COMP3000 Final Project

Raspberry Pi Door Alarm

Tri Do 101006966
William Findlay 101015157

April 13, 2018

Contents

1 Introduction

1.1 Context

1.1.1 Acronyms

- **GPIO Pins:** general purpose input output pins, outlined in Figure 1
- **I/O:** input and output
- **UART:** universal asynchronous reader/transmitter (a simple communication protocol)
- **ID_SC and ID_SD:** provide read only memory for a device attached to the GPIO header to transfer important information like necessary drivers, etc.
- **IP**: internet protocol
- **TPC**: transmission control protocol
- **RAM**: random access memory
- **LED**: light emitting diode

1.1.2 Relevant Details

This project requires an understanding of what a *Raspberry Pi* is, *GPIO pins*, *character device drivers*, *client/server programs*, *sockets*, and the *Raspbian Linux distribution*.

A **Raspberry Pi**, a very small and not overly powerful computer primarily used for hardware-related computer science projects, provides the hardware component for this project. **Raspbian**, a distribution of Linux based on *Debian Stretch Linux* is an operating system built specifically for Raspberry Pis with useful programs, header files, and utilities specifically geared toward development on the Pi. The most critical aspect of the Raspberry Pi, for the purposes of this project, is the GPIO pin header, an array of 40 pins (some GPIO and some not, see Figure 1).

The **GPIO pins** on this header[1] are those with which the implemented driver will be interfacing. There are two reserved pins and two UART pins which can be treated like regular GPIO pins since we only care about sending/receiving a simple binary signal (1 or 0).

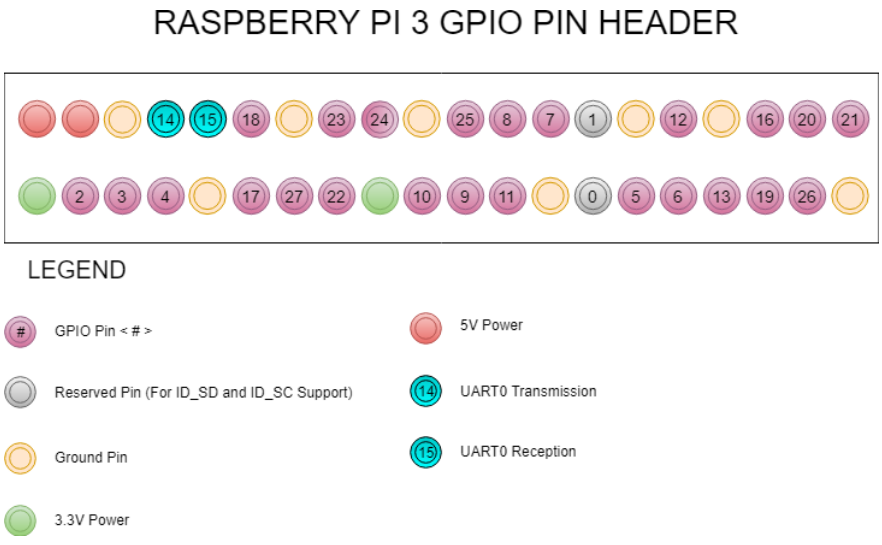

Figure 1: Layout of GPIO pin header on a Raspberry Pi 3 [1].

[1]Header in this case means a *motherboard header* and not a header file.

A **character device driver** (in this case, in the form of a kernel module) behaves like a file which can be read from and written to by programs. It uses this behavior to provide an interface in the kernel for user space programs and hardware to commuunicate. Specifically, this project uses several such character devices (one for each pin) to provide a read/write I/O interface for the GPIO pins on the *Raspberry Pi 3 board*.

A client/server model provides a method for a user to interface (through a client) with a more complex program built to process requests and return responses (the server). For the purposes of this project, a client program interfaces with a server tasked with monitoring the GPIO input from an alarm trigger. The server continuously reads input from the character device asssociated with the GPIO pin and, when the alarm is triggered, sends a message to the client, causing it to display a message to the user.

Communication between the *client and server* is accomplished via **sockets** which provide a basic method of sending data between processes and over a network.

