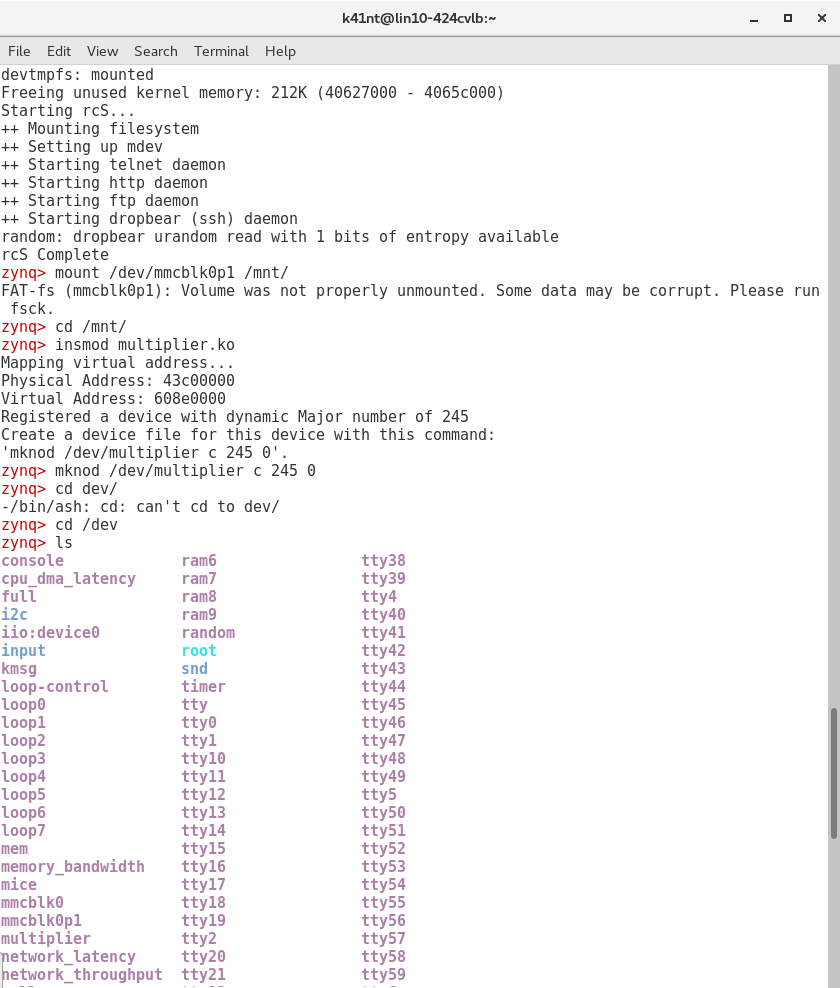
2/ Complete the multiplier module and modify the ‘Makefile’ to compile ‘multiplier.c’

3/ Load ‘multipler.ko’ into Zybo Linux system.

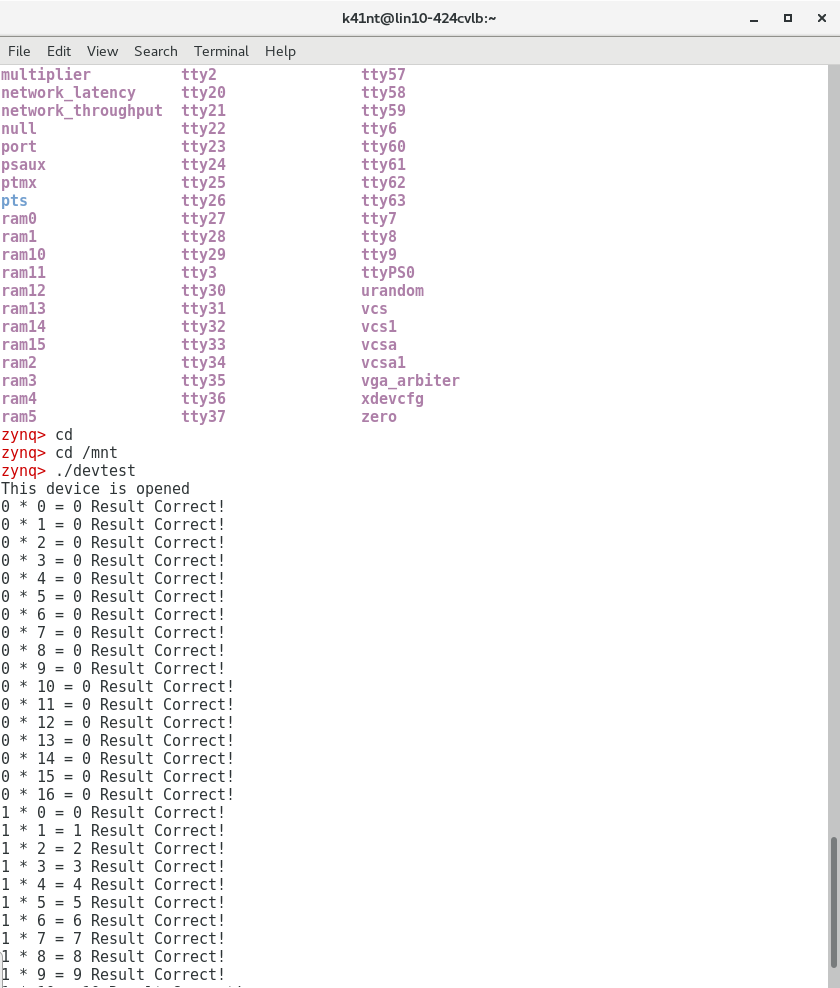
4/ Create a new file called ‘devtest.c’ and complete the skeleton code provided.

