Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, July 30, 2022. Git commit: 3c5bfb0

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 8

Physical computing in Rust

Up until now, all the programs you wrote only exists in the virtual world. However, a big part of the physical world you live in is controlled by software. Traffic lights, self-driving cars, airplanes, and even rockets and satellites are just a few examples. Many of this software has to be compiled and executed in a drastically different environment than the usual Linux, Windows, or macOS desktop or laptop computers. They usually have to run on relatively weaker CPUs with less available memories. They might sometimes need to run without an operating system, or on specialized operating systems designed specifically for embedded systems.

Traditionally, these applications are usually written in C or C++ for maximum performance and low-level control of memory. Many of the embedded platforms are so limited that garbage collection is not feasible. But this is where Rust shines. Rust can provide performance and low-level control as C or C++ but guarantees higher safety. A Rust program can be compiled to run on many different CPUs like Intel, ARM, MIPS. It also supports various mainstream operating systems and even no operating systems.

8.1 What are you building?

In this chapter, you'll be focusing on using Rust on a Raspberry Pi. Raspberry Pi is an inexpensive computer with a credit card size footprint, created to make computer education more accessible. It has a few key important features that help us demonstrate the points for this chapter:

- It has an ARM CPU. You can demonstrate how to compile and cross-compile code for an ARM platform
- It has GPIO pins. You can use it to control physical circuits like LEDs and buttons.
- It's powerful enough to run a full Debian-based operating system (Raspbian), so you can demonstrate physical computing and cross-compilation without going too deep into bare-metal programming. But if you are feeling adventurous, you can try writing your own mini operating system on it with Rust.

To begin with, you'll install a full operating system on the Raspberry Pi. Then you'll install the complete Rust toolchain on it. You'll build two circuits, one for output and one for input, and use Rust to interact with them:

- Output: The first circuit will allow us to generate output to the physical world with light. You'll create a simple LED circuit connected to a GPIO output pin. You can write a Rust program to turn the LED on and off and blink it regularly.
- Input: You can take input from the physical world as well. You'll add a push button to the circuit. The Rust program can detect button clicks and then toggle the LED on and off.

These two examples will help us gain an understanding of how Rust code interacts with the physical world. However, you are compiling them on the Raspberry Pi itself. In many of the embedded applications, the target platform (i.e., the Raspberry Pi or similar board) is not powerful enough to compile the code. You'll move the compilation to another computer, which is more powerful but has a different CPU and OS than the target platform. This way of compiling is called cross-compilation. You'll set up a cross-compilation toolchain and cross-compile the previous example on it. Finally, to give you a sneak-peek into how the GPIO pin works, you'll use lower-level APIs to control it. You'll be able to get a sense of how the high-level GPIO libraries work.

8.2 Physical Computing on Raspberry Pi

Getting to know your Raspberry Pi

You'll be using a Raspberry Pi 3 B+ board for this chapter.

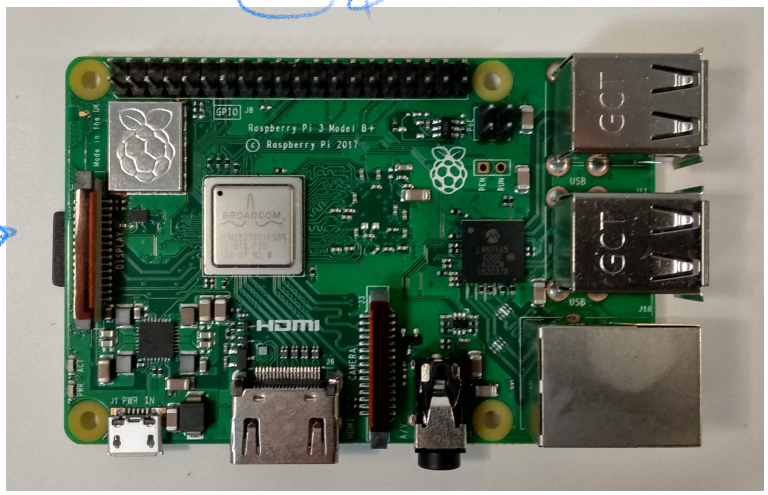

Figure 8.1: Raspberry Pi 3 B+

A Raspberry Pi board is like a mini-computer. It has all the necessary components of a computer: CPU, memory, WiFi, Bluetooth, HDMI output, USB, etc.

One significant difference between Raspberry Pi and typical desktop or laptop computer is that it uses an ARM CPU. Most of the mainstream desktop or laptop computers nowadays use the Intel x86/x86_64 architecture CPUs. However, ARM CPUs are more common in mobile, embedded, or IoT devices due to lower power consumption. Since Rust is a language that compiles to machine code, the CPU architecture dictates how the final output is.

Raspberry Pi features many peripherals. It has an SD card reader so you can load the program and an operating system onto an SD card. It has a micro-USB power input so it can run on a phone charger or even a portable power bank. For video output, you can use its HDMI output to connect to an HDMI monitor. To control the device, you can use a USB mouse and keyboard. Finally, you can see two rows of metal pins (on the top left edge of Figure 8.1). These are GPIO (General-Purpose Input/Output) pins, which you'll use in the later chapters to interact with external circuits like LEDs and buttons.

Install Raspbain through NOOBS

In the first example, you are going to show how to run a Rust program on an operating system. There are many operating systems available for Raspberry Pi. What you need to do is to install an operating system image onto the SD card and let the Raspberry Pi boot from the image. The official Raspberry Pi operating system is called Raspbian. Raspbian is a Debian-based operating system that has a friendly desktop environment and many useful software packages like the Firefox browser, text editor, calculator, and also programming environments for education, like Scratch, Python, Java, etc.

