Raspberry Pi Temperature Sensor

CMP408: System Internals and Cybersecurity

BSc Ethical Hacking Year 4

2021/22

Note that Information contained in this document is for educational purpose

.

Contents

.

1 INTRODUCTION

This project aims to develop an intuitive IoT device that displays and stores the current air temperature using various hardware connected to a Raspberry Pi Zero Wireless. The temperature will be recorded using a DS18B20 thermometer connected to the RPi via a breadboard. The recorded data will then be displayed using three methods, these are the terminal on the RPi, a small OLED display, and within a DynamoDB database hosted on AWS (Amazon Web Services). This project is relevant to System Internals & Cybersecurity because it will be combining hardware, software, cloud, and security into one project. The hardware element will be covered by the various components connected to the RPi such as the DS18B20 sensor, wires, LED lights, buttons, and the OLED screen. A python script that runs the sensor code and an LKM developed in C will cover the software component. The security aspect will be fulfilled by following the best practices used in the industry. Finally, the cloud element will be covered by the DynamoDB table on AWS, making sure to always keep security in mind. The use of DynamoDB has many benefits for the RPi project, these benefits include high performance, scalability, NoSQL, and automatic backup (Rockset, 2019). In order to fulfil the project requirements, the following objectives will need to be met.

- Assemble the hardware e.g., Adafruit OLED display, and DS18B20 sensor.
- Develop a script using python that reads data from the DS18B20 sensor, then either illuminates three LEDs and displays the data on the OLED display or uploads the data to DynamoDB.
- Develop a LKM using C that starts the python script when the button is pressed.
- Create table in DynamoDB that stores temperature readings.

2 PROCEDURE AND RESULTS

2.1 PROJECT HARDWARE

The hardware section of the project makes use of the D18B20 temperature sensor to record and save temperatures to a folder on the RPi. A userspace program then reads the stored temperatures and scripts the output. There are three LEDs that connected to a medium sized breadboard that individually light up according to the air temperature. The project also makes use of a small Adafruit OLED display that displays the output in Celsius or Fahrenheit. There is also a green LED and simple button that communicates between an LKM and the userspace program. In order to prevent uncontrolled current from damaging hardware components, the project makes use of two 10k **Ω** resistors and four 100 **Ω** resistors. The power and ground rails on the medium breadboard are also used so fewer ground/ power pins on the RPi are used. A full image of the RPi hardware can be seen in Figure 2 in Appendix A.

Table 1

Hardware Component	GPIO Pin	Use of Component
DS18B20 – One Wire Digital Temperature Sensor	GPIO 4	This sensor records the current air temperature and sends the data to a folder located on the RPi.
Red LED	GPIO 23	This LED lights up when the temperature reaches warm
Yellow LED	GPIO 20	This LED lights up when the temperature is mi
Blue LED	GPIO 21	This LED lights up when the temperature goes below 19°C and when the temperatures get uploaded to DynamoDB
Green LED	GPIO 16	This LED lights up when the LKM module is inserted and when the button is clicked
Adafruit OLED Display	GPIO 2 (SDA), GPIO 3 (SCL)	This OLED display displays the temperature in Celsius and Fahrenheit
Button	GPIO 17	This button communicates between the userspace application and the LKM. When the button is pressed a char is sent from the LKM to the char device folder located in /dev/ and starts runs the main python script function.

2.2 Project Software

A python script and a Linux Kernel Module covered the software section of the RPi project. In short, the python script reads temperature data from the DS18B20 sensor and outputs the data to the terminal or an Adafruit display in either Celsius or Fahrenheit. The python script also gives the user the opportunity to upload the temperature data to AWS DynamoDB. The LKM on the other hand includes an interrupt handler for a push button and functions to create and read from a char device driver.

2.2.1 Python Script

The first section of the python script contains all the essential imports and third-party python libraries that are required to run the code. This is followed by two OS modules that that load the 1 wire drivers to interface with the DS18B20 sensor, three variables that configure the pixels on the Adafruit display such as width and height and two variables that initialize the I2C interface on the RPi's firmware.

The next important section of the code is the class that manages all AWS DynamoDB functionality within the project. The first function initializes the table name and connects the RPi to the server region that contains the DynamoDB server. An important security aspect of the code is the fact that no AWS session credentials are stored within the script, instead they are stored in a separate folder within the RPi. Next is the essential 'put' function which initializes the item attribute IDs within the DynamoDB table.

