Creating a Basic LED Driver for Raspberry Pi

Author: Ramiro Rubio / Author: Rafael Lopez / Author: Victor Ramirez

April 27 2017

1 Abstract

Manipulate a LED sounds pretty simple when you do it by analog logic, or digital logic. But when you need to communicate the software and hardware, and from the user space to Kernel space is much more complicated, there are restrictions that the software have to protect the hardware. So we need to create a new driver to communicate the user space and the Kernel space because we need to manipulate the GPIO, in this case plugged to a LED. We need this to set the instructions for the LED to blink, turn on, turn off and set the PWM time and duty cycle.

2 Introduction

This report covers the solution of the final project of the Operating System course, this project approaches us more to the practical functionality of some topics learned at the course. And let us to define a more clear line between the different kinds of drivers (Char, Block, and Network). Implementing and using one of them (char driver), to solve the problem proposed below. The purpose of this paper is to present a solution to the LED problem, using a driver on a RaspberryPi 2, we will use systemcalls as the communication from user space to kerne space to use the GPIO that we need to turn on/off the LED.

3 Theoretical Framework

There are a lot of documentation about turn on a LED but with high level language, but there are few documents talking about the Kernel and how to manipulate it in comparison. Implement in a Linux Embedded board (RaspberryPi2) a char driver able to control a GPIO, the challenge was no using any high level language like python or C++, neither their libraries that simplify the GPIO handling. The solution includes libraries with basic functions to drive high or low the output of an onboard pin. An application of this library needs to be presented to prove the proper functioning of the char driver.

4 Objective

The main objective of this work will be to prove that a LED can be manipulated with out the use of a high level language and with out libraries, just using a char driver to control a GPIO.

5 Justification

This project was developed for learning purpose, the need for more knowledge about the Kernel and drivers. We can actually see the effects of a driver working physically by manipulating the software and watching the performance on a LED.

6 Development

First we need a Kernel and its headers, we found a lot of problems because our Kernel version doesn't match with our headers version. When we try to run our driver there was a lot of error about the libraries, so that was our first problem. We solved it matching the Kernel version with the headers version, it was a little bit complicated because we didn't find both of them that fast.

7 Solution

A char driver was implemented as a loadable Kernel module, since it's a driver a node needs to be created in order to transmit information between the application and the driver. An option to transmit information to this driver is trough user space using commands like "echo" to channel a string into the driver's node. The other way is using a library that we made using system calls like "write" to change the value of the GPIO inside of the application.

8 Conclusion

Kernel versions and Kernel headers must match for a module to build. We actually see a working char driver and also how to communicate an application with a Kernel module. Currently the PWM mode is not exact, because we don't use a high definition timer. The PWM mode is implemented in a blocking fashion. If we have more time, we surely can implement more precise timers and tools like alarms to avoid the blocking implementation that is actually running.

9 References

References

[1] https://github.com/Aribababa/Sistemas-Operativos

10 Appendix

Manipulate a LED sounds pretty simple when you do it by analog logic, or digital logic. But when you need to communicate the software and hardware, and from the user space to Kernel space is much more complicated, there are restrictions that the software have to protect the hardware. So we need to create a new driver to communicate the user space and the Kernel space because we need to manipulate the GPIO, in this case plugged to a LED. We need this to set the instructions for the LED to blink, turn on, turn off and set the PWM time and duty cycle.

