LAB REPORT 6

ECEN 749

Name: Swarna Prabhu

UIN:932003046

An introduction to character device driver development

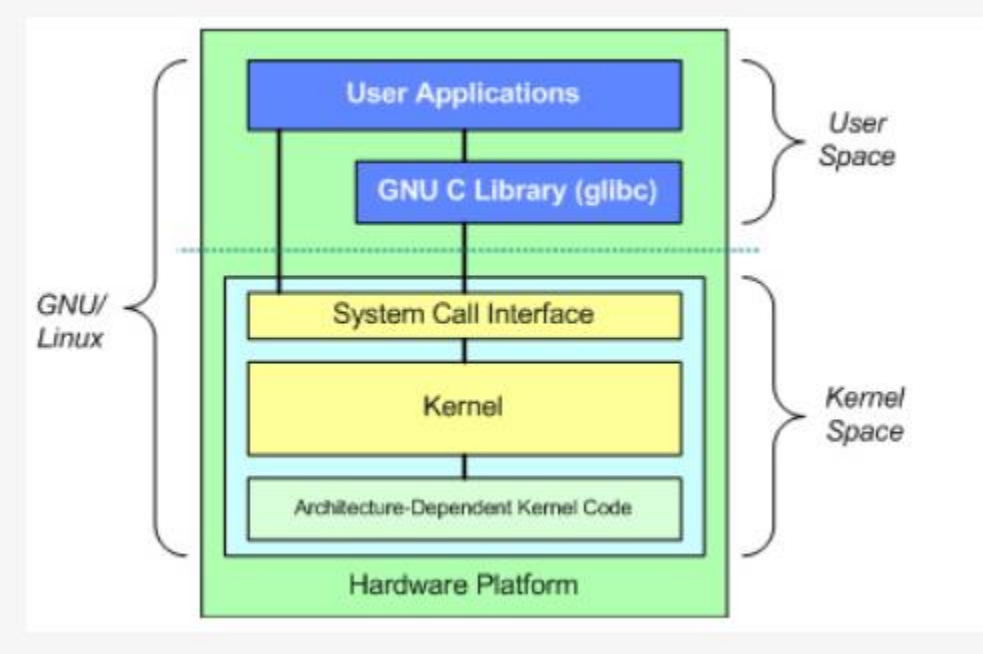
Introduction:

Purpose of the lab 6 is develop a character-based device driver in the form of a Linux kernel module that performs multiplication in our case, when booted on a zybo z7 board. A device driver can be considered as a software application that is designed to ensure interaction between different hardware devices.

Character based device drivers are specifically used for IO operations where we need to transfer the data in the form of byte by byte like input from keyboard, mouse etc. The character device files are always stored in dev/directory from where we can directly read or write of data byte by byte.

Once we design the module multiplier we load the module and test it by writing a simple user application called devtest.c.

Procedure:



- The procedure for lab 6 is divided into 2 main parts. In the first part we create multiplier module that is loaded into the linux kernel booted on the zybo board. In the 2nd part we develop the test application in user space which is written in devtest.c file.
- We initially implement a character based device driver by writing appropriate functions in file named multiplier.c. This is done by referring to the basic template given in my_Chardev.c and mychar_devmeme.c
- In the beginning the necessary header files used for implementing the functions are declared. Device name and a buffer length (constant) is declared followed by list of function prototypes.
- File operation structure is defined –
  static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
  }; // the file operation structure basically defines the pointers that will point to functions that is used to perform various operations on the device. Each function is defined in the driver to perform some requested operation be it read /write etc.The linux kernel sees each device as a file structure and is defined in header `linux/fs.`

- In the initialization routine at first the logical address issued is mapped to the physical address of the multiplier using ioremap command. Every device of a particular class contains a major number and the devices under the major number are assigned a minor number. By registering the device we are adding the driver to the system. register_chrdev(0, DEVICE_NAME, &fops) returns the major number which is used in the function to check if the device has been registered correctly or not. If major is negative device registration failed if its correct we return 0 in the function
- Routine for device clean up -my_cleanup(void). Here we unregister the device and unmap the virtual address space.
- Device open and device release functions are defined where only print statements is written informing the use r that a device has been opened or released.
- Device read function is written where the device driver should perform read operation bytes wise but the buffer that we need to fill with the read information is not present in kernel buffer. Hence we use put_user to copy the data from kernel space to the user space. The count of total bytes of data that is put into the user space is calculated and is returned.
- The write function is implemented where the device driver has to write into the multiplication peripheral. Therefore get_user is used to copy the data from the user space to kernel space.We keep a count of total bytes of data that is written and the same value is returned for the write function. The length of buffer for which data has to be written is passed as one of the arguments.
- We generate the object file after running the below make
- make ARCH=arm CROSSCOMPILE=arm-xilinx-linux-gnueabi and the multiplier. Ko generated is loaded as module. The result for this is attached below in result section.

