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

Lab #8:

**Interrupt Driven IR Remote**

**Device Driver**

Khanh Nguyen

UIN# 525000335

ECEN 449– 503

Date: April 20, 2018

**INTRODUCTION:**

In this lab we added an output to the module that we have created from previous lab which was the IR interrupt. Next, we wrote the driver for the circuit then load and test it on the Zybo board.

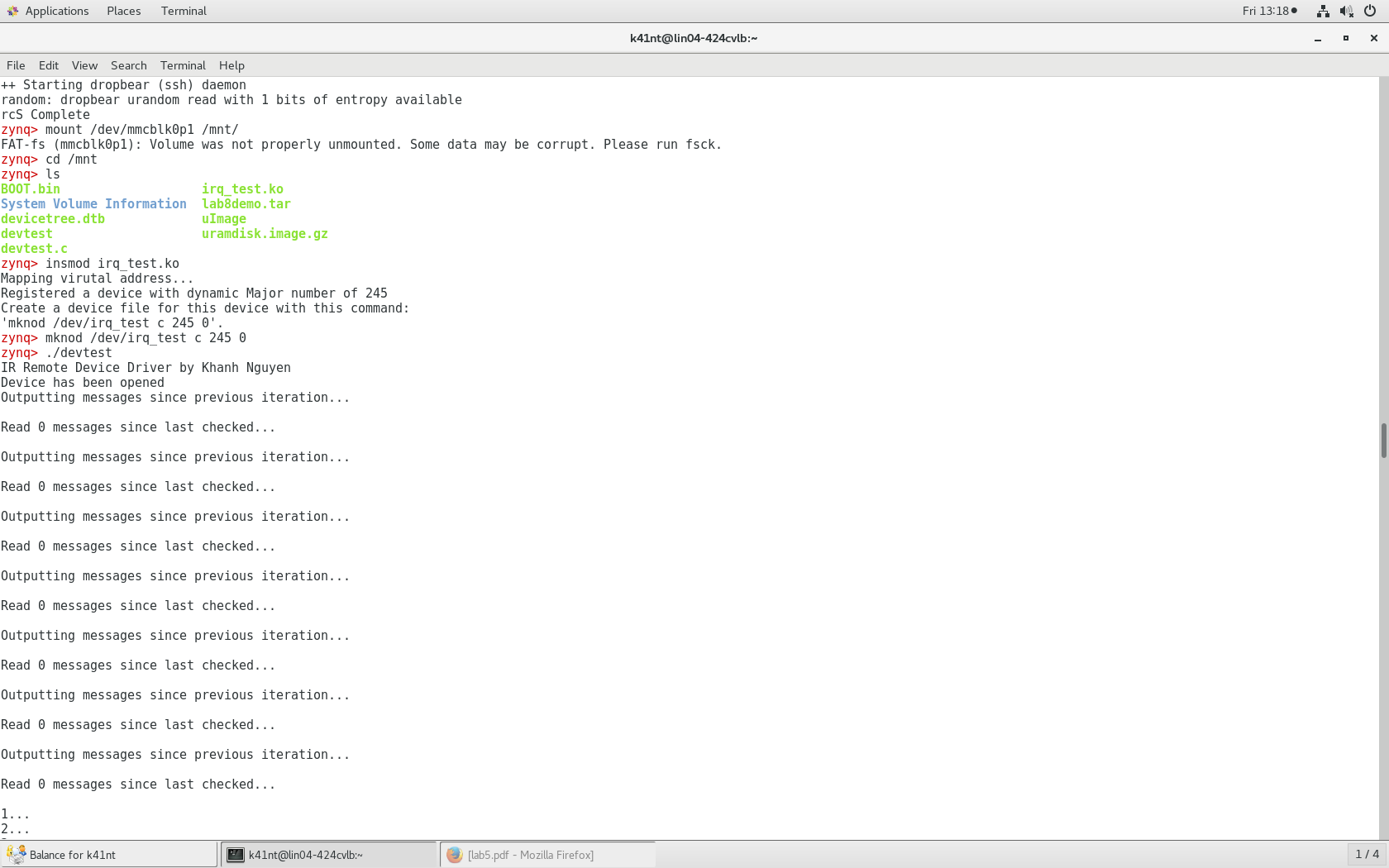
**PROCEDURE:**

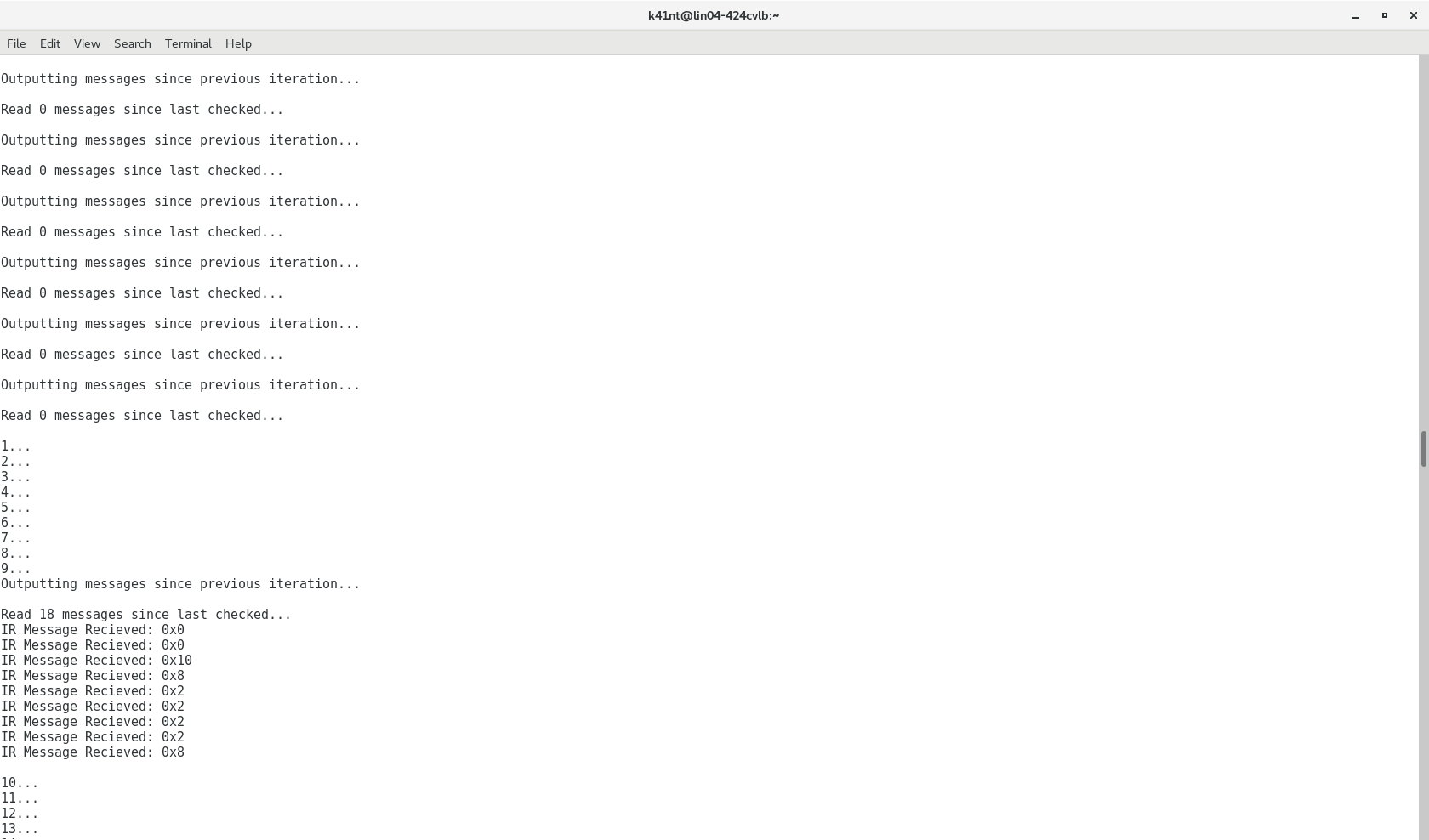
1/ The first part of the lab was to add the IR interrupt output to the IP module that we have created from pervious lab. The IR interrupt goes high whenever a full message is received. After that, we created the test C code for circuit.

