Embedded Linux device drivers/Markku Nuutinen 03 May 2014
Student name: Vu Nguyen (1005157)

Final project – Writing a GPIO device driver for Raspberry Pi platform

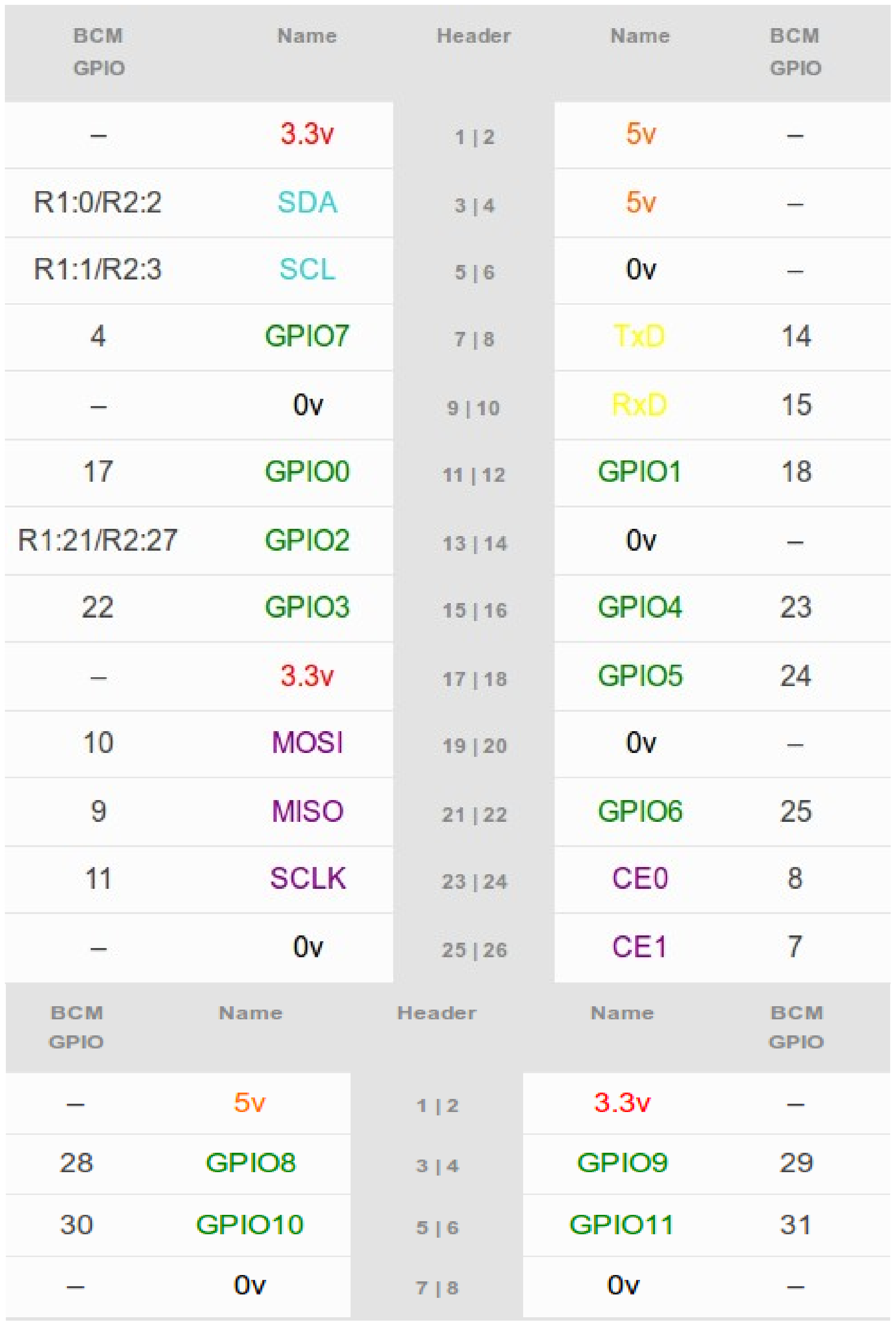
1. Introduction

The Raspberry Pi is a small Linux computer which is cheap and versatile. This platform was used in the Embedded Linux device drivers course at Helsinki Metropolia University of Applied Sciences. As part of the course, a final project was carried out to implement a GPIO device driver for the Raspberry Pi. The Linux kernel that comes with the Raspberry Pi supports a built-in device driver for GPIO by exposing the driver's interface for use in userspace via “/sys/class/gpio” interface. However, the purpose of the final project was to reinvent the wheel by implementing a new custom kernel module that plugs into the Linux kernel to manage access to the GPIO pins. Userspace applications can make requests to the kernel module to use the Raspberry Pi GPIO pins in a safe way. This report is going to represent briefly the design, implementation and testing of the GPIO device driver.

2. Design

The GPIO device driver (kernel module) offers a set of ioctl commands which userspace applications can use to make requests to the device driver for accessing GPIO pins. This set of commands allows applications to request, release, set mode (direction), read, write, and toggle GPIO pins on Raspberry Pi. Since reading operation does not modify the state of a GPIO pin, any userspace process can read a GPIO pin simultaneously. There is a static array of the device driver in which the index of the array represents the GPIO pin number and the value in each array element stores state information of a GPIO pin. The purpose of having the array is to control access to a GPIO pin so that race conditions are prevented. Each array element has three states: reserved, busy, and free. The reserved state is for system use. The device driver rejects any request to these reserved state pins. The busy state indicates the GPIO pin is occupied by a process. The free state indicates the GPIO pin is available for reservation. GPIO pins in reserved or busy state cannot be requested by any process for writing. This is a mechanism used for protecting concurrent access to a shared resource, which could cause problems known as race conditions.

There are in total 54 GPIO pins from the Broadcom SoC on the Raspberry Pi. But only 21 pins can be used as GPIO pins. Among these pins, 17 pins are in P1 header and four pins are in P5 header. The P5 header only exists in Raspberry Pi revision 2.0 model B which was used in the final project. Figure 1 shows 21 GPIO pins that are on the P1 and P2 headers.



BCM GPIO	Name	Header	Name	BCM GPIO
–	3.3v	1 | 2	5v	–
R1:0/R2:2	SDA	3 | 4	5v	–
R1:1/R2:3	SCL	5 | 6	0v	–
4	GPIO7	7 | 8	TxD	14
–	0v	9 | 10	RxD	15
17	GPIO0	11 | 12	GPIO1	18
R1:21/R2:27	GPIO2	13 | 14	0v	–
22	GPIO3	15 | 16	GPIO4	23
–	3.3v	17 | 18	GPIO5	24
10	MOSI	19 | 20	0v	–
9	MISO	21 | 22	GPIO6	25
11	SCLK	23 | 24	CE0	8
–	0v	25 | 26	CE1	7

BCM GPIO	Name	Header	Name	BCM GPIO
–	5v	1 | 2	3.3v	–
28	GPIO8	3 | 4	GPIO9	29
30	GPIO10	5 | 6	GPIO11	31
–	0v	7 | 8	0v	–

Figure 1. Mapping between GPIO pin numbers of Broadcom SoC and Raspberry PI pin header
Reprinted from Henderson (2013) [1]

In particular, these 21 GPIO pins are: 2, 3, 4, 7, 8, 9, 10, 11, 14, 15, 17, 18, 22, 23, 24, 25, 27, 28, 29, 30, and 31. When a certain process makes a request to reserve a GPIO pin, the GPIO device driver makes sure that the pin requested is in the range from 0 to 31. Next it refers to the pin array to check for the state of the requested GPIO pin. If the pin is in free state, the state of the pin will be replaced by the process id (pid) of the requesting process. The state values are represented by 32-bit unsigned integers. The free state is represented by 0 and the reserved state is represented by 1. The maximum value of pid in Linux is 32768; therefore, a 32-bit unsigned integer is sufficient to represent the pid of a process in the static pin array.

