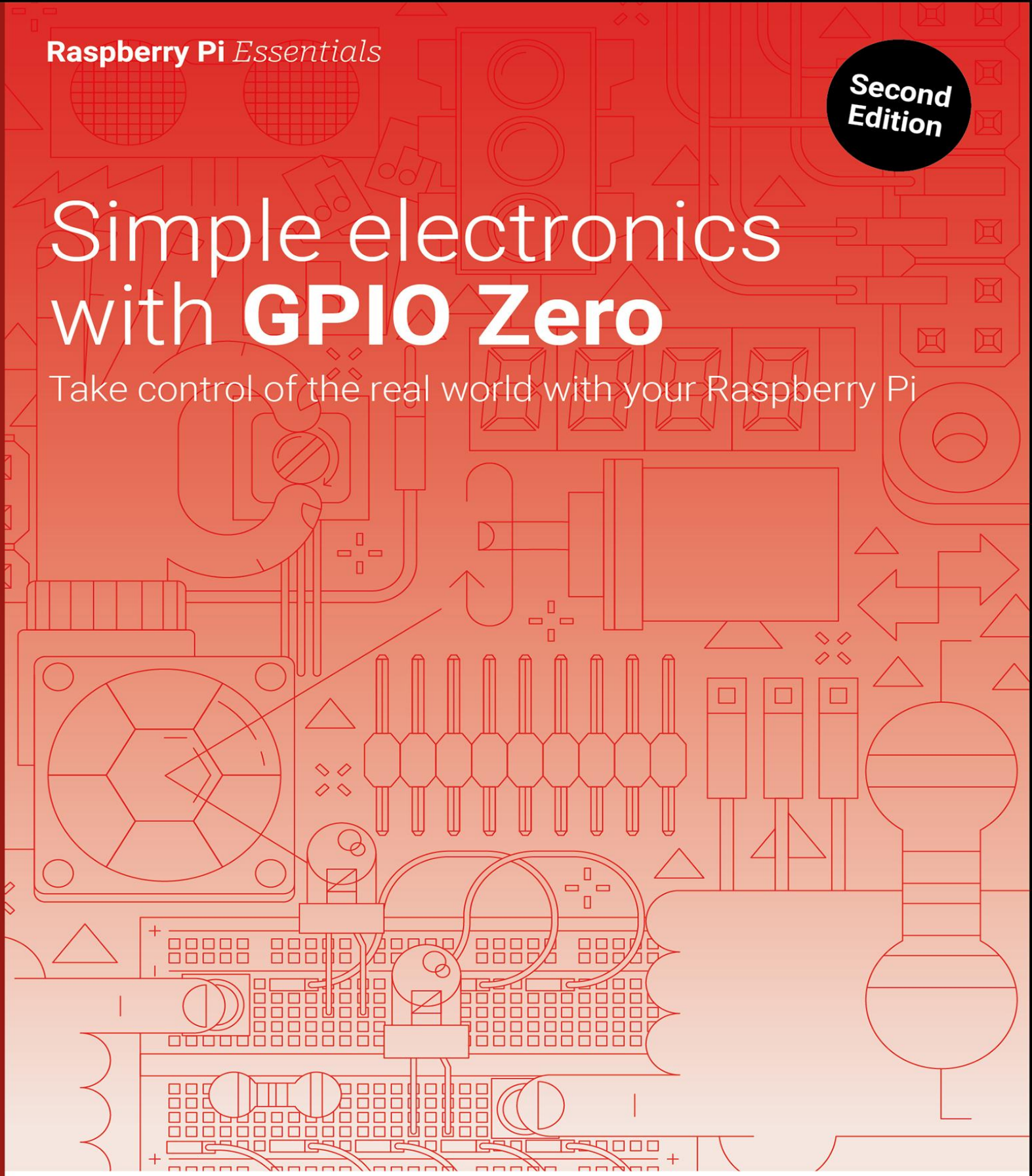
Raspberry Pi *Essentials*

Second Edition

Simple electronics with **GPIO Zero**

Take control of the real world with your Raspberry Pi



Raspberry Pi PRESS

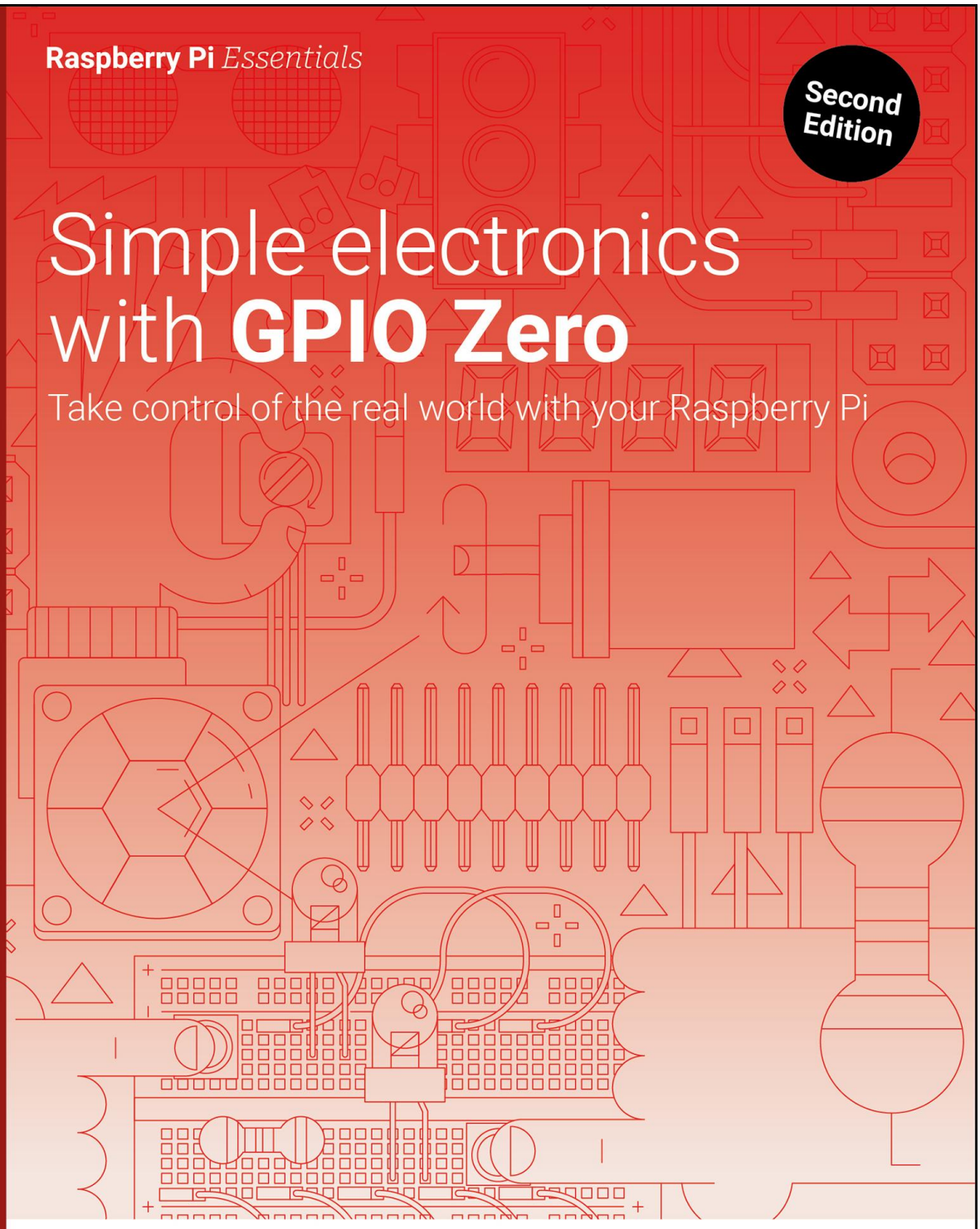
Phil King

Raspberry Pi *Essentials*

Second Edition

Simple electronics with **GPIO Zero**

Take control of the real world with your Raspberry Pi



Raspberry Pi PRESS

Phil King

Simple electronics with GPIO Zero, 2nd Edition

Raspberry Pi Essentials: *Simple electronics with GPIO Zero*, 2nd Edition
by Phil King
ISBN: 978-1-916868-44-1

Printed in the United Kingdom
Published by Raspberry Pi, Ltd., 194 Science Park, Cambridge, CB4 0AB
Raspberry Pi Ireland Ltd, 3 Dublin Landings, D01 C4E0, compliance@raspberrypi.com

Editor: Andrew Gregory
Copy Editor: Nicola King
Interior Designer: Sara Parodi
Production: Brian Jepson
Photographer: Brian O'Halloran
Illustrator: Sam Alder
Graphics Editor: Natalie Turner
Publishing Director: Brian Jepson
Head of Design: Jack Willis
CEO: Eben Upton

June 2025: Second Edition
August 2016: First Edition

Welcome

There are many things that distinguish Raspberry Pi from other computers. The GPIO header might just be the most significant of them all — it allows you to connect electronic components to your Raspberry Pi and control them with code you've written yourself.

The most popular programming language for controlling electronics on a Raspberry Pi is Python, particularly the code in the GPIO Zero library, which you can use to control LEDs, sensors, motors, and many more components. Before its creation by Ben Nuttall and Dave Jones, connecting electronics required numerous lines of code just to get everything set up. GPIO Zero simplifies those complicated parts of GPIO programming so you can focus on controlling the physical devices. As well as resulting in far fewer lines of code, it makes it a lot easier for newcomers to understand.

Using the GPIO pins on your Raspberry Pi opens up a whole new world of possibilities. While it might seem daunting at first glance, with our help you'll be creating electronic circuits and controlling them with the Python programming language within minutes. Grab your breadboard and start taking control of the real world with your Raspberry Pi today!

You can find example code and other information about this book, including errata, in its GitHub repository at **rpimag.co/gpiobookgit**. If you've found what you believe is a mistake or error in the book, please let us know by using our errata submission form at **rpimag.co/gpiobookfeedback**.

About the author

Phil King is a Raspberry Pi enthusiast and regular contributor to Raspberry Pi Official Magazine. Growing up in the ‘golden era’ of 8-bit computers in the 1980s, he leapt at the chance to write about them in magazines such as CRASH and ZZAP!64. When consoles took over the video games world, he missed the opportunity to program… until the Raspberry Pi came along.

Colophon

Raspberry Pi is an affordable way to do something useful, or to do something fun.

Democratising technology — providing access to tools — has been our motivation since the Raspberry Pi project began. By driving down the cost of general-purpose computing to below $5, we’ve opened up the ability for anybody to use computers in projects that used to require prohibitive amounts of capital. Today, with barriers to entry being removed, we see Raspberry Pi computers being used everywhere from interactive museum exhibits and schools to national postal sorting offices and government call centres. Kitchen table businesses all over the world have been able to scale and find success in a way that just wasn’t possible in a world where integrating technology meant spending large sums on laptops and PCs.

Raspberry Pi removes the high entry cost to computing for people across all demographics: while children can benefit from a computing education that previously wasn’t open to them, many adults have also historically been priced out of using computers for enterprise, entertainment, and creativity. Raspberry Pi eliminates those barriers.

Raspberry Pi Press

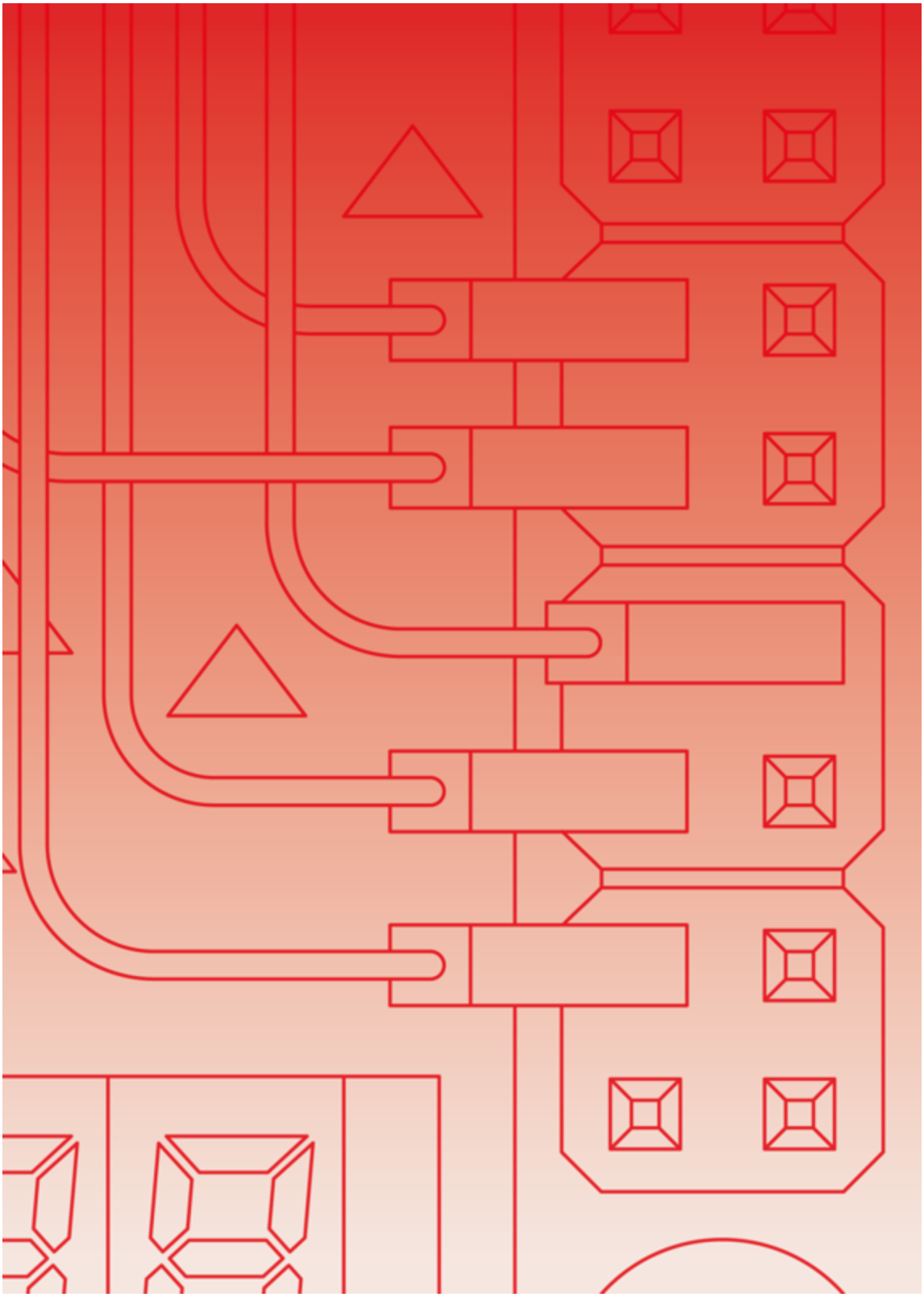
store.rpipress.cc

Raspberry Pi Press is your essential bookshelf for computing, gaming, and hands-on making. We are the publishing imprint of Raspberry Pi Ltd. From building a PC to building a cabinet, discover your passion, learn new skills, and make awesome stuff with our extensive range of books and magazines.

Raspberry Pi Official Magazine

magazine.raspberrypi.com

Raspberry Pi Official Magazine is written for the Raspberry Pi community. It's packed with Raspberry Pi-themed projects, computing and electronics tutorials, how-to guides, and the latest community news and events.



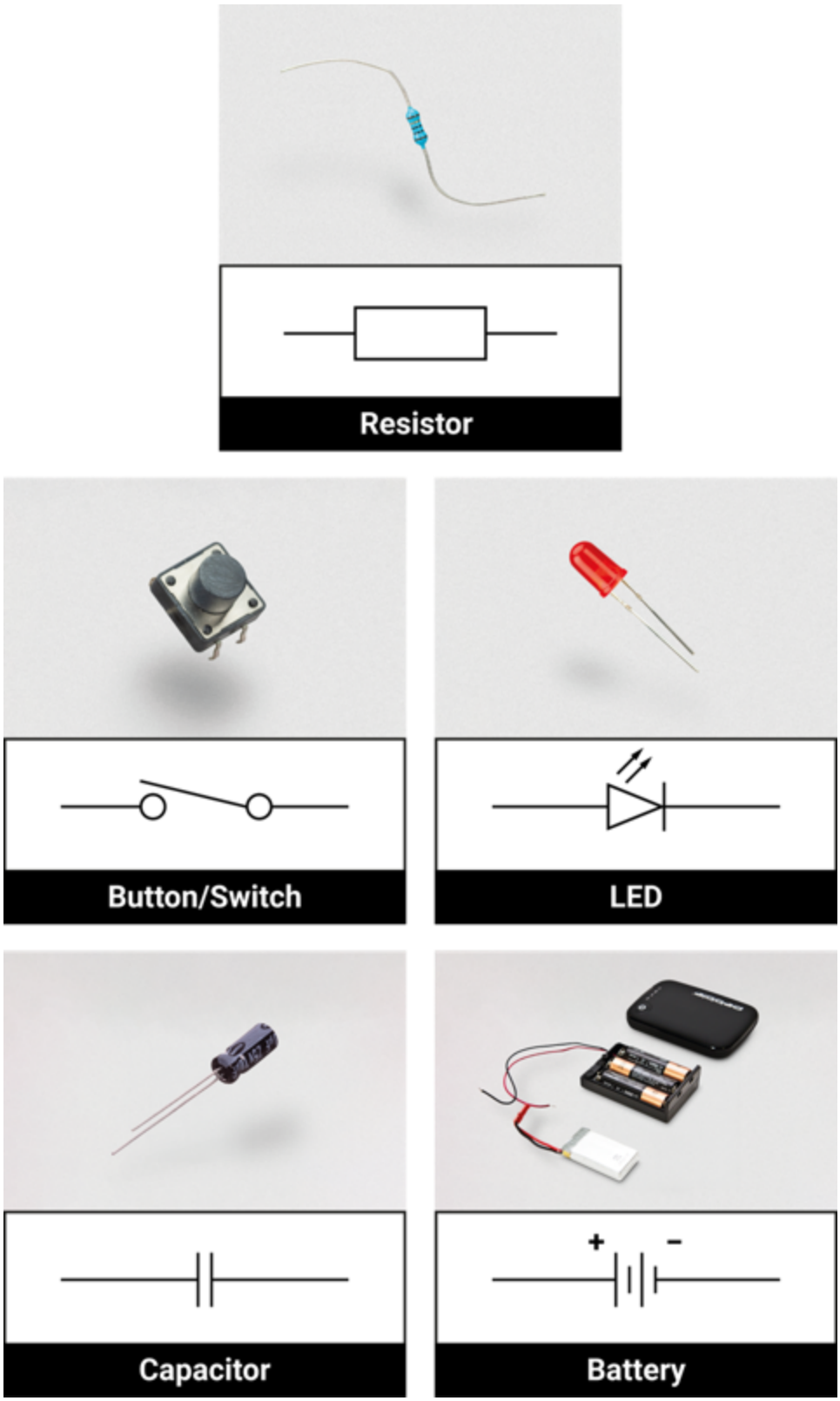
Chapter 1

Get started with electronics and GPIO Zero

Connect electronic components to your Raspberry Pi and write code to interact with the real world

Raspberry Pi is a great platform for learning computing. Whether that's writing your own programs or building a media server, Raspberry Pi has the tools, resources, and community support to help you learn how to build what you want. Raspberry Pi is also very good at *physical computing* — programming and interacting with the real world through electronics. As the name suggests, physical computing is all about controlling things in the physical world with your programs: using hardware alongside software. When you set the program on your washing machine, change the temperature on your programmable thermostat, or press a button at traffic lights to cross the road safely, you're using physical computing.