1.2 Problem Statement

The initial goal is to create a door alarm system using the Raspberry PI and its GPIO pins. We accomplish this by providing a method in the operating system by which programs in multiple languages can interface with the GPIO pins on the Raspberry Pi. Then it is simply a matter of using the interface in the implementation of the client/server code for the door arlarm system.

In order to provide an interface with a high degree of extensibility for any future work, we choose to implement a set of character device drivers in the form of a kernel module. For more on this, see Section 2.3 on why a kernel module was the correct choice.

Finally we need to create the client/server interface to allow the end user to use the door alarm. This client/server code will make use of the implemented character device driver.

The relevance of the initial problem extends beyond the desire to make a functioning door alarm with a Raspberry Pi. In fact, the problem statement could be abstracted to the following: implement a character device driver interface for the Raspberry Pi which could be used for a wide variety of projects *including the door alarm*. It now becomes clear that the **extensibility** of the solution is even more significant than the particular use case which initially motivated its creation.

1.3 Result

All of the intially set goals were achieved. We created a kernel module which adds a set of character device drivers to the `/dev` directory, each one interfacing with its own GPIO pin. We were then able to use these character device drivers to read and write input to and from the GPIO pins. Using this functionality, we implemented a client/server model in Python such that the server was able to successfully notify the client when the alarm had been tripped.

In addition to the main goals of our project, we also implemented a few other programs to test the GPIO driver. These programs include: another more general client/server model in Python that uses input and output threads to allow the user to read and write to and from pins and get information about those pins in real time; a testerout.c program capable of writing outputs to GPIO pins; and a testerin.c program capable of reading `low` or `high` inputs from GPIO pins.

1.4 Outline

The rest of this report is structured as follows: **section 2** presents background information and concepts related to the project; **section 3** describes the implementation of the kernel module, the client/server user interface, and the door alarm circuit; **section 4** contains a self-evaluation and steps taken to ensure quality;

section 5 will conclude the report with a summary of what was done, a description of the project's relevance, and a brief subsection on the possibility of future work.

2 Background Information

The basic model for the project can be described with a simple diagram.

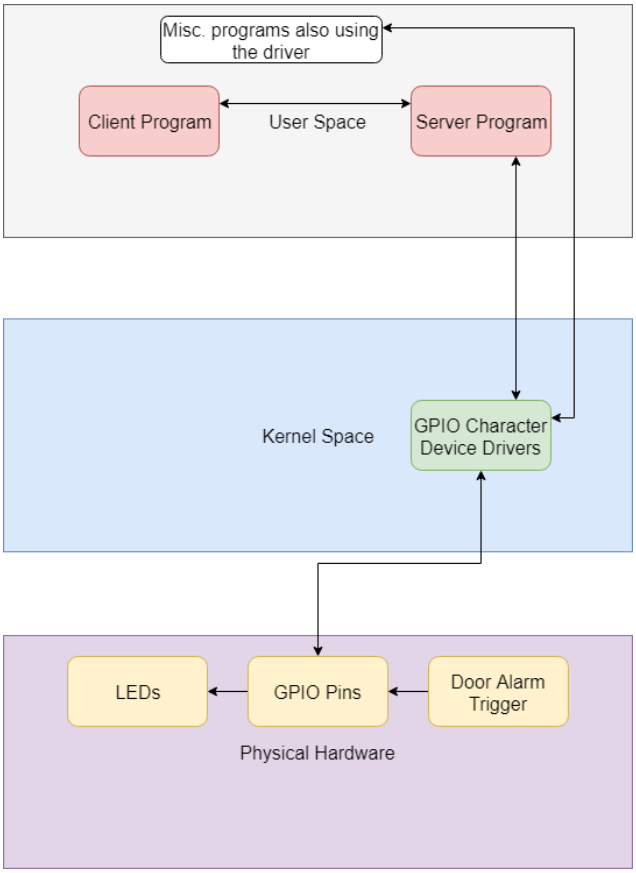

Figure 2: The layout of the project divided in terms of hardware/software abstraction levels. In a more sophisticated future iteration, the client and server would run on separate systems.

2.1 Raspbian OS and Setup

Raspbian is the best flavor of Linux to use with a Raspberry Pi since it was designed specifically for that purpose. Critically, its kernel headers come with important files such as `linux/gpio.h` which provides an interface with which to implement the character device driver for the GPIO pins.

