System Design & Administration

Practice Phase 1

Óscar Cubeles Ollé

October 2021

Table of Contents

1 Introduction

This practice is about creating a Linux Kernel Module (LKM) in a Raspberry Pi that uses Raspbian as a Debian based operating system. Given a Raspberry Pi connected using its GPIOS (General Purpose Input Outputs) to four pushbuttons and two LEDs, the programmed LKM must be able to perform a different action each time a pushbutton is pressed. That is, running a different bash script for each pushbutton and turning on and off the LEDs according to the pressed pushbutton.

Moreover, the LKM needs to store the number of times that a button has been pressed and it has to generate a log message each time any of the buttons is pressed.

2 Previous Installations

Before programming the LKM, some tasks need to be done so that the Raspberry Pi works properly. In this section will describe in detail all those steps needed to make the Raspberry Pi work so a user could do it and use Raspberry Pi without any problems.

2.1 Installing Raspberry Pi Operating System

The first step to make Raspberry PI work is to install the operating system on a microSD card.

In order to do that, the Raspberry Pi imager needs to be installed on the computer. It can be installed using this link.

Once the imager is installed on the computer, the microSD card needs to be inserted in the computer. Then, the imager has to be opened.

After that, the microSD needs to be selected using the "Choose Storage" tab and the Operating System using the "Choose OS" tab. Finally, pressing the "Write" button, the OS will be installed in the selected microSD card.

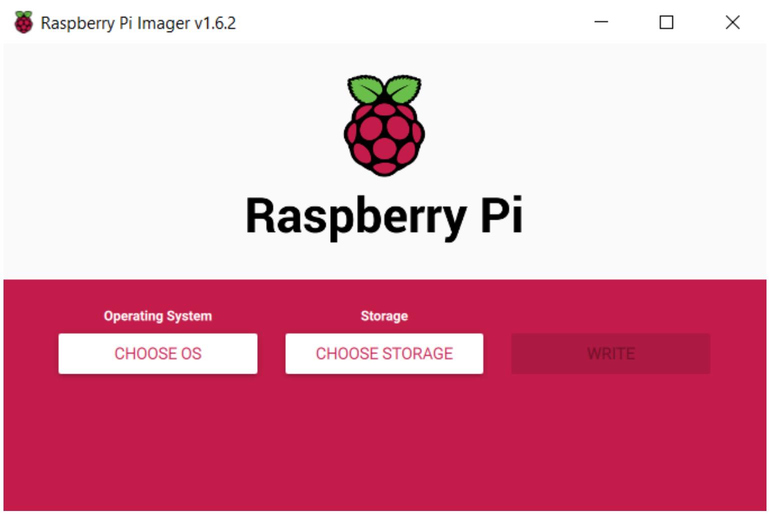


Figure 1: Raspberry PI imager

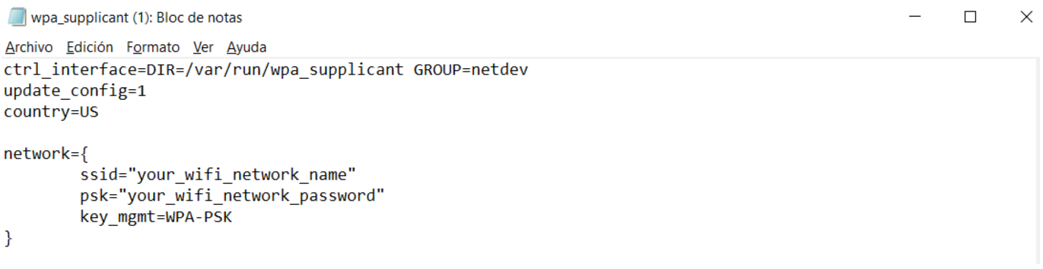
2.2 Setting Up SSH Wi-Fi Connection to Raspberry Pi

In order to being able to connect to our Raspberry Pi to program the LKM, the SSH and the Wi-Fi connection need to be set up.

Firstly, to set up the SSH, it is required to put a SSH file in the microSD card where the OS has been installed. This file can be downloaded using this link.