- In the 2nd part of the procedure we create a devtest.c which is basically a user application to test read and write operations on the created device file.use mknod dev/multiplier c 245 0 for creating the multiplier file in dev
- Once devtest.o exectutable file is generated we load the same in sd card and execute the test application using ./devtest and the output of performing multiplication will be displayed.

Result :

Result of running the multiplier module:On adding the multiplier module and dmseg will give the result of running of the module where it displays the registration of the device is done with major numer 0 and minor numher 245

```
Booting Linux on physical CPU 0x0
Linux version 3.18.0-xilinx (swarnatind-abc88@zach-354a-02.engr.tamu.edu) (gcc version 4.9.1
(Sourcery CodeBench Lite 2014.11-30) ) #4 SMP PREEMPT Mon Oct 25 18:04:14 CDT 2021
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Zynq ZYBO Z7 Development Board
cma: Reserved 16 MiB at 0x1e400000
Memory policy: Data cache writealloc
On node 0 totalpages: 131072
free_area_init_node: node 0, pgdat 4069be00, node_mem_map 5fbf0000
  Normal zone: 1024 pages used for memmap
  Normal zone: 0 pages reserved
  Normal zone: 131072 pages, LIFO batch:31
PERCPU: Embedded 10 pages/cpu @5fbd3000 s8768 r8192 d24000 u40960
pcpu-alloc: s8768 r8192 d24000 u40960 alloc=10*4096
pcpu-alloc: [0] 0 [0] 1
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttyPS0,115200 root=/dev/ram rw earlyprintk
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 492632K/524288K available (4650K kernel code, 258K rwdata, 1616K rodata, 212K
init, 219K bss, 31656K reserved, 0K highmem)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xffe00000   (2048 kB)
    vmalloc : 0x60800000 - 0xff000000   (2536 MB)
    lowmem  : 0x40000000 - 0x60000000   ( 512 MB)
    pkmap   : 0x3fe00000 - 0x40000000   (   2 MB)
    modules : 0x3f000000 - 0x3fe00000   (  14 MB)
```

```
      .text : 0x40008000 - 0x40626b1c   (6267 kB)
      .init : 0x40627000 - 0x4065c000   ( 212 kB)
      .data : 0x4065c000 - 0x4069cb60   ( 259 kB)
       .bss : 0x4069cb60 - 0x406d3a78   ( 220 kB)
Preemptible hierarchical RCU implementation.
Dump stacks of tasks blocking RCU-preempt GP.
RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS:16 nr_irqs:16 16
L2C-310 erratum 769419 enabled
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 ID prefetch enabled, offset 1 lines
L2C-310 dynamic clock gating enabled, standby mode enabled
L2C-310 cache controller enabled, 8 ways, 512 kB
L2C-310: CACHE_ID 0x410000c8, AUX_CTRL 0x76360001
ps7-slcr mapped to 60804000
zynq_clock_init: clkc starts at 60804100
Zynq clock init
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 3298534883328ns
ps7-ttc #0 at 60806000, irq=43
Console: colour dummy device 80x30
Calibrating delay loop... 1332.01 BogoMIPS (lpj=6660096)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 1024 (order: 0, 4096 bytes)
Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes)
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x467598 - 0x4675f0
CPU1: Booted secondary processor
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
Brought up 2 CPUs
SMP: Total of 2 processors activated.
CPU: All CPU(s) started in SVC mode.
devtmpfs: initialized
VFP support v0.3: implementor 41 architecture 3 part 30 variant 9 rev 4
regulator-dummy: no parameters
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
cpuidle: using governor ladder
cpuidle: using governor menu
hw-breakpoint: found 5 (+1 reserved) breakpoint and 1 watchpoint registers.
hw-breakpoint: maximum watchpoint size is 4 bytes.
zynq-ocm f800c000.ps7-ocmc: ZYNQ OCM pool: 256 KiB @ 0x60880000
vgaarb: loaded
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
```

