ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Sunil P Khatri
(Lab exercise created and tested by Ramu Endluri, He Zhou and Sunil P Khatri)

Laboratory Exercise #6

An Introduction to Linux Device Driver Development

Objective

The purpose of lab this week is to create device drivers in an embedded Linux environment. Device drivers are a part of the operating system kernel which serves as the bridge between user applications and hardware devices. Device drivers facilitate access and sharing of hardware devices under the control of the operating system.

Similar to Lab 5, you will create a kernel module, which prints messages to the kernel's message buffer. However, this kernel module will also moderate user access to the multiplication peripheral created in Lab 3. You will then extend the capabilities of your kernel module, thereby creating a complete character device driver. To test your multiplication device driver, you will also develop a simple Linux application which utilizes the device driver, providing the same functionality as seen in Lab 3.

System Overview

The hardware system you will use this week is that which was built in Lab 4. Figure 1 depicts a simplified view of the hardware and software you will create in this lab. Please note the hardware/software boundary in the figure below. Above this boundary, the blocks represent compiled code executing within the ARM processor. Below this boundary, the blocks represent hardware attached to the microprocessor. In our particular case, the hardware is a multiplication peripheral attached to the ARM Processor. Obviously, other hardware/software interactions exist in our system, but Figure 1 focuses on that which you will provide. Also

notice the existence of the kernel-space/user-space boundary. The kernel module represents the character device driver executing in kernel space, while the user application represents the code executing in user space, reading and writing to the device file, '/dev/multiplier'. The device file is not a standard file in the sense that it does not reside on a disk, rather it provides an interface for user applications to interact with the kernel.

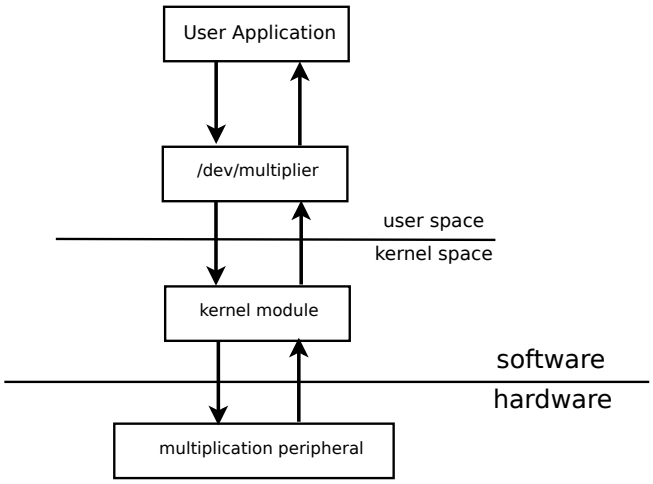

Figure 1: Hardware/Software System Diagram

Procedure

1. Compile a kernel module that reads and writes to the multiplication peripheral and prints the results to the kernel message buffer.

   (a) Create a lab6 directory and copy the 'modules' directory from your lab5 directory to your lab6 directory.

   (b) Navigate to the 'modules' directory and ensure that the 'Makefile' points to the Linux kernel directory.

(c) Create a new kernel module source file called 'multiply.c'.

(d) Copy the 'xparameters.h' and 'xparameters_ps.h' files in to modules directory

(e) Copy the following skeleton code into 'multiply.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 "xparameters.h" /*needed for physical address of multiplier*/

/*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 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
    /*write 7 to register 0 */
    printk(KERN_INFO "Writing a 7 to register 0\n");
    iowrite32( 7, virt_addr+0); //base address + offset
    /* Write 2 to register 1*/
    printk(KERN_INFO "Writing a 2 to register 1\n");
    //use iowrite32
    printk("Read %d from register 0\n", ioread32(virt_addr+0));
    printk("Read %d from register 1\n", \\use ioread32);
    printk("Read %d from register 2\n", ioread32(virt_addr+8));


    // 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");
    iounmap((void*)virt_addr);
}

