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

Lab #9:

**Linux Built-in Kernel Modules**

Khanh Nguyen

UIN# 525000335

ECEN 449– 503

Date: April 27, 2018

**INTRODUCTION:**

In this lab we learned how to add device drivers created from previous labs into Linux OS and boot it with Zybo board. We also learned how to remove these drivers to reduce the size of the Linux OS.

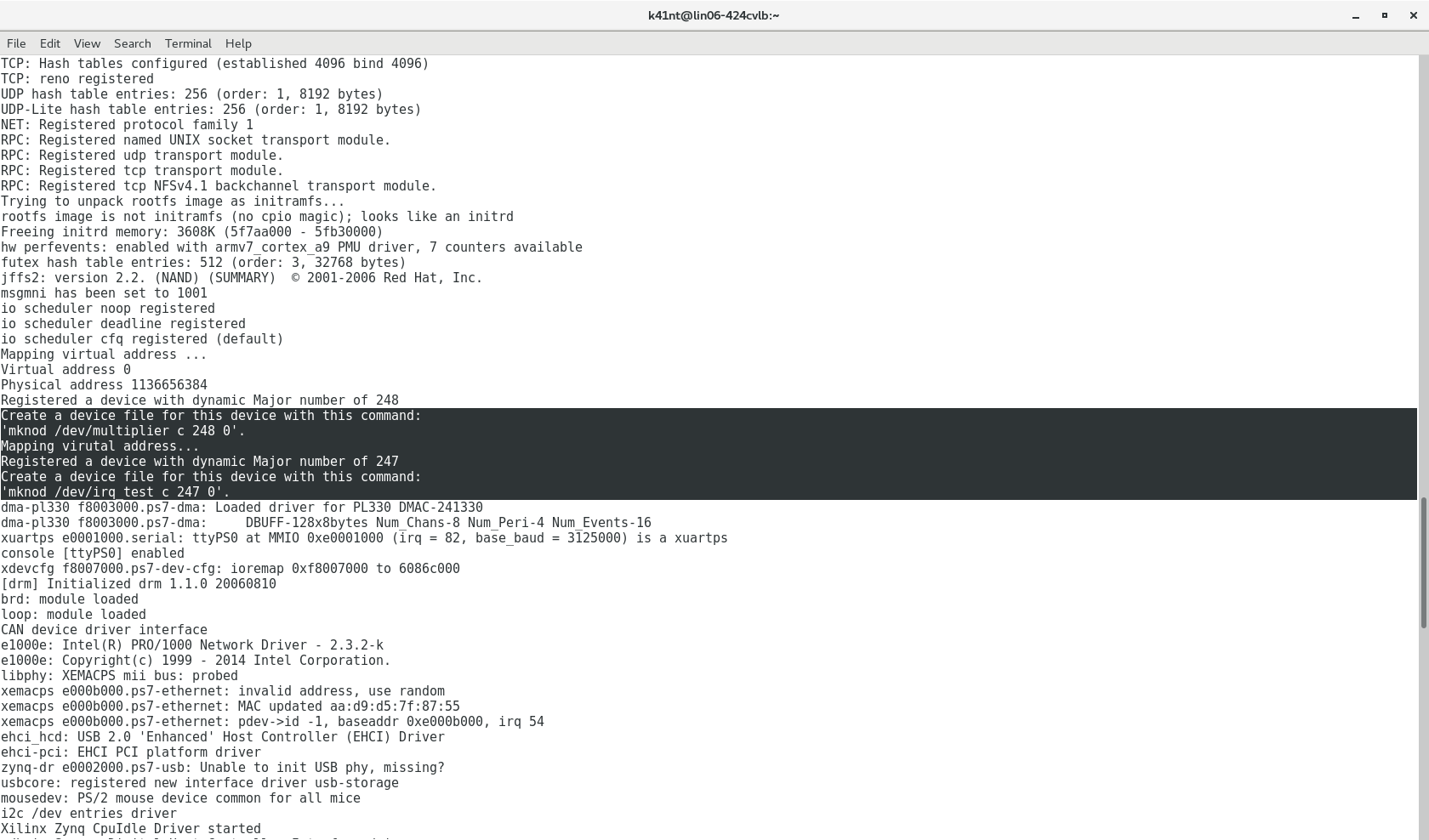
**PROCEDURE:**

1/ The first part of the lab was to use menuconfig to check that multiplier device driver have been selected to be built in. That process allows us to run the “multiply” peripheral without using the “insmod” command.