2/ After having the IR interrupt working, we created the device driver for the circuit. Boot files were also created. After that, we made a devtest file and ran it on the Zybo board Linux system to test the whole combined system.

**RESULT:**

The results showed from Picocom terminal:





**CONCLUSION:**

This lab is a good review of what we did in previous labs (lab4, lab5, lab6. Building the circuit and adding the IR interrupt to the IP peripheral were pretty simple. However, making the device driver and testing it on the zybo board were quite demanding and took me a lot of time. There were so many error occurred during the process (syntax, logical etc.). Debugging was a very time-consuming part of this lab. Despite of some problems I had, it was a good learning experience.

**QUESTIONS:**

**1/ Contrast the use of an interrupt based device driver with the polling method used in the previous lab.**

In previous lab, we have to monitor the state of the peripheral by polling register continuously. It wastes time and CPU resources.

In this lab, the interrupt only goes high when a new message comes in. That way the CPU will have more time when waiting to other jobs.

**2/ Are there any race conditions that your device driver does not address? If so, what are they and how would you fix them?**

The race condition occurs when a queue receives a new message at the same time the reader reads from the queue. To prevent this, we used a counting semaphore.

**3/ If you register your interrupt handler as a ‘fast’ interrupt (i.e. with the SA INTERRUPT flag set), what precautions must you take when developing your interrupt handler routine? Why is this so? Taking this into consideration, what modifications would you make to your existing IR-remote device driver?**

Different interrupt may cause different computing time. Therefore, we must take the computing time into consideration. The handler function code is expected to complete in small amount of time. I would remove all the print statements for debugging.

**4/ What would happen if you specified an incorrect IRQ number when registering your interrupt handler? Would your system still function properly? Why or why not?**If and incorrect IRQ number if specified, the device driver will not function. The interrupt may trigger and wrong handler and that would cause errors.

**CODE:**

**User logic in ir\_demod\_v1\_0\_S00\_AXI.v:**

reg [31:0] clockCount;

reg clock;

reg temp\_interrupt;

reg Status;

assign reset = ~S\_AXI\_ARESETN; //S\_AXI\_ARESETN is active low

assign mainClock = S\_AXI\_ACLK;

/\* Dividing clock by 1,000 to manage counter better \*/

always@(posedge mainClock) begin

if (clockCount == 1000 && ~reset) begin

clock <= 1;

clockCount <= 0;

end

else if (reset) begin

clockCount <= 0;

clock <= 0;

end

else begin

clockCount <= clockCount + 1;

clock <= 0;

end

end

reg oldSignal;/\* Old Signal used for edge detection \*/

reg [31:0] counter;/\* Counter to count time of ~IR\_signal \*/

reg State;

reg NextState;

reg [11:0] demodulatedMessage;

reg startSignal;

reg [31:0] bitCount;

reg keepCounting;

assign IR\_interrupt= temp\_interrupt;

always@(posedge clock) begin

oldSignal <= IR\_signal;

if (slv\_reg2[0]==1'b1) begin //reset interrupt

Status<=0;

temp\_interrupt<=0;

end

if (oldSignal && ~IR\_signal) begin

/\* We start counting on the negative edge of the clock \*/

keepCounting <= 1'b1;

end

else if (~oldSignal && IR\_signal) begin //if we just resolve a bit

/\* We stop counting when we reach the positive edge again \*/

bitCount <= bitCount + 1; //increament bitCount (we just read a bit)

keepCounting <= 1'b0; //stop counting

counter <= 0; //reset counter

//store the bit we just read into current message

if (startSignal && bitCount >= 12) begin //if we have received a full message

slv\_reg0 <= demodulatedMessage;

slv\_reg1 <= slv\_reg1 + 1;

bitCount <= 0;

startSignal <= 0;

