Lab 6: Introduction to Kernel Modules on Zynq Linux System

Samuel Huang
ECEN 449

1. Introduction

The purpose of this lab is to create a device driver within an embedded Linux environment. The driver will create a bridge between the user applications and hardware devices, facilitating access and sharing of hardware under the control of the operating system. The implemented driver will extend upon the work done in Lab 5, to create a complete character device driver, and will be tested with a simple linux application that uses the device driver to multiply two numbers and obtain the result.

2. Procedure

The first part of the lab creates a new PetaLinux project and creates a character device driver called 'multiplier.c.' The structure of the code starts off with an initialization routine which completes the virtual memory mapping and registers the driver with the major number. The code then sets up an exit routine to unregister and unmap the virtual memory to properly disconnect it. The next part is to implement the open and close functions which simply print a message through the kernel buffer. The last part is to implement the functions that communicate between the user space and into the kernel space. The read function, reads bytes 0 to 11 from the peripherals address range and adds them into the user space, transferring 12 bytes from the kernel space to the user space. The write function copies bytes from the user buffer and transfers them into the kernel space. Both the read and write functions utilize put_user and get_user functions to conduct the transactions between the user and kernel space, along with keeping track of which position in the peripheral memory to read and write the data to to perform the correct mathematical operations.

The next part builds the PetaLinux project and loads the resulting driver '.ko' file into the Linux system. With the code complaining it is then tested by making a devtest.c file, which performs read and write operations through the device driver to compute test products. Once the devtest.c file is complete and checked that it properly compiles, it is loaded into the mounted SD card on the ZYBO Z-10 board and used to test the functionality of the created driver.

3. Results

The lab fully implements a character driver by building off of previous labs. The lab utilizes the created Linux operating system, to utilize the multiplication peripheral. The first part successfully creates the code for setting up a driver and making sure it is properly registered with the kernel, along with being able to communicate between the user and kernel space. The next part successfully creates a test code that checks that the created driver properly functions. Ultimately the lab demonstrates the ability to create a working device driver that can be utilized by programs run in the user space.

4. Conclusion

This lab gave a practical experience on creating a driver. The process of creating the driver base code, taught best practices and what the steps are to properly connect and disconnect a driver to the linux system. The lab also taught the importance of memory mapping, and how a driver can allow programs in the user space to interact with peripherals embedded in the hardware. While it was a struggle to get some of the memory mapping and proper code to make and test the driver, it provided experience on setting up drivers to a linux environment.

5. Questions

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?

- Ioremap is required because the processor does not have direct access to the physical address, but rather uses a virtual memory system in which ioremap maps the physical memory region used by the multiplier into this virtual space.

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?

- Lab 3 would be faster since it only works more directly with the hardware level and does not require communication between the userspace and kernel space to read and write values.

Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach?

- While lab 3 would be faster since it directly connects to the hardware, it is not as practical since values have to be pre-loaded or changed based on a signal input on the zybo-board. This means while it would be faster it would not have as wide spread of an application. The approach in this lab would be slower since there requires communication between the user space and kernel space and takes more complexity; however, the ability to connect software programs that run in the user space and utilize the multiplication peripheral greatly expands its application since it can be used the user relatively easily and incorporated into a wide variety of software programs.

Explain why it is important that the device registration is the last thing that is done in the initial-ization routine of a device driver. Likewise, explain why un-registering a device must happen first in the exit routine of a device driver?

- It is important to register the device after initialization because if any of the initialization steps fail the device should not be registered, and would allow for the possibility of users utilizing an improper device. Unregistering a device must be done first to make sure that the kernel and user know that the device is no longer available and that it can no longer be used, to prevent race conditions with clean up.

6. Appendix

multiply.c

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/io.h>
#include "xparameters.h"

#define DEVICE_NAME "multiplier"
#define BASE_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR
#define MEMSIZE 12

// module variables
static int major_num;
void __iomem *mapped_addr; // mapped address
static u8 kbuf[MEMSIZE];  // kernel buffer of 12 bytes

// function prototypes
static int multiplier_open(struct inode *inode, struct file *file);
static int multiplier_release(struct inode *inode, struct file *file);
static ssize_t multiplier_read(struct file *file, char __user *buf, size_t len, loff_t
*offset);
static ssize_t multiplier_write(struct file *file, const char __user *buf, size_t len,
loff_t *offset);
```