Once the SSH file has been added to the microSD containing the operating system, another configuration file with the Wi-Fi network name and password need to be added so that each time the Raspberry Pi is powered, it will automatically connect to the network. This file can be downloaded using this link.

This file will have two parts that need to be modified, the first one after the ssid word. The network name has to be put there. The second part that needs to be changed after the psk word, there the network password needs to be entered.


wpa_supplicant (1): Bloc de notas
Archivo Edición Formato Ver Ayuda

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=US

network={
        ssid="your_wifi_network_name"
        psk="your_wifi_network_password"
        key_mgmt=WPA-PSK
}
```

Figure 2: Wi-Fi Network configuration file

Note: If the Raspberry Pi is connected through the network using Ethernet, the configuration file is not needed.

Once this is done, the connection will be working.

2.3 Installing Linux Headers to Raspberry Pi

Prior to developing the Linux Kernel Module code, it is essential to have the Linux Headers installed in the machine. The Linux Headers are used to compile Linux Kernel Modules and act as an interface between the internal kernel components and the user space.

To install the Linux Headers in a machine, the following steps need to be followed:

1. The first step is to download package information from all configured sources. To do this, the following shell command needs to be done:
   ```
   $ sudo apt update
   ```

2. After that, the Linux Headers can be installed. It is important to install the Linux Headers with the same version as the machine Kernel version (that can be checked using the uname -r command). The following command needs to be done:
   ```
   $ sudo apt install linux-headers-$(uname-r)
   ```

3. After that, the Linux Headers should be already installed. If they do not work properly, after a reboot should work. That can be done with the following command:
   ```
   $ sudo reboot
   ```

3 Software Installation

To be able to install the software developed in this practice, you need to follow the following steps:

- Step 1: Assembly your hardware as specified in section
- Step 2: Download the code in this github repository: https://github.com/OscarCubeles/ASOPhase1.git
- Step 3: Then, type sudo make in the directory where you downloaded the code.
- Step 4: After that, type sudo make install to install the LKM, then the LKM should work as expected.

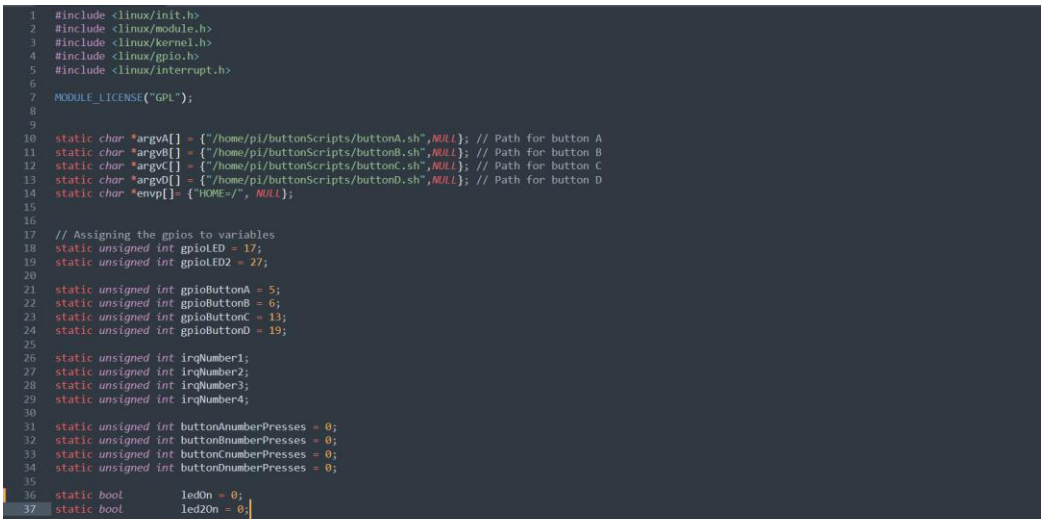
4 Code

4.1 Kernel Module code in C

The code of the LKM has been developed using C as a programming language. Regarding the code structure, it can be divided in different parts. This section includes a detailed explanation of all the sections of the C code of the LKM. However, the main idea is the following: This LKM will map the buttons with an interrupt request number per button. Then, it will set a function for each button that will handle the button interrupts. There, in the handler functions, the code for the actions of each button will be placed.

Part 1: Global Variables & Script Paths

The first part of the code contains the libraries needed for the code to work, the paths to the bash scripts for each button, variables to store the GPIO numbers of the Raspberry Pi, to store the amount of time that the buttons have been pressed, the state of the buttons and the ORQ numbers.