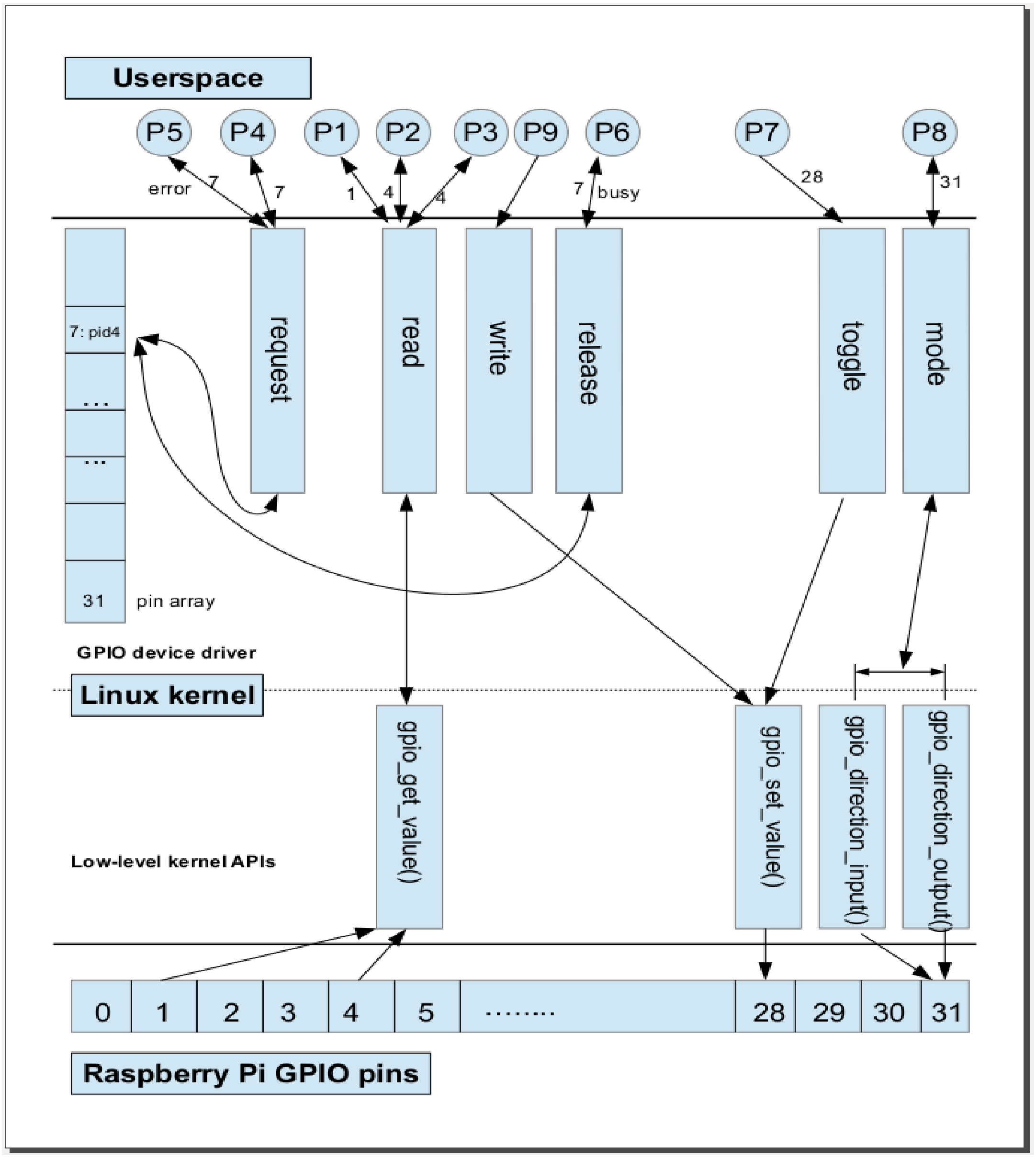


Figure 2. Interaction between userspace processes, GPIO device driver, low-level kernel APIs, and Raspberry Pi GPIO pins

In addition to having a pin array, the device driver also has a pin direction array which stores the direction of GPIO pins. The array is used so that an input GPIO pin cannot be set to low/high logic state by a process.

