EMBEDDED LINUX KERNEL DEVELOPMENT REPORT LAB 2

1 INTRODUCTION

This LAB is dedicated to learning how to develop a Linux Driver, and use it through a userspace application.

We will start from where the LAB1 ends, meaning we will use the custom Linux kernel running on the Zedboard, and develop a LED control application based on e LED Driver.

2 PREPARATION

In this lab we will write a kernel driver that allows communication with a physical hardware device (a character driver). In Linux, hardware devices are accessed by the user through special device files. To create a device file, we will use the mknod command. We will use 255 as a major number because it is not already uses by another device driver.

3 LED DRIVER

The LED Driver must be able to switch on and of each LEDs by writing a character from the userspace application.

For example, if we write 0x1 to the Driver, the LED1 turns on. If we write 0xFF, all LEDs turs on.

We use the character Driver template available at http://users.polytech.unice.fr/~bilavarn/fichier/elec5_linux/Ch6-DeviceDrivers.pdf, and we add the memory allocation and memory mapping of the LEDs.

To test if the character Driver is working, we try sending a Hello_World message to the kernel and displaying it.

```
int mydevice_file;
char *msg_passed = "Hello World !!\n";
char *msg_received;
int msg_length;

msg_length = strlen(msg_passed);
msg_received = malloc(msg_length);

mydevice_file = open(MYDEVICE_PATH, O_RDWR);
if (mydevice_file == -1)
{
  printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
  exit(EXIT_FAILURE);
}

// BASIC WRITE/READ TEST
write(mydevice_file, msg_passed, msg_length);
read(mydevice_file, msg_received, msg_length);
printf("write/read test: %s\n", msg_received);
```

Figure 1- userspace program sending Hello World to kernel

We can install the Driver using the following commands:

$ mknod /dev/mydevice c 255 0

$ insmod mydevice.ko

Then running the userspace program:

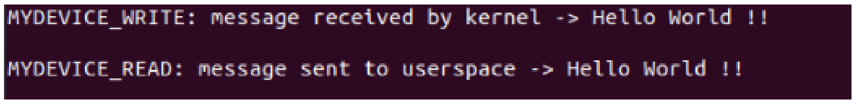
$ ./test_driver

And removing the Driver:

$ rmmod mydevice.ko

We can display the kernel prompt using the command:

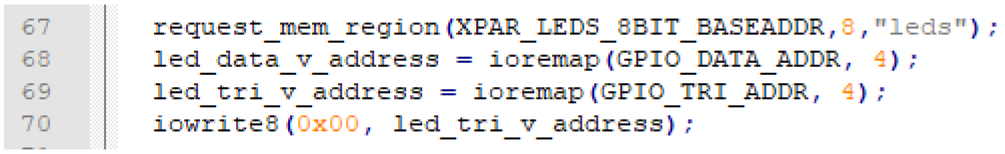
$ dmesg|tail



```
MYDEVICE_WRITE: message received by kernel -> Hello World !!

MYDEVICE_READ: message sent to userspace -> Hello World !!
```

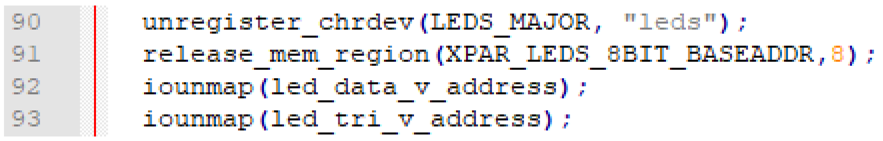
Figure 2- Hello World test program

We cannot write the values of the LEDs directly to the physical address of the LEDs registers, because Linux uses virtual addresses. We first need to reserve the memory space we are going to use to make sure that no other driver writes to that memory space while our program is running. We request the memory space using *request_mem_region*. To get the virtual address associated with the LEDs, we need to use the *ioremap* function. This function returns a pointer corresponding to virtual address of the physical components.



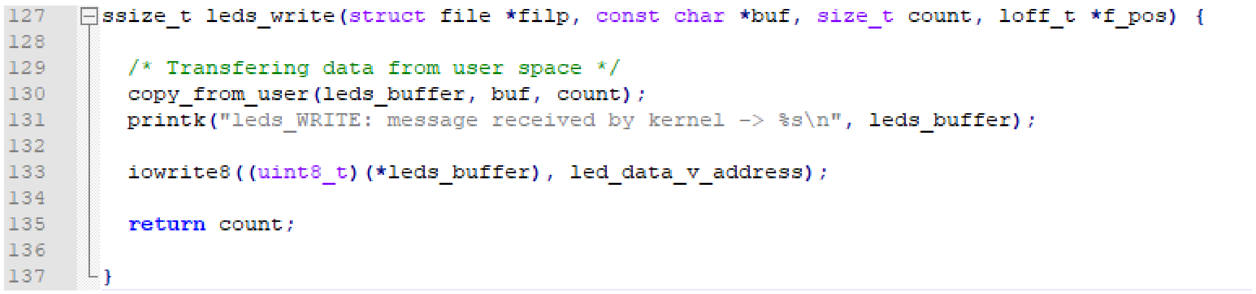
```
    request_mem_region(XPAR_LEDS_8BIT_BASEADDR,8,"leds");
    led_data_v_address = ioremap(GPIO_DATA_ADDR, 4);
    led_tri_v_address = ioremap(GPIO_TRI_ADDR, 4);
    iowrite8(0x00, led_tri_v_address);
```