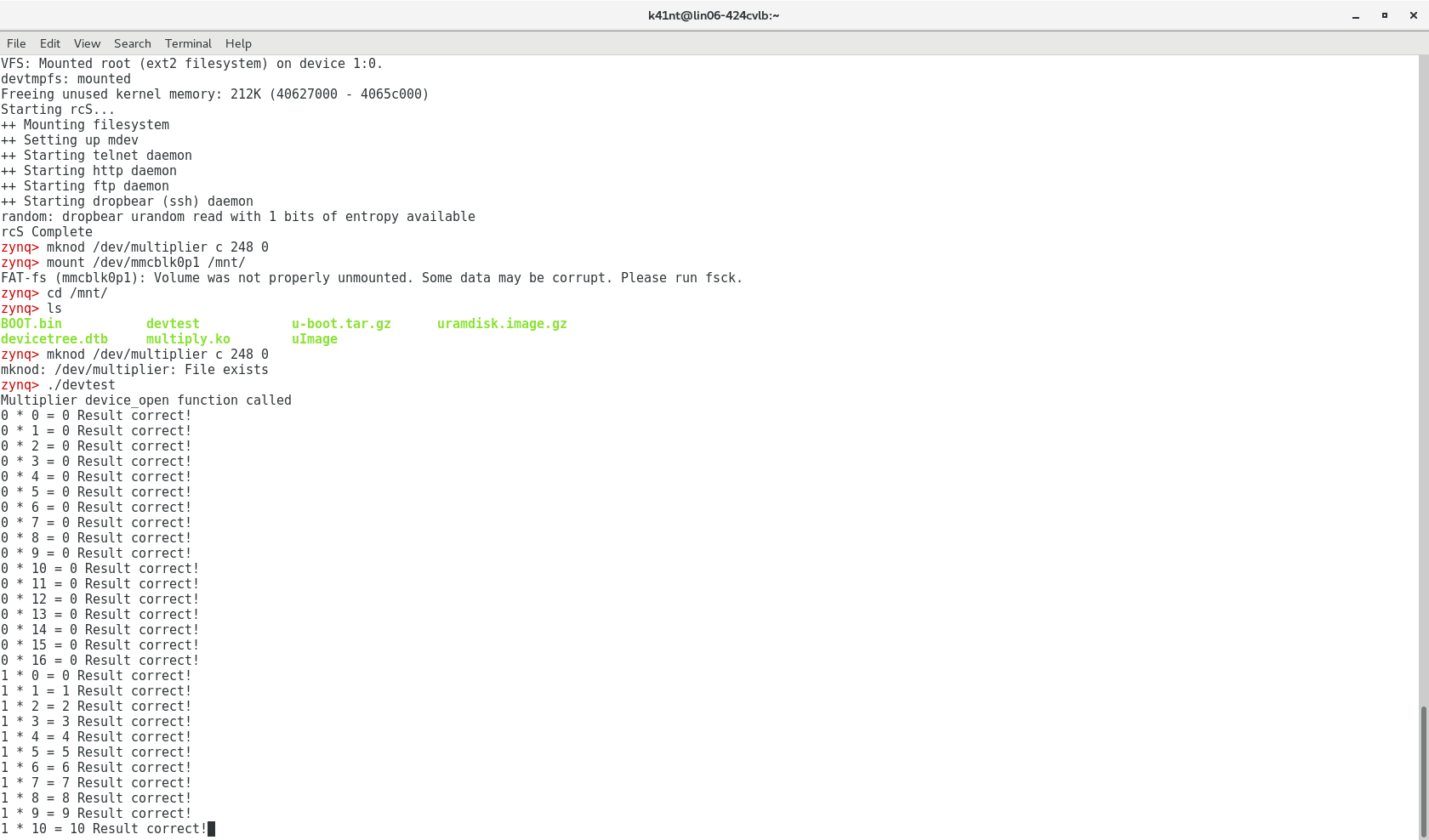
2/ In part 2, we added the ir\_demod module from lab 8 and built it into the kernel. We repeat the step from part 1 to create BOOT.bin and devicetree.dtb. After that, we booted the Linux and observed the size of uImage.

3/ The last part was to remove the sound cards, network and multimedia support drivers. We also observed the reduced size of uImage file.

**RESULT:**

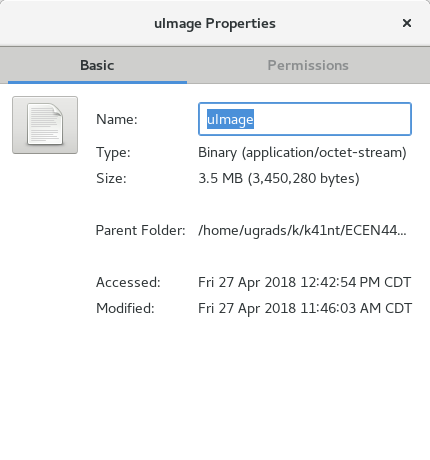


Kernel bootup with multiplier and ir\_demod modules

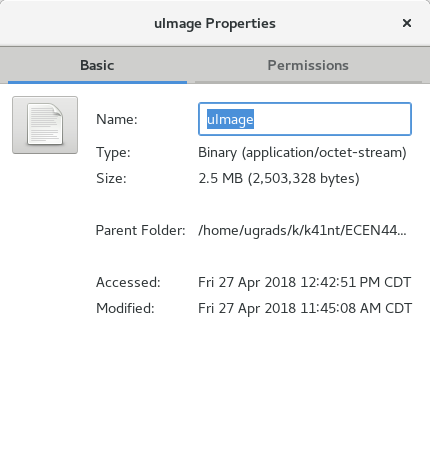


Working multipler module with using “insmod” command

For that third part, after removing the sound cards, network and multimedia support drivers, I observed that the uImage file has reduced to 2,5KB ( the original file was 3,45KB)



Original uImage file



Reduced uImage file

**CONCLUSION:**

From this lab, I learned how to add and remove device drivers to and from Linux. The size changing of uImage file also gave me an idea of the pros and cons of having built-in device drivers. Built-in device drivers will help we reduce the time to connect to the hardware (we don’t need to load it anymore). However, it will increase the size of the Linux if we have too many unnecessary drivers.