The function that displays the recorded temperatures on the Adafruit OLED display is the next function within the script. This function makes use of a third-party library called Adafruit_SSD1306 which greatly simplifies the configuration process within python. In short, the first section of the function clears and prepares the display for the incoming data. The next section of the code improves the display aesthetics by drawing a rectangle around the text. Lastly, the function then loads in the required text font and then outputs the text onto the Adafruit display with the .image and .show functions.

The next section of the script reads data from the directories that contains the temperature values and searches for any folders that has the name format '28' with the use of the glob library as this is where the DS18B20 ROM folder is located. The 'read_temperature' function reads the DS18B20 output file and returns the data into a list.

The next function uses the Argparse module to create the command line input options for the python script. This is followed by the functions that convert the stored temperatures into Celsius and Fahrenheit according to whatever argument the user inputs in the terminal.

The final function in the python script is the main function. The first section of the function calls the Argparse function and declares a variable (global counter) for the DynamoDB table id. The next section initializes the GPIO pins for the LED lights. The last section of the code contains three if statements located inside a while loop. These statements are run if the user enters one of three Argparse options in the terminal. The first if statement calls the Celsius function and outputs the data to the OLED display and the terminal. The three LEDs will also light up or turn off according to the temperature. The next if statement does the exact same but for Fahrenheit.

2.2.2 LKM

A LKM was created using the C programming language, in order to cover the Kernel portion of the project. In the LKM an interrupt handler was created to trigger an interrupt service routine when the button is pressed, and a char device driver was made to communicate between the userspace and the kernel space.

The important sections of the code are the interrupt handler function and the read function for the char device driver. In short, when the interrupt handler fires, it writes to global variable that indicates that the button has been pressed. The read function (dev_read) is called whenever device is being read from user space i.e., data that is being sent from the device to the user. When the read function is called it reads the number set in the global variable if the button is pressed, returns a 1 then resets the global variable to 0. If the button is not pressed, the read function only returns 0. The code is also secure as the LKM device char interface is only readable by root, which prevents any regular user from getting readings.

2.2.3 Cloud

In order to fulfil the cloud element of the project, Dynamo DB was used to store the temperature readings. To connect with Amazon Web Services (AWS), the project makes use of AWS CLI (AWS Command Line Interface), which is installed onto the RPi along with the python module Boto3. Fortunately, when using DynamoDB there are not many security factors to consider because AWS configures the security settings for DynamoDB automatically. One thing to remember though is if a user is transmitting readings to DynamoDB, make sure they encrypted using HTTPs to prevent them being intercepted and/ or changed. An image of the DynamoDB table can be seen in Figure 1 below