Raspbian requires additional setup in order to develop kernel modules on it. Specifically, one needs to update the firmware and kernel, download the headers, and finally install them with a symbolic link.

First, we become root via `sudo su`. Then run the following upgrade routine:

```
apt-get update -y
apt-get upgrade -y
rpi-update
```

The `rpi-update` command updates the firmware/kernel of the Pi.

Next, we get the installation script for the rpi-source code and make it executable. Then update tags and run with the flag `--skip-gcc` to avoid a newer version of gcc raising a conflict.

```
sudo wget https://raw.githubusercontent.com/notro/rpi-source/master/rpi-source -O
  /usr/bin/rpi-source
sudo chmod +x /usr/bin/rpi-source
/usr/bin/rpi-source -q --tag-update
rpi-source
```

2.2 Various Approaches for GPIO I/O

There were several options to consider when dealing with GPIO pins on the Raspberry Pi [2]. In C, for example, one can choose to directly manipulate registers, use an available interface such as `sysfs`, or download a library like wiringPi.

Higher level languages such as Python or Ruby naturally have fewer options, with many lower level solutions like direct register access being beyond the scope of what the language can do.

One attractive option, however, is a kernel module. A kernel module (in this case, a set of character device drivers) offers several advantages not addressed by the above solutions. These advantages are outlined in the section below.

2.3 Why a Kernel Module Was the Correct Choice

The primary advantage of writing a kernel module is portability between languages. A module can be read in C, just as it can be read in Python, Java, or any other language which supports file I/O. This lends itself nicely to any future work which may build upon what has been done here, in addition to broadening the spectrum of possible approaches for the next steps in the project.

A kernel module also provides a layer of abstraction between the interfaces available, for example the `sysfs` interface and the application programmer. The programmer need not know *how the module works internally*, only that he or she can write desired operations to a pin and read a value from a pin, with the promise that it *will work*.

2.4 Pull Up vs. Pull Down Circuits for GPIO Pins

When dealing with circuits and GPIO pins, it is important to consider the two types: pull up and pull down. Each offers a different way of dealing with high/low values in a GPIO pin and each comes with advantages and disadvantages.

2.4.1 Pull Up

In a pull up circuit, when the switch is open, the current runs through resistor R_1 and and then resistor R_2 and sets $GPIO_{18}$ to `1` (see left of Figure 3).

When the switch is closed, the current runs through R_1 and through the magnetic switch into ground, and sets $GPIO_{18}$ to `0` [3] (see right of Figure 3).

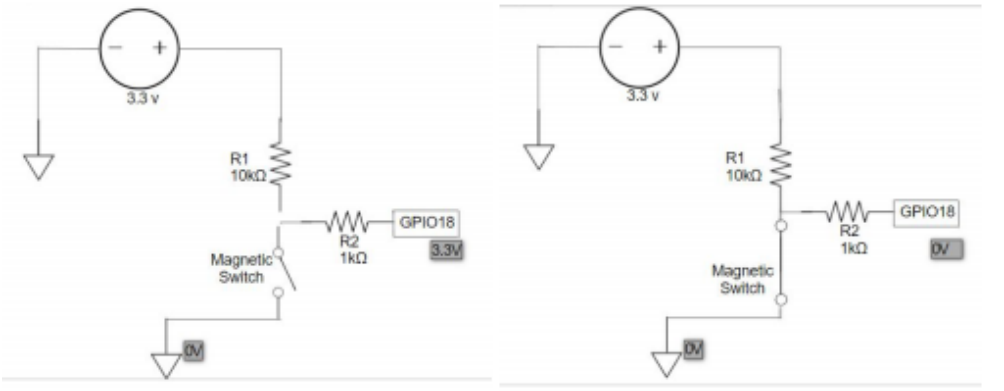

Figure 3: Diagram of a pull up circuit.

2.4.2 Pull Down

In a pull down circuit, when the switch is open, the circuit is broken and $GPIO_{18}$ is set to `0` (see left of Figure 4).

When the switch is closed, the current runs through R_1 and through the magnetic switch, and sets $GPIO_{18}$ to `1` (see right of Figure 4). The function of R_2 here is to protect the pin by grounding the circuit [3].

