ECEN 749

Laboratory Exercise #6

An Introduction to Character Device Driver Development

Name: Pranav Anantharam

UIN: 734003886

Course: ECEN 749 Microprocessor System Design

Section: 603

Introduction:

The objective of the sixth laboratory exercise was to create device drivers in an embedded Linux environment. Device drivers played a crucial role in the operating system kernel by enabling communication between user applications and hardware devices. This lab was important as it helped us understand the mechanisms behind device drivers and their role in facilitating access and sharing of hardware resources controlled by the operating system. We extended the capabilities of the kernel module from previous labs, creating a comprehensive character device driver. Additionally, we developed a Linux application to test the multiplication device driver, highlighting the practical application of this knowledge, like previous exercises, thereby reinforcing essential skills in embedded systems development.

Procedure:

1) Insert the USB drive that contains the installation and project directory of PetaLinux.
2) Run commands on the console to create a module called 'multiplier' using PetaLinux.
3) Using the sample files 'my_chardev.c' and 'my_chardev.h' for reference, write a character device driver to read and write to multiplication peripherals. The character device driver must register itself after virtual memory mapping and unregister itself before virtual memory unmapping.
4) Modify 'muliplier.bb' to include the required header files.
5) Build the multipler module using the command 'petalinux build'.
6) Transfer the built kernel module 'multiplier.ko' into the SD card and insert the SD card into the Zybo board.
7) Using the picocom serial console, run commands given in the manual to mount the SD card.
8) Using the command 'insmod multiplier.ko' load the kernel module onto the Zybo linux system.
9) Use the 'dmesg | tail' command to view the output of the kernel module after it is loaded.
10) Run the 'mknod' command to create the '/dev/multiplier' device node. The outputs were demonstrated to the TA.
11) Using the sample code given in the lab manual, write a user application 'devtest' to read and write to the device file '/dev/multiplier'.
12) Cross compile the user application using the command given in the lab manual.
13) Load the user application onto the SD card and insert the SD card into the Zybo board.
14) Execute the user application on the Zybo board. The outputs were demonstrated to the TA.

(Source code and Output Screenshots given in Appendix Section)

Results:

We were able to build and compile the multiplier device driver on the CentOS workstations and load the kernel module onto the Zybo Z7-10 board. The user application was also developed and built which generated inputs and used 'read' and 'write' operations to pass the inputs and get the outputs from the device driver. The 'multiplier' kernel character device driver interacted with the multiplier peripheral and the user application. The multiplication results were obtained from the multiplier hardware peripheral, loaded onto the kernel buffer by the device driver and passed to the user application. Outputs were obtained and displayed on a serial console using picocom application. The outputs were demonstrated to the TA as well.

Source code notes: multiplier.c:

- Introduced file operations structure to define device_read, device_write, device_open and device_close callback functions
- The device driver initialization function maps the virtual address space to multiplier peripheral physical address space and also performs character device driver registration with dynamic allocation of device driver major number
- Used kmalloc and kfree functions to dynamically allocate kernel buffer memory
- The device read operation reads the registers of the multiplier hardware peripheral using ‘ioread32’ function and writes into the kernel buffer. Next, the kernel buffer is copied into the user space buffer using ‘put_user’ function.
- The device write operations reads the user space buffer and copies into the kernel space buffer ‘get_user’ function. Next, the kernel space buffer is read and the values are written into the multiplier using ‘iowrite32’ function.

Source code changes: devtest.c:

- Defined macros for device file name, input length and output length
- Initialized an integer array buffer of size 3 to store values
- Use write function to write buffer values into device file (with typecasting to char datatype)
- Use read function to read from device file into buffer (with typecasting to char datatype)

(Source Code and Output Screenshots given in Appendix Section)

Conclusion:

In conclusion, the sixth laboratory exercise enhanced our understanding of device drivers in an embedded Linux environment. By exploring kernel modules and their role in facilitating communication between user applications and hardware devices, we gained valuable insights into system-level operations. Developing a character device driver and its accompanying Linux application reinforced essential skills in embedded systems development while highlighting the practical significance of device drivers in enabling seamless interaction with hardware resources. This knowledge equips us with the necessary tools to navigate and innovate within the domain of embedded systems, laying a solid foundation for future explorations in this field.

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?

Answer:

The multiplier block interfaces as an I/O device to the ARM processor, and the code necessary for interacting with this I/O device runs on the ARM processor in virtual memory space. A successful call to ioremap() returns a kernel virtual address corresponding to the start of the requested physical address range. Since the I/O memory is accessed through the page tables, the kernel must execute ioremap, which provides the entries of the physical address of the device in the page table, through which the device is accessed.

(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?

Answer:

No, I expect the multiplication operation would be faster in the original Lab 3 implementation, as the C code is directly interfacing with the hardware. In this lab, however, the C code must pass from the device driver to reach the hardware. Also, there might be an additional delay if the OS is busy with some operation.