Figure 3- memory allocation of LEDs



```
    unregister_chrdev(LEDS_MAJOR, "leds");
    release_mem_region(XPAR_LEDS_8BIT_BASEADDR,8);
    iounmap(led_data_v_address);
    iounmap(led_tri_v_address);
```

Figure 4- memory and MAJOR NUMBER release

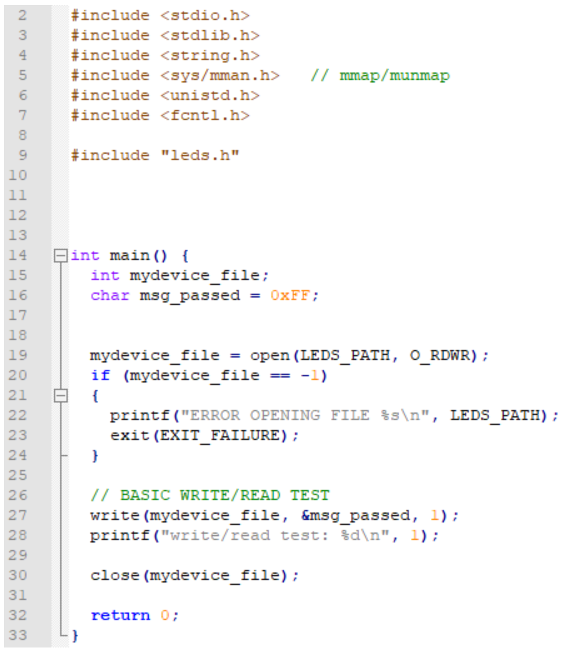


```c
ssize_t leds_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos) {

    /* Transfering data from user space */
    copy_from_user(leds_buffer, buf, count);
    printk("leds_WRITE: message received by kernel -> %s\n", leds_buffer);

    iowrite8((uint8_t)(*leds_buffer), led_data_v_address);

    return count;

}
```

Figure 5- write function to switch on/off the LEDs

4 USERSPACE TEST APPLICATION

To test the driver we wrote earlier, we write a userspace application that turns all the LEDs on using our LED driver.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>   // mmap/munmap
#include <unistd.h>
#include <fcntl.h>

#include "leds.h"




int main() {
  int mydevice_file;
  char msg_passed = 0xFF;


  mydevice_file = open(LEDS_PATH, O_RDWR);
  if (mydevice_file == -1)
  {
    printf("ERROR OPENING FILE %s\n", LEDS_PATH);
    exit(EXIT_FAILURE);
  }

  // BASIC WRITE/READ TEST
  write(mydevice_file, &msg_passed, 1);
  printf("write/read test: %d\n", 1);

  close(mydevice_file);