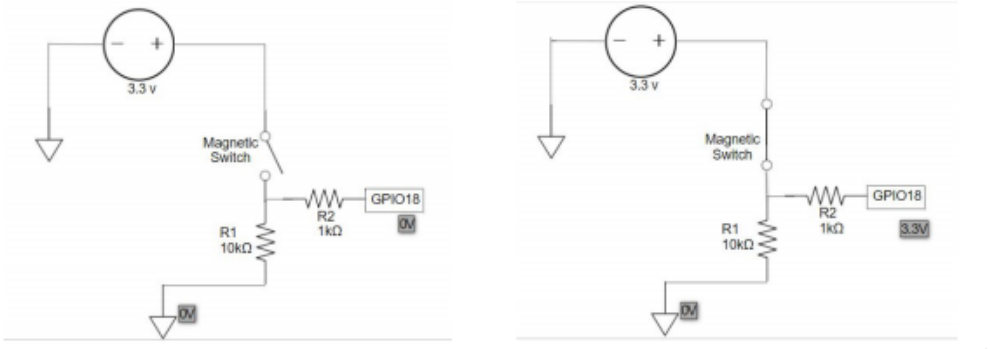

Figure 4: Diagram of a pull down circuit.

2.4.3 Which to Choose?

Since this application involves a relatively simple circuit, there is no advantage to using pull up over pull down. Furthermore, a pull down circuit is safer due to excess volatage always running through a resistor to ground. So here, pull down is the obvious choice; in more complex circuits with multiple components, more nuanced behavior offered by a pull up circuit may be desirable [4].

2.5 Interfacing With the End User

There are a few options to consider for the actual door alarm application. One possibility would be to create a server which runs in the background and watches the pin connected to the alarm trigger. When the alarm is tripped, the server writes to another pin, turning it on, either illuminating an LED or sounding an alarm.

Another possibility is to create a client application and a server application. This offers the possibility of remotely notifying the client program over a network connection. Additionally, should the Pi lose power and the server shut down, the client program would be able to notify the end user appropriately that the connection has been lost and the user would be able to take the appropriate steps.

The client/server approach seems the most appealing due to its practicality, versitility, and extensibility.

2.6 Sockets

Sockets provide a relatively simple method of sending data between two processes either on a local machine or network. Each socket represents a combination of an IP address and a port number. Together, they provide the TCP layer with the ability to identify the application to which data needs to be sent. Python has a built-in socket module which is perfect for the purposes of a simple client/server model. See Figure 5 for an example.

```
s = socket.socket();
host = socket.gethostname();
port = 4002;
s.bind((host,port));

s.listen(5);
```

Figure 5: Python code to set up a socket for a simple server on localhost:4002. It listens on 127.0.0.1 (the default argument for `socket.gethostname()`) for connections to port 4002. It will store a queue of up to 5 of these connections for processing.

3 Result

3.1 Kernel Module - Character Device Driver for GPIO Pins

The kernel module creates 28 *character devices*[5], each with a unique minor number and corresponding to a unique GPIO pin on the Raspberry Pi 3 (Figure 6). Each GPIO character device supports the following operations:

- `read`
  - copy either 1 or 0 to the user space depending on value of pin
- `write`
  - values include: 1, 0, in, out
- `open`
  - open pin for reading/writing
- `release`
  - close pin from reading/writing

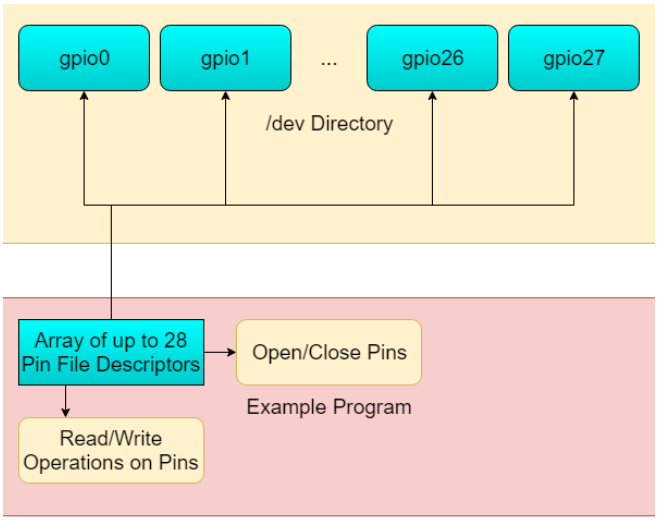

Figure 6: The `/dev` directory with 28 GPIO pin character devices and how an example program might interface with one or more of them.