Electronic circuits are the physical part of a physical computing project. You'll connect these circuits to your Raspberry Pi, which, together with the code you'll write, is the computing part of the project. These circuits can be simple or very complex and are made up of electronic components such as LEDs, buzzers, buttons, resistors, capacitors, and even integrated circuit (IC) chips.



At its simplest, an electronic circuit lets you route electricity to certain components in a specific order, from the positive end of a circuit to the

ground (or zero volts) end. Think of a light in your house: the electricity passes through it, so it lights up. You can add a switch that breaks the circuit, so it only lights up when you press the switch. Now it's an interactive electronic circuit.

Reading circuit diagrams

Building a circuit can be easy if you know what you're doing, but if you're making a new circuit or are new to electronics in general, you'll most likely have to refer to a circuit diagram. This is a common way you'll see a circuit represented, and these diagrams are much easier to read and understand than a photo of a circuit. However, components are represented with symbols which you'll need to learn or look up.

Figure 1-1 is an example of a light circuit. Here we have a power source (a battery in this circuit), a switch, a resistor, and an LED. The lines represent how the circuits are connected, either via wire or other means. Some components can be inserted any way round, such as the resistor or switch. However, others have a specific orientation, such as the LED. LED stands for *Light-Emitting Diode*, and diodes only let electricity flow freely in one direction; luckily, real-life LEDs have markers such as a longer leg or a flat edge to indicate which side is positive, making them easier to wire up.

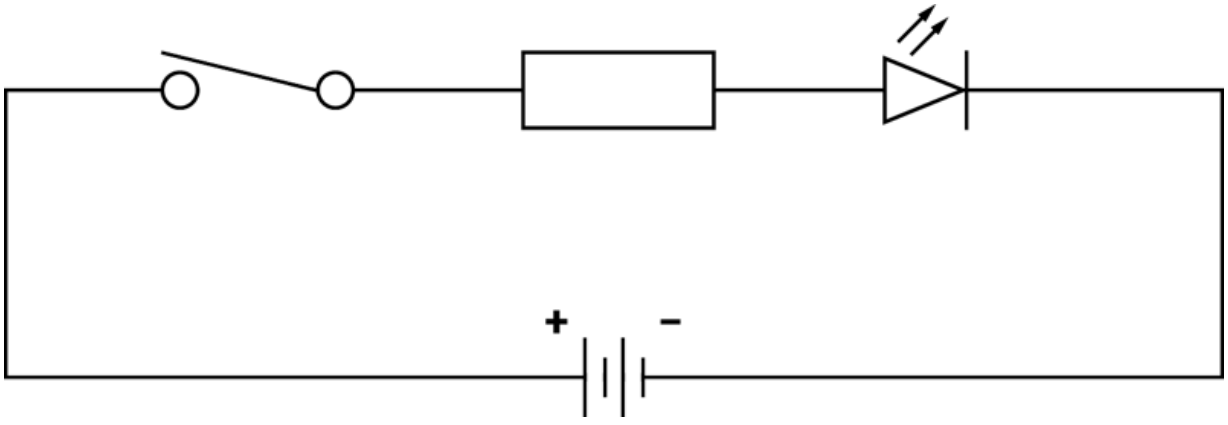


Figure 1-1: Switch circuit

Introducing the GPIO header

At the top of Raspberry Pi's circuit board, or at the back of a Raspberry Pi 400 or 500, you'll find two rows of metal pins. This is the GPIO (general-purpose input/output) header and it's there so you can connect electronic components to the Raspberry Pi. As the name suggests, these pins can be used for both input and output.

Raspberry Pi's GPIO header is made up of 40 pins as shown in Figure 1-2. Some pins are available for you to use in your physical computing projects, some pins provide power, and other pins are used to communicate with add-on hardware.

The Raspberry Pi 400 and 500 compact keyboard computers have the same GPIO header with all the same pins, but it's turned upside-down compared to other Raspberry Pi models. Figure 1-3 assumes you're looking at the GPIO header from the back of Raspberry Pi 400 or 500. Always double-check your wiring when connecting anything to the GPIO header on one of the compact computer models — it's easy to forget, despite the Pin 40 and Pin 1 labels on the case!

Raspberry Pi Zero 2 W has a GPIO header too but doesn't necessarily have header pins attached. If you want to do physical computing with Raspberry Pi Zero 2 W, or another model in the Raspberry Pi Zero family, you'll need to *solder* the pins into place using a soldering iron. If that sounds a little adventurous for now, check with an approved Raspberry Pi reseller for a Raspberry Pi Zero 2 WH with the header pins already soldered into place for you.

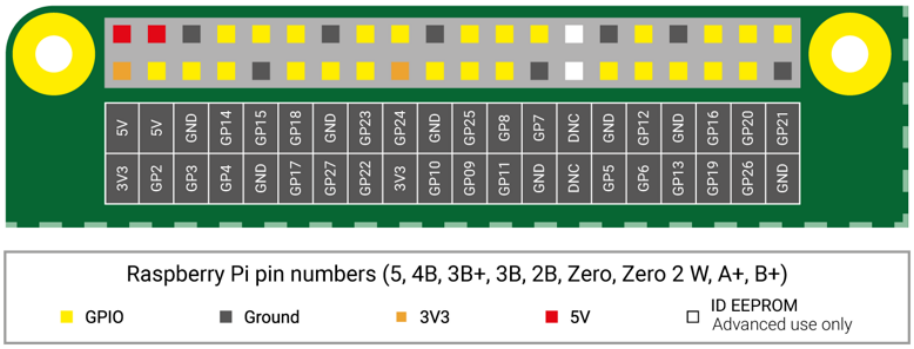


Figure 1-2: Raspberry Pi GPIO pinout

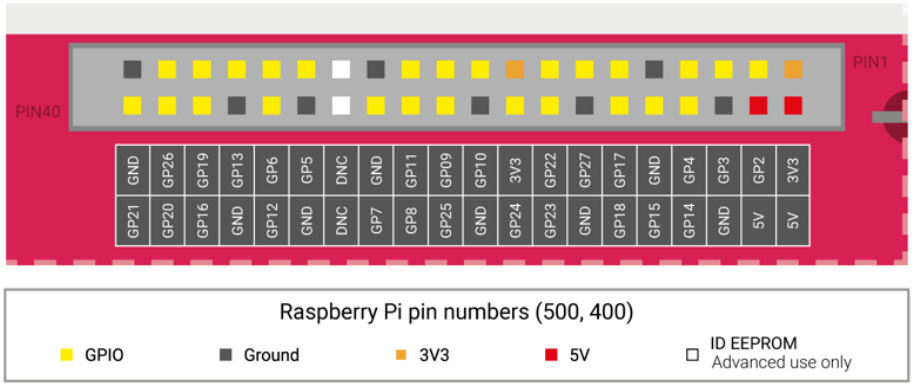


Figure 1-3: Raspberry Pi 400 and 500 GPIO pinout

Raspberry Pi and electronic circuits

Involving a Raspberry Pi computer in such a circuit is quite easy. At its most basic, it can provide power to a circuit, as well as a ground (abbreviated as GND) end through the GPIO pins. Some pins are always powered, mostly at 3.3V, and several pins offer a ground connection. Most pins can be programmed to create or recognise a HIGH or LOW signal, though; in the case of the Raspberry Pi, a HIGH signal is 3.3V and a LOW signal is ground or 0V.

STAYING GROUNDED

You'll sometimes see ground referred to as *negative*, particularly in descriptions of a battery's positive and negative terminal, but also as the minus symbol (–) on a breadboard and some

components. In the kind of circuits you'll see in this book, you'll be working with 5 volts, 3.3 volts, and a 0 volts (ground).

In an LED circuit, you can wire up an LED to a 3.3V pin and a ground pin and it will turn on, but you will need a low-value resistor (around 330Ω is good) in there somewhere to keep from burning out the LED. If you instead put the positive end of the LED onto a programmable GPIO pin, you can only turn it on by running some code that makes that pin go to HIGH. See *Chapter 2, Control LEDs with GPIO Zero* for more details on controlling LEDs.

Wiring up a circuit to a Raspberry Pi computer is simple. To create the physical circuits in the guides throughout this book, we're using prototyping breadboards, as shown in Figure 1-4. These allow you to insert components and wires to connect them all together, without having to fix them permanently. You can modify your circuits and completely reuse your components because of this.

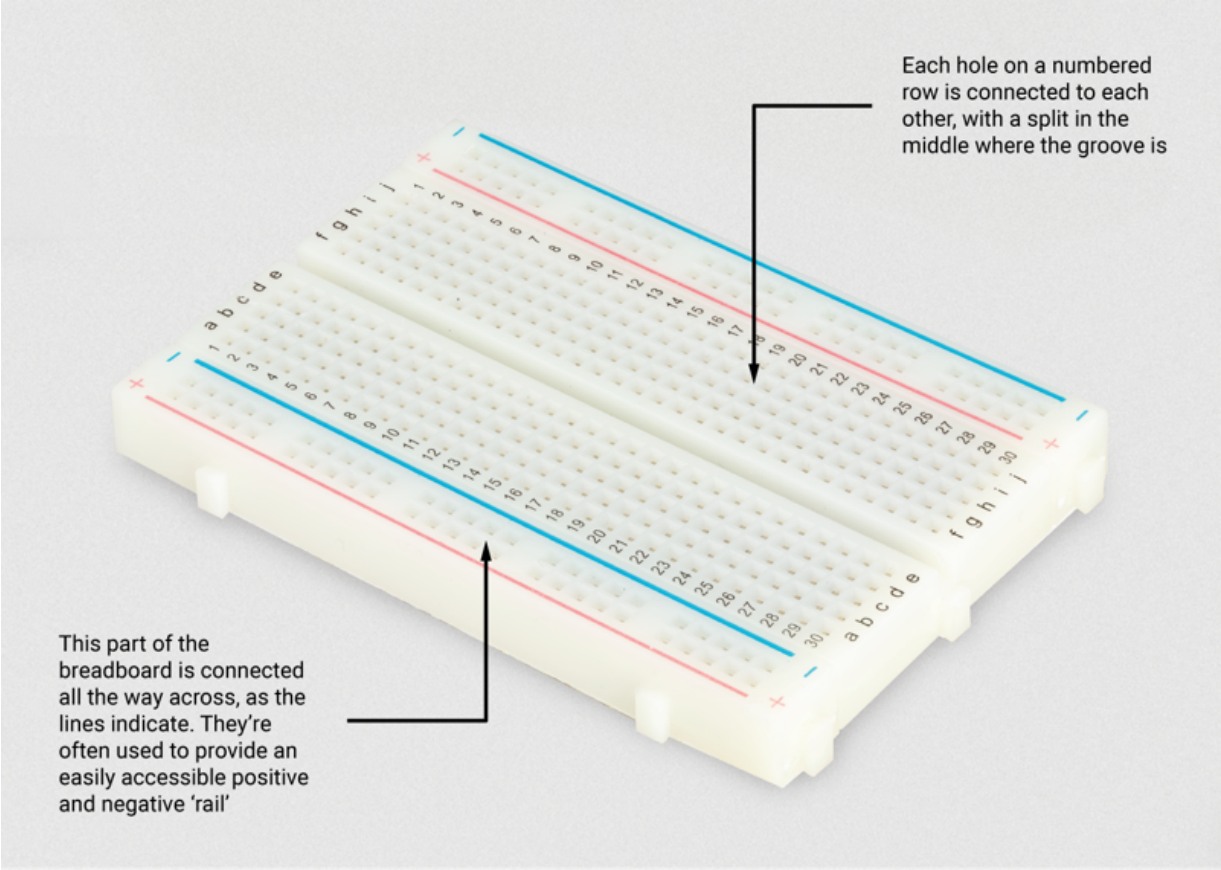


Figure 1-4: A prototyping breadboard

Using GPIO Zero

Once the components are all hooked up to your Raspberry Pi, you need to be able to control them. Raspberry Pi is set up to allow you to program it with the Python language. GPIO Zero makes it easy to program components in Python. It comes pre-installed in the latest Raspberry Pi OS desktop image. If you don't have it, however, you can install GPIO Zero manually: after performing a package list update by entering `sudo apt update` in a terminal, run `sudo apt install python3-gpiozero`.

GPIO Zero was created to simplify the process of physical computing, helping new coders to learn. It's a Python library which builds upon the existing GPIO libraries RPi.GPIO and pigpio. However, while those libraries provide an interface to the GPIO pins themselves, GPIO Zero sits above them and provides a way to interface to the devices that you connect to those pins.

This simplifies thinking about physical computing. Consider wiring a simple push button to GPIO 4 and ground pins. In order to react to this button, we need to know that the pin should be configured with a pull-up resistor, and that the pin state when the button is pushed will be LOW. Here's what this would look like in the classic RPi.GPIO library:

```
from RPi import GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(4, GPIO.IN, GPIO.PUD_UP)
GPIO.wait_for_edge(4, GPIO.FALLING)
print("Button pressed")
```

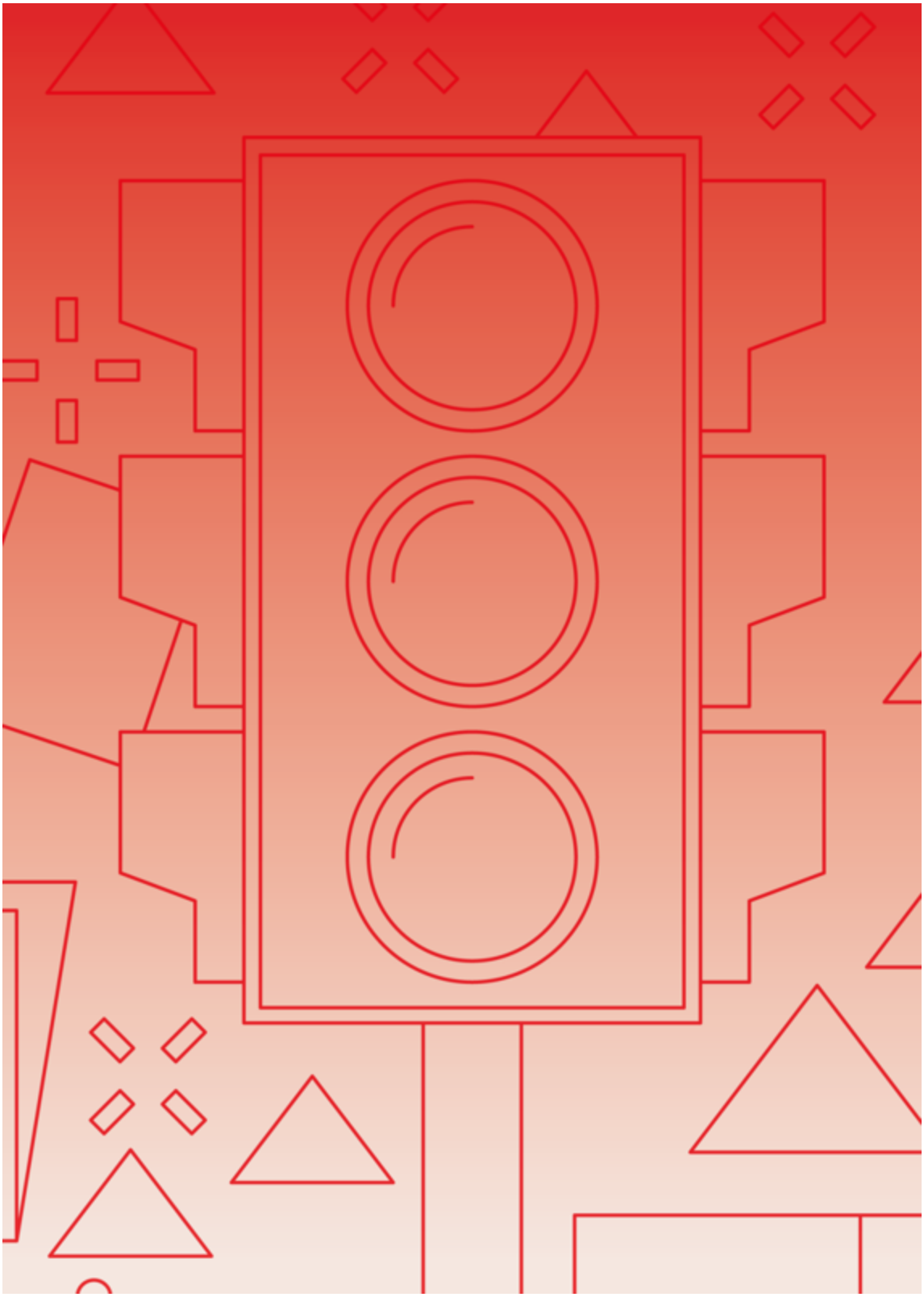
To complete beginners, there's quite a lot going on there, which gets in the way of experimenting with it and even learning the simple logic required. Here's the equivalent code in GPIO Zero:

```
from gpiozero import Button

btn = Button(4)
btn.wait_for_press()
print("Button pressed")
```

The *boilerplate*, the setup code you must write without necessarily understanding its inner workings, is reduced to the bare minimum that we need. The name 'GPIO Zero' derives from this 'zero boilerplate' philosophy, which was first espoused by Daniel Pope's Pygame Zero library.

The logic is straightforward, with no curious inversion of the input value.

So, now you've learnt about GPIO Zero and how it makes coding much simpler, it's time to get started doing some physical computing with it. In the next chapter, we'll show you how to wire up some LEDs on a breadboard and control them using GPIO Zero's `LED` class.



Chapter 2

Control LEDs with GPIO Zero

Turn LEDs on and off with just a few lines of code, and build a traffic light system

One of the first physical computing projects you'll want to try with GPIO Zero is lighting an LED. This is very simple with the library's `LED` class and uses few lines of code. Here we'll show you how to wire up a simple circuit connected to your Raspberry Pi's GPIO pins, then light an LED and make it blink on and off. We'll then add two more LEDs to make a traffic light system.

You'll Need:

- 1× solderless breadboard
- 3× LEDs (one each of red, yellow, and green)
- 3× 330Ω resistors
- 4× pin-to-socket jumper wires

Connect an LED

Before building a circuit, you must shut down your Raspberry Pi and disconnect it from power.

Most breadboards feature numbered columns, each containing one group of five holes, a groove, and another group of five holes. Within each group, all five holes are connected, but the holes on either side of the groove are not connected. If you're holding your breadboard in a different orientation than shown in Figure 2-1, simply swap "columns" for "rows" as you read this section.

Place your red LED's legs in adjacent numbered columns, as shown in the figure. Note that the shorter leg of the LED is the negative end. Next, insert one end of the resistor into one of the other four holes within the same column and group, then place the other end in the outer row marked – (the ground rail). Use a pin-to-socket jumper wire to connect any hole in that ground rail to a GND pin on the GPIO header. Despite the gaps in the rails, all the holes in a rail are usually connected to each other. However, some larger breadboards have discrete groups of holes amongst the rails.

Finally, use a jumper wire to connect a hole within the same column and group of the LED's longer (positive) leg to GPIO 25. Figure 2-2 shows the assembled circuit.