```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/gpio.h>
#include <linux/interrupt.h>

MODULE_LICENSE("GPL");


static char *argvA[] = {"/home/pi/buttonScripts/buttonA.sh",NULL}; // Path for button A
static char *argvB[] = {"/home/pi/buttonScripts/buttonB.sh",NULL}; // Path for button B
static char *argvC[] = {"/home/pi/buttonScripts/buttonC.sh",NULL}; // Path for button C
static char *argvD[] = {"/home/pi/buttonScripts/buttonD.sh",NULL}; // Path for button D
static char *envp[]= {"HOME=/", NULL};


// Assigning the gpios to variables
static unsigned int gpioLED = 17;
static unsigned int gpioLED2 = 27;

static unsigned int gpioButtonA = 5;
static unsigned int gpioButtonB = 6;
static unsigned int gpioButtonC = 13;
static unsigned int gpioButtonD = 19;

static unsigned int irqNumber1;
static unsigned int irqNumber2;
static unsigned int irqNumber3;
static unsigned int irqNumber4;

static unsigned int buttonAnumberPresses = 0;
static unsigned int buttonBnumberPresses = 0;
static unsigned int buttonCnumberPresses = 0;
static unsigned int buttonDnumberPresses = 0;

static bool         ledOn = 0;
static bool         led2On = 0;
```

Figure 3: Code Part 1, Script Paths and Global Variables

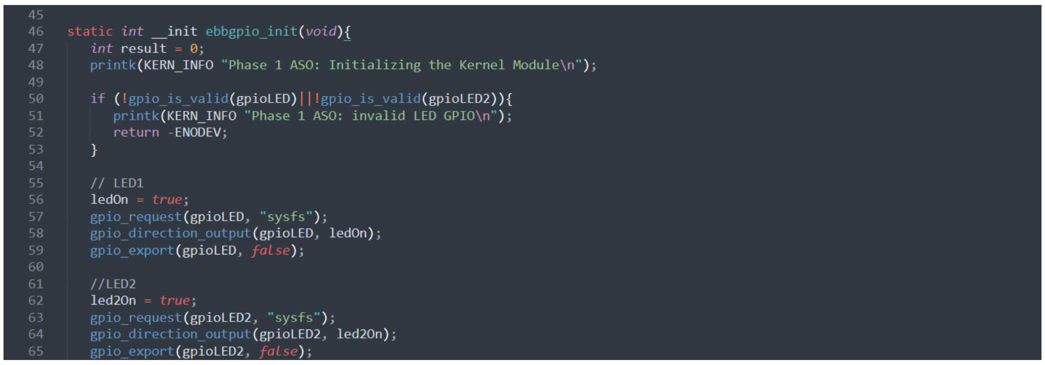
Part 2: Init Function

This function is used to initialize everything related to the LKM. Concretely, it checks if the GPIOs of the LEDs are valid. Then, the LEDs are configured by telling that they are an output and passing the GPIO number.

After configuring the LEDs, the Buttons are configured. Similarly to the LEDs, the GPIO numbers and its direction are configured. The button debouncing time is configured here too.

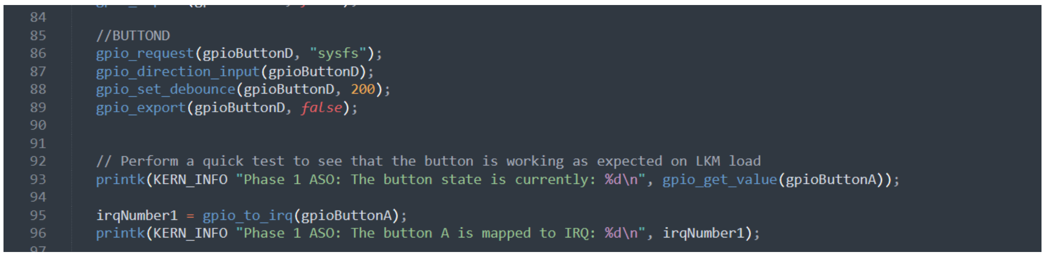
After the configurations, the mapping of the Buttons is made, each button with an IRQ number. And finally, it sets the handlers for the button interrupts.

The following images show the code of this part.



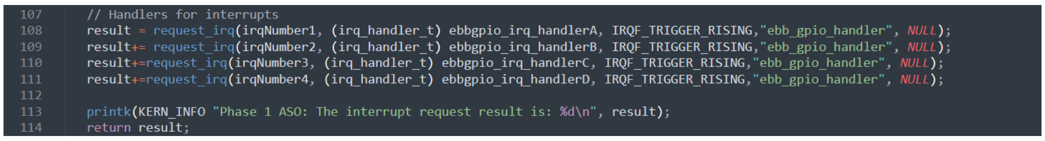
```c