The kernel module has functions mapped to each file operation as well as an `init` function and an `exit` function to take care of initialization and cleanup respectively when the driver is loaded and unloaded from the kernel.

3.1.1 The `gpio_dev_init` Function

We begin by allocating a region for a range of 28 character device drivers (one for each GPIO pin on the Raspberry Pi 3). There are two choices for allocating a region of character devices:

- `alloc_chrdev_region`
- `register_chrdev_region`

We use `alloc_chrdev_region` in order to allow the kernel to determine the major number of the devices automatically. The alternative `register_chrdev_region` function must be given the major number through an input parameter [6].

Now, the module needs a driver class to use in the `device_create` function. This can be accomplished with `driverClass = class_create(THIS_MODULE, DEVICE_NAME);`. Once the driver class has been created, we can loop through each pin and set its initial state. The state of pins is changed using functions from `linux/gpio.h` and stored in pointers to a structure containing metadata relevant to each pin.

Before setting the state of the metadata structure, it must first be allocated with `kmalloc(sizeof(struct gpio_dev),GFP_KERNEL)` where `struct gpio_dev` is the structure containing the metadata and `GFP_KERNEL` is a flag telling the OS to allocate normal RAM in kernelspace.

The next step in the loop is to associate the structure of file operations with each character device. Once this has been accomplished, the device can be registered and created. On creation, each device is given a minor number corresponding to its unique GPIO pin number. See Figure 7 for a screenshot of the `/dev` directory after `gpio_dev_init` has been called.