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

Answer:

Benefits of the approach in this lab:

- A higher level of abstraction is provided to users. Users do not need to have knowledge of the underlying hardware and the methods to interact with the register. Users only need to create an application file, pass inputs, and receive outputs from the same. The internal implementation of the hardware is completely hidden.
- The implementation is not limited by the hardware, it can be repurposed or additional hardware can be supported with code changes.

Costs of the approach in this lab:

- Longer execution time / Performance overhead since additional layer of software is added
- Developing a device driver is complex since it must be compatible with the hardware peripheral
- Lack of visibility of internal implementation

Benefits of approach in Lab 3:

- Lower latency / Faster execution times since the kernel directly interacts with the hardware peripheral without an intermediate device driver layer.
- Increased efficiency since fewer hardware resources (memory, CPU cycles) are used.

Costs of approach in Lab 3:

- In-depth knowledge of the hardware design of the peripheral is required. In Lab 3, we generated the bitstream and loaded it on the FPGA board and hence, we required an FPGA board and knowledge about the register space.
- The implementation is limited to the specific hardware peripheral and cannot be easily extended to support additional hardware.

(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.

Answer:

The device registration is done last in the initialization routine because, once the device is registered, the kernel can make calls to the module even before the initialization is finished. For example, if the call is made before any allocation of buffer, it might lead to unexpected behavior. Also, once device registration is performed, the device will be made available to the user application for read/write operations, and so the device must have allocated memory and performed virtual address mapping before this step. Additionally, the device is unregistered in the exit routine first so that the kernel cannot make calls to the module to access the device once it is in the exit routine.

Appendix:

Multiplier Device Driver Registration Console Output:

File Edit View Search Terminal Help

```
linux_boot:/mnt# ls
BOOT.BIN       boot.scr       devtest        image.ub       multiplier.ko
linux_boot:/mnt# insmod multiplier.ko
multiplier: loading out-of-tree module taints kernel.
Mapping virtual address...
Physical Address = 55b71922
Virtual Address = 10b941d2
Registered a device with dynamic Major number of 244
Create a device file for this device with this command:
'mknod /dev/multiplier c 244 0'.
linux_boot:/mnt#
```

```
Create a device file for this device with this command:
'mknod /dev/multiplier c 244 0'.
linux_boot:/mnt# mknod /dev/multiplier c 244 0
linux_boot:/mnt# ls /dev/
block            loop0       null    ram3    tty10   tty22   tty34   tty46   tty58               vcs
char             loop1       port    ram4    tty11   tty23   tty35   tty47   tty59               vcs1
console          loop2       ptmx    ram5    tty12   tty24   tty36   tty48   tty6                vcsa
cpu_dma_latency  loop3       pts     ram6    tty13   tty25   tty37   tty49   tty60               vcsa1
disk             loop4       ram0    ram7    tty14   tty26   tty38   tty5    tty61               vcsu
fpga0            loop5       ram1    ram8    tty15   tty27   tty39   tty50   tty62               vcsu1
full             loop6       ram10   ram9    tty16   tty28   tty4    tty51   tty63               vga_arbiter
gpiochip0        loop7       ram11   random  tty17   tty29   tty40   tty52   tty7                watchdog
iio:device0      mem         ram12   shm     tty18   tty3    tty41   tty53   tty8                watchdog0
initctl          mmcblk0     ram13   snd     tty19   tty30   tty42   tty54   tty9                zero
kmsg             mmcblk0p1   ram14   tty     tty2    tty31   tty43   tty55   ttyPS0
log              mtab        ram15   tty0    tty20   tty32   tty44   tty56   udev_network_queue
loop-control     multiplier  ram2    tty1    tty21   tty33   tty45   tty57   urandom
linux_boot:/mnt#
```

User Application Console Output:

```
File  Edit  View  Search  Terminal  Help
Kernel Buffer[2] = 1
1 * 1 = 1
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 2
Kernel Buffer[2] = 2
1 * 2 = 2
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 3
Kernel Buffer[2] = 3
1 * 3 = 3
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 4
Kernel Buffer[2] = 4
1 * 4 = 4
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 5
Kernel Buffer[2] = 5
1 * 5 = 5
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 6
Kernel Buffer[2] = 6
1 * 6 = 6
Result Correct!



Kernel Buffer[0] = 1
Kernel Buffer[1] = 7
Kernel Buffer[2] = 7
1 * 7 = 7
Result Correct!
```

```
File  Edit  View  Search  Terminal  Help
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 10
Kernel Buffer[2] = 110
11 * 10 = 110
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 11
Kernel Buffer[2] = 121
11 * 11 = 121
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 12
Kernel Buffer[2] = 132
11 * 12 = 132
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 13
Kernel Buffer[2] = 143
11 * 13 = 143
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 14
Kernel Buffer[2] = 154
11 * 14 = 154
Result Correct!



Kernel Buffer[0] = 11
Kernel Buffer[1] = 15
Kernel Buffer[2] = 165
11 * 15 = 165
Result Correct!
q
Device closed succesfully
linux_boot:/mnt#
```