5/ Compile the ‘devtest.c’, copy the executable file ‘devtest’ onto the SD card and execute with the Zybo Linux system to test the driver created.

**RESULT:**

****

Loading multiplier.ko onto Zybo linux system



Result of devtest

**CONCLUSION:**

We were able to create a module like we did in lab 3 with different approach. We also learned new functions such as ioremap(), iounmap(), ioread8(), iowrite8()… and used them to created other functions of the driver like device\_release, device\_open, device\_read, device\_write.

**QUESTIONS:**

**1/ 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?**

The ioremap is needed because it maps the physical address of the hardware to the virtual address. The way the kernel handles memory is based on the virtual addresses so this step is necessary.

**2/ 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?**

The approach in lab 3 would be faster because in lab 3 we had direct connections to the hardware. Implementation was directly with hardware therefore the mem read and write would be faster. Also in lab 6, the linux system would need some more time to transfer the virtual address to the physical hardware address.

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

The benefit of Lab 3 approach is that it has lower level of abstraction and can be performed faster because it is connected to the hardware directly. However, the driver is limited to the given hardware.

The benefit of Lab 6 approach is that it is not connected to the hardware directly, so the it won’t be limited by the hardware. However, the procedure would be slower.

**4/ 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.**

In the initialization part, all the necessary settings will be configured, therefore, the device can be registered and available by user.

The device should be unregister before exit to make sure it’s not accessible while removing and all memory is freed.

**CODE:**

**Multiplier.c**

#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 macros \*/

#include <asm/io.h> /\* Needed for IO reads and writes \*/

#include <linux/moduleparam.h> /\* Needed for module parameters \*/

#include <linux/fs.h> /\* Provides file ops structure \*/

#include <linux/sched.h> /\* Provides access to the "current" process task structure \*/

#include <asm/uaccess.h> /\* Provides utilities to bring user space \*/

#include "xparameters.h" /\* Needed for physical address of multiplier \*/

#include <linux/slab.h>

#define PHY\_ADDR XPAR\_MULTIPLY\_0\_S00\_AXI\_BASEADDR //physical address of multiplier

/\*size of physical address range for multiple \*/

#define MEMSIZE XPAR\_MULTIPLY\_0\_S00\_AXI\_HIGHADDR - XPAR\_MULTIPLY\_0\_S00\_AXI\_BASEADDR+1

#define DEVICE\_NAME "multiplier"

/\* 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 Device\_Open=0;

void\* virt\_addr; //virtual address pointing to multiplier

static int Major; /\* Major number assigned to our device driver \*/

/\* 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

};

/\* 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 virtual address...\n");

/\*map virtual address to multiplier physical address\*/

//use ioremap

virt\_addr = ioremap(PHY\_ADDR, MEMSIZE);

//msg\_ptr = kmalloc

printk("Physical Address: %x\n", PHY\_ADDR); //Print physical address

printk("Virtual Address: %x\n", virt\_addr); //Print virtual address

/\* 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);

//iounmap((void\*)virt\_addr);

return Major;

} else {

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);

}

//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 for a reboot). \*/

static void \_\_exit my\_exit(void) {

printk(KERN\_ALERT "unmapping virtual address space...\n");

unregister\_chrdev(Major, DEVICE\_NAME);

iounmap((void\*)virt\_addr);

}

/\*

\* 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)

{

printk(KERN\_ALERT "This device is opened\n");

if (Device\_Open)

return -EBUSY;

Device\_Open++;

try\_module\_get(THIS\_MODULE);

return 0;

}

/\*

\* Called when a process closes the device file.

\*/

static int device\_release(struct inode \*inode, struct file \*file)

{

printk(KERN\_ALERT "This device is closed\n");

Device\_Open--;

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 \*file, /\* 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 i;

for(i=0; i<length; i++) {

put\_user((char)ioread8(virt\_addr+i), buffer+i);

bytes\_read++;

}

/\*

\* Most read functions return the number of bytes put into the

\* buffer

\*/

return bytes\_read;

}

/\*

\* This function is called when somebody tries to write into our

\* device file.

\*/

static ssize\_t device\_write(struct file \*file, const char \_\_user \* buffer, size\_t length, loff\_t \* offset)

{

int i;

char message;

/\* get\_user pulls message from userspace into kernel space \*/

for(i=0; i<length; i++) {

get\_user(message, buffer+i);

iowrite8(message, virt\_addr+i);

}

/\*

\* Again, return the number of input characters used

\*/

return i;

}

/\* These define info that can be displayed by modinfo \*/

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("ECEN449 Khanh Nguyen");

MODULE\_DESCRIPTION("Simple multiplier module");

/\* Here we define which functions we want to use for initialization and cleanup \*/

module\_init(my\_init);

module\_exit(my\_exit);

**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 = open("/dev/multiplier",O\_RDWR);

int i,j;

unsigned int read\_i;

unsigned int read\_j;

char input = 0;

int buffer[3];

if(fd == -1){

printf("Failed to open device file!\n");

return -1;

}

while(input != 'q')

{

for(i=0; i<=16; i++)

{

for(j=0; j<=16; j++)

{

buffer[0]=i;

buffer[1]=j;

write(fd,(char\*)&buffer,8);

read(fd,(char\*)buffer,12);

read\_i=buffer[0];

read\_j=buffer[1];

result=buffer[2];

printf("%u \* %u = %u ",read\_i,read\_j,result);

if(result==(i\*j))

printf("Result Correct!");

else

printf("Result Incorrect!");

input = getchar();

}

}

}

close(fd);

return 0;

}