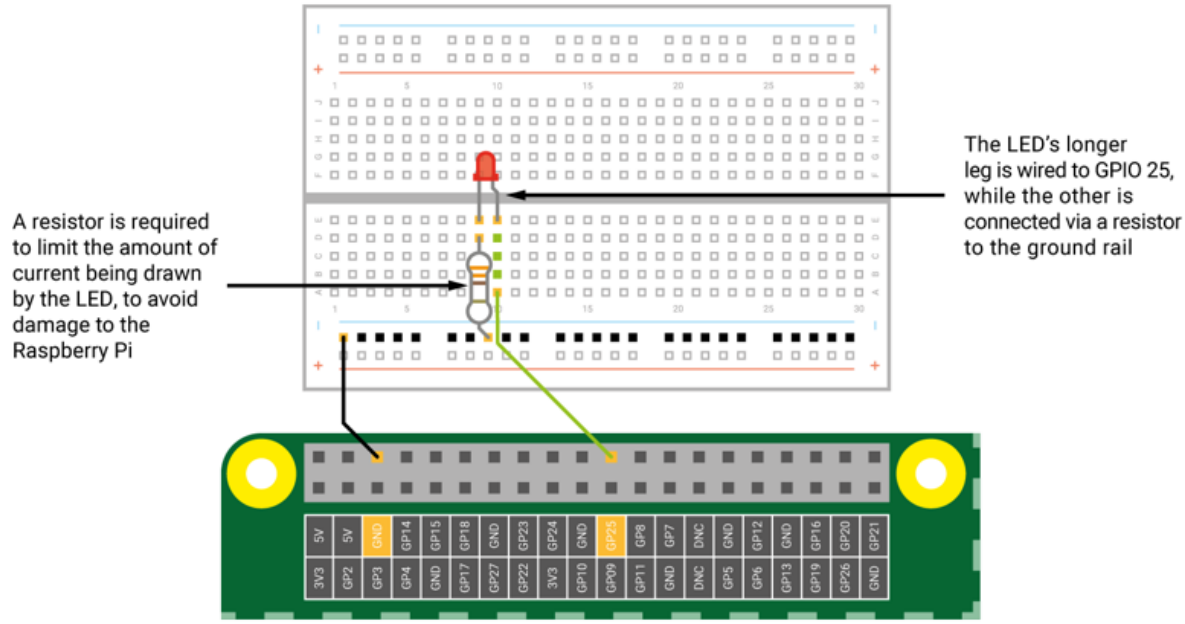


Figure 2-1: Wiring up an LED

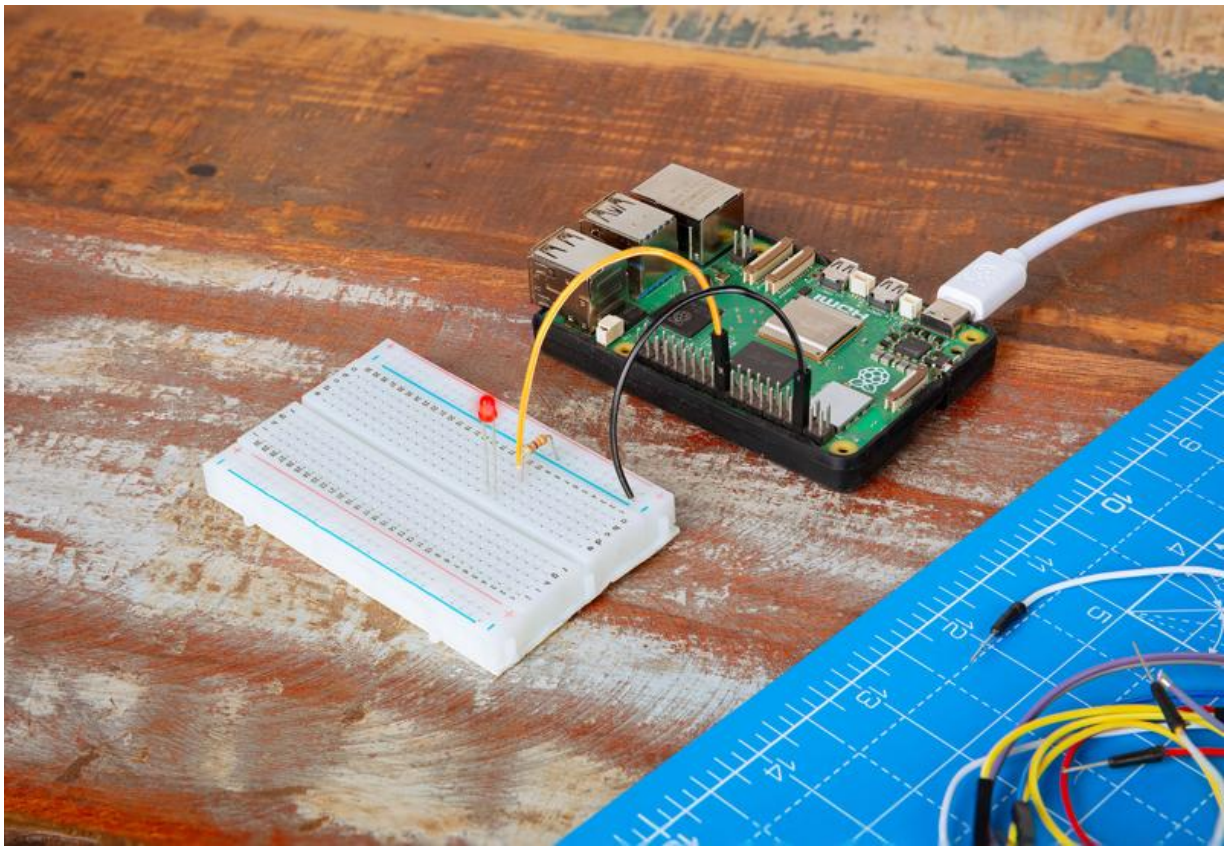


Figure 2-2: While it's possible to connect an LED and resistor directly to the Raspberry Pi, it's better to use a solderless breadboard

Light the LED

Let's test our circuit with a simple Python program to turn the LED on and off. To edit your Python code, you can use any plain text editor, including console-based editors such as nano, emacs, or vi. If you prefer a more full-featured development environment, try Thonny: click the Raspberry Pi menu, then choose Programming > Thonny. Create a new file by clicking New (the green Plus icon). Next, enter the following code and save it by clicking the Save icon, naming it blink_led.py.

To use a console-based editor such as nano, open a Terminal window or SSH into your Raspberry Pi. Enter the following code and save it (in nano, press CTRL+O, type the filename, and press ENTER).

```
from gpiozero import LED
```

```
from time import sleep
led = LED(25)
while True:
  led.on()
  sleep(1)
  led.off()
  sleep(1)
```

At the start of the program, we import the `LED` class from GPIO Zero, and the `sleep` function from the `time` library (to enable us to pause between turning the LED on and off). We then set the `led` variable to the GPIO 25 pin, which will power it whenever we set it to on in the code. Finally, we use `while True:` to create a never-ending loop that switches the LED on and off, pausing for 1 second between each change.

In Thonny, press F5 or click the Run icon to run the code, and your LED should be flashing on and off. To stop the program, click the Stop icon. If you're using a console-based editor, exit it (in nano, use CTRL+X), then run the code with the command `python blink_led.py`. To stop the program, type CTRL+C.

Easier blinking

Alternatively, to make things even easier, GPIO Zero features a special `blink` method. You could try this code, which does exactly the same thing as the first listing, but with even fewer lines of code:

```
from gpiozero import LED
from signal import pause
red = LED(25)
red.blink()
pause()
```

Note that between the brackets for `led.blink`, you can add parameters to set the on and off times, number of blinks, and determine whether it runs as a background thread or not, as in:

```
red.blink(on_time=1, off_time=2, n=3, background=True)
```

Blink multiple LEDs

Now that we've got the hang of controlling one LED, let's use three LEDs to create a traffic light sequence. You can add the optional push button and a buzzer if you like, but in this chapter, we'll only explain the LEDs.

Connect them as shown in the diagram, with the longer (positive) legs connected to a resistor that straddles the groove in the breadboard. Use jumper wires to connect the side of the resistor that's across the groove to the following GPIOs: 25 (red), 8 (yellow), and 7 (green).

Note that the placement of the resistor is different than the previous circuit: instead of putting the resistor between the LED and ground, we're putting it between the LED and the positive voltage supplied by the GPIO pins. It doesn't matter which way you do it, but the resistor must be part of the circuit.

You'll also need to connect each LED's shorter (negative, or ground) leg to the – rail. Like before, the – rail must be connected to one of the GPIO GND pins. Figure 2-3 shows the circuit diagram, and Figure 2-4 shows the assembled project.

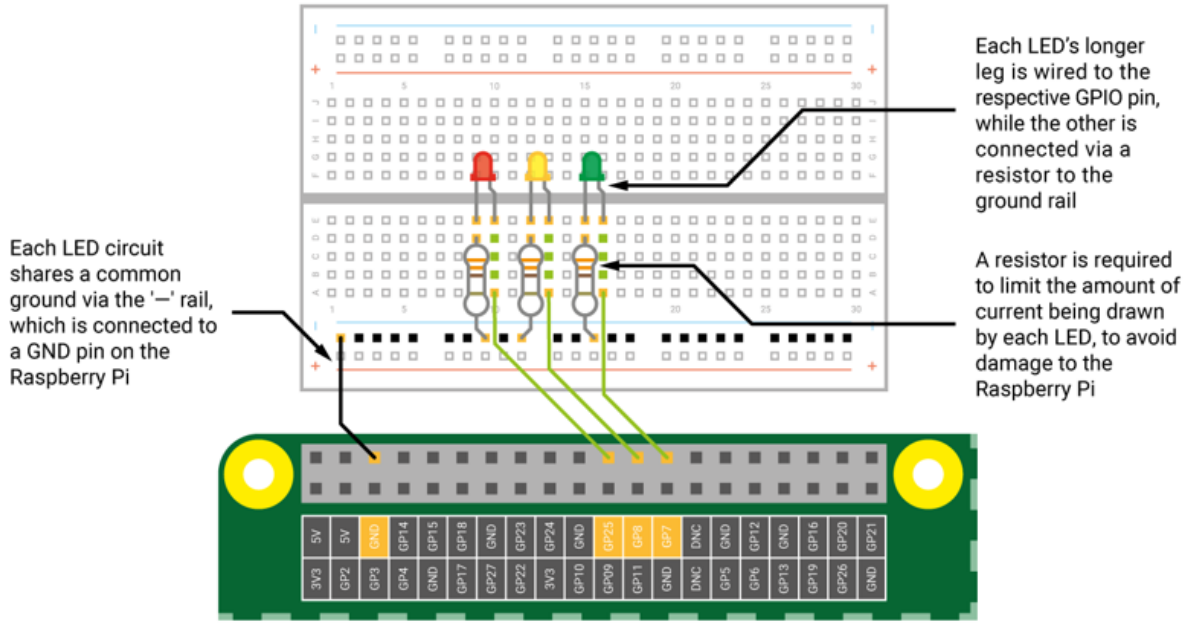


Figure 2-3: Wiring up a traffic signal

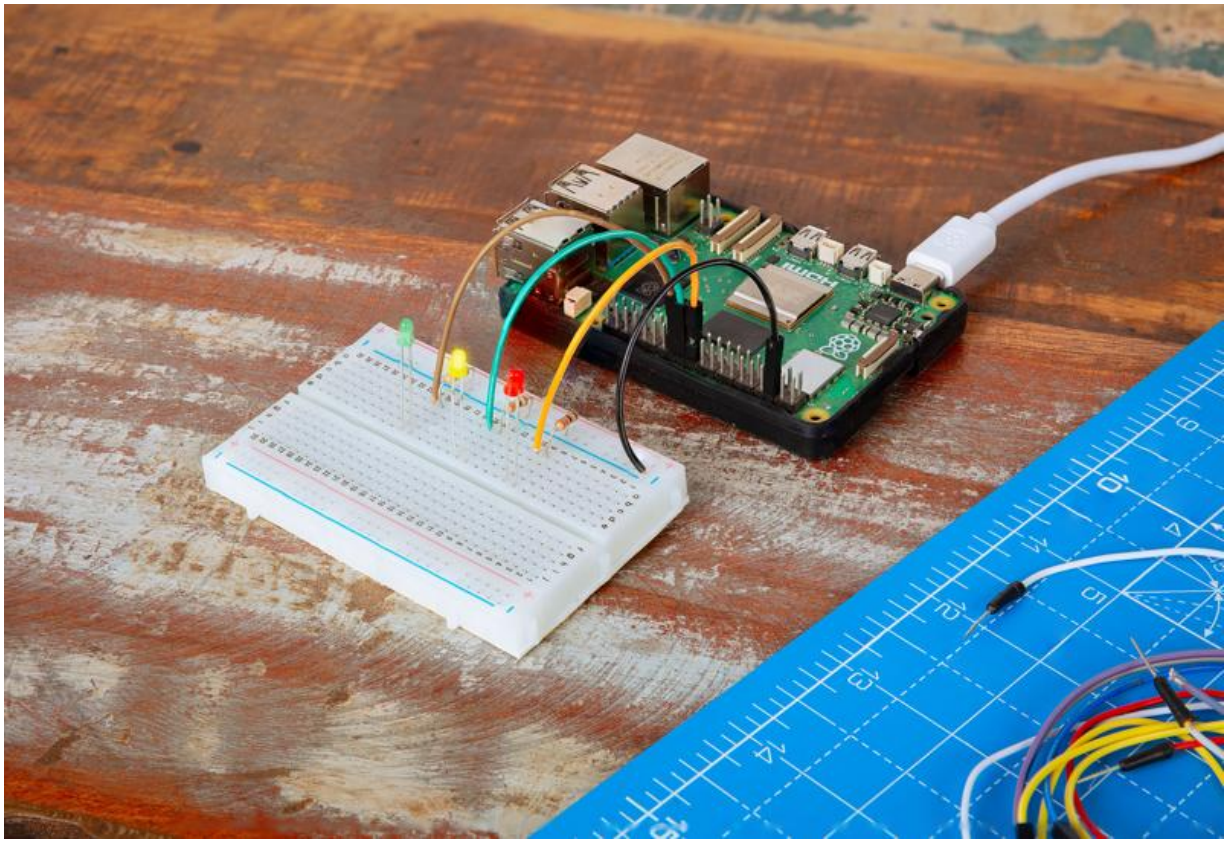


Figure 2-4: The assembled traffic signal

Enter the code

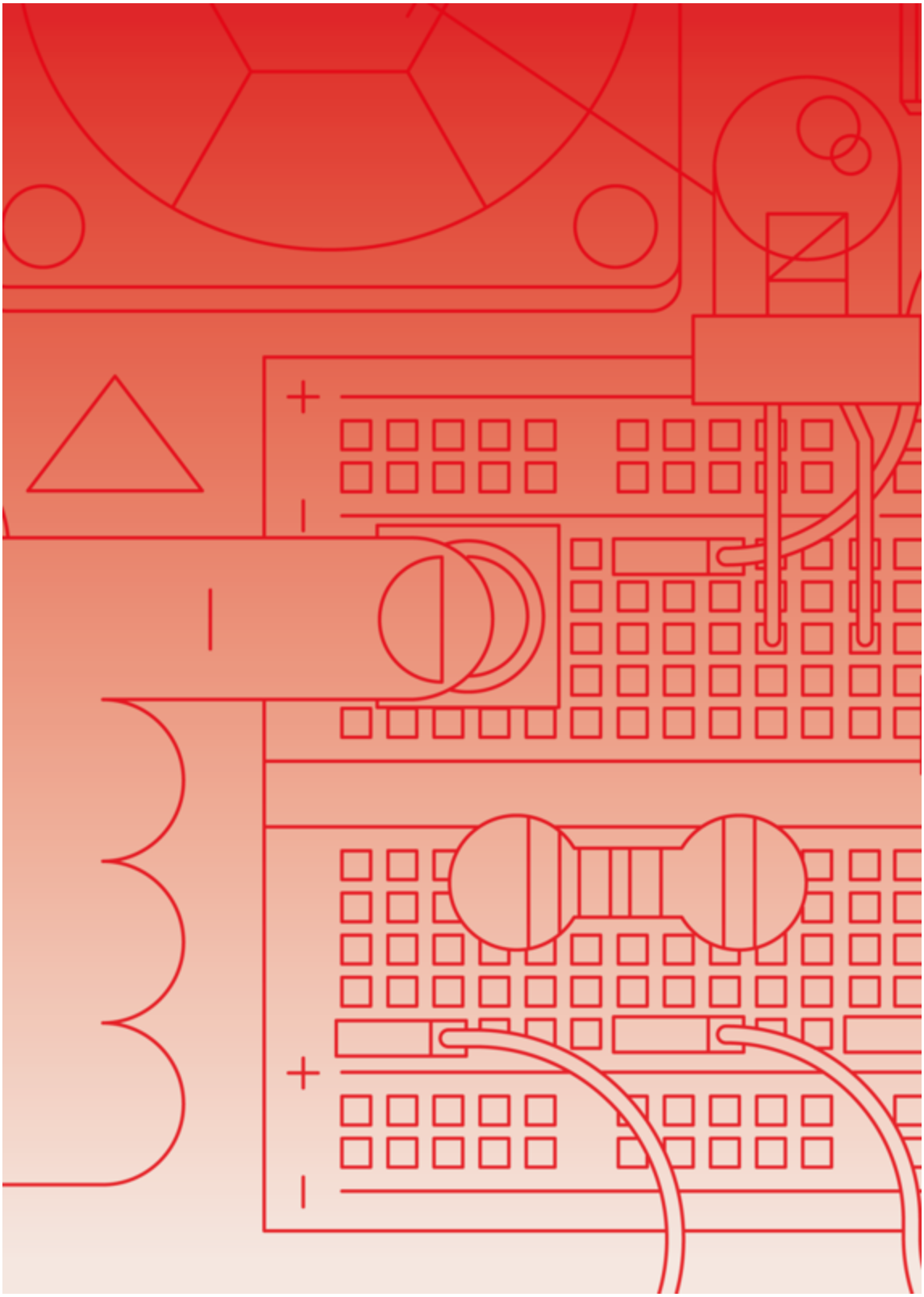
Open your code editor, type in the following code, then save the file as traffic_signal.py:

```
from gpiozero import LED
from time import sleep

red = LED(25)
amber = LED(8)
green = LED(7)

green.on()
amber.off()
red.off()
while True:
  sleep(10)
  green.off()
  amber.on()
  sleep(1)
  amber.off()
  red.on()
  sleep(10)
  amber.on()
  sleep(1)
  green.on()
  amber.off()
  red.off()
```

As before, we import the `LED` class and `sleep` function from GPIO Zero and the `time` library respectively. We then assign `red`, `amber`, and `green` variables to the relevant GPIO pins. To start with, we turn the green LED on and the others off. Finally, we use `while True:` for a never-ending loop; this waits 10 seconds before showing amber then red, then waits another 10 seconds before showing red/amber then green. If you're in Thonny, press F5 or click the Run icon to run the program and wait for the traffic light sequence to start. If

you're using the command line, exit your editor and run the code with `python traffic_signal.py`.

Rather than running through a fixed sequence, you could request that the signal stop traffic with the addition of a push button (see *Chapter 3, User input with a push button*) and a buzzer (see *Chapter 6, Make a motion-sensing alarm*) to alert pedestrians when it's safe to cross. In both cases, you'd need to write some code to work with those components.



Chapter 3

User input with a push button

Make things happen at the press of a button, and create a fun two-player reaction game

Raspberry Pi's GPIO pins aren't just for outputs like LEDs — they can connect to inputs, too. Consider the simple push button, which can be used to trigger other components or functions. Let's connect a button and write a program to print a message when it's pushed. After that, we'll use it to light an LED, then add a second button for a fun two-player reaction game.

You'll Need:

- 1× solderless breadboard
- 2× push buttons
- 1× LED
- 1× 330Ω resistor
- 4× pin-to-socket jumper wires
- 2× pin-to-pin jumper wires

Connect the button

Shut down your Raspberry Pi and unplug it from power before building your circuit. Connect the LED as shown in Figure 3-1 (see "Connect an LED" for more details). Add the push button to the breadboard as shown in the diagram, with its pins straddling the central groove. Connect a pin-to-socket jumper wire from one pin's column to GPIO 21 on the GPIO header. Then

connect a pin-to-pin jumper wire from the other column (on either side of the groove) to the – (ground) rail. Finally, connect a pin-to-socket jumper wire from the ground rail to a GND pin on the GPIO header.