Multiplier Device Driver Source Code:

```c
#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 <asm/uaccess.h>   /* Provides utilities to bring user space data into kernel
space.  Note, it is processor arch specific. */
#include <linux/slab.h>    /* kmalloc and kfree definitions */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <asm/io.h>
#include "xparameters.h"
#include <linux/ioport.h>   // IO memory allocation

// From xparameters.h
#define PHY_ADDR  XPAR_MULTIPLY_0_S00_AXI_BASEADDR // physical address of multiplier
```

```c
// Size of physical address range for multiply
#define MEMSIZE   XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR + 1

/* 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 init_module(void);
void cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_close(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 *);

// Virtual address pointing to multiplier
void* virt_addr;

/*
 * 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 */
static int Device_Open = 0; /* Flag to signify open device */


/* 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_close
};

static int __init my_init(void)
{

  printk(KERN_INFO "Mapping virtual address... \n");

  // map virtual address to multiplier physical address
  virt_addr = ioremap(PHY_ADDR, MEMSIZE);

  printk(KERN_INFO "Physical Address = %p \n", PHY_ADDR);
  printk(KERN_INFO "Virtual Address = %p \n", virt_addr);
```

```c
  /* 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);
    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 */
}

static void __exit my_exit(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/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)
{
/* 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.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("Device opened successfully \n");
  return 0;
}


/*
 * Called when a process closes the device file.
 */
static int device_close(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("Device closed succesfully \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)
{
  /*
   * Number of bytes actually written to the buffer
   */
  int bytes_read = 0;

  // Allocating kernel buffer
  int * kernel_buffer = (int*)kmalloc(length * sizeof(int), GFP_KERNEL );

  // Read values from registers into kernel buffer
```

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

  printk("Kernel Buffer[0] = %d \n", kernel_buffer[0]);
  printk("Kernel Buffer[1] = %d \n", kernel_buffer[1]);
  printk("Kernel Buffer[2] = %d \n", kernel_buffer[2]);

  // Using char pointer
  char * kbuf = (char*)kernel_buffer;
  int i ;
  for( i = 0; i < length; i++ )
  {
  // Write / copy values from kernel space to user space buffer
  put_user(*(kbuf++), buffer++); // one char at a time...
  bytes_read++;
  }


  kfree(kernel_buffer);

  /*
   * 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)
{
  int i;

  char * kernel_buffer = (char*)kmalloc((len+1)*sizeof(char), GFP_KERNEL);

  for( i = 0; i < len; i++ )
  {
    // Get / copy values from user space into kernel buffer
    get_user(kernel_buffer[i], buff+i);
  }

  kernel_buffer[len] = '\0';

  // Using int pointer
  int * int_buf = (int*)kernel_buffer;

  printk(KERN_INFO, "Writing %d to register 0 \n", int_buf[0]);
```

```
  iowrite32(int_buf[0], virt_addr+0);

  printk(KERN_INFO, "Writing %d to register 0 \n", int_buf[1]);
  iowrite32(int_buf[1], virt_addr+4);

  kfree(int_buf);

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


/* Display information that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Pranav Anantharam");
MODULE_DESCRIPTION("Multiplier Character Device Driver");

/* Functions to use for initialization and cleanup */
module_init(my_init);
module_exit(my_exit);
```

User Application Source Code:

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

#define DEVICE_FILE     "/dev/multiplier"
#define INPUT_LEN       8
#define RESULT_LEN      12

int main()
{
    unsigned int read_i, read_j, result;
    int fd;                 /* file descriptor */
    int i,j;                /* loop variables */
    int buffer[3] = {0};
    char input = 0;

    fd = open(DEVICE_FILE, O_RDWR);

    if( fd == -1 )
```

```c
    {
        printf("Failed to open device file! \n");
        return -1;
    }

    for( i = 0; i <= 16; i++)
    {
        for( j = 0; j <= 16; j++ )
        {
            /* Write values to registers using char dev */
            if( input != 'q' )  /* Continue unless user entered 'q' */
            {
                /* Use write to write i and j to peripheral */
                buffer[0] = i;
                buffer[1] = j;

                write(fd, (char*)&buffer, INPUT_LEN);

                /* Read i, j and result using char dev */

                /* Use read to read from peripheral */
                read(fd, (char*)&buffer, RESULT_LEN);

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

                /* Print unsigned ints to screen */
                printf("%u * %u = %u \n", read_i, read_j, result);

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

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

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