ECEN 749: Microprocessor Systems Design

Lab 6: Introduction to Character Device Driver Development

Manav Gurumoorthy

830000011

Section 603

Table of Contents

Introduction 3
Procedure 4
  Configuration 4
  Creating a character device driver 4
  Creating the user space application 4
  multiplier.c 4
  devtest.c 8
Results 10
Conclusion 10
Questions 11

Introduction

Using specialised hardware on a Linux machine from a user application is made possible using device drivers. A device driver is software that interfaces the user space of the Linux to the Kernel space. In the context of this lab, a device driver was developed for the multiply hardware module. This device driver was then used to access the multiply hardware from a user space application.

Procedure

Configuration

This lab begins from the configuration done in the previous lab. There are no additional steps required once the Linux kernel was compiled for the Zynq board and the board was booted.

Creating a character device driver

The character device driver was created in order to create a way for a user space program to access the multiplication peripheral.

The device driver was created by modifying a skeleton code provided. The driver is responsible for device initialization; which includes memory mapping. The driver is also responsible for assigning a major number to the device. The driver also handles unregistering the device during the exit routine.

For the actual communication of data from the user space to the kernel space the functions get_user and put_user was used to perform read and write functions to the device.

Creating the user space application

The user application is used to read and write to the device character file. The user application uses linux system calls open(), close(), read and write to access the device character file. The user application is used to validate that the multiply module is computing the results accurately.

multiplier.c

```
/*  my_chardev_mem.c - Simple character device module
 *
 *  Demonstrates module creation of character device for user
 *  interaction.  Also utilizes/demostrates kmalloc and write.
 *
 *  (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 "my_chardev_mem.h"
#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
#include <asm/io.h>      //needed for IO read/write
#include <linux/moduleparam.h>//needed for module parameters
#include <linux/fs.h>    //file ops
#include <linux/sched.h>  //access to "current" processes structure
#include <asm/uaccess.h>  //utilites for userspace
#include "xparameters.h"  //physical multipler address
#include <linux/slab.h>
// virtual address pointing to multiplier
void* virt_addr;

// From xparameters.h, physical address of multiplier
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR
// Size of physical address range for multiply
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR
+ 1
//#define DEVICE_NAME "multiplier"
/* 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 called when the module is loaded and registers a
 * device for the driver to use.  We also allocate a little memory for
 * the driver to use as a backing store for data written to the device
 * file from userland, emulating a hardware device. Note: if there
 * were a real hardware device (with associated memory mapped io) we
 * wanted to read and write from we'd have to call ioremap to get a
 * kernel virtual memory address that maps to the physical address of
 * the device.
 */
int my_init(void)
{
        Major = register_chrdev(0, DEVICE_NAME, &fops);


  /* We need to allocate the memspace _BEFORE_ registering the device
     to avoid any race conditions */
  virt_addr = ioremap(PHY_ADDR, MEMSIZE);
  printk(KERN_INFO "PHY ADDR: %d", PHY_ADDR);
  printk(KERN_INFO "VIR ADDR: %p", virt_addr);
  /* Note: kmalloc can fail, even on a non-borked kernel, always exit
     gracefully.  In the event of a failure pointer will be set to
     NULL. */






  /* 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.*/


  /* 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);

    /* We won't need our memory so make sure to free it here... */
    //kfree(msg_bf_Ptr);
    iounmap((void*)virt_addr);


    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);

 iounmap((void*)virt_addr);
 /* free our memory (note the ordering here) */

}



/*
 * Called when a process tries to open the device file, like "cat
 * /dev/my_chardev_mem".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
{

  /* In these case we are only allowing one process to hold the device
     file open at a time. */
  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. */

  cur_Ptr = msg_bf_Ptr;           /* set the ptr to the beginning of the
                          message */

  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);

  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 actually written to the buffer
   */
  int bytes_read = 0;


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

 if (length <0 || length >12){
      printk(KERN_INFO "Invalid Length\n");
      return -1;
      }
    /*
     * 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.
     */
  while (length) {
    put_user(ioread8(virt_addr + bytes_read), buffer++); /* one char at a time...
*/
    length--;
    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)
{

  char* char_buf = (char*)kmalloc(length*sizeof(char),GFP_KERNEL);
  int i;
  int* temp = (int*)NULL;

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

  /* get_user pulls message from userspace into kernel space */
  for (i = 0; i < length; i++){
       get_user(char_buf[i], buffer + i);
       }

  temp = (int*)char_buf;
  //Write to Reg 1
  iowrite32(temp[0],virt_addr+0);
```

```
  //Write to Reg 2
  iowrite32(temp[1], virt_addr+4);

  kfree(char_buf);

  /*
   * Again, return the number of input characters used
   */
  return i;
}


/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Paul V. 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);
```