The easiest way to install Raspbian is to use an installer called *NOOBS* (New Out Of the Box Software). It provides a step-by-step wizard to guide you through the installation process. Here are the steps:

- Head to the NOOBS download page https://www.raspberrypi.org/downloads/noobs/ (Figure 8.2) and download the ZIP file for *Offline and network install*.
- Prepare an SD card (at least 8GB), formatted to the FAT format.
- Unzip the NOOBS ZIP file and copy all the files to the SD card.
- Plug the SD card into the Raspberry Pi.
- Connect your Raspberry Pi with a keyboard, mouse, and an HDMI monitor.
- Connect your Raspberry Pi with a micro-USB power source (usually a phone/tablet charger). This will turn the Raspberry Pi on.
- Once the NOOB installer has booted. You can select the "Raspbian" option from the menu. NOOBS will start the installation process.
- Once it's installed, you can reboot the Raspberry Pi, and the Raspbian OS is ready.

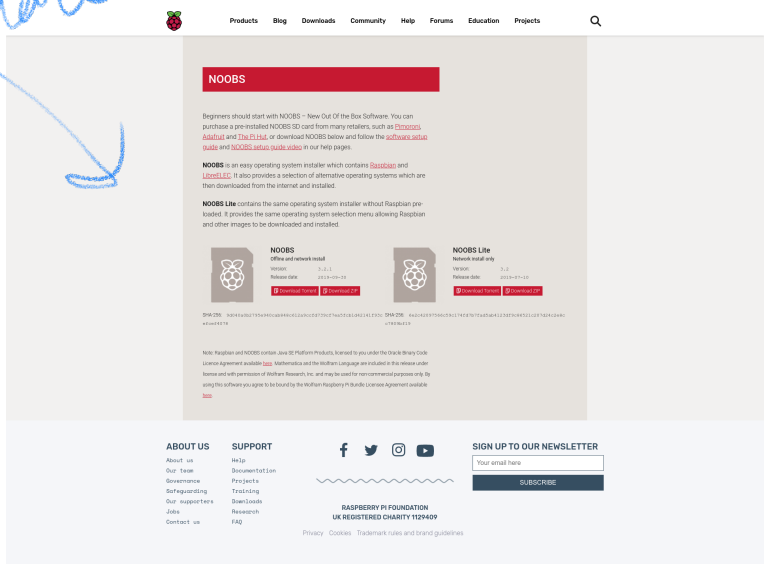

Figure 8.2: The NOOBS download page

■ **NOTE** If you don't want to install the operating system yourself, you can also buy a pre-installed SD card from many electronics or educational store online.

Install the Rust toolchain

As you did for the Linux desktop, you can install the Rust compiler and cargo on the Raspbian with `rustup`. Run the following command copied from the Rust official installation page https://www.rust-lang.org/tools/install:

```
curl␣https://sh.rustup.rs␣-sSf␣|␣sh
```

This will install the whole Rust toolchain on the Raspberry Pi. One big difference you might notice is that `rustup` detects the ARM CPU and suggests a different target architecture `armv7-unknown-linux-gnueabihf` (Figure 8.3). You want `rustc` to compile the Rust code into the ARM assembly so that the binary can run on the Raspberry Pi. Therefore, you'll take `rustup`'s suggestion and install the toolchain for ARM. Once installed, don't forget to add the cargo folder to the `PATH` environment variable, so the `cargo` command will work.

```
Current installation options:

   default host triple: armv7-unknown-linux-gnueabihf
     default toolchain: stable
               profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>
```

Figure 8.3: Rustup suggests installing the ARM target

Controlling the GPIO pins

Once you set the stage for Raspberry Pi, you are going to look at two rows of metal pins that occupy one side of the circuit board. These pins are called GPIO (General-purpose input/output) pins. These GIOP pins are used for communicating with the outside world. When a pin is acting as an output, you can control it with software to let it output either 3.3 volts (written as "3V3") or 0 volts. When a pin works as an input, it can detect whether the pin has high (3V3) or low (0V) voltage.

Not all pins are used as input/output pins. Some pins have special purposes like consistently providing 5V power, or work as ground (constant 0V). Figure 8.4 shows the layout of the pins. You can also find an interactive pin layout at https://pinout.xyz/.

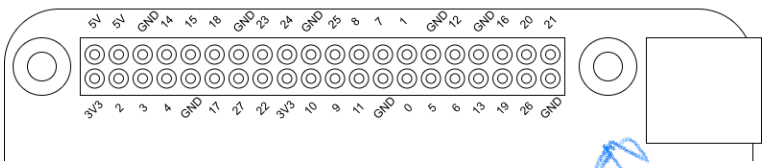

Figure 8.4: Raspberry Pi 3 B+ GPIO layout (BCM numbering)

There are a few different kinds of pins:

- 5V: 5V power supply
- 3V3: 3.3V power supply
- GND: ground
- *number*: the GPIO pin, the number is the BCM number. When you refer to pin by their number in the example code, you are referring to this number. [1]

[1] You might find the number confusing because they seem random. BCM refers to the "Broadcom SOC

Some of the pins can also be configured to communicate using particular protocols like PWM (Pulse-Width Modulation), SPI (Serial Peripheral Interface), I²C (Inter-Integrated Circuit) or Serial, which you'll not use in this book.