Status<=1;

temp\_interrupt<=1;

end

else if (startSignal && bitCount < 12 && bitCount != 0) begin //in order to prevent having to use blocking assignments

demodulatedMessage[11 - (bitCount - 1)] <= State;

end

end

/\* IR Signal is Active Low \*/

if (keepCounting && ~IR\_signal) begin

counter <= counter + 1;

/\* With a start signal time of 2.4 ms, IR\_signal

shoud be low for approximately 180 cycles

Zero Signal = 0.6 ms = 45 cycles

One Signal = 1.2 = 90 cycles

I'm using the halfway point between 0 and 1

to create a fine line between them and resolving

an appropriate signal (.9 ms = 68 cycles)

Halfway point between Start and 1 = 1.8 ms = 135 cycles \*/

if (counter >= 20 && counter <= 68) begin

State <=0;

end

else if (counter >= 69 && counter <= 134) begin

State <= 1;

end

else if (counter >= 135 && counter <= 250) begin

startSignal <= 1;

/\* Initialize bit count to 0 \*/

bitCount <= 0;

end

else begin

/\* Just to be safe (it's probably impossible

to get to here anyway) \*/

State <= 0;

end

end

end

// User logic ends

**Irq\_test.h**

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

data into kernel space. Note, it is

processor arch specific. \*/

#include <linux/semaphore.h> /\* Provides semaphore support \*/

#include <linux/wait.h> /\* For wait\_event and wake\_up \*/

#include <asm/io.h> //needed for IO reads and writes

#include <linux/interrupt.h> /\* Provide irq support functions (2.6

only) \*/

#include <linux/slab.h> //needed for kmalloc() and kfree()

#include "xparameters.h" //needed for address of the ir perpheral

/\* Some defines \*/

#define DEVICE\_NAME "irq\_test"

#define PHY\_ADDR XPAR\_IR\_DEMOD\_0\_S00\_AXI\_BASEADDR //physical address of ir demod

//size of physical address range for multiply

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

#define BUF\_LEN 80

#define IRQ\_NUM 61

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

/\*

\* Global variables are declared as static, so are global but only

\* accessible within the file.

\*/

typedef struct message {

//each message has two bytes to it (16 bits)

char byte0;

char byte1;

} MESSAGE;

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

static int Device\_Open = 0; /\* Flag to signify open device \*/

static char msg[BUF\_LEN]; /\* The msg the device will give when asked \*/

static char \*msg\_Ptr;

static struct semaphore sem; /\* mutual exclusion semaphore for race

on file open \*/

static wait\_queue\_head\_t queue; /\* wait queue used by driver for

blocking I/O \*/

static int counter = 0; /\* keep track of the number of

interrupts handled \*/

static int message = 0;

static int writeIndex = 0;

static char\* byteBuff = (char\*)&message; //will use to extract individual bytes of msg

static MESSAGE\* messageQueue; //stores 100 messages at a time

**irq\_test.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);

**devtest.c**

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main() {

printf("IR Remote Device Driver by Khanh Nguyen!\n");

unsigned int message;

char\* msg\_Ptr = (char\*)&message; //used to write 1 byte at a time to message

int fd; //file descriptor

char input = 0;

char\* outputBuffer = (char\*)malloc(200 \* sizeof(char)); //enough to read all 100 messages

//open device file for reading and writing

//user open to open 'dev/ir\_demod'/

fd = open("/dev/irq\_test", O\_RDWR);

//handle error opening file

if (fd == -1) {

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

return -1;

}

int i = 0;

for (;;) {

int bytesRead = read(fd, outputBuffer, 200); //read up to 100 messages at a time

for (i = 0; i < bytesRead / 2; i++) { //print all 100 messages

msg\_Ptr[0] = outputBuffer[i \* 2];

msg\_Ptr[1] = outputBuffer[i \* 2 + 1];

msg\_Ptr[2] = 0;

msg\_Ptr[3] = 0;

printf("IR Message: 0x%x\n", message);

}

printf("\n");

sleep(1); //sleep for () seconds

}

close(fd);

free(outputBuffer);

return 0;

}