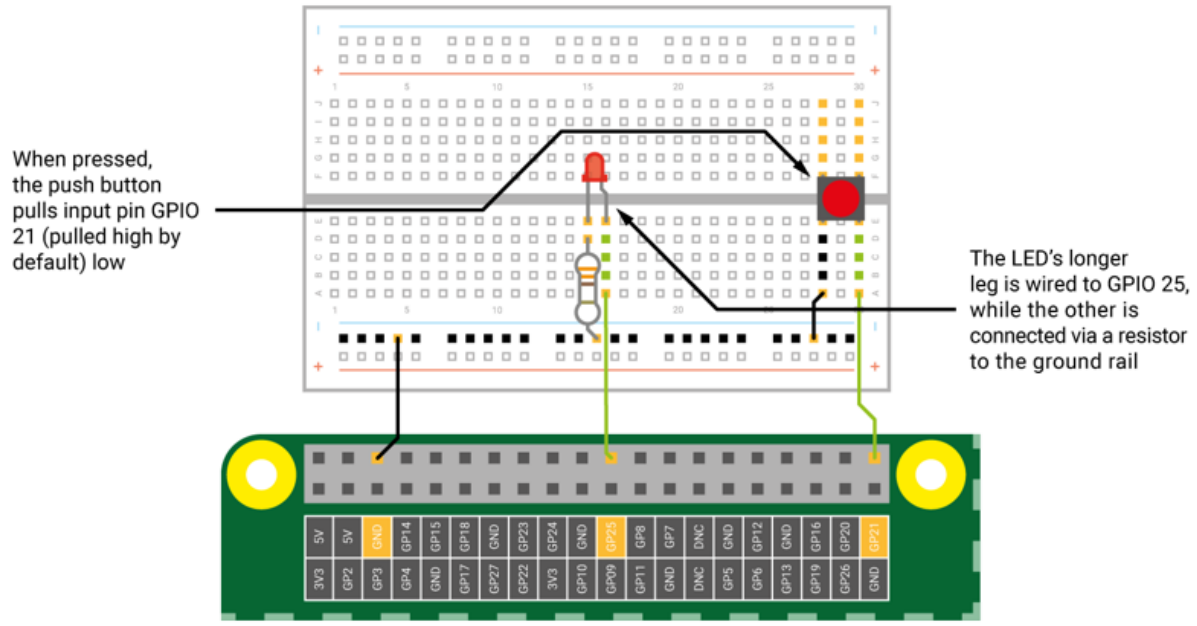


Figure 3-1: Wiring up a button

Button pushed

We'll now test our circuit with a simple Python program to show a message on the screen whenever the button is pushed. Create a new file with the following code, then save it as button.py (see "Light the LED" for an overview of editing and running code):

```
from gpiozero import Button
button = Button(21)
while True:
  if button.is_pressed:
    print("Button is pressed")
  else:
    print("Button is not pressed")
```

At the start of this short program, we import the `Button` class from GPIO Zero. We then set the `button` variable to the GPIO 21 pin, so we can read its value. Finally, we use `while True:` to create a never-ending loop that checks whether the button has been pressed or not and prints a status message on the screen. When you run the code, you'll get a scrolling list of messages that change when you press the button. To stop the program, press CTRL+C. Figure 3-2 shows the assembled project on a breadboard.

You can also trigger a Python function when the button is pressed, as shown in the following example. Note that we don't use parentheses when assigning the function name to the `when_pressed` event. Unlike the previous example, this will only print text when you press the button. Save this as button_func.py and run it. You can exit it with CTRL+C.

```
from gpiozero import Button
from signal import pause
button = Button(21)

def button_pressed():
  print("Button was pressed")

button.when_pressed = button_pressed
pause()
```

Wait for it

GPIO Zero's `Button` class also includes a `wait_for_press` method which pauses the script until the button is pressed. Create a new file, enter the following code and save it as button_wait.py. This will only print the message at the bottom on the screen once the button has been pressed, and the program will exit immediately after you press the button.

```
from gpiozero import Button
button = Button(21)
button.wait_for_press()
print("Button was pressed")
```

Light an LED

Let's try using the button to control the LED. Create a new file, enter the following code, and save it as button_led.py.

```
from gpiozero import LED, Button
from signal import pause

led = LED(25)
button = Button(21)

button.when_pressed = led.on
button.when_released = led.off
pause()
```

At the top, we import the `LED` and `Button` classes from GPIO Zero, along with the `pause` function from `signal`. We then allocate variables to the LED and button on GPIOs 25 and 21 respectively. When the button is pressed, the LED is turned on; when released, it's turned off.

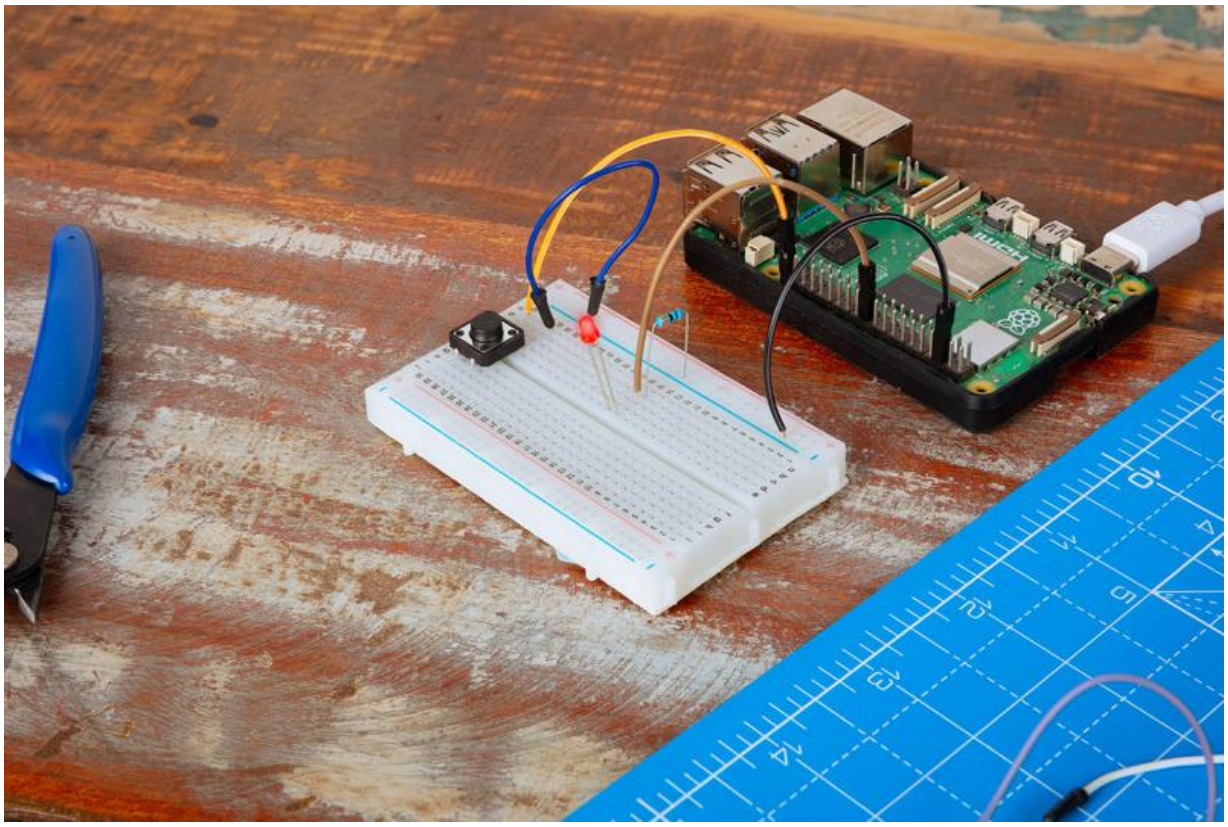


Figure 3-2: When the button is pressed, GPIO 21 registers the LOW signal and our program turns the LED on

It's also possible to keep the LED lit for a set period after pressing. Open a new file, enter the following code and save it as button_off_delay.py. This time, we trigger a function when the button is pressed as before, but we also set up another function that's triggered when the button is released — it waits three seconds before turning the LED off. As with other programs that use a `pause` or infinite `while` loop, you'll need to stop the program with CTRL+C to exit it.

```
from gpiozero import LED, Button
from time import sleep
from signal import pause

led = LED(25)
button = Button(21)
```

```
def button_released():
    sleep(3)
    led.off()

button.when_pressed = led.on
button.when_released = button_released
pause()
```

Reaction game

If you add a second push button to the circuit, you can make a simple two-player reaction game. Shut down your Raspberry Pi and disconnect it from power. Add a second button as shown in Figure 3-3, connecting it to the ground rail and GPIO 2; move the LED and its connections to the middle, if not there already.

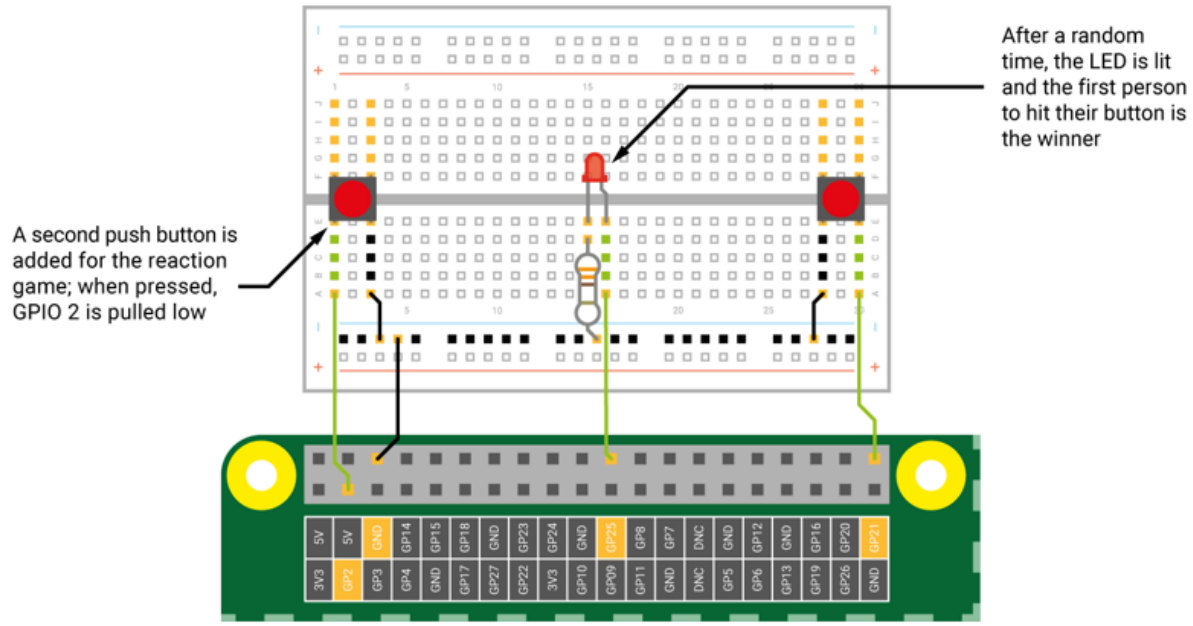


Figure 3-3: There are now two buttons and an LED

Now you're ready to plug your Raspberry Pi in and boot it back up. Create a new file, enter the following code, and save it as reaction_game.py:

```
from gpiozero import Button, LED
from time import sleep
import random
led = LED(25)
player_1 = Button(21)
player_2 = Button(2)
time = random.uniform(5, 10)
sleep(time)
led.on()
while True:
  if player_1.is_pressed:
    print("Player 1 wins!")
    break
  if player_2.is_pressed:
    print("Player 2 wins!")
    break
led.off()
```

As before, we import the classes required (along with the `random` module) at the top. We assign variables to the LED and two buttons, then create a `time` variable equal to a random number between 5 and 10; after sleeping for this number of seconds, the LED is turned on, as shown in Figure 3-4. The `while True:` loop is terminated with `break` when someone presses their button, after printing the appropriate victory message.

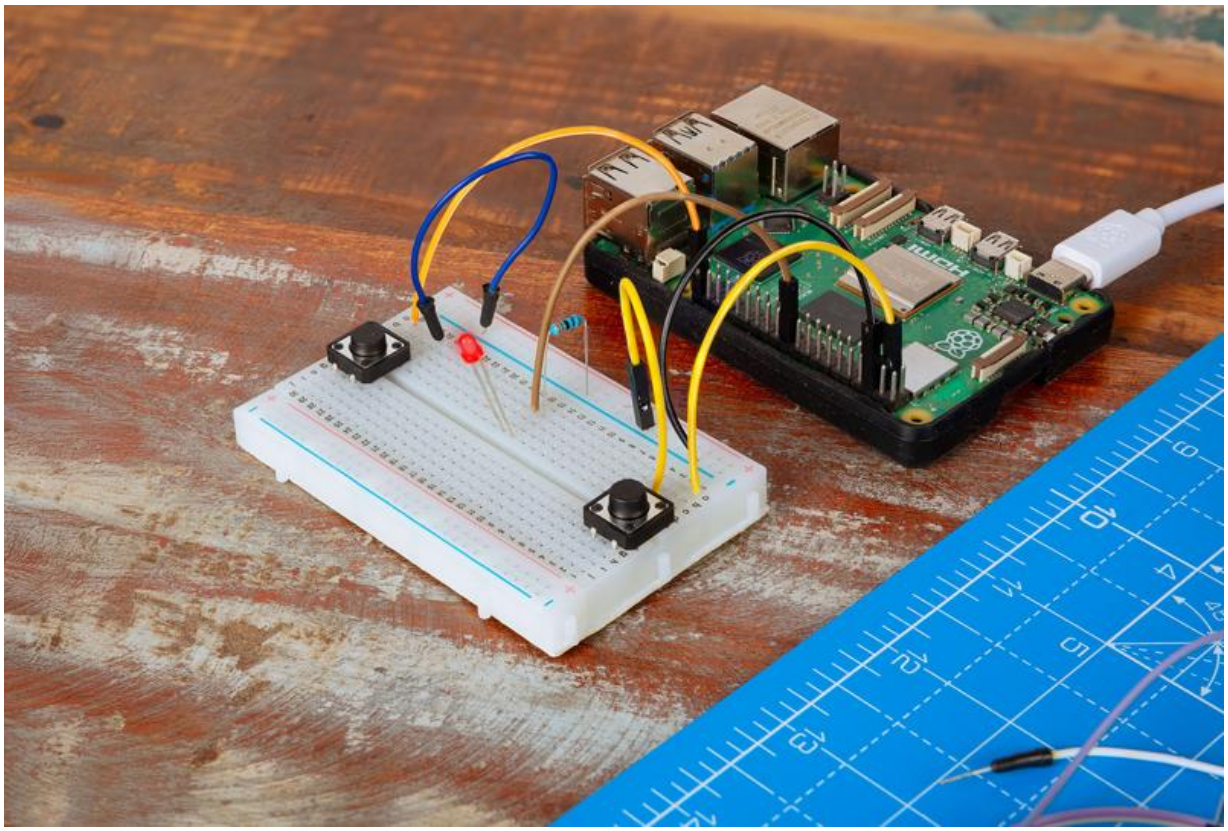
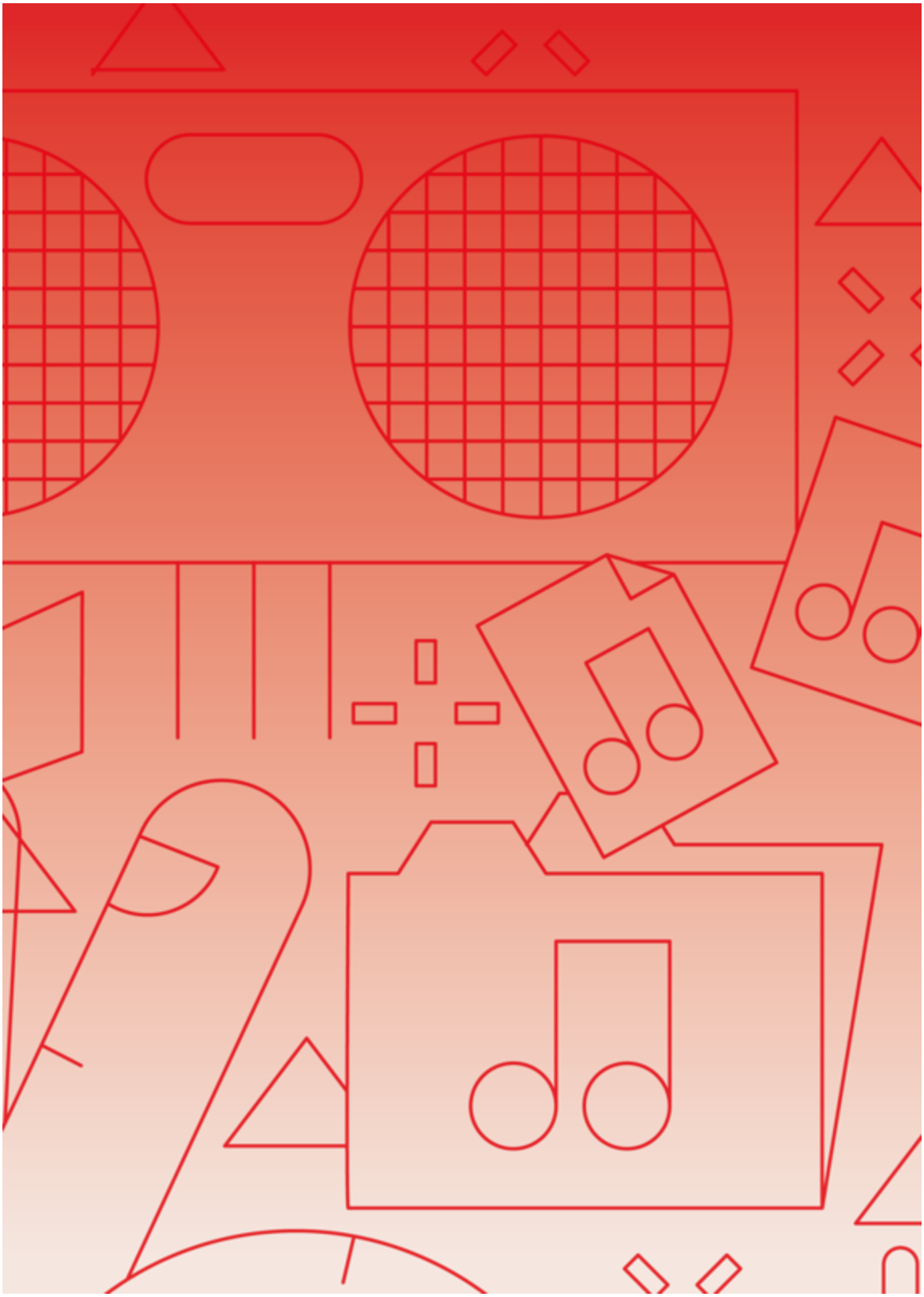


Figure 3-4: The LED is lit! In this reaction game, the first person to now press their button will win



Chapter 4

Make a push button music box

Use two or more tactile push buttons to play different sound samples

So far, we've added a push button to a simple circuit to light an LED and then added a second button to make a reaction game. In this chapter, we'll use several push buttons to make a GPIO music box that triggers different sounds when we press different buttons. For this, we'll make use of GPIO Zero's Button class again, as well as using the Python dictionary structure to assign sounds to buttons.

You'll Need

- 1× solderless breadboard
- 2× push buttons
- 3× pin-to-socket jumper wires
- 2× pin-to-pin jumper wires
- Headphones or speaker

Get some sounds

Before we start building our GPIO music box circuit, we'll need to prepare some sound samples for it to play. The easiest way to do this is to install the samples from Sonic Pi, a live coding environment that lets you create music with code that you write. Helpfully, Sonic Pi includes a large collection of sound samples that it installs in /usr/share/sonic-pi/samples. You can install the samples with `sudo apt install sonic-pi-samples`.