usbcore: registered new interface driver hub
usbcore: registered new device driver usb
media: Linux media interface: v0.10
Linux video capture interface: v2.00
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti <giometti@linux.it>
PTP clock support registered
EDAC MC: Ver: 3.0.0
Advanced Linux Sound Architecture Driver Initialized.
Switched to clocksource arm_global_timer
NET: Registered protocol family 2
TCP established hash table entries: 4096 (order: 2, 16384 bytes)
TCP bind hash table entries: 4096 (order: 3, 32768 bytes)
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.
PCI: CLS 0 bytes, default 64
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)
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 = 6249999) 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 f6:af:44:ec:fe:ee
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
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
sdhci-arasan e0100000.ps7-sdio: No vmmc regulator found
sdhci-arasan e0100000.ps7-sdio: No vqmmc regulator found
mmc0: SDHCI controller on e0100000.ps7-sdio [e0100000.ps7-sdio] using ADMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
TCP: cubic registered
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
zynq_pm_ioremap: no compatible node found for 'xlnx,zynq-ddrc-a05'
zynq_pm_late_init: Unable to map DDRC IO memory.
Registering SWP/SWPB emulation handler
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SC16G 14.8 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 the virutal address...
Physical Address: 0x43c00000
Virtual Address: 0x608e0000
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'

Result of testing the device file-

```
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SC16G 14.8 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 the virutal address...
Physical Address: 0x43c00000
Virtual Address: 0x608e0000
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'.
zynq> mknod /dev/multiplier c 245 0
zynq> ./devtest
Device has been opened correctly
Writing 0 to register 0
Writing 0 to register 1
0 * 0 = 0
Result Correct!

Writing 0 to register 0
Writing 1 to register 1
0 * 1 = 0
Result Correct!
h
Writing 0 to register 0
Writing 2 to register 1
0 * 2 = 0
Result Correct!
Writing 0 to register 0
Writing 3 to register 1
0 * 3 = 0
Result Correct!
k
Writing 0 to register 0
Writing 4 to register 1
```

0 * 4 = 0
Result Correct!
Writing 0 to register 0
Writing 5 to register 1
0 * 5 = 0
Result Correct!

Conclusion:

In this lab we learnt how to implement a character device driver for multiplication. We learnt the main purpose of using device driver is that it provides an abstraction layer between the underlying the hardware and the user application layer in the top most level .

Also in this approach the users will get the flexibitlity to write the user application code without making any modifications to the hardware used .

We also learnt that applications make use of system calls to have an interaction with the device driver using function arguements.

Questions:

1>ioremap is used to map the I/O devices's physical address to Kernel's virtual address. However, kernel developers designed ioremap command so that it does nothing even when mapped to directly mapped I/O addresses. Given that multiplier hardware uses memory mapped I/O, ioremap allows the device driver to access any I/O memory address whether it is directly mapped or not. Also, direct usage of pointers to access I/O memory is considered as a bad practice. Hence ioremap is used.

2> In lab 3 using hardware software codesign a custom peripheral was designed for multiplication which was integrated to the zybo board. Overall time for performing multiplication was faster in lab 3 than in part 3 of lab 6. There we use hardware software codesign principle to implement the multiplication where we created an .c file in sdk to test the multiply peripheral. The baud rate determined the speed of the multiplier.

3> Hardware approach is always faster than software approach, cost of implementing the hardware based solution like the one we use in lab 3 will always be relatively higher than the software approach. Using device driver we can readily and easily add the modules without disturbing the operating system.Cost for software solutions are usually lesser.

4>In the initialization routine for kernel, we add the character device driver to the kernel by the means of registering the device. Registration of the device requires the major number , device name and a pointer that points to the file operation structure and since the device can be immediately accessed once it is registered its is important to complete the intializtions before than we register the character device. In unregistering we are trying to release the module. Once we release the device file only then we can perform the unmappping of the virtual address using iounmap command.

Appendix:

:

Devtest.c file code:

```
#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, read_i, read_j;
        int fd; //file descriptor
        int i, j; //loop variables
        char input = 0;
```

```
//handle error opening file
        if (fd == -1) {
                printf("Failed to open the device  file\n");
                return -1;
        }



    char* output = (char*)malloc(3 * sizeof(int)); //for output
        unsigned int* mul = (unsigned int*)malloc(2 * sizeof(int));  //for input

        int* intBuff;

        fd = open("/dev/multiplier", O_RDWR); //file open



        while (input != 'q') { //continue until user decides to quit
                for (i = 0; i <= 16; i++) {
                        for (j = 0; j <= 16; j++) {

                  //for write function
                                char* chbuff = (char*)mul;


                  mul[0] = i;
                                mul[1] = j;
```

```
                write(fd, chbuff, 2 * sizeof(int)); //write to device file
                read(fd, output, 3 * sizeof(int)); //read to the device file

                intBuff = (int*)output; //casting

        read_i = intBuff[0];
                read_j = intBuff[1];
                result = intBuff[2];

                /*validate result */
                printf("%u * %u = %u\n", read_i, read_j, result);
                if (result == (i * j)) {
                        printf("Result   Correct!\n");
                }
                else {
                        printf("Result Incorrect!\n");
                }
                //read from terminal
                input = getchar();
            }
        }
    }
    close(fd);
    //free memory

free(output);
free(mul);
```

```
        return 0;
}
```

