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.

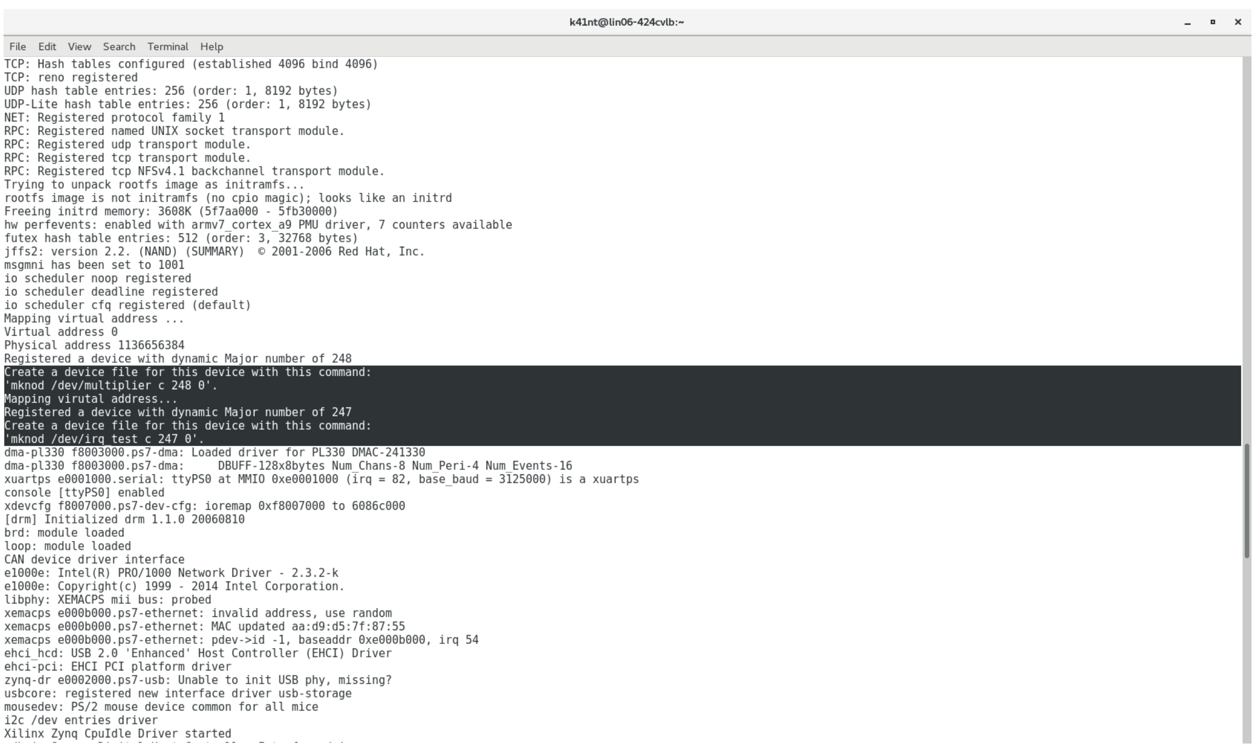
PROCEDURE:

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.