  return 0;
}
```

Figure 6 - led_driver.c program

The program opens the program file used to drive the LEDs, then writes 0xFF to that file. Each bit of the constant represents the state of one of the eight LEDs. Writing 0xFF should turn on all the LEDs.

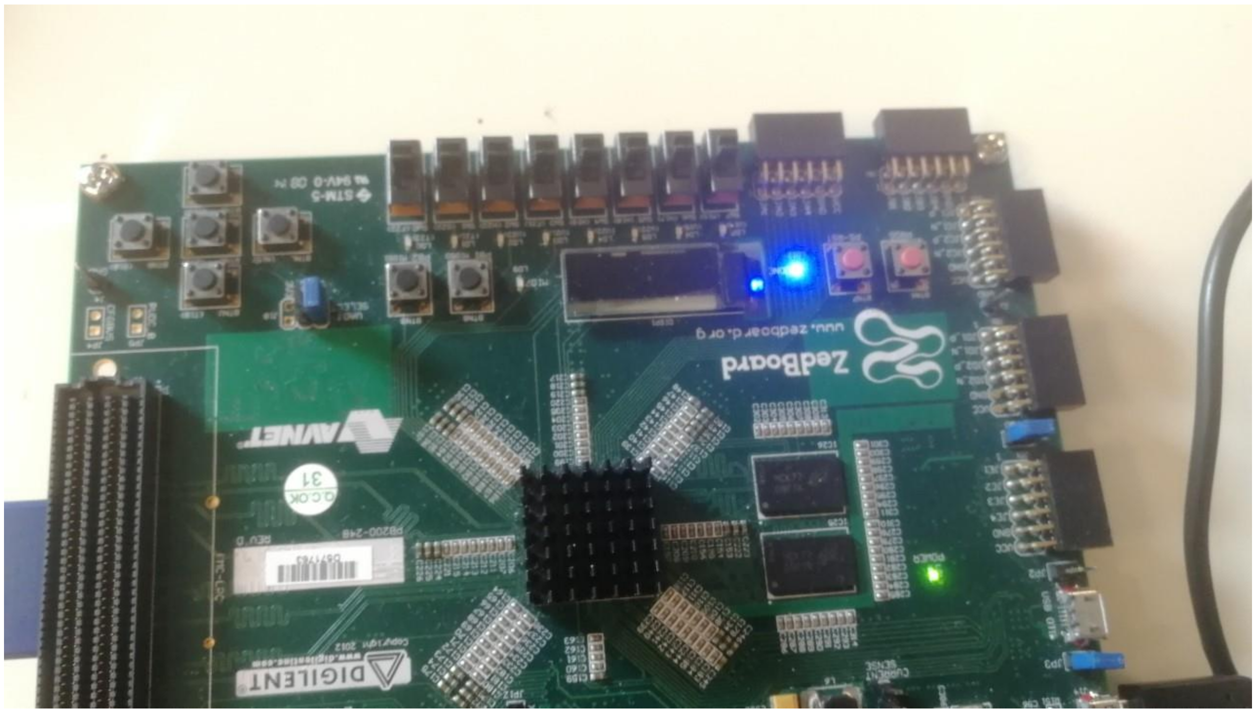


Figure 7- State of the LEDs before executing the userspace program

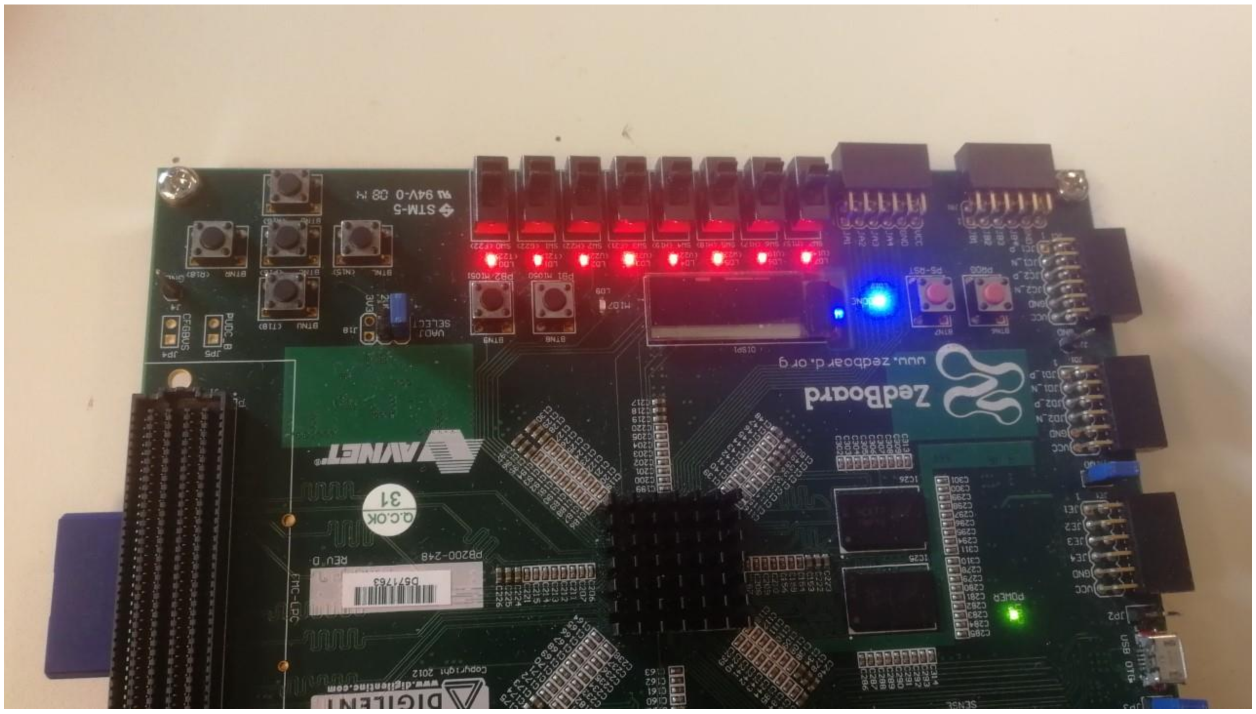


Figure 8- State of the LEDs after executing the userspace program

5 CONCLUSION

Writing a Linux kernel Driver is difficult and error prone.

But using custom Drivers can be really interesting in an embedded application, as we can control every aspects of our program.

After writing and adding a Driver to our kernel, we can interface with it really easily from the userspace using the device file of the Driver.