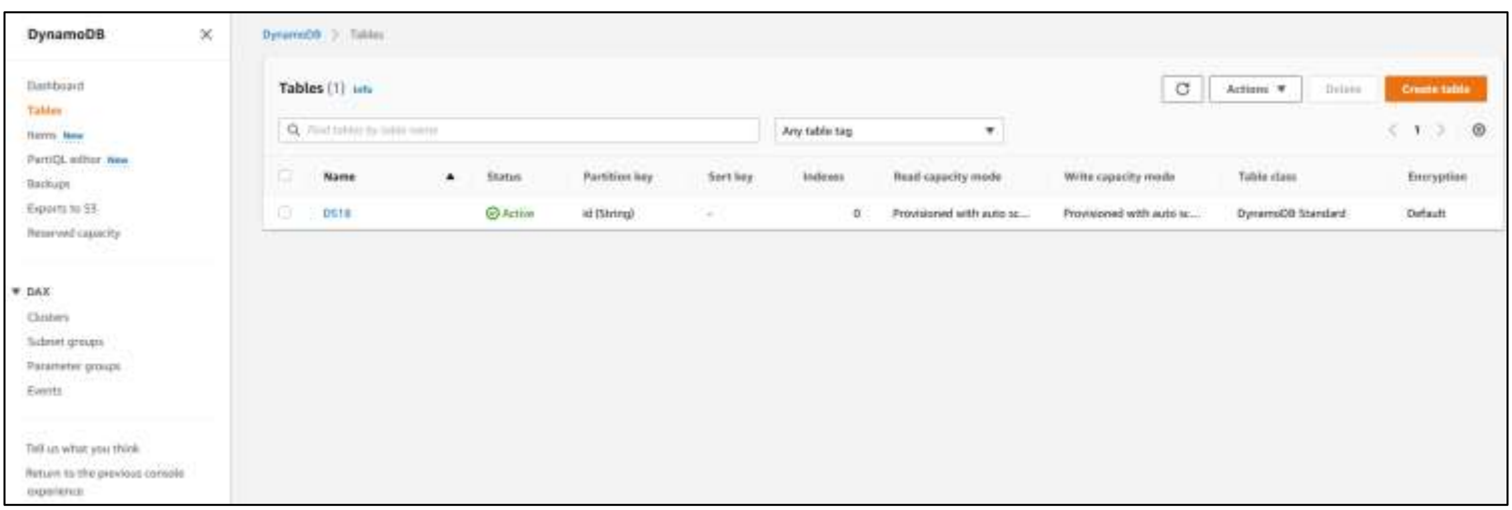

Figure 1

3 Conclusion

In summary, this goal of this project was to create a secure and intuitive IoT sensor using a Raspberry Pi that records the temperature in real time and displays or uploads the data to DynamoDB. This project successfully covers the GPIO hardware element of the project through the use of complex wiring, four LED lights, an OLED display, and a button. The software element of the project covered by the python script and the kernel module was also a success. The python script managed to run various functions based on the users input and also managed to connect to AWS. The kernel module controlled two GPIO components, an LED, and the button, which further increased the overall complexity of the project. Overall, there were not many problems that affected the project, the only minor problem occurred when trying to connect with AWS with the boto3 python module. However, this was quickly resolved, and the project carried on without issues.

4 References

Basics, C., 2016. *Raspberry Pi DS18B20 Temperature Sensor Tutorial.* [Online]
Available at: https://www.circuitbasics.com/raspberry-pi-ds18b20-temperature-sensor-tutorial/
[Accessed 28 12 2021].

boto3.amazonaws.com, n.d. *Boto3 documentation — Boto3 Docs 1.16.56 documentation.* [Online]
Available at: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[Accessed 10 1 2022].

Derek, n.d. *Writing a Linux Kernel Module — Part 2: A Character Device..* [Online]
Available at: http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/
[Accessed 5 1 2022].

Hut, T. P., 2015. *Turning on an LED with your Raspberry Pi's GPIO Pins.* [Online]
Available at: https://thepihut.com/blogs/raspberry-pi-tutorials/27968772-turning-on-an-led-with-your-raspberry-pis-gpio-pins
[Accessed 4 1 2022].

Rockset, 2019. *5 Use Cases for DynamoDB.* [Online]
Available at: https://rockset.com/blog/5-use-cases-for-dynamodb/
[Accessed 16 1 2022].

System, A. L., n.d. *Monochrome OLED Breakouts.* [Online]
Available at: https://learn.adafruit.com/monochrome-oled-breakouts/python-wiring
[Accessed 3 1 2022].

Appendix

Appendix A:

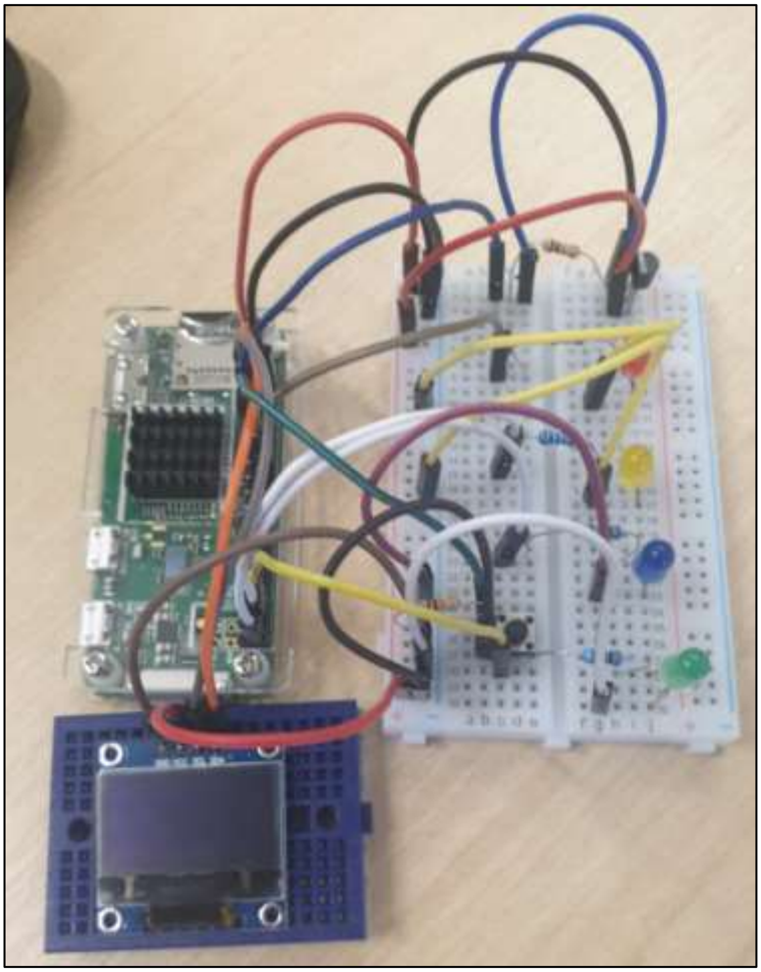

Figure 2

APPENDIX : SENSOR.PY

```
try:
        #module imports
        import os
        import glob
        import argparse
        import time

        #other imports
        import adafruit_ssd1306
        import board
        import busio
        import boto3
        import digitalio
        import pyfiglet
        import RPi.GPIO as GPIO
        from PIL import Image, ImageDraw, ImageFont
        from termcolor import colored, cprint
        from pyfiglet import Figlet
        print ("Modules loaded")
except Exception as e:
        print ("Error {}".format(e))

"""
sensor.py
"""
__version__ = "0.2"

os.system('modprobe w1-gpio')
os.system('modprobe w1-therm')

#LCD DISPLAY
WIDTH = 128
HEIGHT = 64
BORDER = 5

i2c = board.I2C()
```

```
oled = adafruit_ssd1306.SSD1306_I2C(WIDTH, HEIGHT, i2c, addr=0x3c)

#Gimmick Title Art
def titleArt():

        f = Figlet(font="slant")
        cprint(colored(f.renderText('RPi Thermometer'), 'cyan'))
        print(r"""                                     By

        An Temperature Sensor with Cloud functionality

        Use -c for Celsius, -f for Fahrenheit, -d to upload data to DynamoDB
        """ )
        print ("        Version ",__version__)
#dynamodb class
class MyDb(object):

        def __init__(self, Table_Name='DS18'):

                self.Table_Name=Table_Name
                self.db = boto3.resource('dynamodb',
                        region_name = 'us-east-1',
                        endpoint_url="https://dynamodb.us-east-1.amazonaws.com"
                )
                self.table = self.db.Table(Table_Name)

        @property
        def get(self):

                response = self.table.get_item(
                        Key = {
                                'Sensor_Id':"1"
                        }
                )
                return response