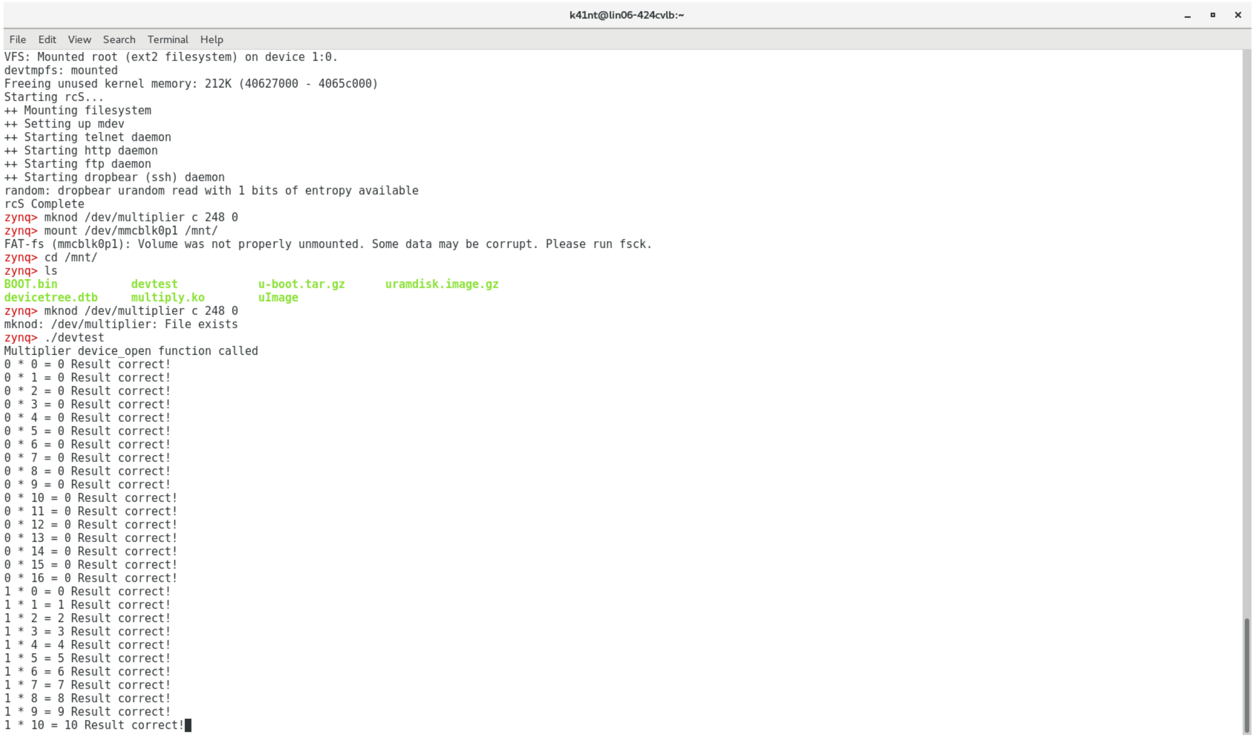
3/ The last part was to remove the sound cards, network and multimedia support drivers. We also observed the reduced size of uImage file.

RESULT:



```
k41nt@lin06-424cvlb:~
File Edit View Search Terminal Help
TCP: Hash tables configured (established 4096 bind 4096)
TCP: reno registered
UDP hash table entries: 256 (order: 1, 8192 bytes)
UDP-Lite hash table entries: 256 (order: 1, 8192 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 3608K (5f7aa000 - 5fb30000)
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
futex hash table entries: 512 (order: 3, 32768 bytes)
jffs2: version 2.2. (NAND) (SUMMARY)  © 2001-2006 Red Hat, Inc.
msgmni has been set to 1001
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
Mapping virtual address ...
Virtual address 0
Physical address 1136656384
Registered a device with dynamic Major number of 248
Create a device file for this device with this command:
'mknod /dev/multiplier c 248 0'.
Mapping virutal address...
Registered a device with dynamic Major number of 247
Create a device file for this device with this command:
'mknod /dev/irq_test c 247 0'.
dma-pl330 f8003000.ps7-dma: Loaded driver for PL330 DMAC-241330
dma-pl330 f8003000.ps7-dma:     DBUFF-128x8bytes Num_Chans-8 Num_Peri-4 Num_Events-16
xuartps e0001000.serial: ttyPS0 at MMIO 0xe0001000 (irq = 82, base_baud = 3125000) is a xuartps
console [ttyPS0] enabled
xdevcfg f8007000.ps7-dev-cfg: ioremap 0xf8007000 to 6086c000
[drm] Initialized drm 1.1.0 20060810
brd: module loaded
loop: module loaded
CAN device driver interface
e1000e: Intel(R) PRO/1000 Network Driver - 2.3.2-k
e1000e: Copyright(c) 1999 - 2014 Intel Corporation.
libphy: XEMACPS mii bus: probed
xemacps e000b000.ps7-ethernet: invalid address, use random
xemacps e000b000.ps7-ethernet: MAC updated aa:d9:d5:7f:87:55
xemacps e000b000.ps7-ethernet: pdev->id -1, baseaddr 0xe000b000, irq 54
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ehci-pci: EHCI PCI platform driver
zynq-dr e0002000.ps7-usb: Unable to init USB phy, missing?
usbcore: registered new interface driver usb-storage
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
Xilinx Zynq CpuIdle Driver started
```