devtest.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

    int fd;     // File descriptor
    int i, j;   // Loop variables
    int rtrn = 0;
    char input = 0;
      char* temp = (char*)malloc(12); //char temp[12];
      unsigned int result=0, read_i =0, read_j=0;

    // Open device file for reading and writing
    // Use 'open' to open '/dev/multiplier'
   fd = open("/dev/my_chardev_mem", O_RDWR);
    // Handle error opening file
    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++) {
            // Write value to registers using char dev
            // Use write to write i and j to peripheral
                 int* array = (int*)malloc(sizeof(int)*2); //int data [2];
                 array[0] = i;
                 array[1] = j;
            // Read i, j, and result using char dev
                 rtrn = write(fd,array,sizeof(int)*2);
            // Use read to read from peripheral
```

```
            // print unsigned ints to screen
                 if(rtrn < 1) {
                       printf("Failed to write\n");
                       return -1;
                 }
                 rtrn = 0;
                 free(array);
                 rtrn = read(fd,temp,12);
                 if (rtrn == -1){
                       printf("Failed to read\n");
                       return -1;
                 }
                 printf("Bytes read: %d", rtrn);
                 int *int_data = (int*)temp;
                 read_i = int_data[0];
                 read_j = int_data[1];
                 result = int_data[2];
            printf("\n%u * %u = %u\n\r", read_i, read_j, result);

            // Validate result
            if(result == (i*j))
                printf("Result Correct!\n");
            else
                printf("Result Incorrect!");

            // Read from terminal
            input = getchar();
            }
        }
    }

    close(fd);
    return 0;
}
```

Results

The results the multiplication module is shown below. This is the output from the user application.

```
Result Correct!

Bytes read: 12
1 * 12 = 12
Result Correct!

Bytes read: 12
1 * 13 = 13
Result Correct!

Bytes read: 12
1 * 14 = 14
Result Correct!

Bytes read: 12
1 * 15 = 15
Result Correct!

Bytes read: 12
1 * 16 = 16
Result Correct!

Bytes read: 12
2 * 0 = 0
Result Correct!

Bytes read: 12
2 * 1 = 2
Result Correct!

Bytes read: 12
2 * 2 = 4
Result Correct!

Bytes read: 12
2 * 3 = 6
Result Correct!

Bytes read: 12
2 * 4 = 8
Result Correct!

Bytes read: 12
2 * 5 = 10
Result Correct!

Bytes read: 12
2 * 6 = 12
Result Correct!

Bytes read: 12
2 * 7 = 14
Result Correct!
```

Conclusion

Lab 6 describs the way the user space and kernel space interact with each other through device drivers.

Questions

1. The ioremap command is used to generate a virtual address for a physical address. The kernel can directly access the physical addresses of the hardware, but the user space is not given access to the physical memory space in linux, hence it needs a virtual address that is translated to a physical address by the kernel to access the device driver.
2. I expect it to be faster in Lab 3's implementation. I believe this is because in the implementation in this lab there is an overhead for the movement of data between the user space and the kernel space. Whereas in lab 3 this overhead was not present.
3. The approach in this lab requires developing a driver software and a user application. In terms of cost this is quite low but requires a deep understanding of the linux kernel. In contrast lab3's approach is a lot easier to implement but is a lot more expensive as we require the use of proprietary software.
4. Device registration must be the last step in the init routine as we want to ensure the device is ready and has all it's setup done before it is registered on the kernel. Un-registering the device must be the first step in the exit routine so that the device can't be accessed during the execution of the exit routine.