```

```
        #put meta
        def put(self, Sensor_Id='', dbCelsiusVal='', dbFahrenheitVal='',):
                self.table.put_item(
                        Item={
                                'id':Sensor_Id,
                                'CelsiusVal':dbCelsiusVal,
                                'FahrenheitVal':dbFahrenheitVal
                        }
                )

        def describe_table(self):
                response = self.client.describe_table(
                        TableName='DS18'
                )
                return response
#displays text on Adafruit LCD display
def display_text(text):
        #Clear Display
        oled.fill(0)
        oled.show()

        image = Image.new("1", (oled.width, oled.height))

        #Get drawing object to draw on image
        draw = ImageDraw.Draw(image)

        #Draw white background
        draw.rectangle((0, 0, oled.width, oled.height), outline=255, fill=255)

        #Draw smaller rectangle inside larger rectangle
        draw.rectangle(
                (BORDER, BORDER, oled.width - BORDER - 1, oled.height - BORDER - 1),
                outline=0,
                fill=0,
        )

        #Load font
```

```
        font = ImageFont.load_default()

        #Test Text
        (font_width, font_height) = font.getsize(text)
        draw.text(
                (oled.width // 2 - font_width // 2, oled.height // 2 - font_height // 2),
                text,
                font=font,
                fill=255,
        )

        #Display Text
        oled.image(image)
        oled.show()

#Open file containing the recorded temperatures
base_dir = '/sys/bus/w1/devices/'
therm_folder = glob.glob(base_dir + '28*')[0]
therm_file = therm_folder + '/w1_slave'

def read_temperature():
        f = open(therm_file, 'r')
        temps = f.readlines()
        f.close()

        return temps
#Arguments for Argparse
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("-c", "--celsius", help="output temperature in Celsius", action="store_true")
    parser.add_argument("-f", "--fahrenheit", help="output temperature in Fahrenheit", action="store_true")
    parser.add_argument("-d", "--dynamo", help="upload temperatures to DynamoDb", action="store_true")
    return parser.parse_args()

#Convert temperature into Celsius
def celsius():
        temps = read_temperature()
```

```
        while temps[0].strip()[-3:] != 'YES':
                time.sleep(0.2)
                temps = read_temperature()
        equals_pos = temps[1].find('t=')
        if equals_pos != -1:
                temp_string = temps[1][equals_pos+2:]
                temp_c = int(temp_string) / 1000.0 #Temp string is the sensor
                temp_c = str(round(temp_c, 1)) #Round the result to 1 place a
                return temp_c

#Convert temperature into Fahrenheit
def fahrenheit():
        temps = read_temperature()
        while temps[0].strip()[-3:] != 'YES':
                time.sleep(0.2)
                temps = read_temperature()
        equals_pos = temps[1].find('t=')
        if equals_pos != -1:
                temp_string = temps[1][equals_pos+2:]
                temp_f = int(temp_string) / 1000.0 * 9.0 / 5.0 + 32.0 #Temp s
                temp_f = str(round(temp_f, 1))
                return temp_f
#Main function
def main():

        args = parse_args()
        global counter

        GPIO.setmode(GPIO.BCM)
        #Assign GPIO pins
        ledR = 23
        ledY = 20
        ledB = 21
        GPIO.setup(ledR, GPIO.OUT)
        GPIO.setup(ledY, GPIO.OUT)
        GPIO.setup(ledB, GPIO.OUT)
```

```
    try:
        while True:
            #prints celsius
            if args.celsius:

                print(celsius()+u"\N{DEGREE SIGN}"+ "C")
                display_text(str(celsius())+u"\N{DEGREE SIGN}"+ "C")

                cTemp = float(celsius())
                if float(cTemp) >= 19 and float(cTemp) <= 23:
                    GPIO.output(ledY, GPIO.HIGH)
                    GPIO.output(ledB, GPIO.LOW)
                    GPIO.output(ledR, GPIO.LOW)
                elif float(cTemp) < 19:
                    GPIO.output(ledB, GPIO.HIGH)
                    GPIO.output(ledY, GPIO.LOW)
                    GPIO.output(ledR, GPIO.LOW)
                elif float(cTemp) > 24:
                    GPIO.output(ledR, GPIO.HIGH)
                    GPIO.output(ledB, GPIO.LOW)
                    GPIO.output(ledY, GPIO.LOW)
            #prints fahrenheit
            elif args.fahrenheit:

                print(fahrenheit()+u"\N{DEGREE SIGN}"+ "F")
                display_text(str(fahrenheit())+u"\N{DEGREE SIGN}"+ "F")

                fTemp = float(fahrenheit())
                if float(fTemp) >= 65 and float(fTemp) <= 75:
                    GPIO.output(ledY, GPIO.HIGH)
                    GPIO.output(ledB, GPIO.LOW)
                    GPIO.output(ledR, GPIO.LOW)
                elif float(fTemp) < 64:
                    GPIO.output(ledB, GPIO.HIGH)
                    GPIO.output(ledY, GPIO.LOW)
                    GPIO.output(ledR, GPIO.LOW)
```

```
                    GPIO.output(ledR, GPIO.LOW)
                elif float(fTemp) > 76:
                    GPIO.output(ledR, GPIO.HIGH)
                    GPIO.output(ledB, GPIO.LOW)
                    GPIO.output(ledY, GPIO.LOW)
            #uploads data to dynamodb
            elif args.dynamo:
                obj = MyDb()
                obj.put(Sensor_Id=str(counter), dbCelsiusVal = str(celsius()), dbFahrenheitVal = str(fahrenheit()))
                counter = counter + 1
                print ("Data uploaded to DynamoDB C: {} ".format(celsius()+u"\N{DEGREE SIGN}"+ "C"))
                print ("Data uploaded to DynamoDB F: {} ".format(fahrenheit()+u"\N{DEGREE SIGN}"+ "F"))
                time.sleep(1.0)

    except KeyboardInterrupt:
        GPIO.output(ledB, GPIO.LOW)
        GPIO.output(ledY, GPIO.LOW)
        GPIO.output(ledR, GPIO.LOW)
        GPIO.cleanup()

if __name__ == "__main__":
    global counter
    counter = 0
    titleArt()
    main()
```