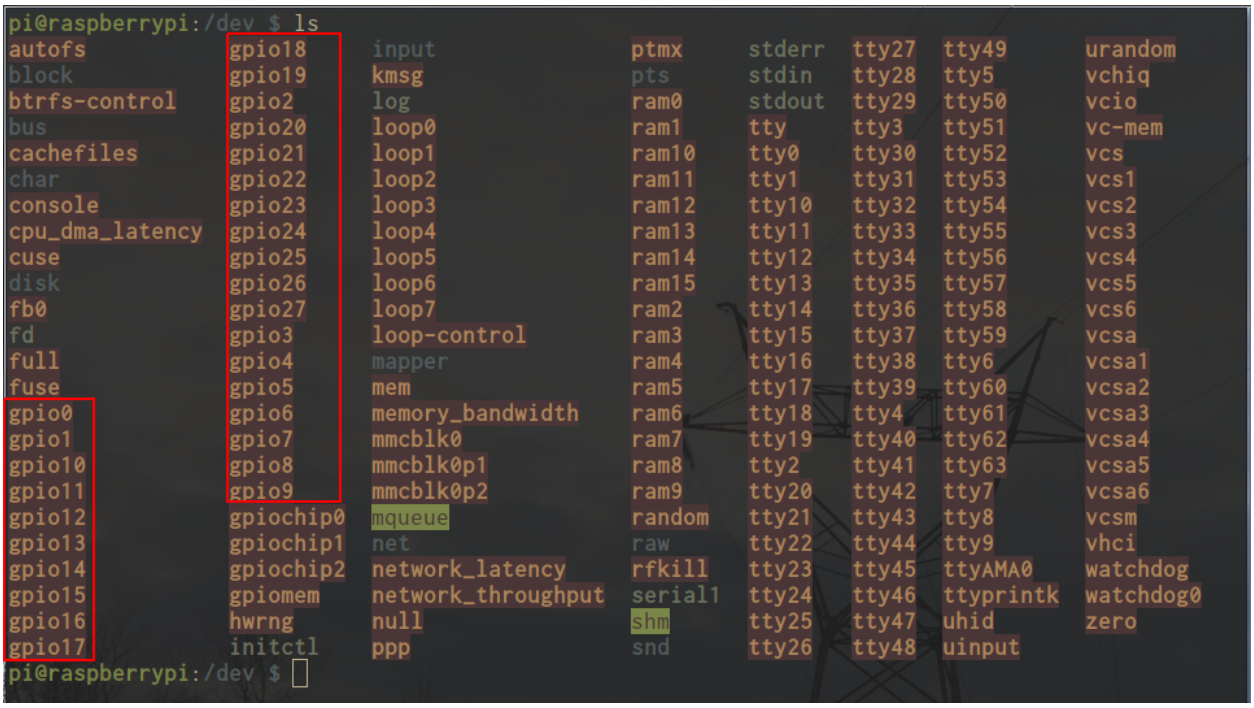

Figure 7: The `/dev` directory after `gpio_dev_init` has been called via inserting the kernel module. The module registers a set of 28 character device drivers.

3.1.2 The `gpio_dev_exit` Function

The exit function performs two cleanup tasks:

- unregister the character device region allocated in the init function
- reset each pin to its initial state and destroy the pin device

3.1.3 The `dev_open` Function

This function prints information about the device opened (i.e., its minor number) to `dmesg` with a call to `printk`.

3.1.4 The `dev_release` Function

This function is a mirror of `dev_open`, printing information about the device closed to `dmesg` with `printk`.

3.1.5 The `dev_read` Function

First, we get the GPIO pin number from the minor number of the device. We then clear any bytes in the message buffer with a call to `memset`. After clearing the message, the state of the GPIO pin with $pinnumber = minornumber$ is copied to the message buffer. After a quick sanity check on the size of the message, it can then be copied to user space.

3.1.6 The `dev_write` Function

Again, we start by getting the GPIO pin number from the minor number of the device. `memset` the message to clear it. Perform a quick sanity check on the message to make sure that the buffer will not overflow.

Now, we need to null terminate the message to prevent undefined behavior. Additionally, due to `read` being weird in some languages, all `\n` characters in the message must also be set to `\0`.

From here, we parse the message and perform the appropriate operation on the GPIO pin. If the message is invalid, we return an error. One other constraint that is important to bear in mind is that the user should not be able to write a `1` or `0` to a pin set to input mode.

3.1.7 Setting the Kernel Module to be Automatically Loaded on Boot

We configure the kernel module to be automatically loaded upon booting the Raspberry Pi as follows:

```
# change directory to project directory
cd /directory/of/the/project

# become root
sudo su

# add the kernel object to the modules directory
mv gpiodriver.ko /lib/modules/$(uname -r)

# run a quick dependency check to make sure everything is OK
depmod

# this step is to confirm it is working
modprobe gpiodriver

# set up autoloading
# edit /etc/modules to include the line: gpiodriver

# reboot the pi
reboot now
```

3.2 The Circuit

The door alarm circuit is a simple pull down circuit with a magnetic switch and two LEDs (a red and a green). The magnetic switch triggers the alarm and is hooked to GPIO_{18} and a ground pin. The red LED is connected to GPIO_{15} and a ground pin. The green LED is connected to GPIO_{14} and a ground pin. See Figure 8.