**QUESTIONS:**

**What are the advantage and disadvantages of loadable kernel modules and built-in modules?**

\*Having loadable kernel modules can help reduce the size of the kernel. However, it will take more time to configure the module after bootup.

\*Having built-in kernel modules is fast because they are always ready after bootup but it will increase the size of the kernel and also increase the bootup time.

=> I think the best is to remove the unnecessary and rarely used modules, only keep the important drivers.

**C CODE:**

**ir\_demod.c**

/\* irq\_test.c - Simple character device module

\*

\* Demonstrates interrupt driven character device. Note: Assumption

\* here is some hardware will strobe a given hard coded IRQ number

\* (200 in this case). This hardware is not implemented, hence reads

\* will block forever, consider this a non-working example. Could be

\* tied to some device to make it work as expected.

\*

\* (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 "irq\_test.h"

/\* 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

};

void\* virt\_addr; //virtual address pointing to ir peripheral

/\*

\* This function is called when the module is loaded and registers a

\* device for the driver to use.

\*/

int my\_init(void)

{

printk(KERN\_INFO "Mapping virutal address...\n");

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

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

printk("Physical Address: 0x%x\n", PHY\_ADDR);

printk("Virtual Address: 0x%x\n", virt\_addr);

init\_waitqueue\_head(&queue); /\* initialize the wait queue \*/

/\* Initialize the semaphor we will use to protect against multiple

users opening the device \*/

sema\_init(&sem, 1);

Major = register\_chrdev(0, DEVICE\_NAME, &fops);

if (Major < 0) {

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

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

}

/\*

\* Called when a process tries to open the device file, like "cat

\* /dev/irq\_test". Link to this function placed in file operations

\* structure for our device file.

\*/

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

{

int irq\_ret;

if (down\_interruptible(&sem))

return -ERESTARTSYS;

/\* We are only allowing one process to hold the device file open at

a time. \*/

if (Device\_Open) {

up(&sem);

return -EBUSY;

}

Device\_Open++;

/\* OK we are now past the critical section, we can release the

semaphore and all will be well \*/

up(&sem);

/\* request a fast IRQ and set handler \*/

irq\_ret = request\_irq(IRQ\_NUM, irq\_handler, 0 /\*flags\*/, DEVICE\_NAME, NULL);

if (irq\_ret < 0) { /\* handle errors \*/

printk(KERN\_ALERT "Registering IRQ failed with %d\n", irq\_ret);

return irq\_ret;

}

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. \*/

msg\_Ptr = NULL;

printk("Device has been opened\n");

//allocating messageQueue with enough bytes to store 100 of MESSAGE

messageQueue = (MESSAGE\*)kmalloc(100 \* sizeof(MESSAGE), GFP\_KERNEL);

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 \*/

free\_irq(IRQ\_NUM, NULL);

/\*

\* 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\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 bytes\_read = 0;

/\* In this driver msg\_Ptr is NULL until an interrupt occurs \*/

//wait\_event\_interruptible(queue, (msg\_Ptr != NULL)); /\* sleep until

//interrupted \*/

/\*

\* Actually put the data into the buffer

\*/

int i = 0;

//if we go past the amount of messages we've written

/\*if (length > counter \* 2 || length > 200) {

length = writeIndex \* 2;

}\*/

length = writeIndex \* 2;

printk("Read %d messages since last checked...\n", length);

writeIndex = 0;

msg\_Ptr = (char\*)messageQueue;

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

/\*

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

bytes\_read++;

}

/\* completed interrupt servicing reset

pointer to wait for another

interrupt \*/

msg\_Ptr = NULL;

/\*

\* 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)

{

/\* 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 \*/

}

irqreturn\_t irq\_handler(int irq, void \*dev\_id) {

sprintf(msg, "IRQ Num %d called, interrupts processed %d times\n", irq, counter++);

printk("%d...\n", counter);

msg\_Ptr = (char\*)messageQueue; //pointer array to the start of the queue

message = ioread32(virt\_addr + 0);

if (writeIndex == 100) {//every 100 messages we send a wake signal

//reset writeIndex when it becomes large

/\* Just wake up anything waiting

for the device \*/

//wake\_up\_interruptible(&queue);

writeIndex = 0;

}

messageQueue[writeIndex].byte0 = byteBuff[0]; //write to the message queue

messageQueue[writeIndex].byte1 = byteBuff[1];

writeIndex++;

iowrite32(0x80000000, virt\_addr + 8); //clear the interrupt

return IRQ\_HANDLED;

}

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

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Khanh Nguyen");

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);

**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);