APPENDIX : LKMT.C

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>
#include <linux/device.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <linux/spinlock.h>
#define DEVICE_NAME "tempchar"
#define CLASS_NAME "temp"

/* Meta Information */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("");
MODULE_DESCRIPTION("An LKM for a gpio interrupt that lights up an LED");
MODULE_VERSION("0.1");

static unsigned int Led = 16;
static unsigned int Button = 17;
static unsigned int counter = 0;
static bool state = 0;
/* variable contains pin number interrupt controller to which GPIO 17 is m
static unsigned int irq_number;
static bool buttonPress;
/* spinlock to syncronise access to buttonPress */
DEFINE_SPINLOCK(defLock);

extern unsigned long volatile jiffies;
unsigned long old_jiffie = 0;

static int majorNumber;
static struct class* tempcharClass = NULL;
static struct device* tempcharDevice = NULL;

// The prototype functions for the character driver -- must come before th
static int     dev_open(struct inode *, struct file *);
```

```c
static int     dev_open(struct inode *, struct file *);
static int     dev_release(struct inode *, struct file *);
static ssize_t dev_read(struct file *, char *, size_t, loff_t *);
static ssize_t dev_write(struct file *, const char *, size_t, loff_t *);

/** @brief Devices are represented as file structure in the kernel. The file_operations structu
 *  /linux/fs.h lists the callback functions that you wish to associated with your file operati
 *  using a C99 syntax structure. char devices usually implement open, read, write and release
 */
static struct file_operations fops =
{
    .open = dev_open,
    .read = dev_read,
    .write = dev_write,
    .release = dev_release,
};

/**
 * @brief Interrupt service routine is called when the interrupt is triggered
 */
static irq_handler_t gpio_irq_handler(unsigned int irq, void *dev_id, struct pt_regs *regs) {
    printk("gpio_irq: Interrupt was triggered and ISR was called!\n");

    unsigned long diff = jiffies - old_jiffie;
    if (diff < 20) {
        return (irq_handler_t) IRQ_HANDLED;
    }

    state = !state; /* Toggle LED */
    if (state){
        gpio_set_value(Led, 1);
    }
    else {
        gpio_set_value(Led, 0);
    }

```

```
    spin_lock(&defLock);
    buttonPress = true;
    spin_unlock(&defLock);

    old_jiffie = jiffies;

    printk(KERN_INFO "Led state is: [%d] \n", gpio_get_value(Led));
    printk(KERN_INFO "Button state is: [%d] \n", gpio_get_value(Button));

    counter++;
    return (irq_handler_t) IRQ_HANDLED;
}

/**
 * @brief The device open function that is called each time the device is opened
 *  This will only increment the numberOpens counter in this case.
 */
static int dev_open(struct inode *inodep, struct file *filep){
    printk(KERN_INFO "tempchar: Device has been opened\n");
    return 0;
}

/** @brief This function is called whenever device is being read from user space i.e. da
 *  being sent from the device to the user. In this case is uses the copy_to_user() func
 *  send the buffer string to the user and captures any errors.
 */
static ssize_t dev_read(struct file *filep, char *buffer, size_t len, loff_t *offset) {

    printk(KERN_INFO "reading now!!!\n");
    unsigned char sendByte = 0;
    unsigned long flags;
    int error_count;

    if (buttonPress) {
       sendByte = 1;
       spin_lock_irqsave(&defLock, flags);
       buttonPress = false;
```

```
       spin_unlock_irqrestore(&defLock, flags);
       printk(KERN_INFO "button has been pressed\n");
    }
    copy_to_user(buffer, &sendByte, 1);

    if (error_count==0){            // if true then have success
      printk(KERN_INFO "TempChar: Sent %d characters to the user\n", 1);
      return (1);  // clear the position to the start and return 0
    }
    else {
      printk(KERN_INFO "TempChar: Failed to send %d characters to the user\n", error_count);
      return -EFAULT;              // Failed -- return a bad address message (i.e. -14)
    }
    return 1;
}