On a very high level, these GPIO pins are controlled by hardware registers. Registers are components in the chip that act like computer memory. You can read or write bits to them. To set the mode (input, output, or special protocol) of a pin, you can write a specific bit pattern to some register. These registers are exposed as memory addresses (`/dev/gpiomem`), so you can change their value as if you are writing to a particular memory location. But direct manipulation of memory is too low-level for most use cases, so there are a few abstractions on top of it. On the Raspbian OS, these registers are exposed as device files (`/sys/class/gpio/*`[2]). You can read from these virtual files to get the register's value. If you write to these files like regular files, the register will be set accordingly.

But manipulating these virtual files is still very tedious. To further conceal the complexity, you can use the `rust_gpiozero` crate. The `rust_gpiozero` is inspired by the Python `gpiozero` library, which exposes easy-to-use components like `LED` or `Button` so you can control these GPIO-connected hardware components with ease. The `rust_gpiozero` crate is built on top of the `rppal` (Raspberry Pi Peripheral Access Library) crate, which allows low-level access to various peripherals like the GPIO pins.

To start using `rust_gpiozero`, simply go to your Raspberry Pi desktop, open the terminal, and create a new project by `cargo new physical-computing`. Then you add the following to the generated `Cargo.toml` file:

```
[dependencies]
rust_gpiozero = "0.2.0"
```

Now you have the software environment ready. You can start connecting the hardware circuits and start coding.

Building a LED circuit

First, you would like to light up an LED (light-emitting diode) (Figure 8.5). LED is a small electronic component that will emit light when electrical current flows through it. The "D" in LED stands for *diode*, which means it only allows the electrical current to flow in one direction. The positive leg is called the *anode*, which is usually the longer leg of the two. The negative leg is called the *cathode*. You can give a high voltage to the anode, say 3.3V, and ground the cathode. This will create a current flowing from the anode to the cathode, and LED will light up.

■ **TIP** As you mentioned, the LED only works in one direction. If your LED is not lighting up, don't panic. Try to swap the direction of the LED, and it will probably work.

channel", which is the internal numbering of pins in the Broadcom brand CPU used by the Raspberry Pi. Some Raspberry Pi GPIO libraries also support the "board" numbering, which is the sequential left to right, bottom to top numbering from 1 to 40.

[2]They are provided by Sysfs, a Linux virtual file system.

Figure 8.5: A yellow LED

Although Raspberry Pi doesn't provide a very high voltage and current, it's still possible that too much current will pass through the LED and break it. To protect against such a scenario, you can add a resistor (Figure 8.6) into the circuit. A resistor creates resistance to the current, effectively limiting the current that goes through the LED.

It's pretty hard to work with free-floating LEDs and resistors by just connecting them with wires. You'll probably need to solder them together, but then it's hard to break them apart and re-arrange. To make it easier to work with experimental circuits, you can use a breadboard (Figure 8.6). A breadboard is a plastic board with tiny holes that LEDs, wires, and other electronic parts can plug into. Inside the holes are rows of metal pieces that act as temporary wires. A breadboard is perfect for prototyping because you can easily plug electronic parts and form a circuit. You can easily unplug them if you make any mistakes. You'll also be using jumper wires to connect the circuits. A jumper wire is a pre-cut wire with rigid plastic and metal head on each end. The head makes it easy to plug the wire into the breadboard and make them more durable than raw, unprotected wires.

Let's connect a circuit like Figure 8.7. A photo is shown in Figure 8.8. In this circuit, the electric current goes from GPIO pin 2 to the anode leg of the LED (connected through the breadboard) and comes out of the cathode leg. The current then goes through the current-limiting resistor, and to the ground rail (the column on the breadboard that is marked blue with a "-" logo), and finally goes to the ground GPIO pin. You can turn the LED on and off by setting the GPIO pin 2 to high (3V3) or low (0V) with Rust.

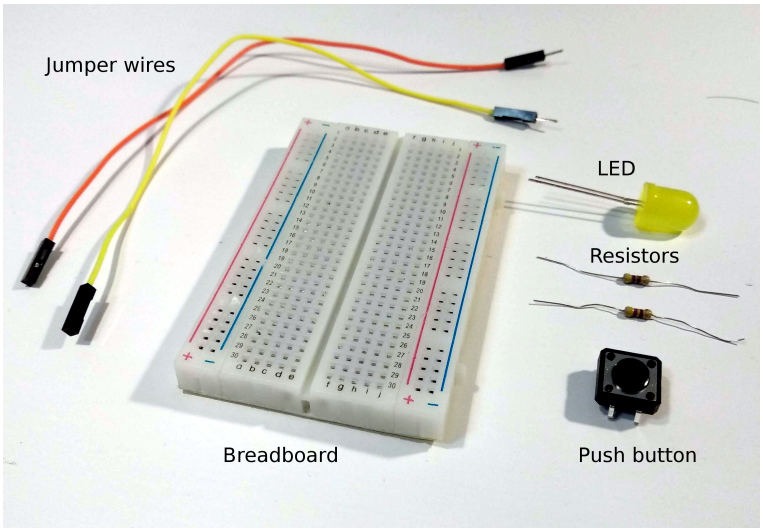

Figure 8.6: A Breadboard and jumper wires

Controlling the GPIO output from Rust

You already created a Rust project and added the `rust_gpiozero` crates to the dependencies. The `rust_gpiozero` page makes it really simple to control the GPIO pins. Open `src/main.rs` and write the code as in Listing 8.2.

TODO: Add filename TODO: Add filename

```
extern crate rust_gpiozero;
use rust_gpiozero::*;

fn main() {
    let mut led = LED::new(2);

    led.on();
}
```

You can compile and run this code on the Raspberry Pi by running `cargo run`. The compilation might take longer because the Raspberry Pi ARM CPU might not be as powerful as your desktop or laptop computer's CPU. If your circuit is connected correctly, you should see the LED lights up.