Play a drum

We'll now create a simple Python program to play a drum sample repeatedly, to check everything is working. Create a new file with the following code, then save it as drumbeat.py (see "Light the LED" for an overview of editing and running code):

```
import pygame.mixer
from pygame.mixer import Sound

pygame.mixer.init()
samples = "/usr/share/sonic-pi/samples/"
drum = Sound(f"{samples}/drum_bass_soft.flac")
while True:
  drum.play()
```

At the start of the program, we import Pygame's `mixer` module as well as its `Sound` class, which enables multichannel sound playback in Python. Next, we add a line to initialise the Pygame mixer: `pygame.mixer.init()`. We then create a variable to hold the path to where the samples reside, use Python's *f-string* notation to embed that variable in a string and combine it with the sample's filename (drum_bass_soft.flac), and create a `Sound` object for it.

Finally, we add a `while True:` loop to repeatedly play the drum sound. Run the program and listen to it play. When you get tired of the drumming, press CTRL+C to stop the program.

Wire up a button

As usual, you must turn the Raspberry Pi off while wiring up a circuit on the breadboard. First, we'll add a single button. As before, place the button so it straddles the central groove of the breadboard. One leg is connected to GPIO 2, and the other to the common ground rail on the breadboard, which in turn is wired to a GND pin.

We'll now make a sound play whenever the button is pressed. Open a new file, enter the code below, and save it as play_drum.py (see "Light the LED" for an overview of editing and running code).

```
from gpiozero import Button
import pygame.mixer
from pygame.mixer import Sound
from signal import pause

pygame.mixer.init()
button = Button(2)
samples = "/usr/share/sonic-pi/samples/"
drum = Sound(f"{samples}/drum_bass_soft.flac")

button.when_pressed = drum.play
pause()
```

At the start of the program, we also import the `Button` class from GPIO Zero, and the `pause` class from the `signal` library. We set the button variable to GPIO 2, with `button = Button(2)`. We then tell the sound to play when the button is pressed:

button.when_pressed = drum.play

Finally, we add `pause()` at the end so that the program will continue to wait for the button to be pressed. Run the program and every time you press the button, the drum sound should play.

Add a second button

Add a second button to the circuit — it should now look like the diagram in Figure 4-1. Place it on the breadboard, and wire it up to GPIO 3 and the common ground rail.

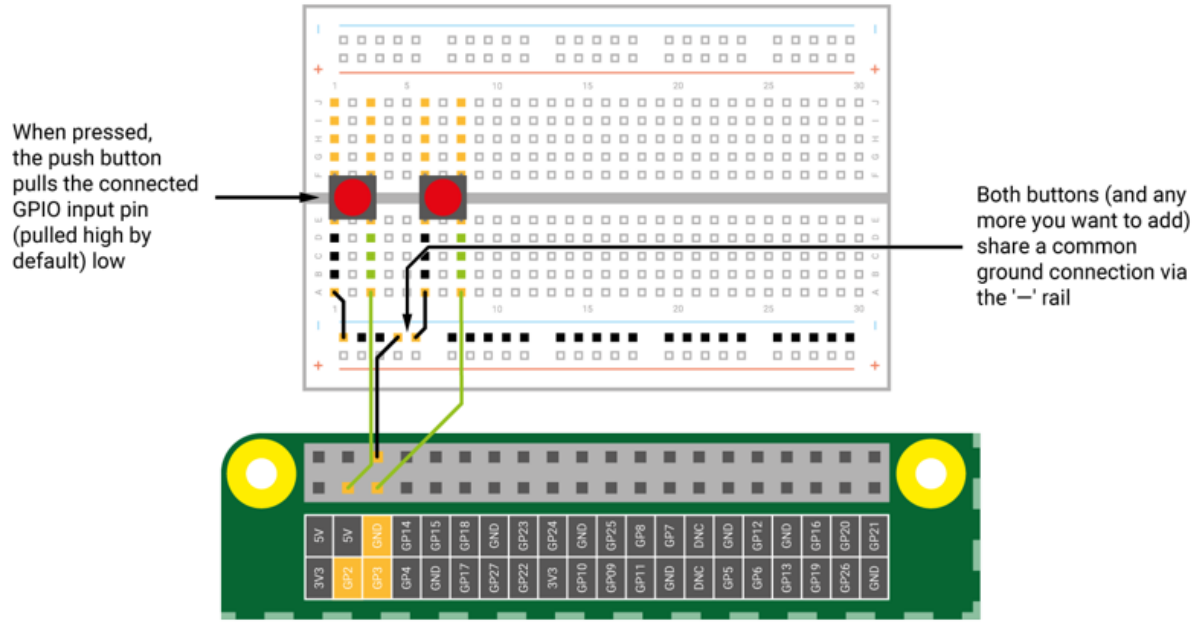


Figure 4-1: Wiring up two buttons

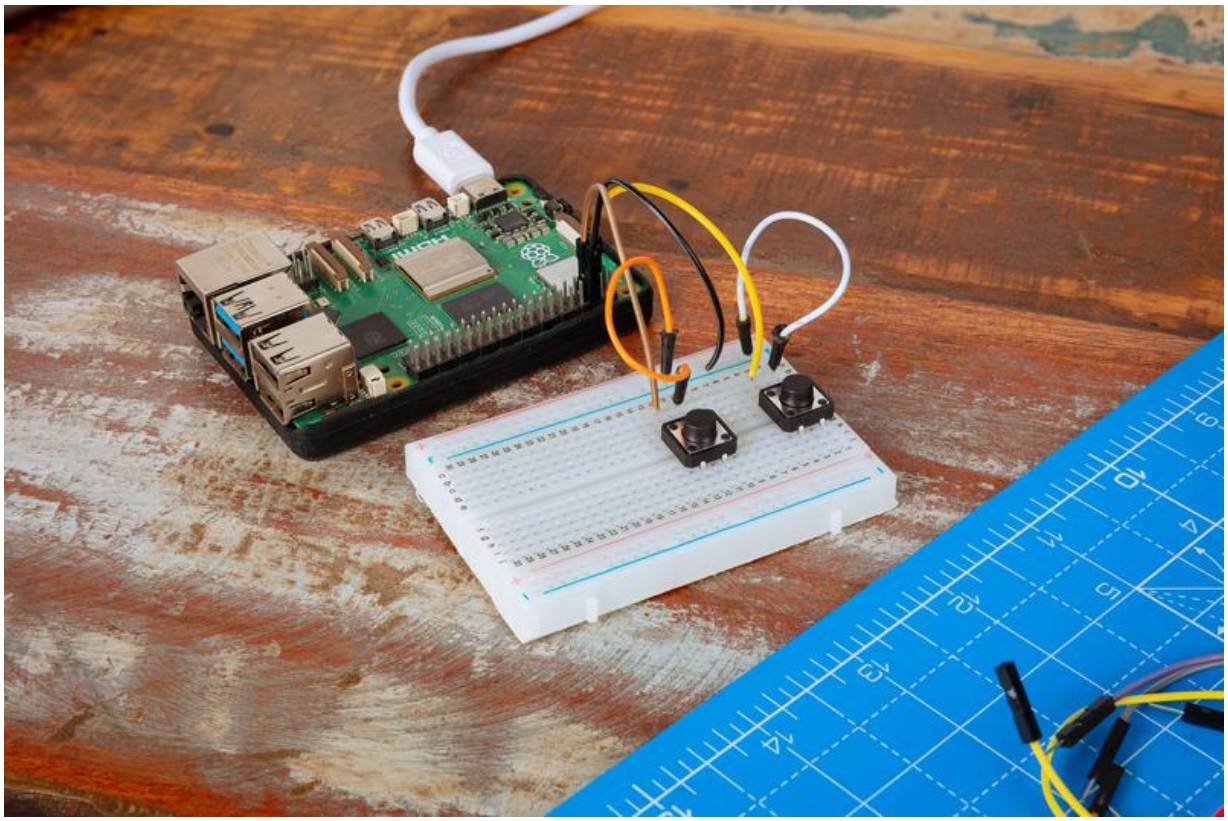


Figure 4-2: Two buttons on a breadboard

Create a new file, type in the code shown below, and save your new program as drum_board.py.

```
from gpiozero import Button
import pygame.mixer
from pygame.mixer import Sound
from signal import pause

pygame.mixer.init()
samples = "/usr/share/sonic-pi/samples/"
sound_pins = {
  2: "drum_bass_soft.flac",
  3: "drum_cowbell.flac",
}

buttons = [Button(pin) for pin in sound_pins]
for button in buttons:
  file = sound_pins[button.pin.number]
  sound = Sound(f"{samples}/{file}")
  button.when_pressed = sound.play

pause()
```

As with the previous example, we import the libraries we need and, initialise the mixer and define a variable called `samples` that contains the path to the directory that holds our samples. We then define a dictionary (`sound_pins`) that maps GPIO numbers (2 and 3) to filenames.

Next, we create a list of Button objects (buttons) for each pin number in the `sound_pins` dictionary. Finally, we create a `for` loop that looks up each button in the dictionary, constructs a new `Sound` object, and configure the button's `when_pressed` event to play the related sound. Run the program and press each button to hear a different sound.

Add more buttons

The way we have structured the program makes it easy to add extra buttons and assign them to sound samples. Just connect each button to a GPIO number pin (not any other type) and the ground rail, as before. Then add the GPIO pin numbers and sound file names to the dictionary, as in the following example. You can see all the samples by running the command `ls /usr/share/sonic-pi/samples/`.

```
sound_pins = {
   2: "drum_bass_soft.flac",
   3: "drum_cowbell.flac",
   4: "tabla_ghe1.flac",
  14: "vinyl_scratch.flac",
}
```

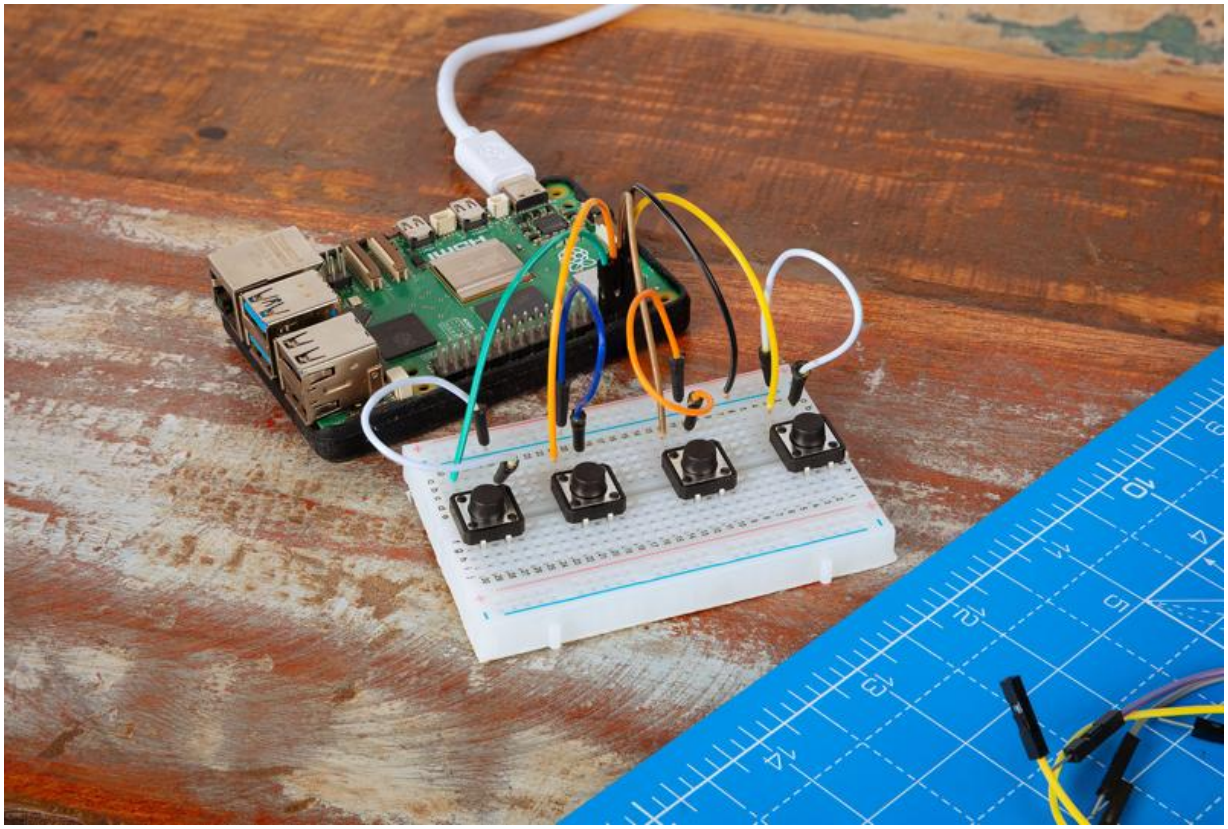
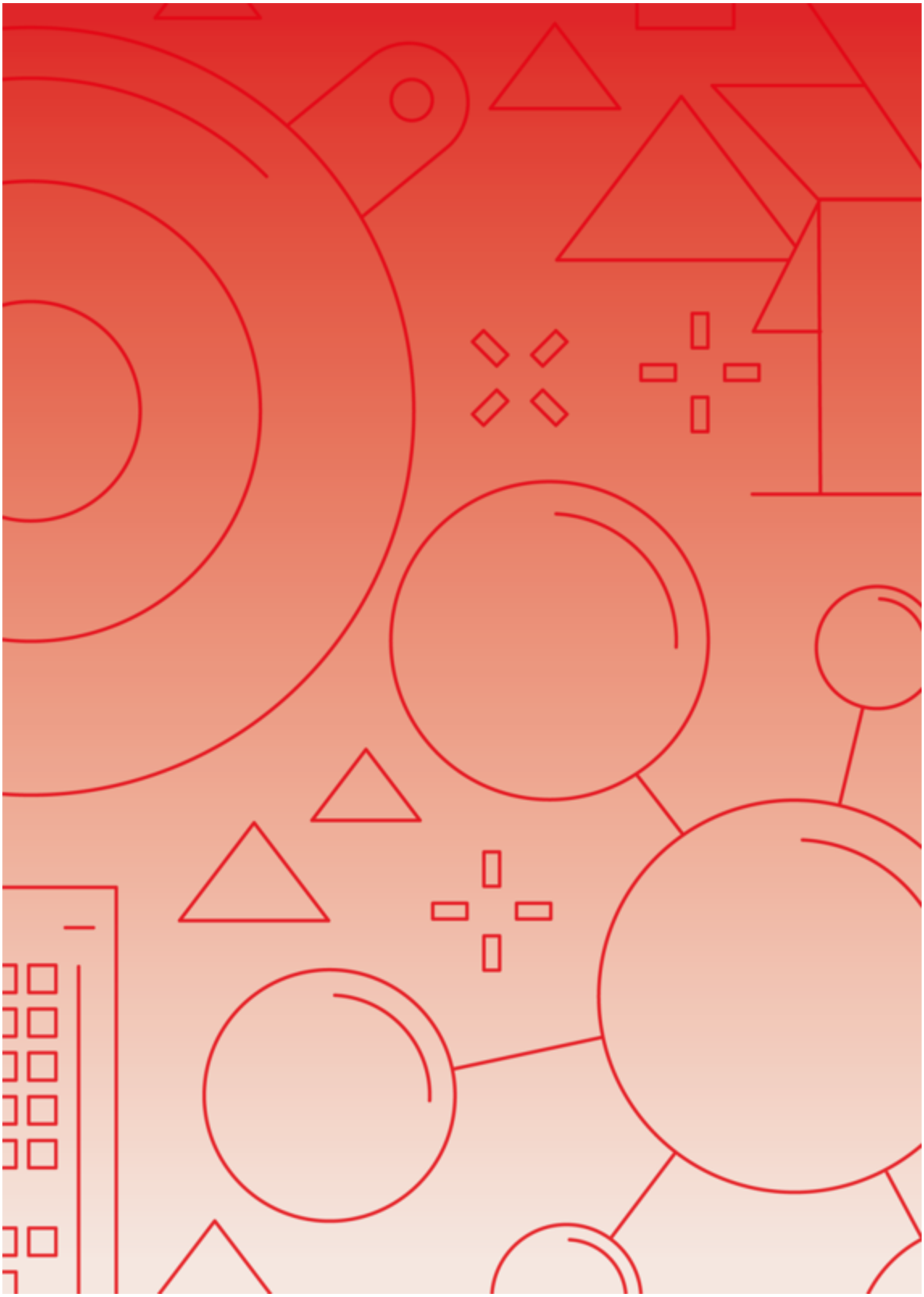


Figure 4-3: Extra buttons can easily be added to the circuit to play more sounds assigned in the Python code



Chapter 5

Measure CPU usage with an RGB LED

Learn how to use an RGB LED and get it to show CPU load

We lit up a standard LED in *Chapter 2, Control LEDs with GPIO Zero*, using the `LED` class. GPIO Zero also offers an `RGBLED` class for controlling — guess what — an RGB LED! In this chapter, we'll make use of this to light up our LED in different shades by altering the red, green, and blue values. Then we'll code up a little program that tracks the Raspberry Pi's CPU usage percentage and adjust the LED between green and red accordingly to show how much processing power it's using.

You'll Need

- 1× solderless breadboard
- 1× RGB LED
- 3× 100Ω resistor
- 4× pin-to-socket jumper wires

Select your RGB LED

Light-emitting diodes (LEDs) are cool. Literally. Unlike a normal incandescent bulb, which has a hot filament, LEDs produce light solely by the movement of electrons in a semiconductor material. An RGB LED has three single-colour LEDs combined in one package. By varying the brightness of each component, you can produce a range of colours, just like mixing paint.

There are two main types of RGB LEDs: common anode and common cathode. We're going to use common cathode for this project.

Connect the RGB LED

As usual, it's best to turn the Raspberry Pi off while connecting our circuit on the breadboard. LEDs need to be connected the correct way round. For a common cathode RGB LED, you have a single ground wire — the longest leg — and three anodes, one for each colour. To drive these from a Raspberry Pi, connect each anode to a GPIO pin via a current-limiting resistor. When one or more of these pins is set to HIGH (3.3V), the LED will light up the corresponding colour. Connect everything as shown in Figure 5-1.

If your RGB LED is common anode, connect the LED's common pin to the 3V3 GPIO pin instead of GND, but leave the other pins wired as shown. You must, however, add `active_high=False` to the call to `RGBLED()`.

Here, we wire the cathode (long leg) to a GND pin, while the other legs are wired via resistors to GPIO 14, 15, and 18 (your red, green, and blue legs may be different than shown here). The resistors limit the amount of current flowing, to avoid damaging your LED or Raspberry Pi; we've used 100Ω, but you could use a slightly higher ohmage, such as 330Ω.