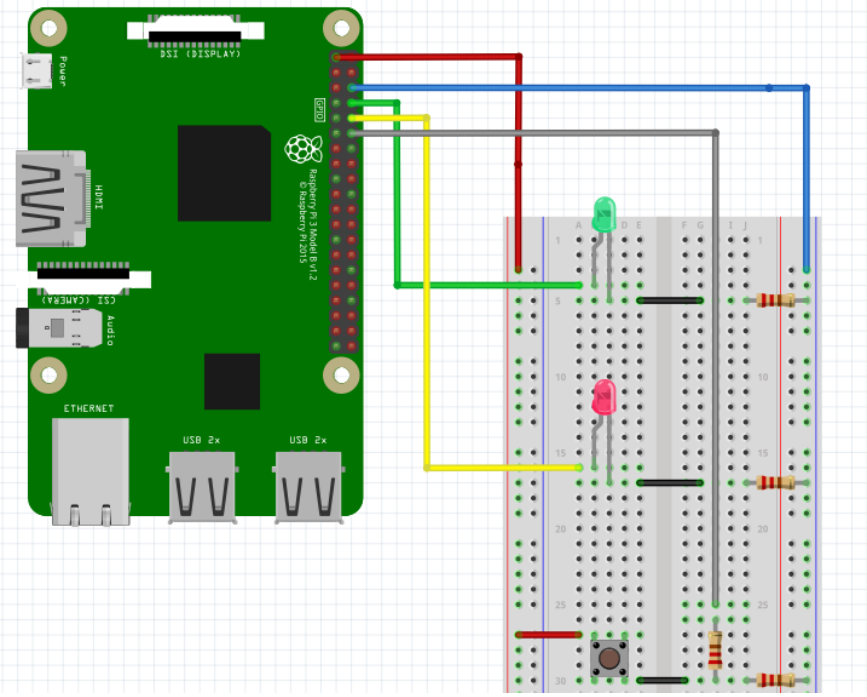

Figure 8: The layout of the door alarm circuit.

3.3 Client/Server Model for User Interface

3.3.1 Server Program

We begin by opening the following pin character devices for reading and writing:

- `/dev/gpio18` for the alarm trigger
- `/dev/gpio15` for the red LED
- `/dev/gpio14` for the green LED

Now wait for a connection on port 4002. When connected, send a message notifying the client and do some initial set up on the pins. That is, do the following:

- write `in` to `/dev/gpio18`
- write `out` to `/dev/gpio14`
- write `0` to `/dev/gpio14`
- write `out` to `/dev/gpio15`
- write `0` to `/dev/gpio15`

Inform the server that the alarm is set and waiting to be tripped and perform the following loop ad infinitum:

- read the value of `/dev/gpio18`
  - if the value of `/dev/gpio18` is `1`
    - send a message to the client telling it that the alarm is still waiting
    - write `1` to `/dev/gpio15`, the red LED
    - write `0` to `/dev/gpio14`, the green LED
  - if the value of `/dev/gpio18` is `0`
    - send a message to the client telling it that the alarm has been tripped
    - write `0` to `/dev/gpio15`, the red LED
    - write `1` to `/dev/gpio14`, the green LED

3.3.2 Client Program

Connect to the server on port 4002. Wait for a reply message.

Perform the following loop ad infinitum:

- wait for a message from the server
- print the message which will be one of the following:
  - "ALARM WAITING" if the switch is closed
  - "ALARM TRIPPED" if the switch is open

4 Evaluation and Quality Assurance

4.1 The Kernel Module

4.1.1 Testing

In order to ensure the robustness of the kernel module, it is critical to test under a wide variety of conditions. This means various combinations of pin I/O, multiple programs manipulating pins at the same time, and I/O in both multiple programming languages and the terminal.

For testing purposes, we create some additional programs which will perform general GPIO pin operations using the kernel module:

- input and output tester programs in C
- test client and test server in Python
- echo and cat in the terminal

Using these various methods, as well as the original door alarm client/server model, we are able to ensure that the kernel module behaves as expected under a wide variety of circumstances.

4.1.2 Steps Taken to Ensure Consistency

Code running in kernelspace needs to be written carefully in order to avoid kernel panics, system crashes, etc. A very important aspect of the initialization function in the module is to make sure that each step in initialization is checked for errors. Should an error occur, the necessary backtracking is performed and all character device drivers which have already been registered by the module are properly destroyed. Care is also taken to make sure that upon initialization of the module, every pin is set to the same default state.

The kernel module includes several checks and balances to ensure consistency, even when being used by several programs. For example, if a program attempts to read from a pin which has been set to output, the

character device driver associated with that pin should return an error which can then either be handled in the userspace program or not.

Whenever dealing with buffers (i.e., in the read and write functions), care must be taken to avoid buffer overflows which would result in the kernel attempting to access memory it shouldn't be. This means checking that the size of the message does not exceed the size of the buffer, and trimming it if necessary. Additionally, care should be taken when writing a message to kernelspace from userspace that the message is null terminated to avoid undefined behavior with functions such as `strlen`.

Finally, when removing the module from the kernel, it is critical that all memory be freed and all pins be reset to their default state, just as in the initialization function.

4.2 The Door Alarm

4.2.1 Testing

Conversely, while testing the kernel module under a wide variety of circumstances, it is also important to test the door alarm while many programs are attempting to manipulate other pins or even pins in use by the door alarm itself.