Code for multiplier.c

```
/* 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 <linux/slab.h> //needed for kmalloc() and kfree()
#include <asm/io.h> //needed for IO reads and writes
#include <asm/uaccess.h>   /* Provides utilities to bring user space
data into kernel space.  Note, it is
processor arch specific. */
#include "xparameters.h" //needed for physical address of the multiplier



/* Some defines */
#define DEVICE_NAME "multiplier"
#define BUF_LEN 80

/* Function prototypes, so we can setup the function pointers for dev
file access correctly. */
int my_init(void);
void my_cleanup(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 *);

/*
* Global variables are declared as static, so are global but only
* accessible within the file.
*/
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
};
//from xparameters.h
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR //physical address of multiplier
//size of physical address range for multiply
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR+1
void* virt_addr; //virtual address pointing to multiplier
```

```
/* This function is run upon mdoule load. This is where I setup data structs
and reserve resources used by the moudle */

/* This function is run just prior to the module's removal from the
system. I will release ALL resources used by my module here*/

int my_init(void) {

//map virtual address to multiplier physical address//use ioremap
virt_addr = ioremap(PHY_ADDR, MEMSIZE);
printk(KERN_INFO "Mapping the virutal address...\n");
printk("Physical Address: 0x%x\n", PHY_ADDR);
printk("Virtual Address: 0x%x\n", virt_addr);

/* 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); //dynamic allocation

/* Negative values indicate a problem */
if (Major < 0) {
printk(KERN_ALERT "Registering a char device has failed  %d\n", Major);
return Major;
}



printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);
```

```
printk(KERN_INFO "Use the command to create a device file:\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);

iounmap((void*)virt_addr);

printk(KERN_ALERT "unmapping the  virtual address space...\n");

}


/*

* 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)

{




// if (Device_Open) /* Device_Open is my flag for the
```

```
 // usage of the device file (definied
 // in my_chardev_mem.h)  */
   // return -EBUSY; /* Failure to open device is given
 // back to the userland program. */

//  Device_Open++; /* Keeping the count of the device
 // opens. */



  //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. */

  printk(KERN_INFO "Device has been opened correctly");
  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 */

  /*
   * 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(KERN_INFO"Device has been released");
 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)
{
/*
* Number of bytes written to the buffer
*/




//allocating kernel buffer (not all may be written to user space though)



    int* kernelBuffer = (int*)kmalloc(length * sizeof(int), GFP_KERNEL);
```

```
kernelBuffer[0] = ioread32(virt_addr);
kernelBuffer[1] = ioread32(virt_addr + 4);
kernelBuffer[2] = ioread32(virt_addr + 8);



/*
* Actually put the data into the buffer
*/



 char* Buffinchar = (char*)kernelBuffer; //bytes will be written one at a time
  int count_bytesread = 0;
for ( int z = 0; z < length; z++) {
/*
* 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(*(Buffinchar++), buffer++); /* one char at a time... */

count_bytesread++;
}

kfree(kernelBuffer);
/*
* Most read functions return the number of bytes put into the
* buffer
```

```
*/
return count_bytesread;
}

/*
* 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) {


  char* buffermessage = (char *)kmalloc(len*sizeof(char), GFP_KERNEL);

  /* printk(KERN_INFO "device_write(%p,%s,%d)", file, buffer, (int)length); */


int t=0;
  /* get_user pulls the data  from user space into kernel space */
while(t<len && t<BUF_LEN-1){

get_user(buffermessage[t], buff++); // place chars from user space into kernel buffer
t++;


}

 int* int_buffer = (int*)buffermessage;

  /* left one char early from buffer to leave space for null char*/
```

```
 buffermessage[t] = '\0';



 printk(KERN_INFO "Writing %d to register 0\n", int_buffer[0]); //wrtiting to reg 0

  iowrite32(int_buffer[0], virt_addr + 0); // base address plus offset

  printk(KERN_INFO "Writing %d to register 1\n", int_buffer[1]); //wrtiting to reg 1

 iowrite32(int_buffer[1], virt_addr + 4); //next address

  /*
   * Again, return the number of input characters used
   */

kfree(int_buffer);

return t;
}



/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("GRatz and others");
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);
```