Code for the driver

```
/*
 * Header for the GPIO driver
 * Authors:
 *     Ramiro Manuel Rubio Contreras (ramirorubioc@gmail.com)
 *     Rafael Lopez Pena           (rfa.lopez.pena@gmail.com)
 * Victor Ramirez Zepeda          (victorrz319@gmail.com)
 *
 */

#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

#include <linux/fs.h>
#include <asm/uaccess.h>

#include <linux/gpio.h>
#include <asm/gpio.h>

#define DEVICE_NAME   "gpio_device"
#define DEVICE_MAJOR  240
#define GPIO_num      4

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("GPIO driver");
MODULE_AUTHOR("RRV");

static unsigned long procfs_buffer_size = 0;
static char buffer_data[3];
static char led_status = 0;


static int dev_open(struct inode *inode,struct file *file);
static int dev_release(struct inode *inode,struct file *file);
static ssize_t dev_write(struct file *file, const char *buffer, size_t
    len, loff_t *offset);


static struct file_operations fops =
{
    .owner        = THIS_MODULE,
```

```c
    .open          = dev_open,
    .release       = dev_release,
    .write         = dev_write,
};

int dev_init(void)
{
    int ret;

    ret = register_chrdev(DEVICE_MAJOR,DEVICE_NAME,&fops);
    if(ret < 0){
        printk(KERN_ALERT "Device registeration failed \n");
    }
    else{
        printk(KERN_ALERT "Device registeration succeed \n");
    }

    ret = gpio_request(GPIO_num,"gpio_test");
    if(ret != 0){
        printk(KERN_ALERT "GPIO%d is not requested\n",GPIO_num);
    }
    else{
        printk(KERN_ALERT "GPIO%d is requested\n",GPIO_num);
    }

    ret = gpio_direction_output(GPIO_num,0);
    if(ret != 0){
        printk(KERN_ALERT "GPIO%d in not set output\n",GPIO_num);
    }
    else{
        printk(KERN_ALERT "GPIO%d is set output and out is
            low\n",GPIO_num);
    }

    return 0;
}

void dev_exit(void)
{
        printk(KERN_ALERT "module exit\n");

    gpio_free(GPIO_num);

    unregister_chrdev(DEVICE_MAJOR,DEVICE_NAME);

}
static int dev_open(struct inode *inode,struct file *file)
{
    gpio_set_value(GPIO_num, led_status);

    printk(KERN_ALERT "GPIO%d is set high\n",GPIO_num);

    return 0;
}
```

```c
static int dev_release(struct inode *inode,struct file *file)
{
    gpio_set_value(GPIO_num, led_status);

    printk(KERN_ALERT "GPIO%d is ser low\n",GPIO_num);

    return 0;
}

static ssize_t dev_write(struct file *file, const char *buffer, size_t
     len, loff_t *offset)
{
    procfs_buffer_size = len;
    if ( copy_from_user(buffer_data, buffer, procfs_buffer_size) )
    {
        return -EFAULT;
    }

    *offset += len;

    if(buffer_data[0] == '1')
    {
        led_status = 1;

    }
    else if(buffer_data[0] == '0')
    {
        led_status = 0;
    }

    pr_info("user input string: %s\n",buffer_data);
    pr_info("user input string len: %lu\n",procfs_buffer_size);

    return procfs_buffer_size;
}

module_init(dev_init);
module_exit(dev_exit);
```

Library for the driver

```c
/*
 * Header for the GPIO driver
 * Authors:
 *     Ramiro Manuel Rubio Contreras (ramirorubioc@gmail.com)
 *     Rafael Lopez Pena       (rfa.lopez.pena@gmail.com)
 * Victor Ramirez Zepeda   (victorrz319@gmail.com)
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

void led_on(void);
       /* Prende el LED asociado con el driver */

void led_off(void);
       /* Apaga el LED asociado con el driver */

void PWM(unsigned long duty_cycle);
       /* Genera una senal de PWM*/

void PWM_period_us(unsigned long _period);
       /* Coloca un nuevo periodo al PWM */

unsigned long period = 1000000; /* El periodod actual del PWM*/
                              /* Por default el periodo es de 1 segundo
                                  */

void led_on(void)
{
       signed int fd, ret;
       fd = open("/dev/gpio_driver", O_RDWR);

       if (fd < 0)
       {
       perror("Failed to open the device...");
   }

   ret = write(fd, "00", 1);
   if (ret < 0){
       perror("Failed to write the message to the device.");
  }
}


void led_off(void)
{
       signed int fd, ret;
       fd = open("/dev/gpio_driver", O_RDWR); /* Abrimos el archivo del
           driver */
       /* Verificamos si es que pudimos hacer el SysCall */
       if (fd < 0)
       {
           perror("Failed to open the device...");
       }

   ret = write(fd, "11", 1); /* Escribimos en el nodo para cambiar el
       led */
   /* En caso de que ocurra un error */
   if (ret < 0)
   {
         perror("Failed to write the message to the device.");
```

```
    }
}

void PWM(unsigned long duty_cycle)
{
        duty_cycle = duty_cycle * (period/100);
        while(1)
        {
                led_off();
                usleep(duty_cycle);

                led_on();
                usleep(period - duty_cycle);
        }
}


void PWM_period_us(unsigned long _period)
{
    period=_period;
}
```

Code for a simple test

```
/*
*  Basic test for GPIO Driver
*  Authors:
*     Ramiro Manuel Rubio
*     Victor Ramirez
*     Rafael Lopez
*/

#include "gpio_driver.h"

int main(void)
{
   PWM_period_us(2000000);
   PWM(50);
}
```