3. Implementation

The implementation of the GPIO device driver resulted in three files: a C source for, a header file and a Makefile file. Listing 1 shows the source code implemented for the GPIO device driver.

```
/*
 * modgpio.c - GPIO device driver for Raspberry Pi rev 2.0 model B
 *
 * Created on: 29 April 2014
 * Author: Vu Nguyen <quangngmetro@gmail.com>
 * License: GPL
 * Modified from Blake Bourque
 */
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/stat.h>
#include <linux/device.h>
#include <linux/fs.h>
#include <linux/err.h>
#include <uapi/asm-generic/errno-base.h>
#include <linux/ioctl.h>
#include <linux/gpio.h>
#include <asm-generic/uaccess.h>
#include <linux/types.h>
#include <linux/sched.h>

#include "modgpio.h"

// Prefix "rpigpio_ / RPIGPIO_" is used in this module to avoid name pollution
#define RPIGPIO_MOD_AUTH     "Vu Nguyen"
#define RPIGPIO_MOD_DESC     "GPIO device driver for Raspberry Pi"
#define RPIGPIO_MOD_SDEV     "Raspberry Pi rev 2.0 model B"
#define RPIGPIO_MOD_NAME     "rpigpio"

#define PIN_RESERVED   1     // reserved pins for system use
#define PIN_FREE       0     // available pins for request
#define PIN_ARRAY_LEN 32

static int rpigpio_open(struct inode *inode, struct file *filp);
static int rpigpio_release(struct inode *inode, struct file *filp);
static long rpigpio_ioctl( struct file *filp,
                           unsigned int cmd,
                           unsigned long arg);

// Global variables
static int deviceOpenCounter = 0;
struct rpigpio_dev {
      int               mjr;
      struct class      *cls;
      spinlock_t        lock;
      uint32_t          pin_state_arr[PIN_ARRAY_LEN];
      PIN_DIRECTION_t pin_dir_arr[PIN_ARRAY_LEN];
};
```

```c
static struct rpigpio_dev std = {
            .mjr = 0,
            .cls = NULL,
};

static const struct file_operations rpigpio_fops = {
            .owner      =                 THIS_MODULE,
            .open =                 rpigpio_open,
            .release =              rpigpio_release,
            .unlocked_ioctl =       rpigpio_ioctl,
};

// Implementation of entry points
static int
rpigpio_open(struct inode*inode, struct file *filp)
{
      spin_lock(&std.lock);
      deviceOpenCounter++;
      spin_unlock(&std.lock);
      try_module_get(THIS_MODULE);

      return 0;
}

static int
rpigpio_release(struct inode *inode, struct file *filp)
{
      int i = 0;

      spin_lock(&std.lock);
      deviceOpenCounter--;
      if (deviceOpenCounter == 0) {
            for (i = 0; i < PIN_ARRAY_LEN; i++) {
                  if (  i != 0 && i != 1 && i != 5 && i != 6 &&
                        i != 12 && i != 13 && i != 16 && i != 19 &&
                        i != 20 && i != 21 && i != 26) {
                        printk(KERN_DEBUG "[FREE] Pin:%d\n", i);
                        std.pin_state_arr[i] = PIN_FREE;
                  }
            }
      }
      spin_unlock(&std.lock);
      module_put(THIS_MODULE);

      return 0;
}

static long
rpigpio_ioctl(  struct file *filp, unsigned int cmd, unsigned long arg)
{
      int pin;
      unsigned long ret;
      int retval;
      uint8_t val;
```

```
struct gpio_data_write wdata;
struct gpio_data_mode mdata;

switch (cmd) {
case GPIO_REQUEST:
      get_user (pin, (int __user *) arg);
      spin_lock(&std.lock);
      if (  pin > PIN_ARRAY_LEN || pin < 0 ||
            std.pin_state_arr[pin] == PIN_RESERVED) {
            spin_unlock(&std.lock);
            return -EFAULT;
      } else if (std.pin_state_arr[pin] != PIN_FREE) {
            spin_unlock(&std.lock);
            return -EBUSY;
      }
      std.pin_state_arr[pin] = current->pid;
      spin_unlock(&std.lock);
      printk(KERN_DEBUG "[REQUEST] Pin:%d Assn To:%d\n", pin, current->pid);
      return 0;

case GPIO_FREE:
      get_user (pin, (int __user *) arg);
      spin_lock(&std.lock);
      if (  pin > PIN_ARRAY_LEN || pin < 0 ||
            std.pin_state_arr[pin] == PIN_RESERVED) {
            spin_unlock(&std.lock);
            return -EFAULT;
      } else if (std.pin_state_arr[pin] != current->pid) {
            spin_unlock(&std.lock);
            return -EACCES;
      }
      std.pin_state_arr[pin] = PIN_FREE;
      spin_unlock(&std.lock);
      printk(KERN_DEBUG "[FREE] Pin:%d From:%d\n", pin, current->pid);
      return 0;

case GPIO_MODE:
      ret = copy_from_user(   &mdata, (struct gpio_data_mode __user *)arg,
                              sizeof(struct gpio_data_mode));
      if (ret != 0) {
            printk(KERN_DEBUG "[MODE] Error copying data from userspace\n");
            return -EFAULT;
      }
      spin_lock(&std.lock);
      if (  mdata.pin > 31 || mdata.pin < 0 ||
            std.pin_state_arr[mdata.pin] == PIN_RESERVED) {
            spin_unlock(&std.lock);
            return -EFAULT;
      } else if (std.pin_state_arr[mdata.pin] != current->pid) {
            spin_unlock(&std.lock);
            return -EACCES;
      }

      if(mdata.data == MODE_INPUT) {
            retval = gpio_direction_input(mdata.pin);
```

```
                if (retval < 0) {
                        spin_unlock(&std.lock);
                        return retval;
                }
                std.pin_dir_arr[mdata.pin] = DIRECTION_IN;
                printk(KERN_DEBUG "[MODE] Pin %d set as Input\n", mdata.pin);
        } else if (mdata.data == MODE_OUTPUT) {
                retval = gpio_direction_output(mdata.pin, 1);
                if (retval < 0) {
                        spin_unlock(&std.lock);
                        return retval;
                }
                std.pin_dir_arr[mdata.pin] = DIRECTION_OUT;
                printk(KERN_DEBUG "[MODE] Pin %d set as Output\n", mdata.pin);
        } else {
                spin_unlock(&std.lock);
                return -EINVAL;
        }
        spin_unlock(&std.lock);
        return 0;

case GPIO_READ:
        get_user (pin, (int __user *) arg);
        val = gpio_get_value(pin);
        printk(KERN_DEBUG "[READ] Pin: %d Val:%d\n", pin, val);
        put_user(val, (uint8_t __user *)arg);
        return 0;

case GPIO_WRITE:
        ret = copy_from_user(   &wdata, (struct gpio_data_write __user *)arg,
                                sizeof(struct gpio_data_write));
        if (ret != 0) {
                printk(KERN_DEBUG "[WRITE] Error copying data from userspace\n");
                return -EFAULT;
        }
        spin_lock(&std.lock);
        if (std.pin_state_arr[wdata.pin] != current->pid) {
                spin_unlock(&std.lock);
                return -EACCES;
        }
        if (std.pin_dir_arr[wdata.pin] == DIRECTION_IN) {
                printk(KERN_DEBUG "Cannot set Input pin\n");
                spin_unlock(&std.lock);
                return -EACCES;
        }
        if (wdata.data == 1)
                gpio_set_value(wdata.pin, 1);
        else
                gpio_set_value(wdata.pin, 0);
        spin_unlock(&std.lock);
        printk(KERN_INFO "[WRITE] Pin: %d Val:%d\n", wdata.pin, wdata.data);

        return 0;

case GPIO_TOGGLE:
			get_user (pin, (int __user *) arg);
			spin_lock(&std.lock);
			if (  pin > PIN_ARRAY_LEN || pin < 0 ||
				std.pin_state_arr[pin] == PIN_RESERVED) {
				spin_unlock(&std.lock);
				return -EFAULT;
			} else if (std.pin_state_arr[pin] != current->pid) {
				spin_unlock(&std.lock);
				return -EACCES;
			}
			if (std.pin_dir_arr[pin] == DIRECTION_IN) {
				printk(KERN_DEBUG "Cannot set Input pin\n");
				spin_unlock(&std.lock);
				return -EACCES;
			}
			val = gpio_get_value(pin);
			if (val > 0) {
				gpio_set_value(pin, 0);
			} else {
				gpio_set_value(pin, 1);
			}
			put_user(val?0:1, (uint8_t __user *)arg);
			spin_unlock(&std.lock);
			printk(KERN_DEBUG "[TOGGLE] Pin:%d From:%.1d To:%.1d\n", pin, val,
				 val?0:1);

			return 0;

		default:
			return -ENOTTY;
		}
}

// Sets permissions for device file
static char *st_devnode(struct device *dev, umode_t *mode)
{
		if (mode) *mode = 0666;//add a leading 0 to make number octal
		return NULL;
}

static int __init
rpigpio_minit(void)
{
		int i = 0;
		int retval;
		struct device *dev;

		// Register char device
		std.mjr = register_chrdev(0, RPIGPIO_MOD_NAME, &rpigpio_fops);
		if (std.mjr < 0) {
			printk(KERN_ALERT "[gpio] Cannot Register");
			return std.mjr;
		}
		printk(KERN_INFO "[gpio] Major #%d\n", std.mjr);
```

```
	// Create class in /sys directory
	std.cls = class_create(THIS_MODULE, "std.cls");
	if (IS_ERR(std.cls)) {
		printk(KERN_ALERT "[gpio] Cannot get class\n");
		unregister_chrdev(std.mjr, RPIGPIO_MOD_NAME);
		return PTR_ERR(std.cls);
	}

	std.cls->devnode = st_devnode;

	// Create device file in /dev directory
	dev = device_create(std.cls,
			    NULL,
			    MKDEV(std.mjr, 0),
			    (void*)&std,
			    RPIGPIO_MOD_NAME);
	if (IS_ERR(dev)) {
		printk(KERN_ALERT "[gpio] Cannot create device\n");
		class_destroy(std.cls);
		unregister_chrdev(std.mjr, RPIGPIO_MOD_NAME);
		return PTR_ERR(dev);
	}

	// Initialize the spinlock
	spin_lock_init(&(std.lock));

	// Initialize the per-device structure
	for (i = 0; i < PIN_ARRAY_LEN; i++) {
		if (i != 0 && i != 1 && i != 5 && i != 6 &&
			i != 12 && i != 13 && i != 16 && i != 19 &&
			i != 20 && i != 21 && i != 26) {
			std.pin_state_arr[i] = PIN_FREE;
			retval = gpio_request(i, NULL);
			if (retval < 0)
				return retval;
		} else {
			std.pin_state_arr[i] = PIN_RESERVED;
		}
		std.pin_dir_arr[i] = DIRECTION_OUT;
	}
	printk(KERN_INFO "[gpio] %s Installed\n", RPIGPIO_MOD_NAME);

	return 0;
}

static void __exit rpigpio_mcleanup(void)
{
	int i = 0;

	for (i = 0; i < PIN_ARRAY_LEN; i++) {
		if (i != 0 && i != 1 && i != 5 && i != 6 &&
			i != 12 && i != 13 && i != 16 && i != 19 &&
			i != 20 && i != 21 && i != 26) {
			std.pin_state_arr[i] = PIN_FREE;
			gpio_free(i);
```

```
            }
        }
        device_destroy(std.cls, MKDEV(std.mjr, 0));
        class_destroy(std.cls);
        unregister_chrdev(std.mjr, RPIGPIO_MOD_NAME);

        printk(KERN_NOTICE "[gpio] Removed\n");
}

module_init(rpigpio_minit);
module_exit(rpigpio_mcleanup);

MODULE_LICENSE("GPL");
MODULE_AUTHOR(RPIGPIO_MOD_AUTH);
MODULE_DESCRIPTION(RPIGPIO_MOD_DESC);
MODULE_SUPPORTED_DEVICE(RPIGPIO_MOD_SDEV);
```

Listing 1. Source code for GPIO device driver

Listing 2 shows the header file for the C source code of the GPIO device driver. This header file is needed for both the C source file of the GPIO device driver and the application program that uses the GPIO device driver.

```
/*
 * modgpio.h – Header file for GPIO device driver
 *
 * Created on: 29 April 2014
 * Author: Vu Nguyen <quangngmetro@gmail.com>
 * License: GPL
 * Modified from Blake Bourque
 */

#ifndef MODGPIO_H_
#define MODGPIO_H_

#define GPIO_IOC_MAGIC 'k'

typedef enum {MODE_INPUT=0, MODE_OUTPUT} PIN_MODE_t;
typedef enum {DIRECTION_IN = 0, DIRECTION_OUT} PIN_DIRECTION_t;

struct gpio_data_write {
        int pin;
        char data;
};

struct gpio_data_mode {
        int pin;
        PIN_MODE_t data;
};

//in: pin to read //out: value //the value read on the pin
#define GPIO_READ _IOWR(GPIO_IOC_MAGIC, 0x90, int)
```

```
//in: struct(pin, data) //out: NONE
#define GPIO_WRITE _IOW(GPIO_IOC_MAGIC, 0x91, struct gpio_data_write)

//in: pin to request //out: success/fail // request exclusive modify privileges
#define GPIO_REQUEST _IOW(GPIO_IOC_MAGIC, 0x92, int)

//in: pin to free
#define GPIO_FREE _IOW(GPIO_IOC_MAGIC, 0x93, int)

//in: pin to toggle //out: new value
#define GPIO_TOGGLE _IOWR(GPIO_IOC_MAGIC, 0x94, int)

//in: struct (pin, mode[i/o])
#define GPIO_MODE _IOW(GPIO_IOC_MAGIC, 0x95, struct gpio_data_mode)

#endif /* MODGPIO_H_ */
```