static int __init ebbgpio_init(void){
   int result = 0;
   printk(KERN_INFO "Phase 1 ASO: Initializing the Kernel Module\n");

   if (!gpio_is_valid(gpioLED)||!gpio_is_valid(gpioLED2)){
      printk(KERN_INFO "Phase 1 ASO: invalid LED GPIO\n");
      return -ENODEV;
   }

   // LED1
   ledOn = true;
   gpio_request(gpioLED, "sysfs");
   gpio_direction_output(gpioLED, ledOn);
   gpio_export(gpioLED, false);

   //LED2
   led2On = true;
   gpio_request(gpioLED2, "sysfs");
   gpio_direction_output(gpioLED2, led2On);
   gpio_export(gpioLED2, false);
```

Figure 4: Configurating the LEDs



```c

   //BUTTOND
   gpio_request(gpioButtonD, "sysfs");
   gpio_direction_input(gpioButtonD);
   gpio_set_debounce(gpioButtonD, 200);
   gpio_export(gpioButtonD, false);


   // Perform a quick test to see that the button is working as expected on LKM load
   printk(KERN_INFO "Phase 1 ASO: The button state is currently: %d\n", gpio_get_value(gpioButtonA));

   irqNumber1 = gpio_to_irq(gpioButtonA);
   printk(KERN_INFO "Phase 1 ASO: The button A is mapped to IRQ: %d\n", irqNumber1);
```

Figure 5: Configurating Buttons and mapping IRQ example

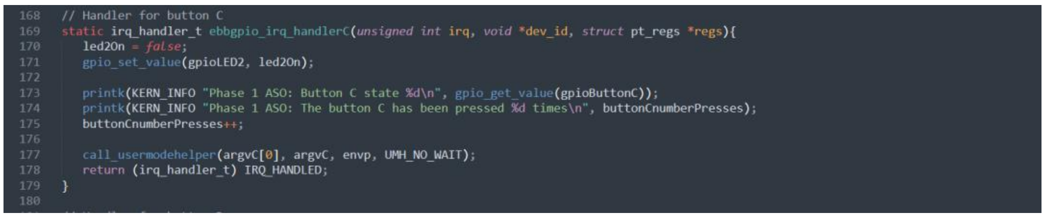