In Listing 8.2, you initiate an `LED`. The integer parameter is the GPIO pin number that this `LED` should control. During initiation, the `rust_gpiozero` will set the GPIO pin in the output mode for you automatically. Then when you call `led.on()`, it will set the correct bit in the register and make the GPIO pin go high (3.3V), which turns the LED on. As you might have guessed, the way to turn off this LED is as simple as changing the

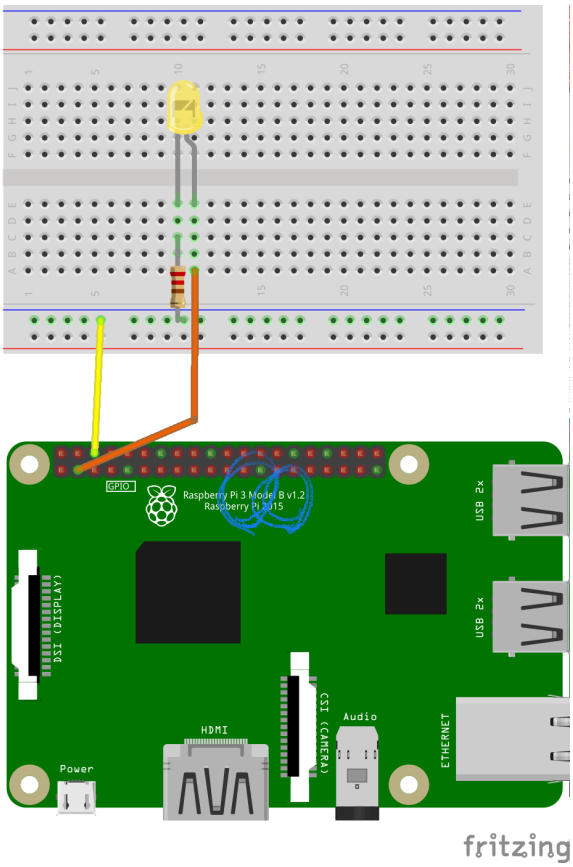

Figure 8.7: The LED circuit diagram (image created with Fritzing)

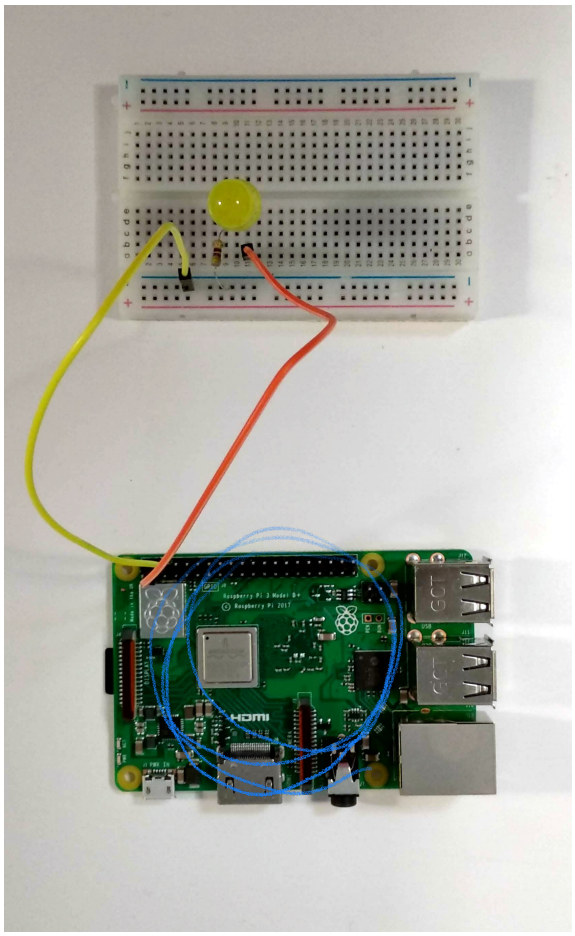

Figure 8.8: The LED circuit

code to `led.off()` and run `cargo run` again.

If you wish to flash the LED, you can use a `loop` and add a one second pause between `led.on()` and `led.off()`. The pause can be easily achieved by `sleep(Duration::from_secs(1))` provided by the standard library. The complete code for it is in Listing 8.2.

TODO: Add filename TODO: Add filename

```
extern crate rust_gpiozero;
use rust_gpiozero::*;
use std::thread::sleep;
use std::time::Duration;

fn main() {
    let led = LED::new(2);

    loop{
        println!("on");
        led.on();

        sleep(Duration::from_secs(1));

        println!("off");
        led.off();

        sleep(Duration::from_secs(1));
    }
}
```

To make it even simpler, `rust_gpiozero` already implemented flashing for us. You can use the `LED::blink()` function as shown in (Listing 8.2)

TODO: Add filename TODO: Add filename

```
extern crate rust_gpiozero;
use rust_gpiozero::*;

fn main() {
    let mut led = LED::new(2);

    led.blink(1.0, 1.0);

    led.wait(); // Prevents the program from exiting
}
```

The `LED::blink()` function takes two parameters: the `on_time` and `off_time`. The `on_time` is how many seconds the LED should stay on; the `off_time` is how many seconds to pause in between. You also have to call the `LED::wait()` function to prevent the program from exiting right after the `LED::blink()` call.

Reading button clicks

Now you learned how to output a light signal from the software world to the physical world. You can take a look at how you can accept physical inputs. You can configure the GPIO pin to use the input mode and receive inputs from a physical button.