Listing 2. Header file for GPIO device driver

Listing 3 shows the Makefile file that was used to compile the GPIO device driver.

```
MODULE_NAME = modgpio

KDIR = /home/quangng/study/linux_device_drivers/Metropolia/labs/final_year_project/
ldd_raspi_gpio_driver/kernel/Rpi_kernel/linux

TOOLCHAIN = /home/quangng/study/linux_device_drivers/Metropolia/labs/
final_year_project/ldd_raspi_gpio_driver/buildtools/tools/arm-bcm2708/
gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-

TARGET = arm
PWD := $(shell pwd)

obj-m := $(MODULE_NAME).o

all:
	make -C $(KDIR) M=$(PWD) ARCH=$(TARGET) CROSS_COMPILE=$(TOOLCHAIN) modules

clean:
	make -C $(KDIR) M=$(PWD) ARCH=$(TARGET) CROSS_COMPILE=$(TOOLCHAIN) clean
```

Listing 3. Makefile file for compiling GPIO device driver

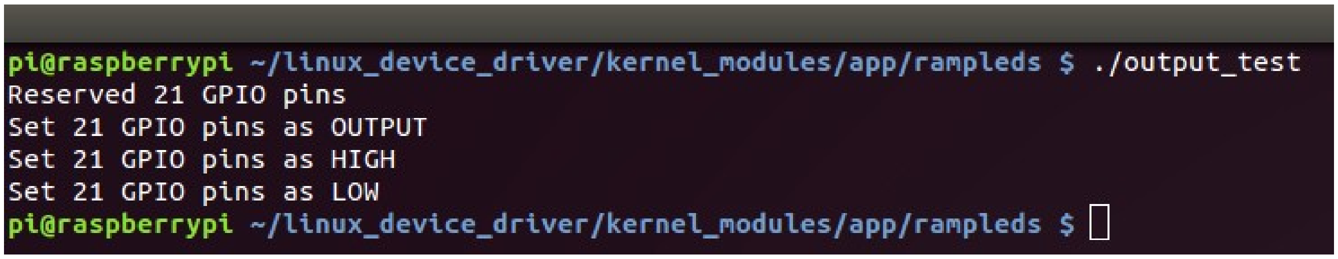
4. Testing

The testing of the GPIO device driver was divided into the following test cases.

- Output functionality test case
- Input functionality test case
- Direction collision test case
- Process collision test case

4.1 Output functionality test case

The purpose of the output functionality test case was to verify to that the GPIO device driver was able to set the direction of GPIO pins as output and set the logic state of GPIO pins as high/low. The application program used for this test case was called *output_test.c.* The application program works by first reserving 21 GPIO pins on Raspberry Pi. Next, it sets all these 21 GPIO pins as output and then sets the logic state of the pins as high. After that, it sleeps for one second and then sets all the state of all the GPIO pins as low. The source code of the application program is included in appendix 1. Figure 3 shows a snapshot of the output functionality test case shown in the terminal.



```
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/rampleds $ ./output_test
Reserved 21 GPIO pins
Set 21 GPIO pins as OUTPUT
Set 21 GPIO pins as HIGH
Set 21 GPIO pins as LOW
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/rampleds $
```

Figure 3. Result of output functionality test case shown in terminal

Figure 4 shows 17 LEDs were lit up when the GPIO pins were set as output, high logic state. Figure 5 shows 17 LEDs were off when the GPIO pins were set as output, low logic state. It should be noted that four GPIO pins in the P5 header were also set as output, low/high logic state. These pins were not connected to LEDs to verify its operating. Instead, a digital multimeter (DMM) was used to verify these GPIO pins. When the GPIO pins were set as output, high logic state, the voltage reading of these four GPIO pins were about 2.99V, meaning that the logic state of the GPIO pins were high. In contrast, when the GPIO pins were set as output, low logic state, the voltage reading of these four GPIO pins were about 0V, meaning that the logic state of the GPIO pins were low.