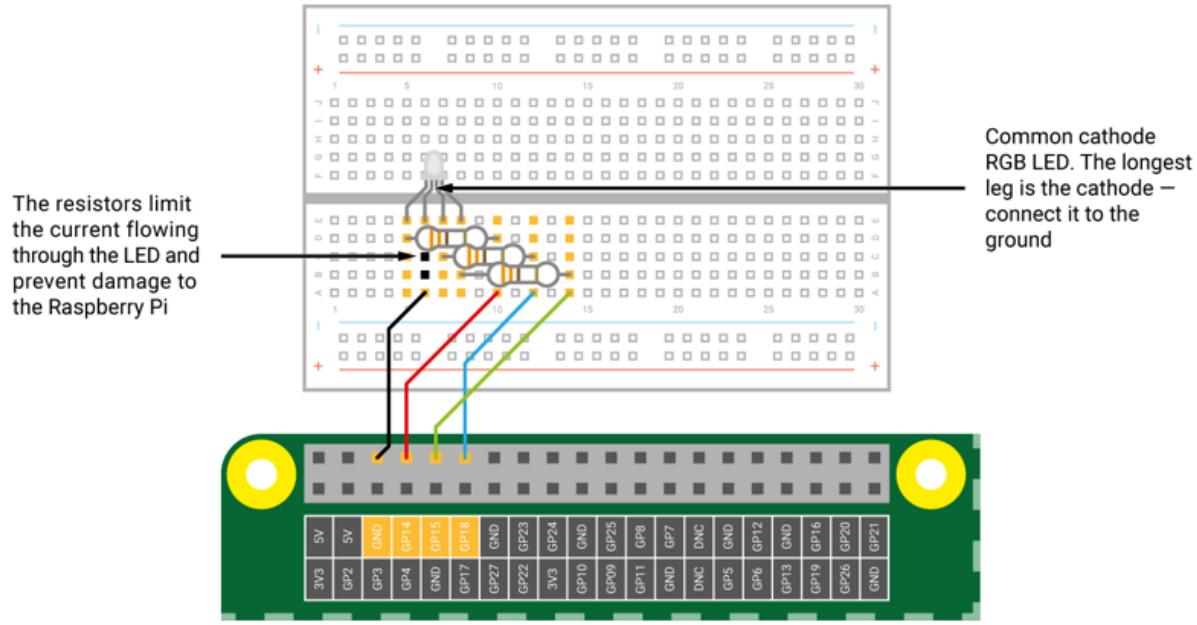


Figure 5-1: Wiring up the RGB LED

Test the LED

With the `RGBLED` class in GPIO Zero, it's easy to alter the colour of the LED by assigning values of between 0 and 1 to red, green, and blue (see Figure 5-2). On the Raspberry Pi, create a new file, enter the following code, and save it as rgb_led.py (see "Light the LED" for an overview of editing and running code).

```
from gpiozero import RGBLED
from time import sleep

led = RGBLED(14, 15, 18)

led.red = 1 # full red
sleep(1)
led.red = 0.5 # half red
sleep(1)
led.color = (0, 1, 0) # full green
sleep(1)
```

```
led.color = (1, 0, 1) # magenta
sleep(1)
led.color = (1, 1, 0) # yellow
sleep(1)
led.color = (0, 1, 1) # cyan
sleep(1)
led.color = (1, 1, 1) # white
sleep(1)
led.color = (0, 0, 0) # off
sleep(1)

# slowly increase intensity of blue
for n in range(100):
  led.blue = n/100
  sleep(0.1)
```

At the top, we import the `RGBLED` class from GPIO Zero, along with the `sleep` function from the `time` library. We then set the variable `led` to the `RGBLED` class on GPIO pins 14, 15, and 18, for red, green, and blue. We then make `led.red` equal to 1 to turn the LED a full red colour. After a second, we then change the value to 0.5 to reduce its brightness.

We then go through a sequence of colours using `led.color`, assigning it a tuple of red, green, and blue values to mix the shades. So, `(1, 0, 1)` shows full red and blue to make magenta. You can vary each value between 0 and 1 to create an almost infinite range of shades. Finally, we use a `for` loop to slowly increase the intensity of blue. Run the program, and it will cycle through its sequence of colours, and then it will exit on its own.

Brightness is an illusion

Truth be told, we're not actually changing the LED's brightness at all. We're using a feature called *pulse-width modulation* or *PWM*. A digital output can only ever be on or off: 0 or 1. Turning a digital output on and off is known as a *pulse* and by altering how quickly the pin turns on and off, you can change,

or *modulate*, the *width* of these pulses — hence 'pulse-width modulation'. By default, GPIO Zero sends 100 pulses per second (100Hz).

The PWM *duty cycle* controls the pin's output: a 0 percent duty cycle leaves the pin switched off for all 100 pulses per second, and effectively turns the pin off; a 100 percent duty cycle leaves the pin switched on for all 100 pulses per second, and is functionally equivalent to just turning the pin on as a fixed digital output; a 50 percent duty cycle has the pin on for half the pulses and off for half the pulses. Although 100Hz is not fast from a computer's perspective, it is fast enough to trick the human eye and create an illusion that the brightness is changing.

If you're using a common anode RGB LED, you must set `active_high` to `False`, which inverts the logic used to illuminate the LED, where the pin is driven HIGH (a 3.3V signal) to turn the LED off and LOW (0V) to turn it on, which is why we asked you to connect the common anode pin to the 3V3 GPIO pin earlier. You'll need to change:

```
led = RGBLED(14, 15, 18)
```

to:

```
led = RGBLED(14, 15, 18, active_high=False)
```

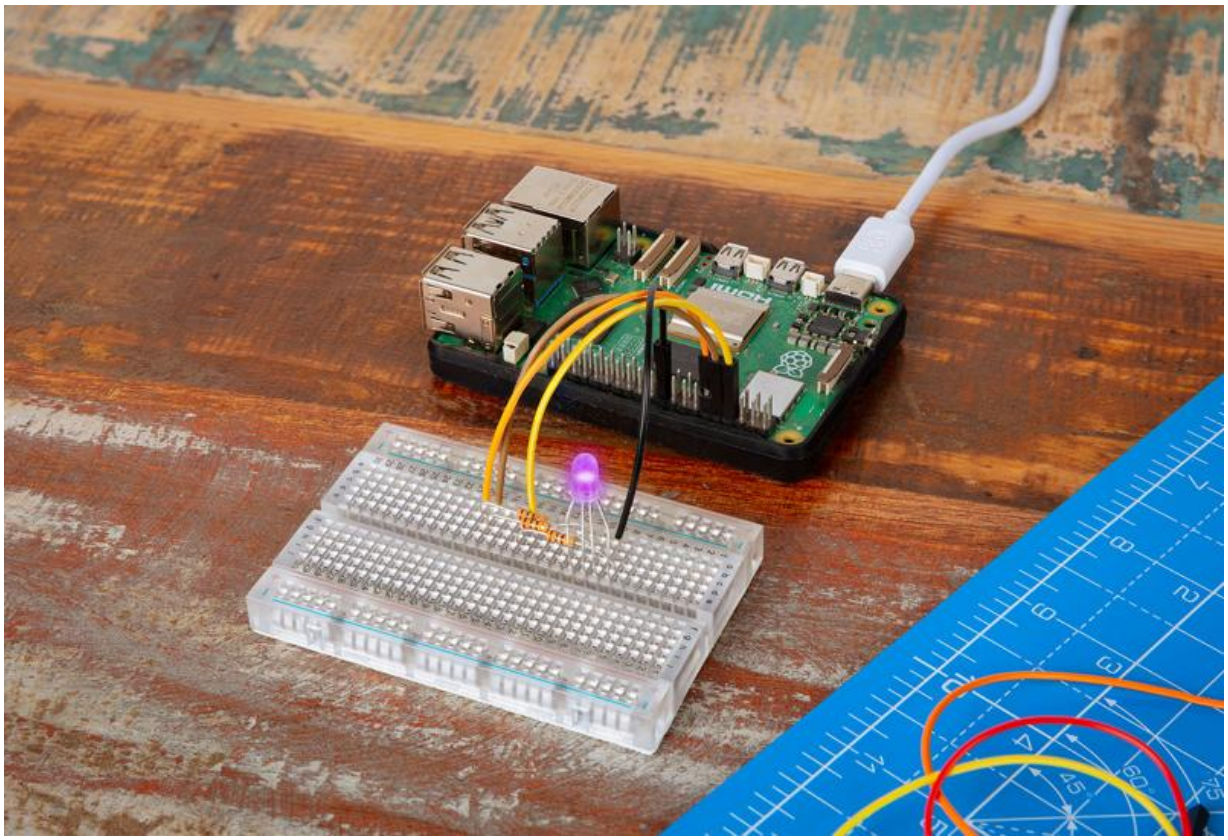


Figure 5-2: By altering the three RGB values, you can light the LED in any shade you like

Add a new library

We now want to get our RGB LED to change colour between green and red, to show the CPU usage of the Raspberry Pi to which it's connected, so we can track how much of its processing power we're using at any time. For this, we'll need the `psutil` Python library. If it's not already installed, you can install it with this command:

```
sudo apt install python3-psutil
```

This will let us look up the CPU usage of the Raspberry Pi as a percentage number, which can then be used in our code to vary the LED's colour.

Python virtual environments

If you want to install a newer version of `psutil` (or any other Python library), you must set up a virtual environment. This lets you install optional libraries without affecting your main Python environment. If you're happy with the version that's available through `apt`, you don't need to follow these instructions. To set up a virtual environment, use the following command to create a virtual environment in the env folder in your home directory:

```
python -m venv --system-site-packages ~/env
```

Next, you can run the following command from any directory to start using the virtual environment (you'll need to run this for each new Terminal window, SSH session, or console login):

```
source ~/env/bin/activate
```

You should then see a prompt like the following:

```
(env) username@hostname:~ $
```

To leave the virtual environment, run the following command from any directory:

```
deactivate
```

You can find instructions for other configurations, such as per-project environments, at rpimag.co/venv. If you don't feel like running `source ~/env/bin/activate` every time you want to use the environment, you could add that line to your `~/.profile.` After you've activated your virtual environment, you can install `psutil` or any other Python module. Make sure you see the (`env`) prompt (if not, run the `source` command shown earlier to load the virtual environment). Run the following command:

```
pip install psutil --upgrade
```

You should periodically update the pip command itself with `pip install pip --upgrade`.

Measure CPU usage

Create a new file, enter the following code, and save it as cpu_usage.py. As with the previous example, if you're using a common anode RGB LED, you must set `active_high` to `False`.

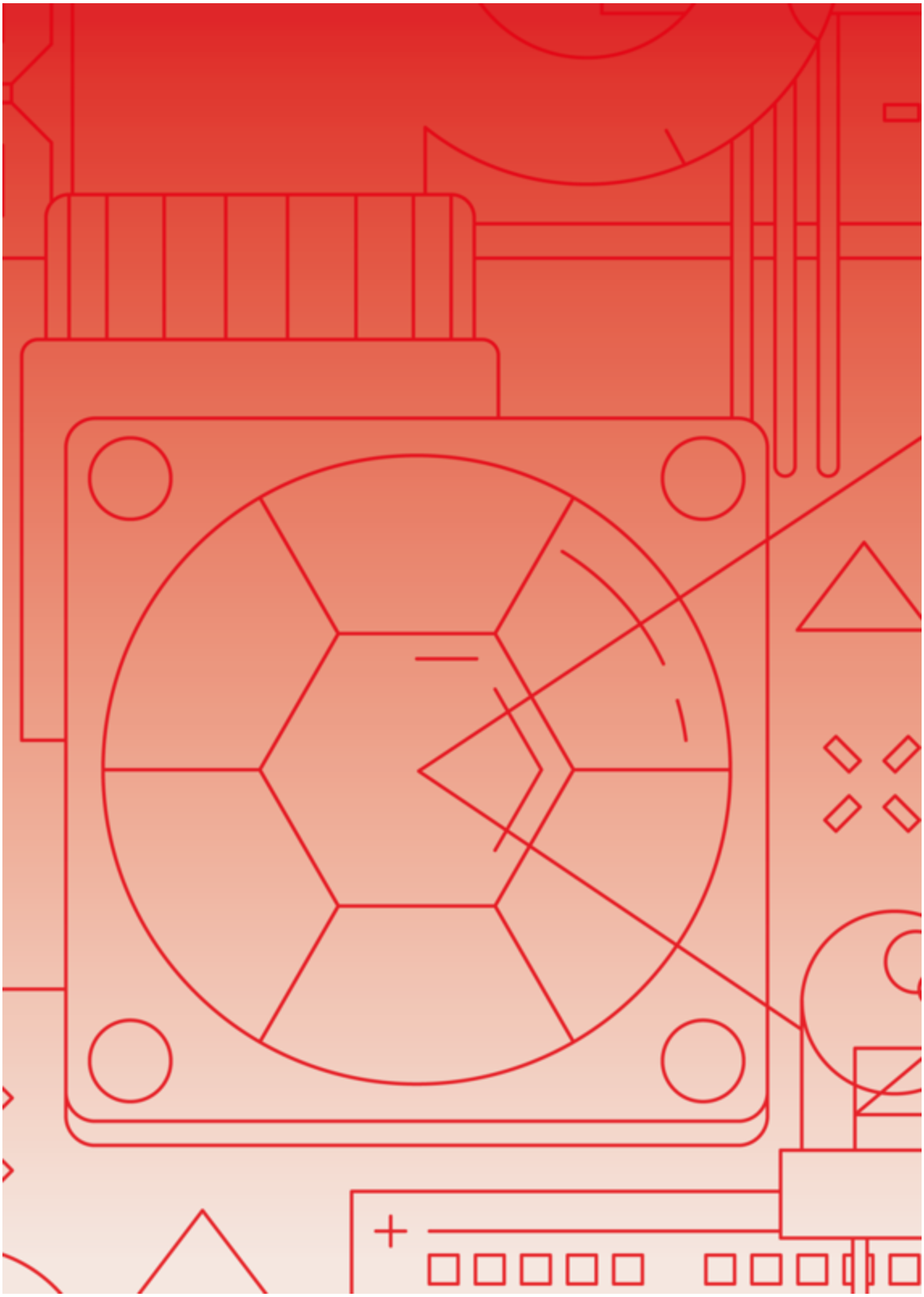
```
from gpiozero import RGBLED
import psutil, time

led = RGBLED(14, 15, 18)

while True:
    cpu = psutil.cpu_percent()
    r = cpu / 100.0
    g = (100 - cpu)/100.0
    b = 0
    led.color = (r, g, b)
    time.sleep(0.1)
```

At the top, we import the modules we need, including `psutil`. We then set the `led` variable to the `RGBLED` class on GPIO 14, 15, and 18, for red, green, and blue. In a never-ending `while True:` loop, we set the `cpu` variable to the percentage of CPU usage via `psutil`, then assign the red and green LED values accordingly, and light the LED.

Try running the code. The LED should light up: its colour will indicate how hard your Raspberry Pi's CPU is working. Green means less busy, turning redder as the CPU becomes more heavily loaded. Start up some other applications to test it. If you have an original Model B, you'll probably find that just running Chrome is enough to turn the LED red. If you have a Raspberry Pi 5, you may need to start lots of things running to have any impact! You can exit the program with CTRL+C.

Customise your project

The example code only uses the red and green components of the LED: the blue value is always set to zero. You could swap things around and create a different colour gradient (e.g. blue to red) or put together a fancy function that maps a percentage value onto all three colours. Have fun with the colours and maybe even have it look at other resources to monitor — GPIO Zero includes several internal *pseudo devices* you can use in your code, including ones that represent times of day, CPU temperature, and disk usage. See rpimag.co/gpiozinternal for details.



Chapter 6

Make a motion-sensing alarm

Stop people from sneaking up with an alarm that buzzes when it “sees” them

Need to protect your room or precious items from miscreants or nosy family members? With just a PIR motion sensor and a buzzer wired up to your Raspberry Pi, it’s very simple to create an intruder alert. Whenever movement is detected in the area, a loud beeping noise will raise the alarm. To take things further, you could add a flashing LED, an external speaker to play a message, or even a hidden Camera Module to record footage of intruders.

You’ll Need

- 1× solderless breadboard
- 1× HC-SR501 PIR sensor
- 1× Mini piezo buzzer
- Jumper wires

Attach components

First, we need to wire the PIR (passive infrared) sensor to the Raspberry Pi. While it could be hooked to the GPIO pins directly using socket-to-socket jumper wires, we’re doing it via a breadboard. The sensor has three pins: VCC (voltage supply), OUT (output), and GND (ground). Use socket-to-pin jumpers to connect VCC to the + rail of the breadboard, and GND to the – (ground) rail. Connect OUT to a numbered row, then use another jumper to connect that row to GPIO 4 (you could also use a socket-to-socket jumper to connect the OUT pin directly to GPIO 4).

Next, we'll hook up the mini buzzer. Place its two legs across the central groove in the breadboard. Note that the longer leg is the positive pin; wire its numbered row to GPIO 3 on the Raspberry Pi to connect it. Wire the row of the buzzer's shorter leg to the – rail.

Finally, connect the – rail to a GND pin on the GPIO header, and the + rail to the 5V pin. Your circuit should now resemble Figure 6-1.

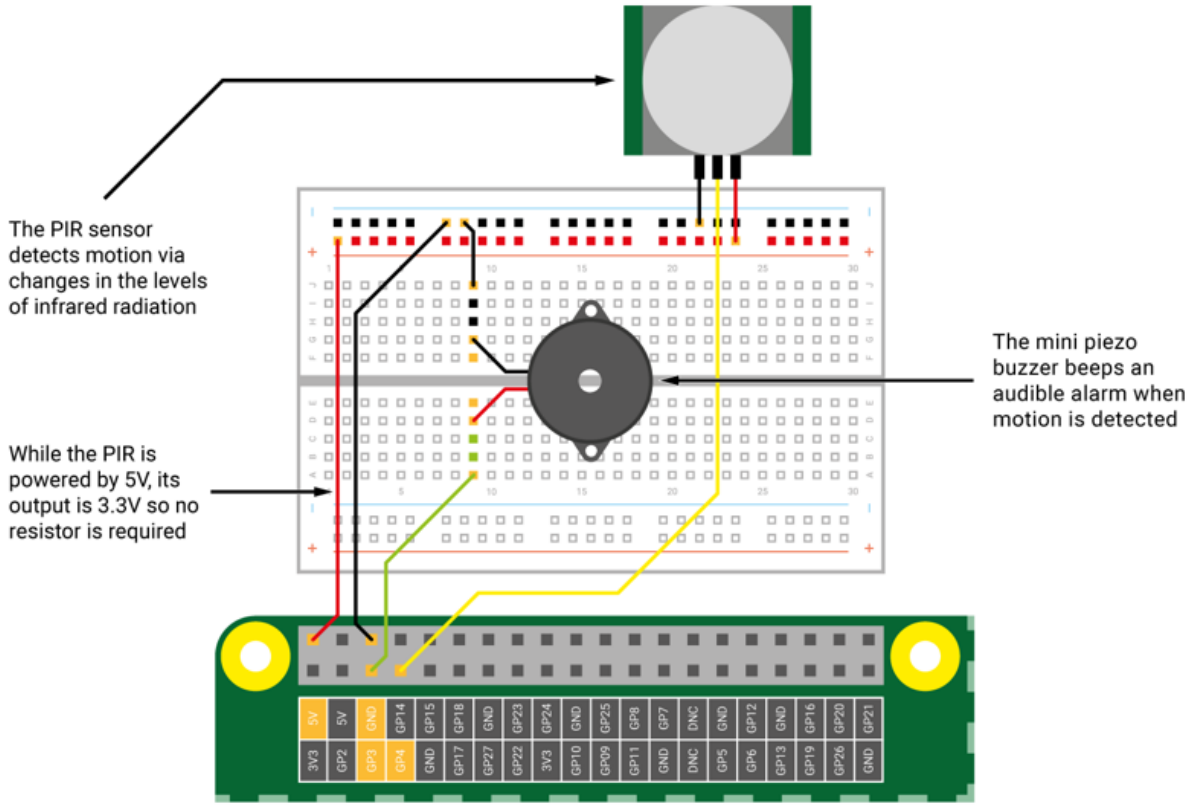


Figure 6-1: Wiring up the alarm

Work on the code

Create a new file, then save the following code as motion_alarm.py, and run it (see "Light the LED" for an overview of editing and running code).

```
from gpiozero import MotionSensor, Buzzer
from time import sleep
```

```
pir = MotionSensor(4)
bz = Buzzer(3)

print("Waiting for PIR to settle")
pir.wait_for_no_motion()

while True:
  print("Ready")
  pir.wait_for_motion()
  print("Motion detected!")
  bz.beep(0.5, 0.25, n=8)
  sleep(3)
```

At the start, we import the `MotionSensor` and `Buzzer` classes from GPIO Zero, each of which contains numerous useful methods; we'll need a few of them for our intruder alarm. We also import the `sleep` function from the `time` library so that we can add a delay to the detection loop. Next, we assign the relevant GPIO pins for the PIR sensor and buzzer; we've used GPIO 4 and 3 respectively in this example, but you could use alternatives if you prefer.