```c
   // Handlers for interrupts
   result = request_irq(irqNumber1, (irq_handler_t) ebbgpio_irq_handlerA, IRQF_TRIGGER_RISING,"ebb_gpio_handler", NULL);
   result+= request_irq(irqNumber2, (irq_handler_t) ebbgpio_irq_handlerB, IRQF_TRIGGER_RISING,"ebb_gpio_handler", NULL);
   result+=request_irq(irqNumber3, (irq_handler_t) ebbgpio_irq_handlerC, IRQF_TRIGGER_RISING,"ebb_gpio_handler", NULL);
   result+=request_irq(irqNumber4, (irq_handler_t) ebbgpio_irq_handlerD, IRQF_TRIGGER_RISING,"ebb_gpio_handler", NULL);

   printk(KERN_INFO "Phase 1 ASO: The interrupt request result is: %d\n", result);
   return result;
```

Figure 6: Setting the handlers for the button interrupts

Part 3: Interrupt Handlers

The third part of the code contains all the interrupt handler functions for the buttons. This handlers change the state value of the LEDs using the gpio_set_value function. They increment the number of times that the button has been pressed, and it calls the function call_usermodehelper which executes the bash script for the button.

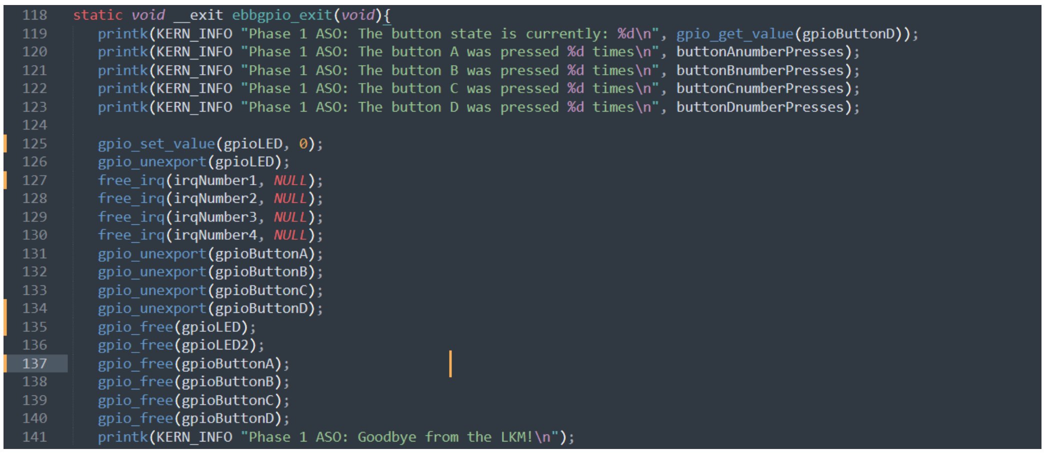


```c
// Handler for button C
static irq_handler_t ebbgpio_irq_handlerC(unsigned int irq, void *dev_id, struct pt_regs *regs){
   led2On = false;
   gpio_set_value(gpioLED2, led2On);

   printk(KERN_INFO "Phase 1 ASO: Button C state %d\n", gpio_get_value(gpioButtonC));
   printk(KERN_INFO "Phase 1 ASO: The button C has been pressed %d times\n", buttonCnumberPresses);
   buttonCnumberPresses++;

   call_usermodehelper(argvC[0], argvC, envp, UMH_NO_WAIT);
   return (irq_handler_t) IRQ_HANDLED;
}
```

Figure 7: Handler function for button C

Part 4: Exit Function

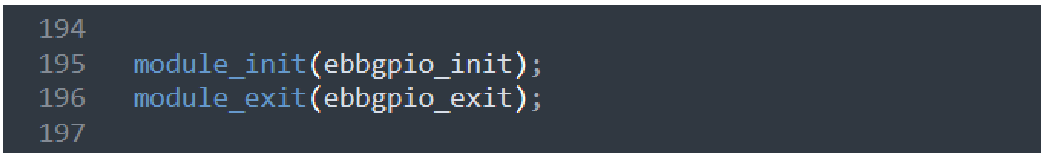
This function is used to free and unexport all the GPIOs, to free the IRQ numbers. Moreover, it prints the log messages with the number of times that the buttons have been pressed.