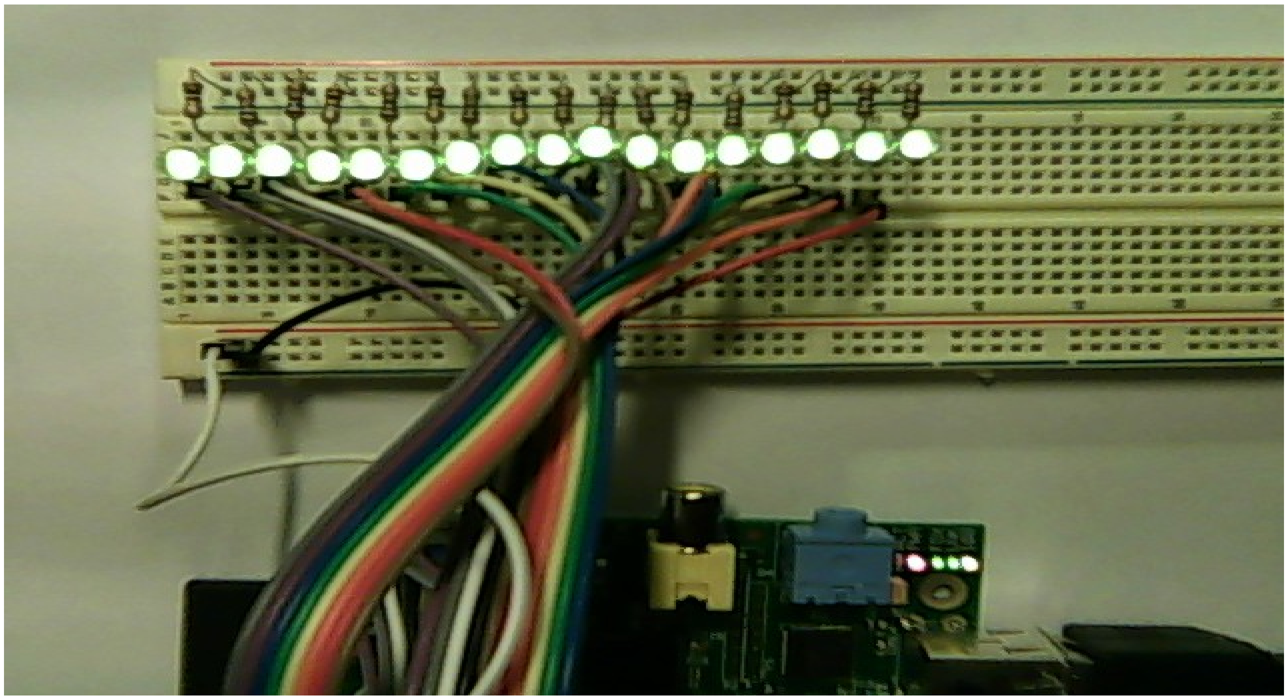


Figure 4. GPIO pins set as output, high state in output functionality test case

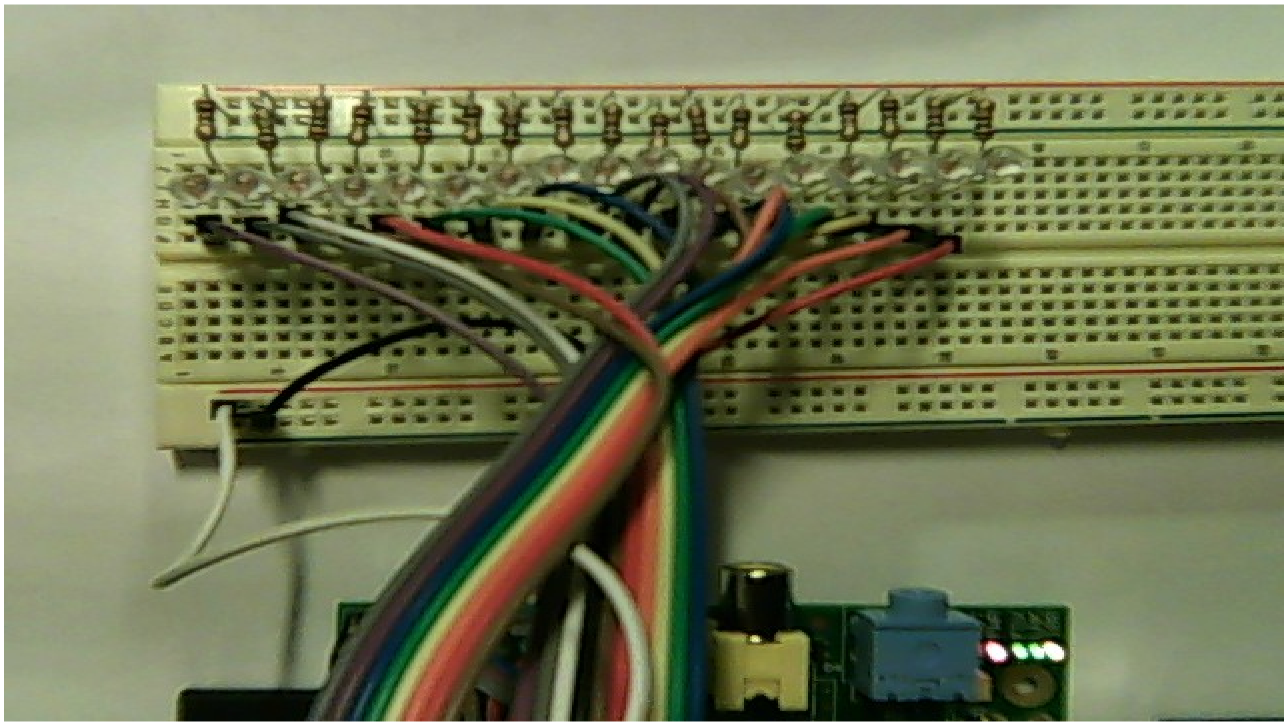


Figure 5. GPIO pins set as output, low state in output functionality test case

4.2 Input functionality test case

The purpose of the input functionality test case was to verify to that the GPIO device driver was able to set the direction of GPIO pins as input and read the logic state of these input GPIO pins. The application program used for this test case was called *input_test.c.* The application program works by first reserving 21 GPIO pins on Raspberry Pi. Next, it sets all these 21 GPIO pins as input and then reads the logic state of the pins. The source code of the application program is included in appendix 2. Figure 6 shows the hardware setup for testing input functionality where all GPIO pins were connected directly to GND. Figure 7 shows a snapshot of the input functionality test case shown in the terminal when 17 GPIO pins on P1 header of Raspberry Pi were all grounded.

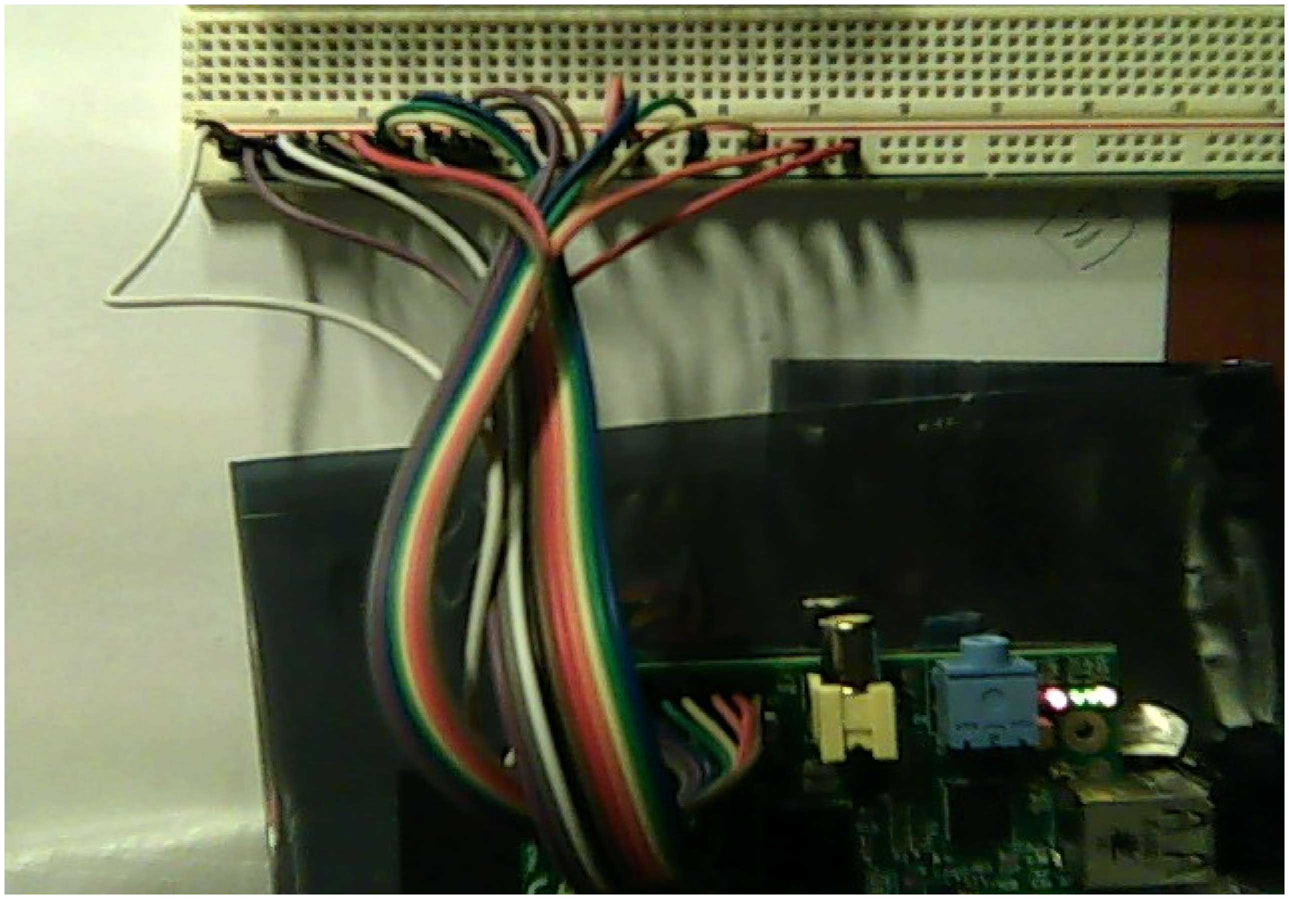
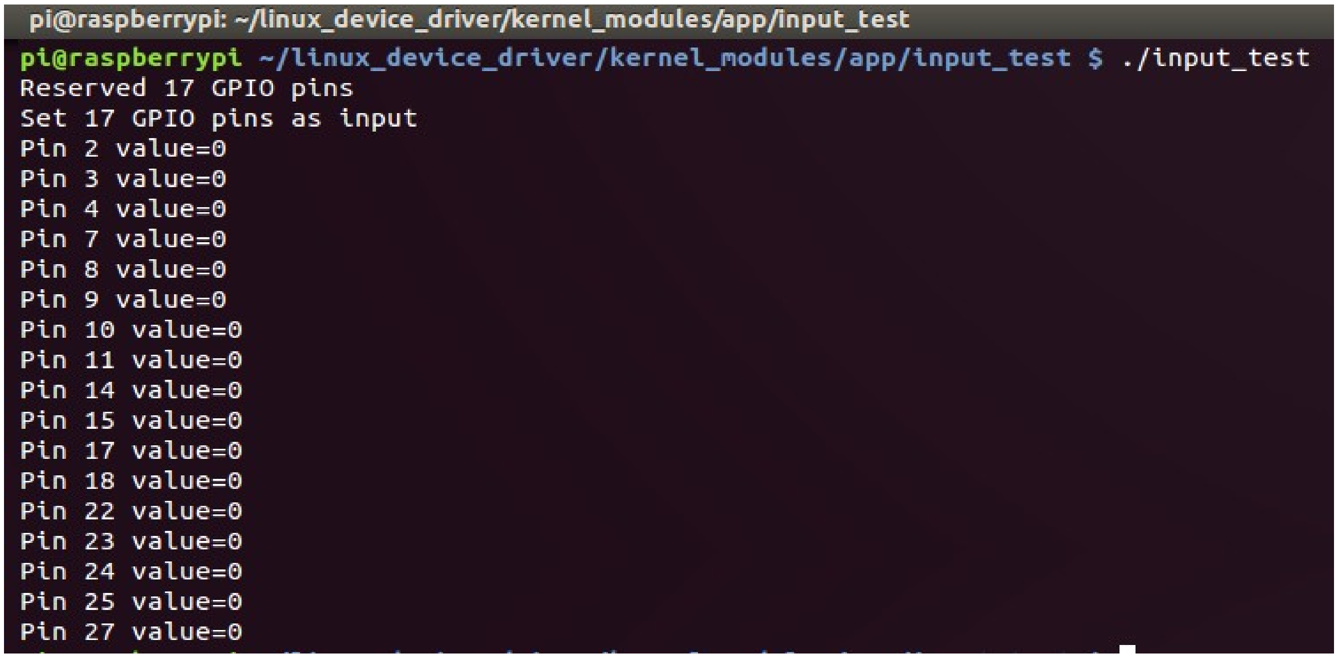