/** @brief This function is called whenever the device is being written to from user space i.e.
 *  data is sent to the device from the user. The data is copied to the message[] array in this
 *  LKM using the sprintf() function along with the length of the string.
 */
static ssize_t dev_write(struct file *filep, const char *buffer, size_t len, loff_t *offset){
    return 0;
}

/** @brief The device release function that is called whenever the device is closed/released by
 *  the userspace program
 */
static int dev_release(struct inode *inodep, struct file *filep){
    printk(KERN_INFO "TempChar: Device successfully closed\n");
    return 0;
}

/**
 * @brief This function is called when the module is loaded into the kernel
 */
static int __init Module_Init(void) {
    pr_info("%s\n", __func__);
```

```
    printk("irq test");
    printk("qpio_irq: Loading module...\n");

    /*  Setup GPIO  */
    if(!gpio_is_valid(Button)){
        printk(KERN_INFO "Invalid GPIO\n");
        return -ENODEV;
    }

    if(gpio_request(Button, "rpi-gpio-17")) {
        printk("Error!\nCan not request GPIO17!\n");
        gpio_free(Button);
        return -1;
    }

    /*  Set GPIO 17 direction   */
    if(gpio_direction_input(Button)) {
        printk("Error!\nCannot set GPIO 17 to input!\n");
        gpio_free(Button);
        return -1;
    }

    gpio_export(Button, false);

    if(!gpio_is_valid(Led)){
        printk(KERN_INFO "Invalid GPIO\n");
        return -ENODEV;
    }

    if(gpio_request(Led, "rpi-gpio-16")) {
        printk("Error!\nCan not request GPIO16!\n");
        gpio_free(Led);
        return -1;
    }

    if(gpio_direction_output(Led, 1)) {
        printk("Error!\nCan not set GPIO 16\n");
```

```c
            gpio_free(Led);
            return -1;
        }

        /* Make it appear in /sys/class/gpio/gpio16 for echo 0 > value */
        gpio_export(Led, false);

        /* Setup the interrupt  */
        irq_number = gpio_to_irq(Button);
        if(request_irq(irq_number,
                (irq_handler_t) gpio_irq_handler,   /* pointer to IRQ handler method */
                IRQF_TRIGGER_RISING,
                "my_gpio_irq",  /* See this string from user console to identify: cat /proc/interr
                NULL) != 0) {

            printk ("Error!\nCannot request interrupt number: %d\n", irq_number);
            gpio_free(Button);
            return -1;
        }

        majorNumber = register_chrdev(0, DEVICE_NAME, &fops);

        if (majorNumber<0){
          printk(KERN_ALERT "TempChar failed to register a major number\n");
          return majorNumber;
        }
        printk(KERN_INFO "TempChar: registered correctly with major number %d\n", majorNumber);

        // Register the device class
        tempcharClass = class_create(THIS_MODULE, CLASS_NAME);
        if (IS_ERR(tempcharClass)){                // Check for error and clean up if there is
           unregister_chrdev(majorNumber, DEVICE_NAME);
           printk(KERN_ALERT "Failed to register device class\n");
           return PTR_ERR(tempcharClass);          // Correct way to return an error on a pointer
        }
        printk(KERN_INFO "TempChar: device class registered correctly\n");
```

```

    // Register the device driver
    tempcharDevice = device_create(tempcharClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
    if (IS_ERR(tempcharDevice)){               // Clean up if there is an error
       class_destroy(tempcharClass);           // Repeated code but the alternative is goto stateme
       unregister_chrdev(majorNumber, DEVICE_NAME);
       printk(KERN_ALERT "Failed to create the device\n");
       return PTR_ERR(tempcharDevice);
    }
    printk(KERN_INFO "TempChar: device class created correctly\n"); // Made it! device was initiali

    printk("Done!\n");
    printk("GPIO 17 is mapped to IRQ Number: %d\n", irq_number);
    return 0;
}

/**
 * @brief This function is called when the module is removed from the kernel
 */
static void __exit Module_Exit(void) {
    printk ("gpio_irq: Unloading module...\n");
    gpio_set_value(Led, 0);
    gpio_unexport(Led);
    free_irq(irq_number, NULL);
    gpio_free(Button);
    gpio_free(Led);
    printk("gpio_irq: Module Unloaded\n");
}

module_init(Module_Init);
module_exit(Module_Exit);
```