```c
static void __exit ebbgpio_exit(void){
   printk(KERN_INFO "Phase 1 ASO: The button state is currently: %d\n", gpio_get_value(gpioButtonD));
   printk(KERN_INFO "Phase 1 ASO: The button A was pressed %d times\n", buttonAnumberPresses);
   printk(KERN_INFO "Phase 1 ASO: The button B was pressed %d times\n", buttonBnumberPresses);
   printk(KERN_INFO "Phase 1 ASO: The button C was pressed %d times\n", buttonCnumberPresses);
   printk(KERN_INFO "Phase 1 ASO: The button D was pressed %d times\n", buttonDnumberPresses);

   gpio_set_value(gpioLED, 0);
   gpio_unexport(gpioLED);
   free_irq(irqNumber1, NULL);
   free_irq(irqNumber2, NULL);
   free_irq(irqNumber3, NULL);
   free_irq(irqNumber4, NULL);
   gpio_unexport(gpioButtonA);
   gpio_unexport(gpioButtonB);
   gpio_unexport(gpioButtonC);
   gpio_unexport(gpioButtonD);
   gpio_free(gpioLED);
   gpio_free(gpioLED2);
   gpio_free(gpioButtonA);
   gpio_free(gpioButtonB);
   gpio_free(gpioButtonC);
   gpio_free(gpioButtonD);
   printk(KERN_INFO "Phase 1 ASO: Goodbye from the LKM!\n");
```

Figure 8: Exit function code

Moreover, it is important to put the module_init and the module_exit functions in the code to specify the functions that the LKM will use as entry point and at removal time.



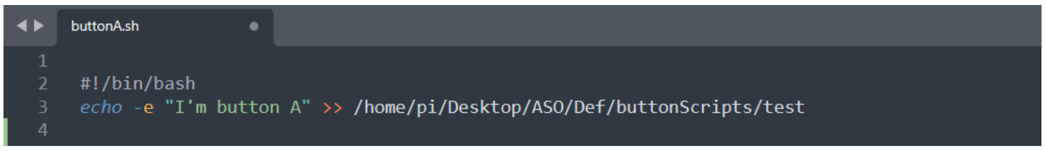
```
module_init(ebbgpio_init);
module_exit(ebbgpio_exit);
```

Figure 9: Module init and Module exit in code

4.2 Bash Scripts

As mentioned in previous sections, each button runs a bash script. The bash scripts do not need to do a specific task. However, they must execute a command. In my case, I decided that each button runs a script that echoes a message to a log file.

The following image shows a script for one of the buttons.



buttonA.sh

```
#!/bin/bash
echo -e "I'm button A" >> /home/pi/Desktop/ASO/Def/buttonScripts/test
```

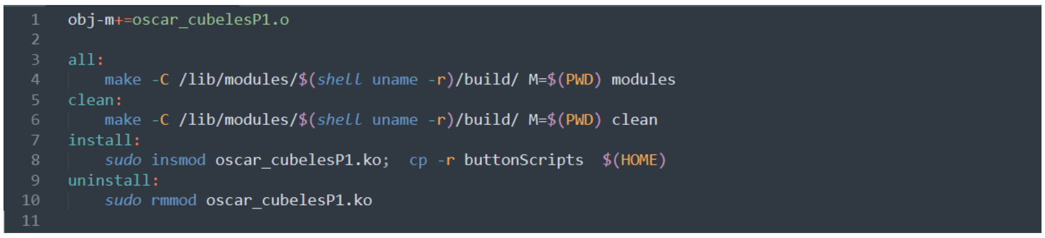
Figure 10: Script for button A

4.3 Makefile

In order to ease the tasks of compiling the module code, inserting the module to the kernel, removing it among others, a Makefile is created.

The Makefile has different parts. The first one is the obj-m+=. There it is specified the object file which are built as loadable kernel modules.

The other part of the Makefile is where the functions are declared, in my case, I had a function to compile the module, one to clean the module objects, another to insert the module in the kernel and to copy the scripts in the Home directory, and another one to remove the module from the kernel.