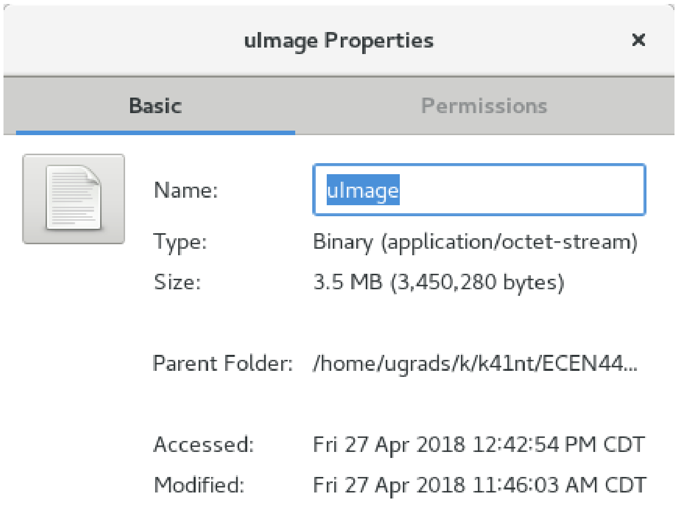
Kernel bootup with multiplier and ir_demod modules



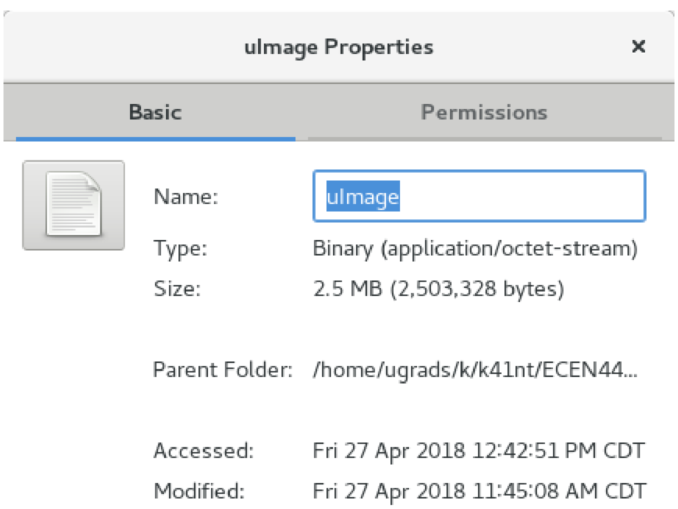
```
k41nt@lin06-424cvlb:~
File  Edit  View  Search  Terminal  Help
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq> mknod /dev/multiplier c 248 0
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> ls
BOOT.bin          devtest           u-boot.tar.gz      uramdisk.image.gz
devicetree.dtb    multiply.ko       uImage
zynq> mknod /dev/multiplier c 248 0
mknod: /dev/multiplier: File exists
zynq> ./devtest
Multiplier device_open function called
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!
1 * 4 = 4 Result correct!
1 * 5 = 5 Result correct!
1 * 6 = 6 Result correct!
1 * 7 = 7 Result correct!
1 * 8 = 8 Result correct!
1 * 9 = 9 Result correct!
1 * 10 = 10 Result correct!
```

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 */
```

```c
#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);
```