Setting things up

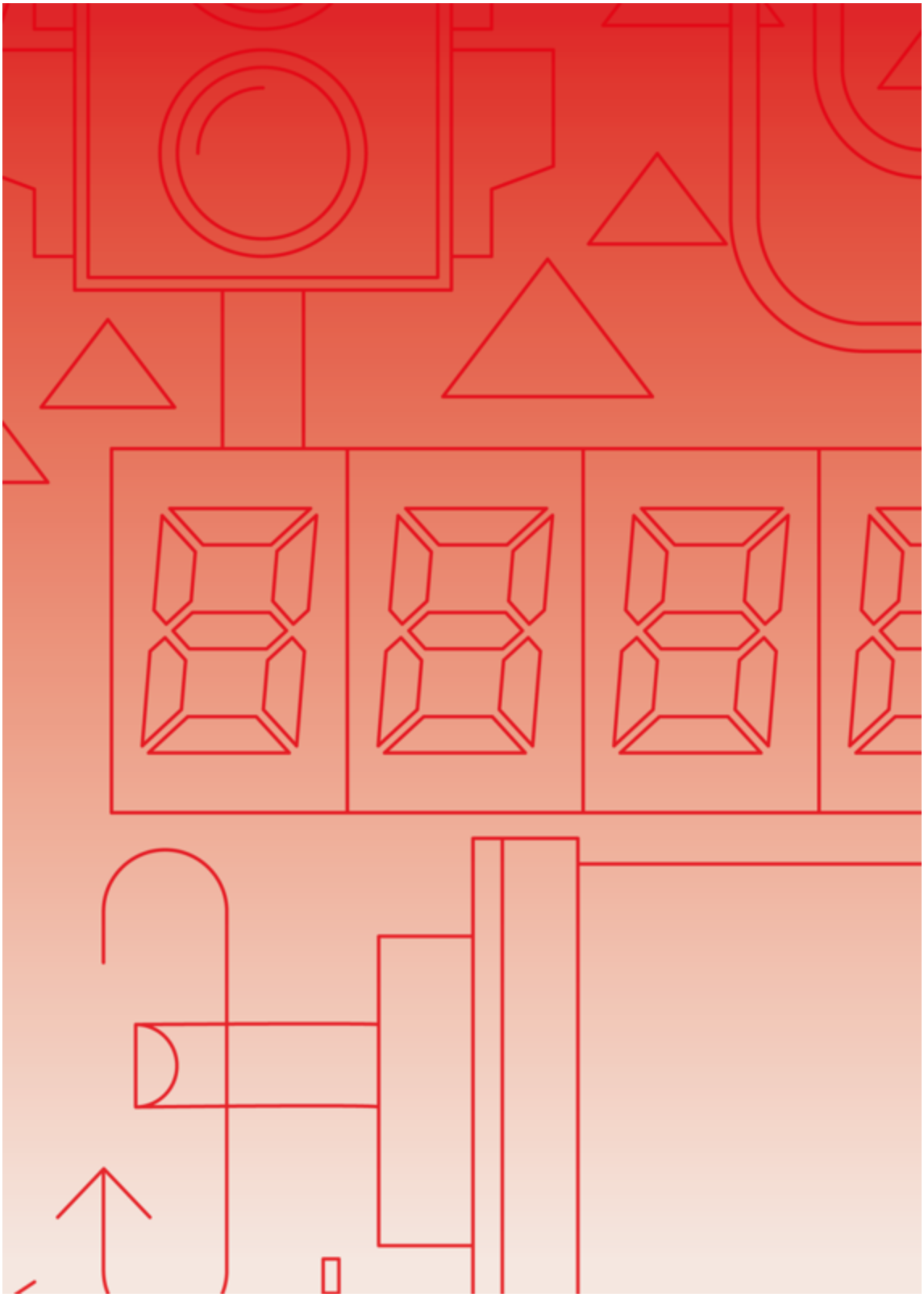
Before starting our motion detection `while` loop, we make use of the GPIO Zero library's `wait_for_no_motion` function to wait for the PIR to sense no motion. This gives you time to leave the area, so that it doesn't immediately sense your presence and raise the alarm when you run the code! Once the PIR has sensed no motion in its field of view, it will print 'Ready' on the screen and the motion detection loop can then commence.

Motion detection loop

Using `while True:` means this is an infinite loop that will run continually, until you stop the program pressing CTRL+C. Whenever motion is detected by the PIR sensor, we get the buzzer to beep repeatedly eight times: 0.5 seconds on, 0.25 seconds off, but you can alter the timings. We then use `time.sleep(3)` to wait 3 seconds before restarting the loop.

Adjust the sensitivity

If you find that the alarm is going off too easily or not at all, you may need to adjust the sensitivity of the PIR sensor. This is achieved by using a small screwdriver to adjust the plastic screw of the left potentiometer, usually labelled Sx; turn it anticlockwise to increase sensitivity. The other potentiometer, Tx, alters the length of time the signal is sent after detection; we found it best to turn it fully anticlockwise, for the shortest delay of 1 second.



Chapter 7

Make a range finder

Link an ultrasonic distance sensor and seven-segment display to measure distances

The HC-SR04 ultrasonic distance sensor is a favourite with Raspberry Pi robot makers. It works by bouncing ultrasonic sound off an object and timing how long it takes for the echo to return. This time is then converted into a distance, which can be displayed on a single-digit, seven-segment display. Using this sensor, you'll improve your skills in working with inputs and outputs. You also get to use seven-segment displays, which are quite cool in a retro kind of way.

You'll Need

- 1× solderless breadboard
- HC-SR04 ultrasonic distance sensor
- Broadcom 5082-7650 seven-segment display
- 9× resistors: 7× 220Ω, 1× 510Ω, 1 or 2× 1kΩ

Because the HC-SR04 uses a 5V power supply, it transmits a 5V pulse. We must use a *voltage divider* to take it down to the 3.3V tolerance of Raspberry Pi's GPIO pins. The maths for a voltage divider is simple:

(source voltage * R2) / (R1 + R2) = output voltage

Resistor 1 (R1) is connected to the source voltage, and Resistor 2 (R2) is connected to ground. With a 510Ω resistor for R1 and a 1kΩ for R2, the formula is:

(5 * 1000) / (510 + 1000) = 3.31

Although the GPIO pins can tolerate no more than 3.3V, they recognize 1.8V or higher as a HIGH signal. So you could use 1kΩ for both R1 and R1, which will give you a (very safe) 2.5V signal.

Lighting the display

The seven-segment display is a collection of LEDs, with one LED corresponding to one of the segments. All the anodes (positive ends) are connected; this should be connected to the 3V3 supply. Each cathode (negative end) should be connected to a resistor to limit the LED current, and the other end of the resistor to a GPIO pin. To turn the LED on, all you have to do is set the GPIO output to be LOW (0V) and it will complete the circuit for the current to flow. This is true for *common-anode* displays; we'll explain what changes you need to make if you're using a *common-cathode* display.

Generating a seven-segment pattern

The display consists of four bars or segments that can be lit. By choosing the segments to light up, you can display a number from 0 to 15, although you have to resort to letters A-F (hexadecimal) for this. There are, in fact, 128 different patterns you can make, but most are meaningless. Each pattern is a list of seven values, each one corresponding to an LED segment. By convention, the LED segments are identified by the letters a–g (not to be confused with the hexadecimal digits we are *displaying* on the LED), starting with the topmost horizontal segment, and continuing clockwise as shown in Figure 7-1. g identifies the middle horizontal segment.

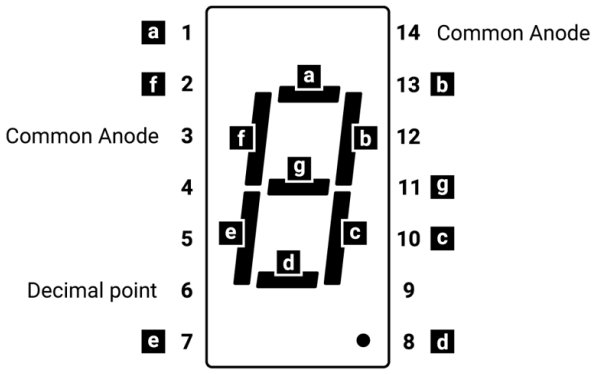


Figure 7-1: LED segment display pins

The patterns identify which segments should be turned on. The first one, with index 0, is (1, 1, 1, 1, 1, 1, 0) and illuminates all the segments except the middle segment, displaying the digit 0. The last one, with index 15, is (1, 0, 0, 0, 1, 1, 1), and illuminates the topmost segment (a), the two leftmost segments (e and f), and the middle segment (g), which displays the letter F. Figure 7-2 shows the breadboard layout.

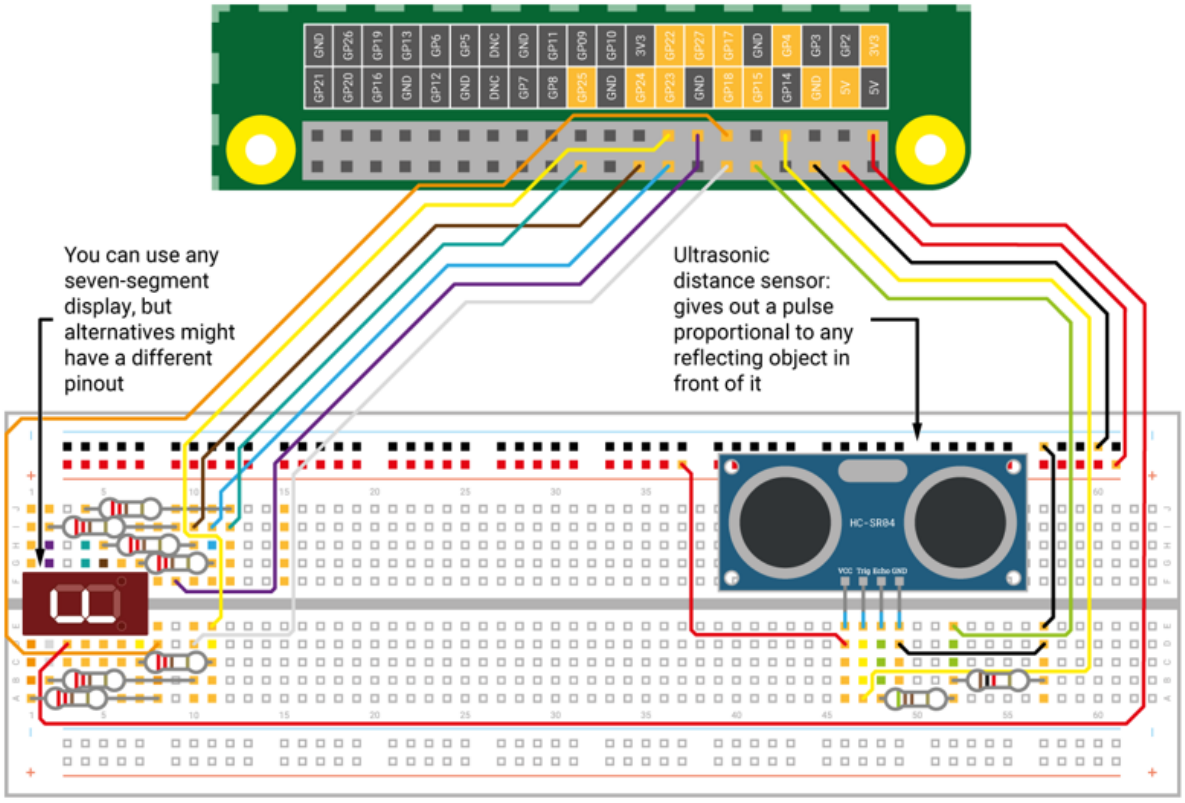


Figure 7-2: Building the range finder

If you have a different seven-segment display, consult its datasheet to find out whether it's common anode or common cathode. We used a common-anode display in Figure 7-2, hence the connection between common anode and 3V3. If you're using a common-cathode display, you'll need to connect its common cathode pin to the GND pin and *not* connect any of its pins to 3V3 (you'll also need to make a change to the code, explained later, to take the GPIO pins HIGH instead of LOW when illuminating a segment). See Figure 7-3 for a common-cathode version (SKU 103527 from The PiHut).

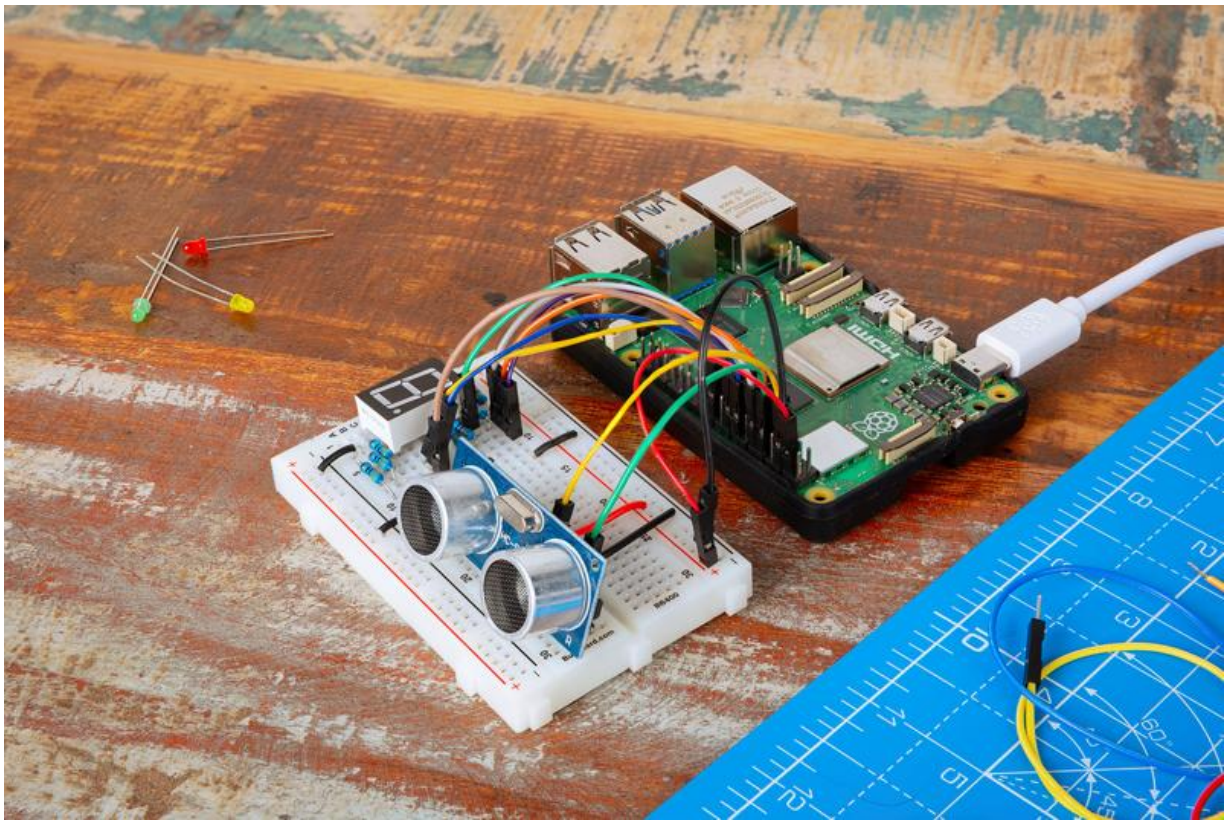


Figure 7-3: The common-cathode version of the range finder project

You'll also need the datasheet to confirm which pins correspond to which segment and adjust your wiring as needed. If you need help figuring out which LED pin goes with which segment, you can use the following code (test_segments.py) to light each segment in sequence, from a to g:

```
# Cycle through each segment in order
from gpiozero import LEDBoard
from time import sleep

#                a   b   c   d   e   f   g
seg = LEDBoard(17, 27, 24, 23, 22, 18, 25, active_high=False)

while True:
  for i in range(0, 7):
    print("abcdefg"[i])
    seg.on(i)
```

```
    sleep(1)
    seg.off(i)
```

Now you're ready to create the range finder code. Create a new file, add the following code, and save it as range_finder.py.

```
# displays the distance in decimetres on a 7-segment display
from gpiozero import LEDBoard, DistanceSensor
from time import sleep

seg = LEDBoard(17, 27, 24, 23, 22, 18, 25, active_high=False)
sensor = DistanceSensor(echo=15, trigger=4)

seg_patterns = [
  (1, 1, 1, 1, 1, 1, 0),
  (0, 1, 1, 0, 0, 0, 0),
  (1, 1, 0, 1, 1, 0, 1),
  (1, 1, 1, 1, 0, 0, 1),
  (0, 1, 1, 0, 0, 1, 1),
  (1, 0, 1, 1, 0, 1, 1),
  (1, 0, 1, 1, 1, 1, 1),
  (1, 1, 1, 0, 0, 0, 0),
  (1, 1, 1, 1, 1, 1, 1),
  (1, 1, 1, 0, 0, 1, 1),
  (1, 1, 1, 0, 1, 1, 1),
  (0, 0, 1, 1, 1, 1, 1),
  (1, 0, 0, 1, 1, 1, 0),
  (0, 1, 1, 1, 1, 0, 1),
  (1, 0, 0, 1, 1, 1, 1),
  (1, 0, 0, 0, 1, 1, 1),
]

print("Display distance on a 7-seg display")
while True:
  distance = sensor.distance * 10 # distance in decimeters
  print("distance", distance)
```

```
    if distance > 15:
        distance = 15
    seg.value = seg_patterns[int(distance)]
    sleep(0.8)
```

In our code, we create an `LEDBoard` object (a collection of LEDs in one object), defining which pins are connected to which segments, and a list called `seg_patterns` defines the LED pattern for each number.

If you are using a common-cathode LED segment display, you'll need to remove the part that sets `active_high` to `False`, changing:

```
seg = LEDBoard(17, 27, 24, 23, 22, 18, 25, active_high=False)
```

to:

```
seg = LEDBoard(17, 27, 24, 23, 22, 18, 25)
```

Run the code and watch the distance values appear on the display. You can stop the program with CTRL+C.

Displaying numbers

The way to set the desired LED configuration to make a number is to set the `LEDBoard`'s value to a 7-tuple of the states of the LEDs (each element of the `seg_patterns` list is a *tuple* — an immutable sequence — because each tuple has seven values, we call them 7-tuples). The `seg_patterns` list contains all the `LEDBoard` values required to display numbers 0-9 and letters A-F. `seg_patterns[0]` gives the pattern for 0 and `seg_patterns[15]` gives the pattern for F. This pattern is given to the `LEDBoard` by setting its value with `seg.value = seg_patterns[number]`.