/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("ECEN449 Student (and others)");
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);
```

(f) Some required function calls have been left out of the code above. Fill in the necessary code. Consult *Linux Device Drivers, 3rd Edition* for help.

Note: When running Linux on the ARM processor, your code is operating in virtual memory, and 'ioremap' provides the physical to virtual address translation required to read and write to hardware from within virtual memory. 'iounmap' reverses this mapping and is essentially the clean-up function for 'ioremap'.

(g) Add a 'printk' statement to your initialization routine that prints the physical and virtual addresses of your multiplication peripheral to the kernel message buffer.

(h) Edit your Makefile to compile 'multiply.c'. Compile your kernel module. Do not for get to source the 'settings64.sh' file.

(i) Load the 'multiply.ko' module onto the SD card and mount the card within the ZYBO Linux system using '/mnt' as the mount point. Consult Lab 5 if this is unclear.

(j) Use 'insmod' to load the 'multiply.ko' kernel module into the ZYBO Linux kernel. Demonstrate your progress to the TA.

2. With a functioning kernel module that reads and writes to the multiplication peripheral, you are now ready to create a character device driver that provides applications running in user space access to your multiplication peripheral.

(a) From within the 'modules' directory, create a character device driver called 'multiplier.c'using the following guidelines:

- Take a moment to examine 'my_chardev.c', 'my_chardev_mem.c' and the appropriate header files within '/home/faculty/shared/ECEN449/module_examples/'. Use the character device driver examples provided in *Linux Device Drivers, 3rd Edition* and the laboratory directory as a starting point for your device driver.

- Use the 'multiply.c' kernel module created in Section 2 of this lab as a baseline for reading and writing to the multiplication peripheral and mapping the multiplication peripheral's physical address to virtual memory.
- Within the initialization routine, you must register your character device driver after the virtual memory mapping. The name of your device should be 'multiplier'. Let Linux dynamically assign your device driver a major number and specify a minor number of 0. Print the major number to the kernel message buffer exactly as done in the examples provided. Be sure to handle device registration errors appropriately. You will be graded on this!
- In the exit routine, unregister the device driver before the virtual memory unmapping.
- For the open and close functions, do nothing except print to the kernel message buffer informing the user when the device is opened and closed.
- Read up on 'put_user' and 'get_user' in *Linux Device Drivers, 3rd Edition* as you will be using these system calls in your custom read and write functions.
- For the read function, your device driver must read bytes 0 through 11 within your peripheral's address range and put them into the user space buffer. Note that one of the parameters for the read function, 'length', specifies the number of bytes the user is requesting. Valid values for this parameter include 0 through 12. Your function should return the number of bytes actually transfered to user space (i.e. into the buffer pointed to by char* buf). You may use 'put_user' to transfer more than 1 byte at a time.
- For the write function, your device driver must copy bytes from the user buffer to kernel space using the system call 'get_user' to do the transfer and the variable 'length' as a specification of how many bytes to transfer. Furthermore, the device driver must write those bytes to the multiplication peripheral . The return value of your write function should be similar to that of your read function, returning the number of bytes successfully written to your multiplication peripheral. Only write to valid memory locations (i.e. 0 through 7) within your peripheral's address space.

(b) Modify the 'Makefile' to compile 'multiplier.c' and run
>make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
in the terminal to create the appropriate kernel module.

(c) As with 'multiply.ko', load 'multiplier.ko' into your ZYBO Linux system.

(d) Use 'dmesg' to view the output of the kernel module after it has been loaded. Follow the instructions provided within the provided example kernel modules to create the '/dev/multiplier' device node. Demonstrate your progress to the TA.

3. At this point, we need to create a user application that reads and writes to the device file, '/dev/multiplier', to test our character device driver, and provides the required functionality similar to that seen in Lab 3.

(a) View the man pages on 'open', 'close', 'read', and 'write' from within the CentOS workstation. You will use these system calls in your application code to read and write to the '/dev/multiplier' device file. Accessing the man pages can be done via the terminal. For example, to view the man pages on 'open', type 'man open' in a terminal window.

(b) Within the 'modules' directory, create a source file called 'devtest.c' and copy the following starter text into that file:

```
#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;
  int fd;                          /* file descriptor */
  int i,j;                         /* loop variables */

  char input = 0;

  /*open device file for reading and writing*/
  /*use 'open' to open '/dev/multiplier'*/

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

  while(input != 'q'){ /*continue unless user entered 'q'*/

    for(i=0; i<=16; i++){
      for(j=0; j<=16; j++){

        /*write values to registers using char dev*/
        /* use write to write i and j to peripheral */
        /*read i, j, and result using char dev*/
        /*use read to read from peripheral*/
        /*print unsigned ints to screen*/
        printf("%u * %u = %u ",read_i,read_j, result);

        /*validate result*/
        if(result == (i*j))
          printf("  Result Correct!");
        else
          printf("  Result Incorrect!");
```

```
            /*read from terminal*/
            input=getchar();
          }
        }
      }
      close(fd);
      return 0;
}
```

(c) Complete the skeleton code provided above using the specified system calls.

(d) Compile 'devtest.c' by executing the following command in the terminal window under the 'modules' directory:
>arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c

(e) Copy the executable file, 'devtest', on to the SD card and execute the application by typing the following in the terminal within the SD card mount point directory:
>./devtest

(f) Examine the output as you hit the 'enter' key from the terminal. Demonstrate your progress to the TA.

Deliverables

1. [5 points.] Demo the working kernel module and device driver to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [4 points.] Commented C files.
4. [2 points.] The output of the picocom terminal for parts 2 through 4 of lab.
5. [4 points.] Answers to the following questions:
   (a) Given that the multiplier hardware uses memory mapped I/O (the processor communicates with it through explicitly mapped physical addresses), why is the ioremap command required?
   (b) Do you expect that the overall (wall clock) time to perform a multiplication would be better in part 3 of this lab or in the original Lab 3 implementation? Why?
   (c) Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach?
   (d) Explain why it is important that the device registration is the last thing that is done in the initialization routine of a device driver. Likewise, explain why un-registering a device must happen first in the exit routine of a device driver.