The GPIO input pin can be configured to detect the voltage and trigger the code when the voltage reached the desired level. However, the GPIO input pin by itself will not stay at either 0V or 3V3. It will be in a *floating* state where its voltage is between 0V and 3V3, which makes it prone to false-trigger. Gladly the Raspberry Pi has an internal resistor that can be configured to keep the GPIO pin at 0V or 3V3. This is the resistor tagged as "Pull-down resistor" and "Pull-up resistor" in Figure 8.9 and 8.10. When you enable the pull-down resistor, it will connect the pin to the ground, thus pulling the voltage down to 0V. When you enable the pull-up resistor, it will connect the pin to an internal 3V3 voltage source and pulling the voltage up to 3V3.

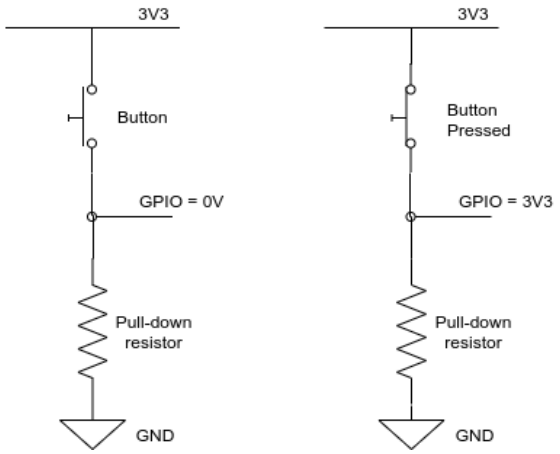

Figure 8.9: Input pin with an internal pull-down resistor

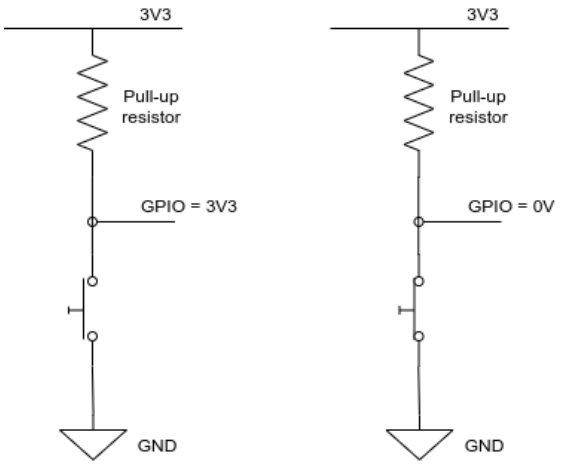

Figure 8.10: Input pin with an internal pull-up resistor

You can imagine a button as two pieces of separated metal, so no current will flow through when it's not pressed. When you press the button, the two pieces of metal touch and short-circuits, allowing the current to flow through. Since the GPIO input pin can detect voltage change, you can use the GPIO pins to detect a button press in two ways:

1. Configure the GPIO pin to use an internal pull-down resistor, so it stays at 0V. Connect one end of the button to the GPIO input pin, the other end to the 3V3 voltage source. When the button is pressed, the GPIO pin is short-circuited with the 3V3 source, so its voltage is drawn up to 3V3. (Figure 8.9)

2. Configure the GPIO pin to use an internal pull-up resistor, so it stays at 3V3. Connect one end of the button to the GPIO input pin, the other end to the ground. When the button is pressed, the GPIO pin is short-circuited with the ground, so its voltage is drawn down to 0V. (Figure 8.10)

In both cases, the GPIO pin will detect a voltage change and triggers the code. For the demonstration, you'll be using option 2. You can add some circuits on top of the LED circuit, as shown in Figure 8.11, and the photo is shown in Figure 8.12. You attach one end of the button to GPIO pin 4, which is configured to have a pull-up resistor that keeps it at 3V3. The other end of the button is connected to the ground through a current-limiting resistor. When the button is pressed, the GPIO pin 4 is short-circuited to the ground and should drop to 0V.

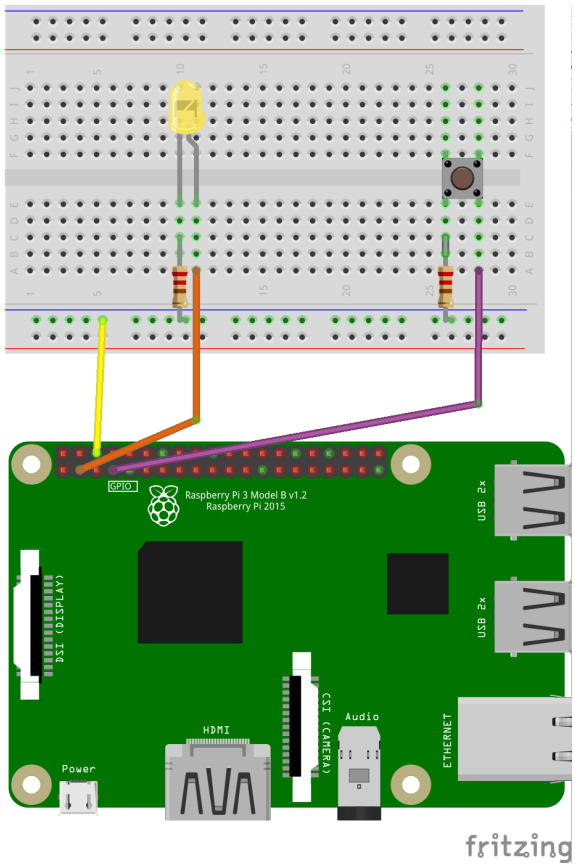

Figure 8.11: The button circuit diagram (image created with Fritzing)