```c
// file operations
static const struct file_operations multiplier_fops = {
    .read = multiplier_read,
    .write = multiplier_write,
    .open = multiplier_open,
    .release = multiplier_release,
};

// initialization routine
static int __init multiplier_init(void) {
    // register the character device and assign a major number
    mapped_addr = ioremap(BASE_ADDR, MEMSIZE);
    if (!mapped_addr) {
        pr_err("Failed to map memory\n");
        return -ENOMEM;
    }

    major_num = register_chrdev(0, DEVICE_NAME, &multiplier_fops);
    if (major_num < 0) {
        pr_err("Failed to register device\n");
        iounmap(mapped_addr);
        return major_num;
    }
    pr_info("Registered device with major number %d\n", major_num);
    pr_info("Create a device file for this device with this command:\n"
        "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, major_num);
    return 0;
```

```
}


// exit routine
static void __exit multiplier_exit(void) {
    unregister_chrdev(major_num, DEVICE_NAME);
    iounmap(mapped_addr);
    pr_info("Unregistered device and unmapped memory\n");
}


// open function
static int multiplier_open(struct inode *inode, struct file *file) {
    pr_info("Multiplier device opened\n");
    return 0;
}


// release function
static int multiplier_release(struct inode *inode, struct file *file) {
    pr_info("Multiplier device closed\n");
    return 0;
}


// read function
static ssize_t multiplier_read(struct file *file, char __user *buf, size_t len, loff_t
*offset) {
    int bytes_to_copy = min(len, (size_t)MEMSIZE);  // only MEMSIZE should be
read
    int i;
```

```
    // read data from device memory into kbuf
    for (i = 0; i < bytes_to_copy; i++) {
        kbuf[i] = ioread8(mapped_addr + i);
    }

    // copy data from kbuf to user space
    if (copy_to_user(buf, kbuf, bytes_to_copy)) {
        pr_err("Failed to copy %zu bytes to user space\n", bytes_to_copy);
        return -EFAULT;
    }

    pr_info("Read %zu bytes from device\n", bytes_to_copy);
    return bytes_to_copy;
}

// write function
static ssize_t multiplier_write(struct file *file, const char __user *buf, size_t len,
loff_t *offset) {
    int bytes_to_copy = min(len, (size_t)8);  // ensure only 8 bytes are written
    int i;

    // copy data from user space to kbuf
    if (copy_from_user(kbuf, buf, bytes_to_copy)) {
        pr_err("Failed to copy %d bytes from user space\n", bytes_to_copy);
        return -EFAULT;
    }
```

```
    // write data from kbuf to device memory
    for (i = 0; i < bytes_to_copy; i++) {
        iowrite8(kbuf[i], mapped_addr + i);
    }

    pr_info("Wrote %d bytes to device\n", bytes_to_copy);
    return bytes_to_copy;
}

// module macros
module_init(multiplier_init);
module_exit(multiplier_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Samuel Huang");
MODULE_DESCRIPTION("Multiplier Device Driver");
```

devtest.c

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

int main() {
```

```c
int fd;  // File descriptor for the device
int i, j;  // Loop variables
char input = 0;

// open device file for reading and writing
fd = open("/dev/" DEVICE_NAME, O_RDWR);
if (fd == -1) {
    perror("Failed to open device file");
    return -1;
}

while (input != 'q') {  // Continue unless user enters 'q'
    for (i = 0; i <= 16; i++) {
        for (j = 0; j <= 16; j++) {
            // Write values i and j to the registers using char device
            int factor[2] = {i,j} // stores the 2 numbers which take up 8 bytes
            if (write(fd, result, 8) == -1) {
                perror("Failed to write i to device");
                close(fd);
                return -1;
            }
            int product[3];
            // Read result from the device
            if (read(fd, product, 12) == -1) {
                // loads i,j and the product as the last four bytes
                perror("Failed to read result from device");
```

```
            close(fd);
            return -1;
        }
        // Print result to screen
        printf("%u * %u = %u\n", product[0], product[1], product[2]);

        // Validate result
        if (product[2] == (i * j)) {
            printf("Result Correct!\n");
        } else {
            printf("Result Incorrect!\n");
        }

        // Read from terminal to continue or quit
        printf("Press 'Enter' to continue, 'q' to quit: ");
        input = getchar();
        getchar();  // Consume the newline character left in the input buffer
        if (input == 'q') break;
      }
      if (input == 'q') break;
    }
  }


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