When running such tests, the door alarm performed as expected under situations where several other programs were attempting to modify the values and settings of pins 18, 14, and 15. This was due to code in the server which continuously wrote the correct values to the LED pins and reset each pin to its proper input/output mode. However, a more elegant solution to this problem will be discussed in Section 5.3 on possible future work.

4.2.2 Steps Taken to Ensure Consistency

Since the door alarm is intended to be used for home security, it is important to bear in mind that should the alarm fail (either by being shut down or the wires being tampered with), *the client should be notified of the failure* so appropriate action can be taken.

An advantage of the client/server approach is that when the client is disconnected from the server (i.e., due to a loss of power), it can be made to behave as if the alarm had been tripped. Additional error handling on the server-side takes care of any issues with GPIO pin I/O which could occur if the circuit was improperly configured.

By the same token, if a wire is cut on the door alarm, the default behavior of a pull down circuit is to leave the pin reading `0`, which trips the alarm. In a pull up circuit, the pin would still read `1` and the end user would have no way of knowing they are not safe. This is a key part of why we use a pull down circuit instead of the pull up alternative.

Another advantage of a pull down circuit is its physical safety. A resistor is always present to protect the GPIO pin from excess voltages resulting from an improperly configured pin setup.

5 Conclusion

5.1 Summary

Implemented a kernel module to create character device drivers for each GPIO pin on a Raspberry Pi 3. Ensured the stability of the kernel module before integration into Raspbian OS. Designed and built a circuit for the door alarm trigger and LEDs. Coded a client application and server application to allow the end user to interface with the door alarm. Created additional test software for general manipulation of the GPIO pins using read and write operations to the character device drivers supplied by the kernel module.

5.2 Relevance

The kernel module implements a set of character device drivers which support read and write operations between userspace and kernelspace. Implementation of the module requires both knowledge about character device drivers from the course and additional research about allocating space for multiple devices in a single module, registering them, and eventually freeing them.

The hardware component of the project is also relevant as the kernel module interfaces with the hardware through `linux/gpio.h` included with the Raspbian Linux Kernel.

The client/server programs use sockets to communicate with each other. The server copies GPIO pin data from kernelspace to userspace by reading from the character devices for the appropriate pins, then uses sockets to send that information to the client program, which decides what to do with it.

5.3 Future Work

Topics to review for future work include the following:

5.3.1 Allow Userspace Programs to Reserve Pins

Add to the write operation to allow a process to reserve/release a GPIO pin to gain exclusive access. This would solve the problem of other processes interfering with pins needed for the door alarm in a much more elegant way and would have implications for other projects as well.

5.3.2 Separate Client/Server Between Two Machines

Extend the client/server userspace programs to work on separate machines. This was the original intended functionality, however this was omitted to make it easier to demonstrate the project to a TA.

Another possibility still is to incorporate the client functionality in the user's smartphone with push notification support.

5.3.3 Add Extra Hardware

Add extra hardware to the project such as a speaker that could play a sound when the alarm trips.

Contributions of Team Members

Tri Do worked on the kernel module, the testerin.c and testerout.c code, the door alarm circuit, and the report.

William Findlay worked on the kernel module, the Python client/server for the door alarm, some miscellaneous tester code, and the report.

The final draft of the report was formatted and edited by **William Findlay** using Rmarkdown compiled into LaTeX with pandoc.

References

[1] “GPIO usage documentation.” https://www.raspberrypi.org/documentation/usage/gpio/README.md. Accessed: 12-Apr-2018.

[2] “GPIO code samples.” https://elinux.org/RPi_GPIO_Code_Samples. Accessed: 20-Mar-2018.

[3] Thales42, “Pull up and pull down resistors.” https://playground.arduino.cc/CommonTopics/PullUpDownResistor. Accessed: 03-Apr-2018.

[4] https://electronics.stackexchange.com/questions/113009/when-to-use-pull-down-vs-pull-up-resistors. Accessed: 03-Apr-2018.

[5] V. Ngyuen, “Implementation of linux gpio device driver on raspberry pi platform.” https://www.theseus.fi/bitstream/handle/10024/74679/Nguyen_Vu.pdf. Accessed: 26-Mar-2018.

[6] https://blog.csdn.net/opencpu/article/details/6764659. Accessed: 01-Apr-2018.