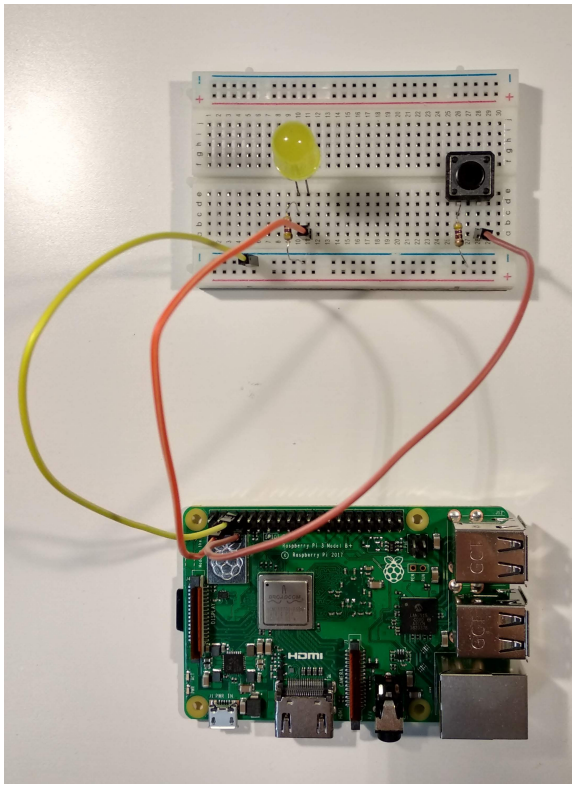

Figure 8.12: The button circuit

Then you can write some code to detect the voltage drop and toggle the LED (Listing 8.2):

TODO: Add filename TODO: Add filename

```
extern crate rust_gpiozero;
use rust_gpiozero::*;

fn main() {
    let mut led = LED::new(2);
    let mut button = Button::new(4);

    loop{
        println!("wait for button");
        button.wait_for_press(None);
        // Make the led switch on
        println!("button pressed!");
        led.toggle();
    }
}
```

You first create a `LED` and a `Button`. The `Button::new()` function will configure that pin to what you describe in option 2. If you wish to use option 1, you can use `Button::new_with_pulldown()` instead. In the loop, you called `button.wait_for_press()`, this will block the program until the button is pressed. You can optionally set a timeout by replacing the `None` parameter with a `Some(f32)`, the f32 number is the timeout in seconds. When the button is clicked, the function will return and continue to the next line `led.toggle()`, which does what its name suggests: toggle the LED's on/off state.

As you mentioned before, inside the button, there are two pieces of metal. When you press the button, you might imagine that the two pieces of metal touch each other immediately and stay touched until you let go. But in reality, the metal pieces might bounce off each other after they make contact for a fraction of a second. So there will be a very short period of time when the metal piece touch and bounce off repeatedly until they finally settled in the touched position. Since the `loop` runs very fast, the bounce might trigger the `button.wait_for_press()` function multiple times, so the LED will flicker and might not reach the final state you want it to be. To counter this issue, you can *debounce* the circuit by only allowing the button press to be triggered once in a short period of time. For example, when you detect a button is pressed, you ignore all the other button press events in the next second. You can improve the button click code as in Listing 8.2:

TODO: Add filename TODO: Add filename

```
extern crate rust_gpiozero;
use rust_gpiozero::*;
use std::time::{Duration, Instant};
```

```
fn main() {
    let mut led = LED::new(2);
    let mut button = Button::new(4);

    let mut last_clicked = Instant::now();
    loop{
        button.wait_for_press(None);

        if last_clicked.elapsed() < Duration::new(1, 0) {
            continue
        }

        led.toggle();
        last_clicked = Instant::now()
    }
}
```

You added a `last_clicked` variable, which keeps track of when the button is last clicked using `std::time::Instant::now()`. When the `button.wait_for_press()` function returns, you first check `last_clicked.elapsed()`, which is the time elapsed since you last called `Instant::now()`. If the elapsed time is less then a second (`Duration::new(1, 0)`[3]), you consider this press event to be a bounce and ignore it by `continuing`. Otherwise, you let it through and toggle the led, then update the `last_clicked` timestamp. With this debounce functionality, the LED no longer flickers. Even if you accidentally click the button multiple times in one second, it will only be toggled once. If you think once second is too long, you can reduce the debounce time to make the button more responsive.

8.3 Cross-compiling to Raspberry Pi

You might notice that the Rust program compiles relatively slow on your Raspberry Pi. This is because the Raspberry Pi CPU is not as powerful as most mainstream desktop CPUs. But Raspberry Pi's CPU is already quite powerful in the embedded world. For some applications, you can expect a much weaker (but more energy-efficient) CPUs or even microcontrollers. Those devices are designed to run on very limited CPU, memory, and battery conditions, so it's not possible to load all the source code onto it and compile it on the device. This is when you need to use cross-compilation. Cross-compilation means you compile the source code on a different machine (the host) than the one running it (the target). For example, you can compile the code on a powerful, Intel-x86-based Linux desktop, and run the binary on the Raspberry Pi target. In this case, the compiler itself is running on an x86 architecture CPU, but it generates machine code (and binary format) for an ARM architecture CPU.

[3] The `Duration::new()` takes two arguments, the first is the seconds, and the second argument is the additional nanoseconds. Therefore, `(1, 0)` means 1 second + 0 nanosecond = 1 second.

To set up the cross-compilation environment, you need to move back to the Linux desktop/laptop. You can install the compiler toolchain for Raspberry Pi, which runs on the ARM Cortex-A53 CPU. You can add a compile target with `rustup`:

```
rustup target add armv7-unknown-linux-gnueabihf
```

■ **NOTE** You might be wondering why you install the `armv7` target while the ARM Cortex-A53 CPU is advertised as ARMv8 architecture? This is because the ARM Cortex-A53 CPU supports both 32-bit mode and 64-bit mode. By default, the Raspbian is built on a 32-bit Linux. Therefore, the CPU will run in 32-bit mode, which only supports ARMv7-compatible features. If you run `cat /proc/cpuinfo`, it will also report itself as an ARMv7 CPU.