Figure 6. Setup for input functionality test case with GPIO pins on header P1 connected to GND



```
pi@raspberrypi: ~/linux_device_driver/kernel_modules/app/input_test
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/input_test $ ./input_test
Reserved 17 GPIO pins
Set 17 GPIO pins as input
Pin 2 value=0
Pin 3 value=0
Pin 4 value=0
Pin 7 value=0
Pin 8 value=0
Pin 9 value=0
Pin 10 value=0
Pin 11 value=0
Pin 14 value=0
Pin 15 value=0
Pin 17 value=0
Pin 18 value=0
Pin 22 value=0
Pin 23 value=0
Pin 24 value=0
Pin 25 value=0
Pin 27 value=0
```

Figure 7. Result of input functionality test case shown in terminal with 17 GPIO pins grounded

Another test in the input functionality test case was to set the 17 GPIO pins high (connected these pin to 3.3V power supply). The the application program input_test was executed again to verify that the will be read as having high state. Figure 8 shows the hardware setup where all the 17 GPIO pins were connected directly to a 3.3V power supply. Figure 9 shows a snapshot of the input functionality test case shown in the terminal when 17 GPIO pins on P1 header of Raspberry Pi were all connected to a 3.3V power supply.

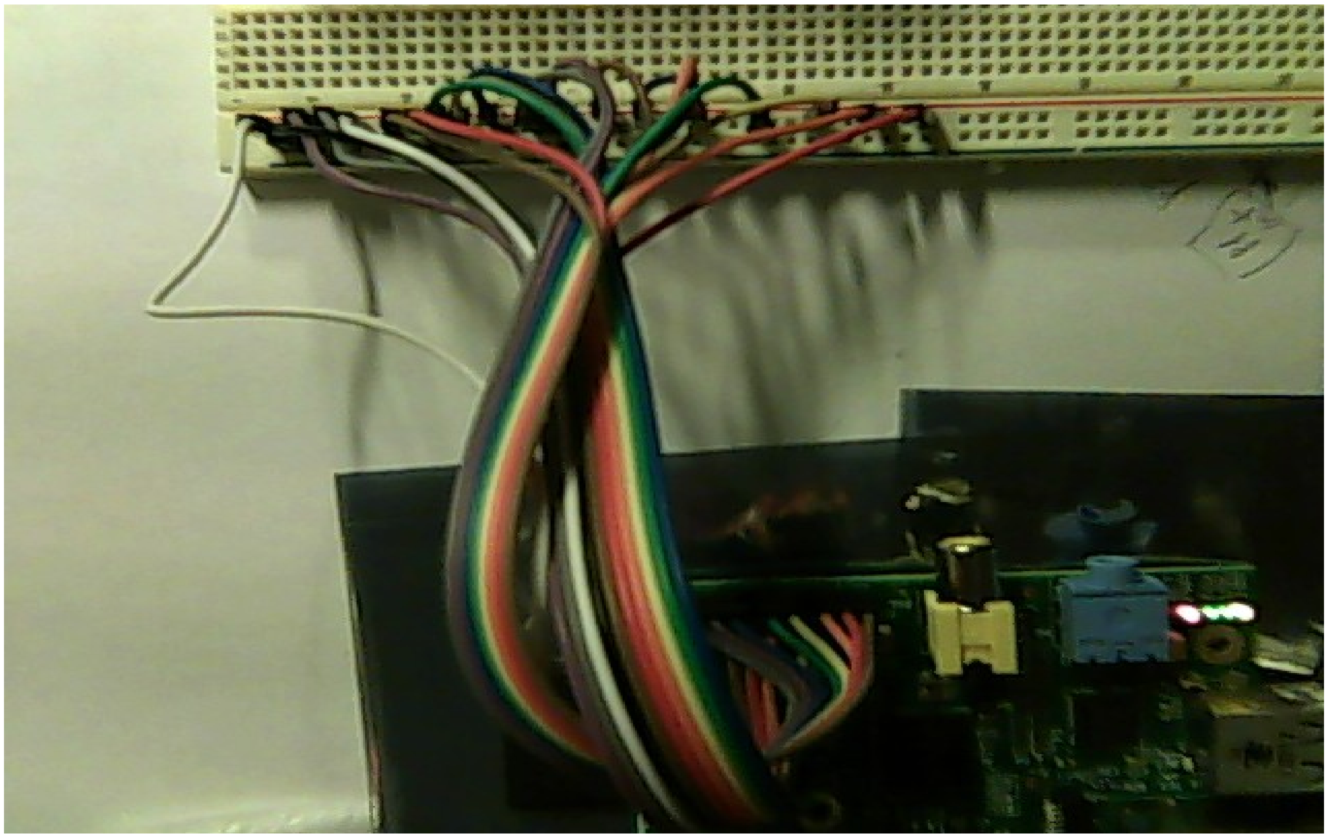
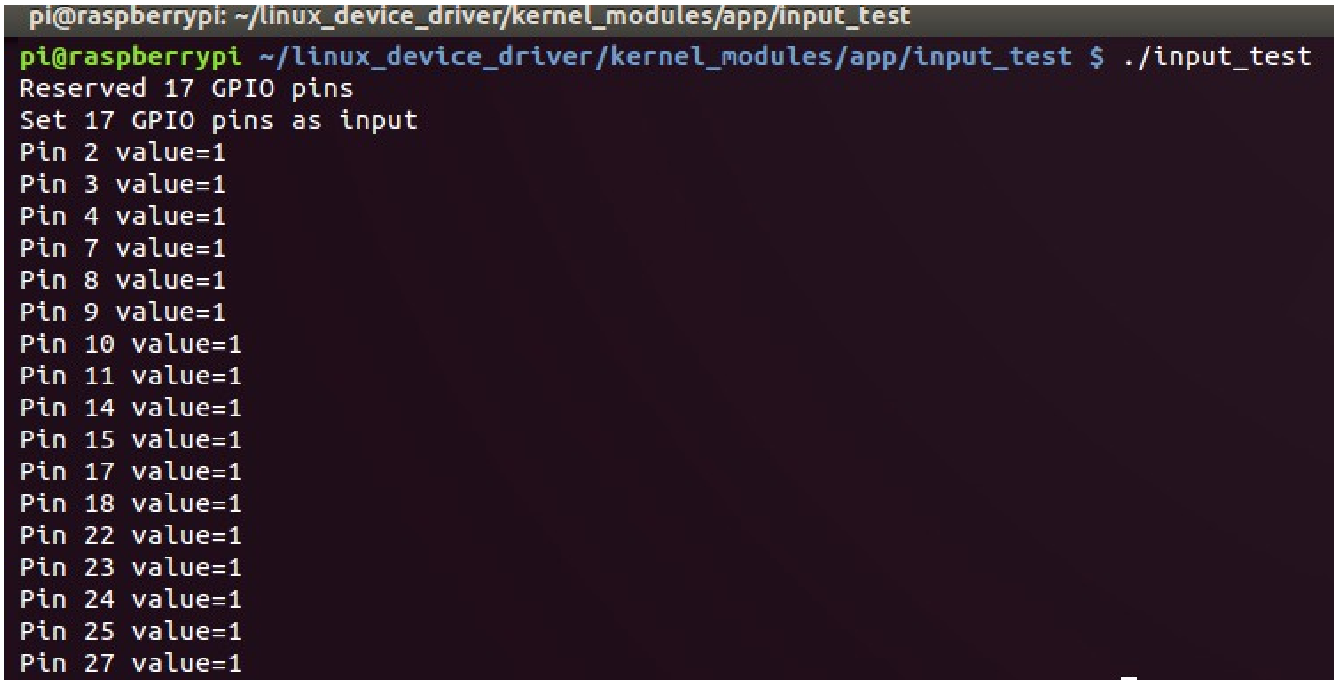