```
obj-m+=oscar_cubelesP1.o

all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
install:
	sudo insmod oscar_cubelesP1.ko;  cp -r buttonScripts  $(HOME)
uninstall:
	sudo rmmod oscar_cubelesP1.ko
```

Figure 11: Makefile used in this project

5 Hardware

The completion of this practice includes a small part of hardware. It consists of connecting the four pushbuttons and the two LEDs to the Raspberry Pi so the Kernel Module receives and sends signals through the Raspberry Pi. The schematic used to complete this task is the following:

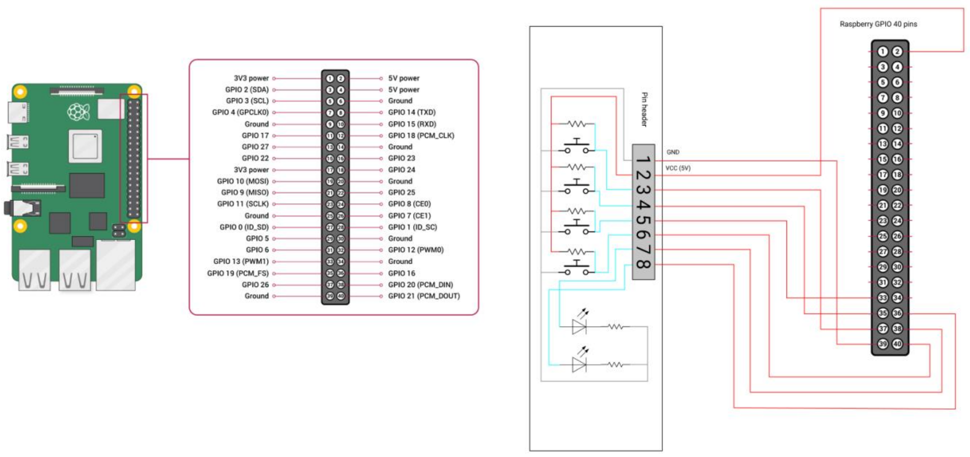


Figure 12: Electrical Schematic of the Board

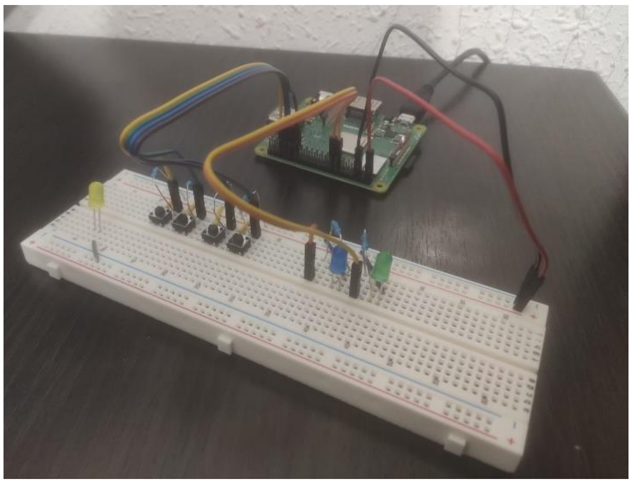


Figure 13: Image of the Physical Hardware

6 Conclusions

Knowing how to design and manage a system properly is essential for a good system administrator. To be able to manage a system correctly, it is necessary to know how the system works in order to be able to adapt to the changing needs of the users who use it.

The knowledge acquired in class has been essential in order to be able to carry out this practice. Specially, the commands to navigate through the file system and the creation programming of scripts. Not only that, but thanks to the development of this practice I have been able to learn more concepts on how to manage a system such as creating a Linux Kernel Module, how to connect using SSH to a machine or how to use the GPIOs of a Raspberry Pi in a Linux Kernel Module.

Through the development of this project I have encountered multiple problems. The main problem I faced was with the installation part, particularly with the installation of the Linux Headers. I had an OS version that did not match with the Linux headers version, therefore I could not compile the module using make. To solve it, I had to install to remove the Linux headers I had, install the generic headers, then reboot the Raspberry Pi and run an update and an upgrade. After that, I managed to solve the problem. Regarding the rest of the project, I faced other problems related to the code of the module but those were easier to solve.

To conclude, for me, this has been a short but challenging academic assignment. However, with dedication I have managed to complete it.