You also need a linker. You might not be aware of the linker when you work on an x86-based Linux. Most of the time, the linker was already installed when you install other programs. Usually, the linker comes in the package of a C compiler so you can install `gcc` (GNU Compiler Collection) to get the ARM linker:

```
sudo apt-get install gcc-5-multilib-arm-linux-gnueabihf
```

Before you compile, you also need to let `cargo` know where to look for the linker. You can open the configuration file `~/.cargo/config` (create one if it doesn't exist yet) and add the following setting:

```
[target.armv7-unknown-linux-gnueabihf]
linker = "arm-linux-gnueabihf-gcc-5"
```

This means "when you compile for the target `armv7-unknonw-linux-gnueabihf`, use the linker provided by `arm-linux-gnueabihf-gcc-5`"

Now let's create a new project with `cargo new blink-cross-compile`. You can open the `src/main.rs` file and copy the code for blinking an LED (Listing 8.2) into it. Also don't forget to add the `rust_gpiozero` dependency to the `Cargo.toml` file.

To compile the Rust project to a specific target, use the `--target` argument like so:

```
cargo build --target=armv7-unknown-linux-gnueabihf
```

This will produce a binary in `target/armv7-unknown-linux-gnueabihf/debug/` named `blink-cross-compile`. Notice that the binary is placed in the `target/armv7-unknown-linux-gnueabihf` folder, not the default `target/debug` folder. If you try to execute this binary on the x86 Linux system, you'll get the following error message:

```
$ ./blink-cross-compile
bash: ./blink-cross-compile: cannot execute binary file: Exec
    format error
```

This is because this binary is cross-compiled for ARM. You can verify this by examining the file with the Unix `file` command:

```
$ file ./blink-cross-compile
./blink-cross-compile: ELF 32-bit LSB shared object, ARM, EABI5
    version 1 (SYSV),
  dynamically linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
  BuildID[sha1]=43d4fc4e17539883185e15c3d442986f2fb2f03d, not
    stripped
```

The final step is to copy this binary onto the Raspberry Pi SD card and boot up the Raspbian OS. Once the Raspberry Pi is booted, open a terminal and `cd` to the location of the binary and executed it. You should see the LED blinking just like before.

8.4 How does the GPIO code work?

The `rust_gpiozero` crate abstract away most of the complexity of setting up the GPIO pins. But you can take a look into how it works at a lower level.

As you mentioned in Section 8.2, the GPIO registers are exposed in two different interfaces: `/dev/gpiomem` and `Sysfs`. Sysfs exposes the GPIO registers as virtual devices, which is easier to understand, so you'll start with those.

To achieve the same effect as Listing 8.2 of turning on an LED, you can write a simple shell script using the Sysfs virtual files (Listing 8.4).

TODO: Add filename TODO: Add filename

```
echo "2" > /sys/class/gpio/export
echo "out" > /sys/class/gpio/gpio2/direction

echo "0" > /sys/class/gpio/gpio2/value
```

First, you need to write the pin number to the file `/sys/class/gpio/export`. This tells Sysfs that you want to work with the specified pin. Before you `export` the pin, the virtual pin device `/sys/class/gpio/gpio2` does not exist. So you first export the pin 2 with

```
echo "2" > /sys/class/gpio/export
```

A new device `/sys/class/gpio/gpio2` will appear. You can then set the direction of the pin by writing either `in` or `out` to the file `/sys/class/gpio/gpio2/direction`. Because you are using this pin as an output device, you `echo` the text `out` to it. This effectively is setting the registers that controls the GPIO 2's mode. Then setting the pin to high or low is as simple as writing `1` or `0` to the `/sys/class/gpio/gpio2/value` file.

These Sysfs files are basically abstractions around the registers that control the GPIO pins. But the `rust_gpiozero` crate uses a more low-level crate, `rppal`, to interact

with GPIO pins. For performance reason, the `rppal` crate does not use the Sysfs interface. Instead, it work with the `/dev/gpiomem` directly. The `/dev/gpiomem` is a virtual device that represents the memory-mapped GPIO registers. If you call the `mmap()` system call on `/dev/gpiomem`, the GPIO registers will be mapped to the designated virtual memory addresses. You can then read or write the bits in memory to control the registers directly.

■ **TIP** The `/dev/gpiomem` virtual device was created to overcome the permission issue. Before `/dev/gpiomem` is available, you can access the GPIO-related memory address with `/dev/mem`. However, `/dev/mem` exposes the whole system memory and requires root permission to access. But since GPIO access is so common on Raspberry Pi, every program that interacts with GPIO needs to use root access, which creates a security hazard. Therefore, `/dev/gpiomem` was created to expose only the GPIO-related part of the memory with no special permission needed. In `rppal` source code, you can see it tries `/dev/gpiomem` first. If any error occurs, it falls back to `/dev/mem` but then requires root access.

If you look into `rppal`'s source code, you can wee that it use `unsafe` code block to `mmap()` the `/dev/gpiomem` device. Once it's mapped into the virtual memory, you can write bits to set the direction and value of pins with low-level memory manipulation. You show the relavant part of the code from `rppal`'s `textsrc/gpio/mem.rs` file in Listing 8.4.