Figure 8. Setup for input functionality test case with GPIO pins on header P1 connected to 3.3V

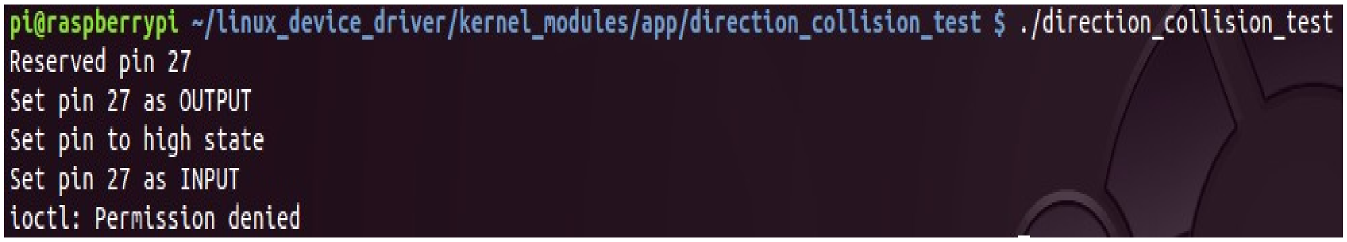


```
pi@raspberrypi: ~/linux_device_driver/kernel_modules/app/input_test
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/input_test $ ./input_test
Reserved 17 GPIO pins
Set 17 GPIO pins as input
Pin 2 value=1
Pin 3 value=1
Pin 4 value=1
Pin 7 value=1
Pin 8 value=1
Pin 9 value=1
Pin 10 value=1
Pin 11 value=1
Pin 14 value=1
Pin 15 value=1
Pin 17 value=1
Pin 18 value=1
Pin 22 value=1
Pin 23 value=1
Pin 24 value=1
Pin 25 value=1
Pin 27 value=1
```

Figure 9. Result of input functionality test case shown in terminal with 17 GPIO pins connected to a 3.3V power supply

4.3 Direction collision test case

One feature of the GPIO device driver is that processes are not allowed to set the state (low/high) of an input GPIO pin. Doing so will be detected by the device driver, and an error -EACCES will be returned in such a case. To test this feature, an application program was developed. This application program first reserves a certain GPIO pin. The it sets the GPIO pin as output, high state to let the tester to know the pin has been set to output. A LED was connected to this GPIO pin. After that, it sets the GPIO pin as input and sets the state of the pin as high to test if the device driver could prevent an input pin from being set to high/low. This application program is called *direction_collision_test.c* and is included in appendix 3. Figure 10 shows a snapshot of the result of the direction collision test case.



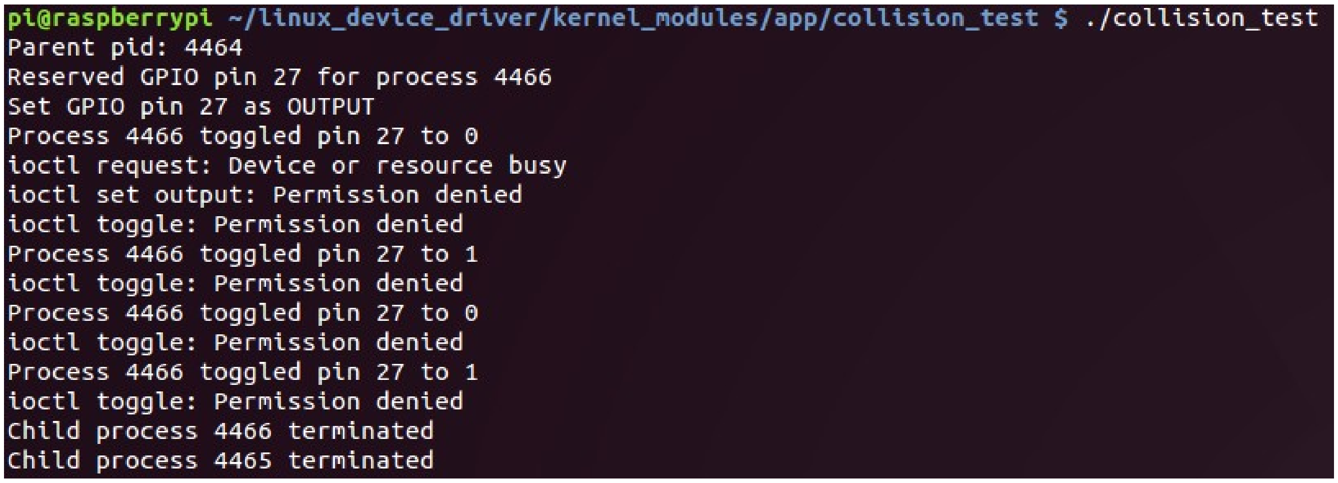
```
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/direction_collision_test $ ./direction_collision_test
Reserved pin 27
Set pin 27 as OUTPUT
Set pin to high state
Set pin 27 as INPUT
ioctl: Permission denied
```

Figure 10. Result of direction collision test case shown in terminal

As can be seen in figure 10, after GPIO pin 27 was set as input. The application program tried to set the state of this pin as high. This was detected by the GPIO device driver, and as a result, an error was returned. In this case, the error was “ioctl: Permission denied”.

4.4 Process collision test case

The last and also the most important was the process collision test case. This test case was carried out to verify that the GPIO device driver could guarantee only one process has exclusive access to one GPIO pin which is a shared resource. To do this, an application program called *collision_test.c* was written. The source code of the program is included in appendix 4. The application program tested this feature of the GPIO device driver by first spawning two child processes. These two child processes tried to access GPIO pin 27 at the same time to perform operations such as setting the direction of GPIO pin 27 as output and toggle the state of the GPIO pin. The GPIO pin was toggled for four times. The parent process waited for all the child processes to terminate after it had created them. Figure 11 shows the a snapshot taken from the terminal after performing the process collision test case.



```
pi@raspberrypi ~/linux_device_driver/kernel_modules/app/collision_test $ ./collision_test
Parent pid: 4464
Reserved GPIO pin 27 for process 4466
Set GPIO pin 27 as OUTPUT
Process 4466 toggled pin 27 to 0
ioctl request: Device or resource busy
ioctl set output: Permission denied
ioctl toggle: Permission denied
Process 4466 toggled pin 27 to 1
ioctl toggle: Permission denied
Process 4466 toggled pin 27 to 0
ioctl toggle: Permission denied
Process 4466 toggled pin 27 to 1
ioctl toggle: Permission denied
Child process 4466 terminated
Child process 4465 terminated
```

Figure 11. Result of the process collision test case shown in terminal

As indicated in figure 11, parent process 4464 created two child processes with process id 4464 and 4466. Only the child process with pid of 4466 was given exclusive access to the GPIO pin 27. The child process with pid 4465 was denied accessing to the GPIO pin 27. This was shown by the line “ioctl request: Device or resource busy”. The child process 4465 tried setting the direction of GPIO pin 27 as output. However, it was denied because it had not been given access to the GPIO pin 27 by the device driver. This error was shown by the line “ioctl set output: Permission denied”. It again tried to toggle the GPIO pin, and the error was “ioctl toggle: Permission denied”. The GPIO device driver had granted the rights so that the child process had exclusive access to GPIO pin 27. It then toggled this GPIO pins four times from 0 to 1 and vice versa. After that, it terminated.

5. References

1. Henderson G. Pins [online]. Gorden Henderson. 14 May 2013.
   URL: https://projects.drogon.net/raspberry-pi/wiringpi/pins/. Accessed 3 May 2014.

Appendices

Note: The header file *modgpio.h* for all of the C source code from appendix 1 to 4 were given in listing 2 of chapter 3.