TODO: Add filename TODO: Add filename

```
const␣PATH_DEV_GPIOMEM:␣&str␣=␣"/dev/gpiomem";

const␣GPFSEL0:␣usize␣=␣0x00;
const␣GPSET0:␣usize␣=␣0x1c␣/␣std::mem::size_of::<u32>();
const␣GPCLR0:␣usize␣=␣0x28␣/␣std::mem::size_of::<u32>();
const␣GPLEV0:␣usize␣=␣0x34␣/␣std::mem::size_of::<u32>();

pub␣struct␣GpioMem␣{}

impl␣GpioMem␣{
␣␣␣␣//␣...
␣␣␣␣fn␣map_devgpiomem()␣->␣Result<*mut␣u32>␣{
␣␣␣␣␣␣␣␣//␣...
␣␣␣␣␣␣␣␣//␣Memory-map␣/dev/gpiomem␣at␣offset␣0
␣␣␣␣␣␣␣␣let␣gpiomem_ptr␣=␣unsafe␣{
␣␣␣␣␣␣␣␣␣␣␣␣libc::mmap(
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ptr::null_mut(),
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣GPIO_MEM_SIZE,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣PROT_READ␣|␣PROT_WRITE,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣MAP_SHARED,
```

```
                gpiomem_file.as_raw_fd(),
                0,
            )
        };

        Ok(gpiomem_ptr as *mut u32)
    }
    #[inline(always)]
    fn write(&self, offset: usize, value: u32) {
        unsafe {
            ptr::write_volatile(self.mem_ptr.add(offset), value)
        ↪ ;
        }
    }

    #[inline(always)]
    pub(crate) fn set_high(&self, pin: u8) {
        let offset = GPSET0 + pin as usize / 32;
        let shift = pin % 32;
        self.write(offset, 1 << shift);
    }

    #[inline(always)]
    pub(crate) fn set_low(&self, pin: u8) {
        let offset = GPCLR0 + pin as usize / 32;
        let shift = pin % 32;
        self.write(offset, 1 << shift);
    }

    pub(crate) fn set_mode(&self, pin: u8, mode: Mode) {
        let offset = GPFSEL0 + pin as usize / 10;
        let shift = (pin % 10) * 3;

        //...

        let reg_value = self.read(offset);
        self.write(
            offset,
            (reg_value & !(0b111 << shift)) | ((mode as u32) <<
        ↪ shift),
        );
    }
}
```

You won't go into detail here, but you can see the `libc::mmap()` call inside the `map_devgpiomem()` function. You can also see that all the operations you did have corresponding functions in it:

- Set direction: `set_mode()`
- Set pin to high: `set_high()`
- Set pin to low`set_low()`

All this function writes directly to memory with `std::mem::transmute()` inside `unsafe` blocks. You might wonder how do they know which memory address is for which functionality? They are all defined in the manual of the Broadcom BCM2837B0 chip.

8.5 Where to go from here?

This chapter only scratched the surface of physical computing with Rust or programming for embedded systems in a general sense. There are many directions you can explore this field further. As usual, there is an "Are You X Yet?" page for embedded systems in Rust: Are You Embedded Yet? (https://afonso360.github.io/rust-embedded/), where you can find exciting progress of the embedded ecosystem.

In Rust's core team, there are some domain-specific working groups. The *Embedded devices working group* is responsible for overseeing the Rust embedded ecosystem. They also maintain a curated list of exciting projects and resources at https://github.com/rust-embedded/awesome-embedded-rust. If you wish to follow the latest development and the working group's future direction, you might want to follow the issues on their coordination repository: https://github.com/rust-embedded/wg.

There are a few directions you can explore. The first is to build up on top of Raspberry Pi. You only touched LED and buttons, but there is much more hardware you can connect to it. For example,

- buzzer
- Light sensor
- Sound sensor
- Orientation sensor
- Camera
- Humidity and temperature sensor
- Infrared sensor
- Ultrasonic sensor
- Touch screen
- Servo motors

There are also add-on boards called "HATs" (Hardware Attached on Top). They are boards with a lot of the hardware components packed into one. They are designed in a way that they can mount directly on top of the Raspberry Pi board and connect with its GPIO pins. The board will also communicate with the Raspberry Pi board and configure the GPIO pins for you. This provides you an easy way to try out many different kinds of hardware without worrying about wiring on a breadboard. You can find crates like `sensehat` that provides an abstraction layer for a specific HAT called the *Sense HAT*. There is also a tutorial accompanying the crate: https://github.com/thejpster/pi-workshop-rs/.

You can also explore other boards and platforms. Rust supports many different computer architectures, so there are many boards available. For instance, the Embedded Working Group published "The Embedded Rust Book" (https://rust-embedded.github.io/book/intro/index.html). In the book, they teach you how to program a STM32F3DISCOVERY board, which runs an STM32F303VCT6 microcontroller. Many of the tutorials are using the QEMU emulator. So you don't need actual hardware to get started. You can build your code and run it on a hardware emulator.

You can also go deeper. One thing that you didn't mention is how to run Rust on a bare-metal environment. Bare metal means the Rust program runs directly on the hardware without an operating system. In such a case, you can't use the standard library because many of the standard library functions depend on the platform. Instead, you need to set the `#![no_std]` attribute on the crate to let Rust compiler know that you can't use `libstd`. It will then use `libcore`, which is a platform-agnostic subset of `libstd`. This will also exclude many features you might not want in an embedded environment like dynamic memory allocation and runtime. The Embedded Rust Book will get you started with bare-metal programming. If you are looking to go deeper, you can read the advanced book "The Embedonomicon" (https://docs.rust-embedded.org/embedonomicon/index.html). You might want to go even further to build your operating system, but you'll touch on that topic in Chapter 10.