Appendix 1 – C source code for output functionality test case

```
/*
 ============================================================================
 Name        : output_test.c
 Author      : Vu Nguyen <quangngmetro@gmail.com>
 Version     : 1.0
 Copyright   : GPL
 Description : This application program was developed to test the output
 functionality of the GPIO device driver implemented for Raspberry Pi rev 2.0
 model B platform.
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>

#include "modgpio.h"

#define NUM_PINS 21

static int pin[NUM_PINS] = {
            2,
            3,
            4,
            7,
            8,
            9,
            10,
            11,
            14,
            15,
            17,
            18,
            22,
            23,
            24,
            25,
            27,
            28,
            29,
            30,
            31,
};


int main(int argc, char * argv[])
{
      int i;
      int fd;
      int ret;
      int t;
      struct gpio_data_mode modeData;
      struct gpio_data_write writeData;

      fd = open("/dev/rpigpio", O_RDWR);
      if (!fd) {
            perror("open(O_RDWR)");
            return errno;
      }

      // Reserve GPIO pins
      for (i = 0; i < NUM_PINS; i++) {
            t = pin[i];
            ret = ioctl(fd, GPIO_REQUEST, &t);
            if (ret < 0)
                  perror("ioctl");
      }
      printf("Reserved %d GPIO pins\n", i);

      // Set the direction of GPIO pins as output
      for (i = 0; i < NUM_PINS; i++) {
            t = pin[i];
            modeData.pin = t;
            modeData.data = MODE_OUTPUT;
            ret = ioctl(fd, GPIO_MODE, &modeData);
            if (ret < 0)
                  perror("ioctl");
      }
      printf("Set %d GPIO pins as OUTPUT\n", i);

      // Set the state of GPIO pins as high
      for (i = 0; i < NUM_PINS; i++) {
            t = pin[i];
            writeData.pin = t;
            writeData.data = 1;
            ret = ioctl(fd, GPIO_WRITE, &writeData);
            if (ret < 0)
                  perror("ioctl");
      }
      printf("Set %d GPIO pins as HIGH\n", i);
      sleep(1);

      // Set the state of GPIO pins as low
      for (i = 0; i < NUM_PINS; i++) {
            t = pin[i];
            writeData.pin = t;
            writeData.data = 0;
            ret = ioctl(fd, GPIO_WRITE, &writeData);
            if (ret < 0)
                perror("ioctl");
        }
        printf("Set %d GPIO pins as LOW\n", i);

        return 0;
}
```

Appendix 2 – C source code for input functionality test case

```
/*
 ============================================================================
 Name        : input_test.c
 Author      : Vu Nguyen <quangngmetro@gmail.com>
 Version     : 1.0
 Copyright   : GPL
 Description : This application program was developed to test the input
 functionality of the GPIO device driver implemented for Raspberry Pi rev 2.0
 model B platform.
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>

#include "modgpio.h"

#define NUM_PINS 17

static int pin[NUM_PINS] = {
            2,
            3,
            4,
            7,
            8,
            9,
            10,
            11,
            14,
            15,
            17,
            18,
            22,
            23,
            24,
            25,
            27,
};

int main(int argc, char * argv[])
{
      int i;
      int fd;
      int ret;
      int t;
      int val;
      struct gpio_data_mode modeData;
```

```
        fd = open("/dev/rpigpio", O_RDONLY);
        if (!fd) {
                perror("open(O_RDONLY)");
                return errno;
        }

        // Reserve GPIO pins
        for (i = 0; i < NUM_PINS; i++) {
                t = pin[i];
                ret = ioctl(fd, GPIO_REQUEST, &t);
                if (ret < 0)
                        perror("ioctl");
        }
        printf("Reserved %d GPIO pins\n", i);

        // Set GPIO pins as input
        for (i = 0; i < NUM_PINS; i++) {
                t = pin[i];
                modeData.pin = t;
                modeData.data = MODE_INPUT;
                ret = ioctl(fd, GPIO_MODE, &modeData);
                if (ret < 0)
                        perror("ioctl");
        }
        printf("Set %d GPIO pins as input\n", i);

        // Read the state of GPIO pins
        for (i = 0; i < NUM_PINS; i++) {
                t = pin[i];
                val = t;
                ret = ioctl(fd, GPIO_READ, &val);
                if (ret < 0)
                        perror("ioctl");
                else
                        printf("Pin %d value=%d\n", t, val);
        }

        return 0;
}
```

Appendix 3 – C source code for direction collision test case

```
/*
 ============================================================================
 Name        : direction_collision_test.c
 Author      : Vu Nguyen <quangngmetro@gmail.com>
 Version     : 1.0
 Copyright   : GPL
 Description : This application program was deleloped to test the feature of
 the GPIO device driver that it does not allow a process to set the state of
 an input GPIO pin.
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>


#include "modgpio.h"

int main(int argc, char * argv[])
{
      int fd;
      int ret;
      int t = 27;
      struct gpio_data_mode modeData;
      struct gpio_data_write writeData;

      fd = open("/dev/rpigpio", O_RDONLY);
      if (!fd) {
            perror("open(O_RDONLY)");
            return errno;
      }
      //reserve pin
      ret = ioctl(fd, GPIO_REQUEST, &t);
      if (ret < 0)
            perror("ioctl");
      else
            printf("Reserved pin %d\n", t);

      //set output
      modeData.pin = t;
      modeData.data = MODE_OUTPUT;
      ret = ioctl(fd, GPIO_MODE, &modeData);
      if (ret < 0)
            perror("ioctl");
      else
            printf("Set pin %d as OUTPUT\n", t);
```

```
        // set pin to high state
        writeData.pin = t;
        writeData.data = 1;
        ret = ioctl(fd, GPIO_WRITE, &writeData);
        if (ret < 0)
              perror("ioctl");
        else
              printf("Set pin to high state\n");
        sleep (3);

        //set input
        modeData.pin = t;
        modeData.data = MODE_INPUT;
        ret = ioctl(fd, GPIO_MODE, &modeData);
        if (ret < 0)
              perror("ioctl");
        else
              printf("Set pin %d as INPUT\n", t);

        // set pin to high state from input pin to test direction collision
        writeData.pin = t;
        writeData.data = 1;
        ret = ioctl(fd, GPIO_WRITE, &writeData);
        if (ret < 0)
              perror("ioctl");
        else
              printf("Set pin to high state\n");

        sleep (3);
        return 0;
}
```

Appendix 4 – C source code for process collision test case

```
/*
 ============================================================================
 Name        : collision_test.c
 Author      : Vu Nguyen <quangngmetro@gmail.com>
 Version     : 1.0
 Copyright   : GPL, modified from Blake Bourque
 Description : This application program was developed to test that only one
 process has exclusive access to one GPIO pin for doing operations such as
 write, toggle, set state and direction. Five child processes were created, and
 all these processes tried to access the same GPIO pin to set its direction
 as output and to set its state as high
 ============================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/wait.h>

#include "modgpio.h"

#define N_CHILDREN 2
#define N 4

int pin = 27;

struct gpio_data_mode modeData;

void forkChildren(int nChildren, int fd);
void child(int fd);

int main(int argc, char *argv[])
{
      int fd, status=0;
      pid_t wpid;

      fd = open("/dev/rpigpio", O_RDWR);
      if (!fd) {
            perror("open(O_RDWR)");
            return errno;
      }
      printf("Parent pid: %d\n", getpid());
      forkChildren(N_CHILDREN, fd);
      while ((wpid = wait(&status)) > 0) {
            printf("Child process %d terminated\n", (int)wpid);
      }
      close(fd);
      return 0;
}

void forkChildren(int nChildren, int fd)
{
	int i;
	pid_t pid;

	for (i = 0; i < nChildren; i++) {
		pid = fork();
		if (pid == 0) {
			child(fd);
			break;
		}
	}
}

void child(int fd)
{
	int i = 0, ret = -1, t;

	// Reserve GPIO pin
	ret = ioctl(fd, GPIO_REQUEST, &pin);
	if (ret < 0)
		perror("ioctl request");
	else
		printf("Reserved GPIO pin %d for process %d\n", pin, getpid());

	// Set GPIO pin as output
	modeData.pin = pin;
	modeData.data = MODE_OUTPUT;
	ret = ioctl(fd, GPIO_MODE, &modeData);
	if (ret < 0)
		perror("ioctl set output");
	else
		printf("Set GPIO pin %d as OUTPUT\n", pin);

	// Toggle the state of GPIO pin N times
	for (i = 0; i < N; i++) {
		t = pin;
		ret = ioctl(fd, GPIO_TOGGLE, &t);
		if (ret < 0)
			perror("ioctl toggle");
		else
			printf("Process %d toggled pin %d to %d\n", getpid(), pin, t);
		sleep(1);
	}
	exit(0);
}
```