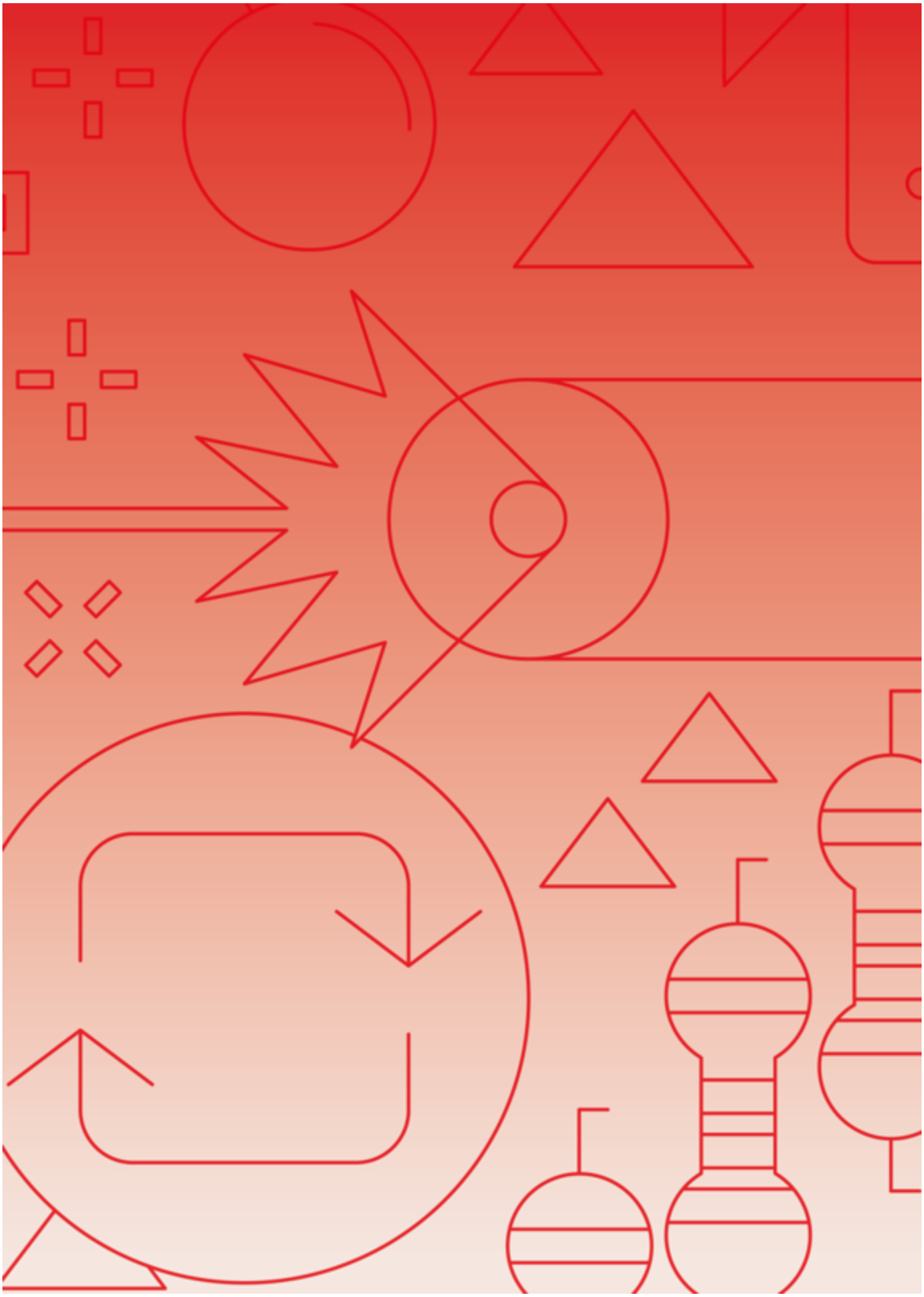
GPIO Zero offers an easier way of working with segment LEDs, but if we'd used it in this chapter, you would not have learned as much about how these displays work. You can use `LEDCharDisplay` to display more characters than we used in this example, though you're still subject to the limits of what characters a seven-segment display can represent. See rpimag.co/ledchardisp for details.

The distance sensor

The HC-SR04 distance sensor reports its reading by producing an output pulse that the Raspberry Pi tries to measure. The GPIO Zero library measures this pulse and converts it into a distance by returning a floating-point number that maxes out at 1 metre. We then multiply this number by 10 to give decimetres. Next, we convert it to an integer to get rid of the fractional part of the measurement, so we can show it on our single-digit display.

Using the range finder

The distance to the reflective object is updated every 0.8 seconds. A display of 0 indicates that the object is less than 10cm away. Don't touch the sensor, otherwise its readings will be wrong. Also, as it has quite a wide beam, you can get reflections from the side. If several objects are in the field of view, then the distance to the closest one is returned.



Chapter 8

Make a laser tripwire

Learn how to use a light-dependent resistor to detect a laser pointer beam

Your Raspberry Pi computer can easily detect a digital input via its GPIO pins: any input that's approximately below 1.8V is considered off, and anything above 1.8V is considered on. An analogue input can have a range of voltages from 0V up to 3.3V, but the Raspberry Pi is unable to detect exactly what that voltage is without the help of an additional component, as you'll see in *Chapter 9, Build an internet radio*.

One way of getting around this is by using a capacitor and timing how long it takes to charge up above 1.8V. By placing a capacitor in series with a *light-dependent resistor* (LDR, also known as a *photocell*), the capacitor will charge at different speeds depending on whether it's light or dark. We can use this to create a laser tripwire!

You'll Need

- 1× solderless breadboard
- 1× light-dependent resistor (LDR)
- 1× 1µF capacitor
- 1× laser pointer
- 5× pin-to-socket jumper wires
- 5× socket-to-socket jumper wires (optional)
- 1× drinking straw or short length of heat-shrink tubing
- 1× plastic box

Connect the LDR

An LDR is a special type of electrical resistor whose resistance is very high when it's dark but reduced when light is shining on it. With the Raspberry Pi turned off and disconnected from power, place your LDR into the breadboard, then add the capacitor. It's essential to get the correct polarity for the latter component: its longer (positive) leg should be in the same breadboard column as one leg of the LDR. Now connect this column (with a leg of both components) to GPIO 4. Connect the other leg of the LDR to a 3V3 pin, and the other leg of the capacitor to a GND pin. Your circuit should now resemble Figure 8-1.

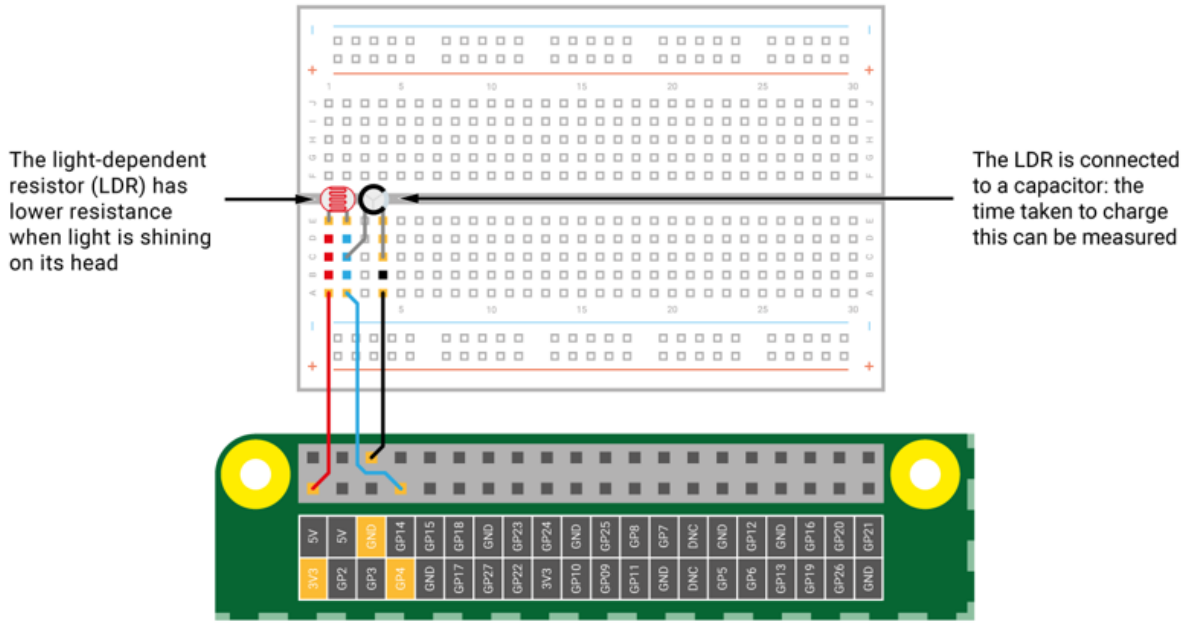


Figure 8-1: Basic LDR breadboard circuit

Test the LDR

GPIO Zero offers a helpful `LightSensor` class that is designed for this exact circuit configuration. However, as of this writing, `LightSensor` did not work on the most recent version of Raspberry Pi OS, which means we'll need to write some code to perform the detection logic ourselves. On your Raspberry Pi, create a new file, enter the code below, and save it as test_ldr.py (see "Light the LED" for an overview of editing and running code).

```
from gpiozero import OutputDevice, DigitalInputDevice
from time import sleep

def LDR_value(pin, charge_time_limit=0.001):

  # Take the pin LOW to discharge the capacitor
  ldr = OutputDevice(pin=pin)
  ldr.off()
  sleep(0.1)
  ldr.close()

  # Configure the pin as a floating input
  ldr = DigitalInputDevice(pin=pin, pull_up=None,
              active_state=True)

  # If the pin goes active before the timeout, we have light
  lit = ldr.wait_for_active(timeout=charge_time_limit)
  ldr.close()

  return lit

ldr_pin = 4
while True:
  print(LDR_value(ldr_pin))
  sleep(1) # Wait for a second
```

At the start, we import the `OutputDevice` and `DigitalInputDevice` classes from GPIO Zero. We then define a function called `LDR_Value` that takes two arguments: a pin number and a `charge_time_limit` (in seconds, with a one-millisecond default).

Next, we create an `OutputDevice` on that pin, then take it LOW for 100 milliseconds. This discharges the capacitor.

Next, we create a digital input device as a floating input, which means the internal pull-up resistors are inactive. This is the workaround for the problem that exists as of this writing. You can read more about the problem and workaround on this GitHub issue: rpimag.co/LightSensorIssue

Next, we wait for the pin to be active (HIGH), which means the capacitor has been charged up. If `wait_for_active()` returns False, it means that it timed out before the pin went high. You can increase `charge_time_limit` to make the code more sensitive to small amounts of light and decrease it to make it less sensitive.

Enclose the LDR

Unless you're working in a darkened room, you'll probably notice little difference between the measured light level when the laser pointer is directed onto the LDR and when it's not. This can be fixed by reducing the amount of light that the LDR receives from other light sources in the room, which will be essential for our laser tripwire device to work effectively. You can achieve this by cutting off a short section — between 2cm and 5cm — of an opaque drinking straw or heat-shrink tubing and inserting the head of the LDR into one end. Now try the test code again and see how the measured light level changes when you shine the laser pointer into the other end of the straw or tubing. You should notice a larger difference in values.

Wire up the buzzer

To create an audible alarm for our laser tripwire, we'll add a piezo buzzer to the circuit. As in *Chapter 6, Make a motion-sensing alarm*, the polarity must be correct: connect the column of the buzzer's longer leg to GPIO 17, and the shorter leg to a GND pin. It should look like Figure 8-2. Let's test it to see if it's working. Create a new file, enter the following code, and save it as test_buzzer.py.

```
from gpiozero import Buzzer
from signal import pause

buzzer = Buzzer(17)
buzzer.beep()
pause()
```

At the top, we import the `Buzzer` class from GPIO Zero and `pause` from the `signal` library. Next, we set the `buzzer` variable to the buzzer output on GPIO 17. Finally, we use `buzzer.beep` to make the buzzer turn on and off repeatedly at the default length of 1 second. To stop it, press CTRL+C while the buzzer is quiet.

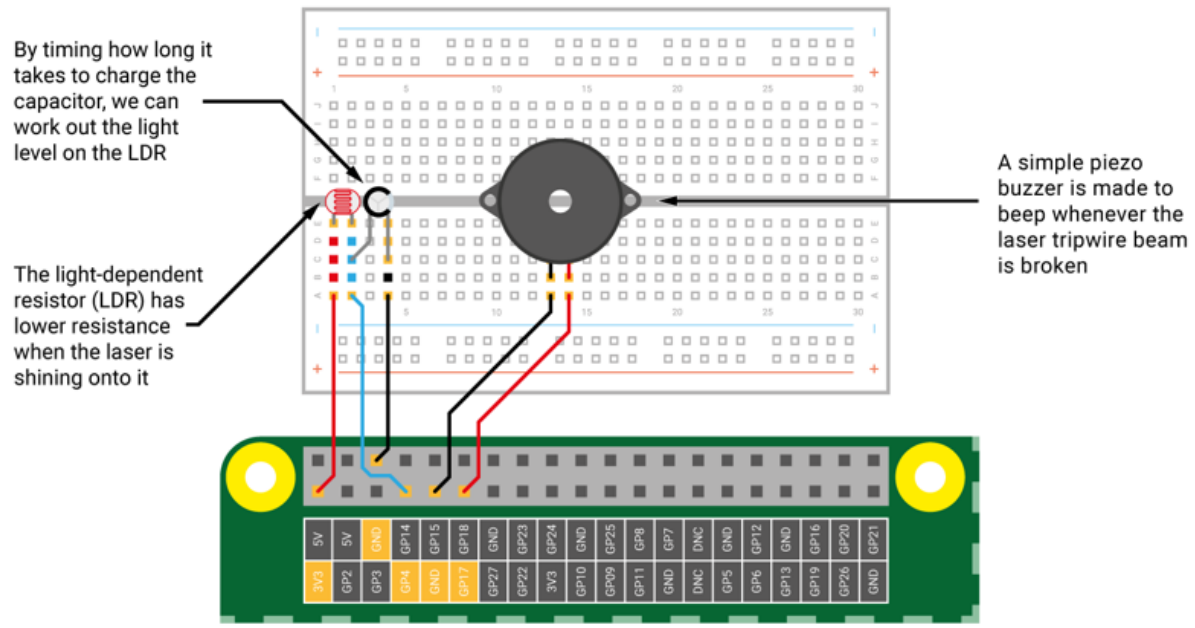


Figure 8-2: Adding the buzzer to the breadboard

Test the tripwire

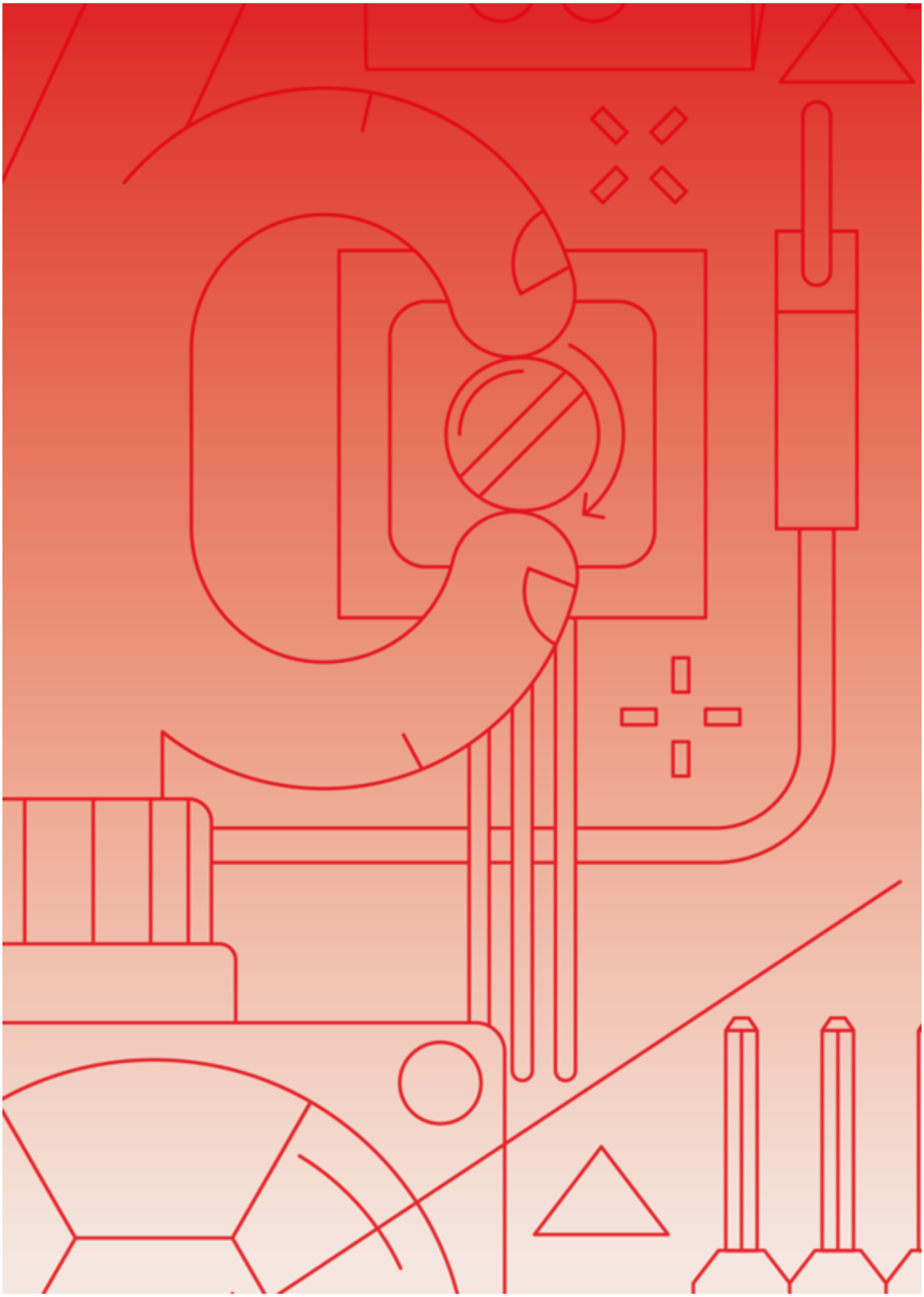
We'll now put everything together so that the laser pointer shines at the LDR (through the straw) and whenever the beam is broken, the buzzer sounds the alarm. In Thonny, create a new file, enter the following code and save it as tripwire.py.

```
from gpiozero import OutputDevice, DigitalInputDevice, Buzzer
from time import sleep

def LDR_value(pin, charge_time_limit=0.001):
  ldr = OutputDevice(pin=pin)
  ldr.off()
  sleep(0.1)
  ldr.close()

  ldr = DigitalInputDevice(pin=pin, pull_up=None,
                active_state=True)
  lit = ldr.wait_for_active(timeout=charge_time_limit)
  ldr.close()

  return lit

buzzer = Buzzer(17)
ldr_pin = 4
while True:
  if LDR_value(ldr_pin):
    buzzer.beep(0.5, 0.5, n=8, background=False)
  sleep(0.1)
```

At the start, we import the `OutputDevice`, `DigitalInputDevice`, and `Buzzer` classes from GPIO Zero. We also import the `sleep` function from `time`; we'll need this to slow the script down a little to give the capacitor time to charge. As before, we assign variables for the buzzer and LDR pin to the respective devices on GPIO 17 and 4. We then use a `while True:` loop to continually check the LDR; if enough light falls on it, we make the buzzer beep. Try running the code; if you break the laser beam, the buzzer should beep for 8 seconds. You can adjust this by altering the `buzzer.beep` parameters and `sleep` time.

Package it up

Once everything is working well, you can enclose your Raspberry Pi and breadboard in a plastic box (such as an old ice cream tub), with the drinking straw poking through a hole in the side. If you prefer, you can remove the breadboard and instead connect the circuit up directly by poking the legs of the components into socket-to-socket jumper wires, with the long capacitor leg and an LDR leg together in one wire end, connected to the relevant pins. Either way, place the tub near a doorway with the laser pointer on the other side, with its beam shining into the straw. Run your code and try walking through the doorway: the alarm should go off!



Chapter 9

Build an internet radio

Use potentiometers to control an LED and tune in to online radio stations

In *Chapter 8, Make a laser tripwire*, we used a trick with a capacitor to get a reading from an analogue sensor. The best — and most reliable — way for Raspberry Pi to detect analogue inputs is by using an analogue-to-digital converter (ADC) chip, such as the MCP3008, which offers eight input channels to connect sensors and other analogue inputs. This not only eliminates the need for the capacitor, but it allows you to get more precise and instant readings across the entire range of the sensor's outputs.

In this tutorial, we'll hook up a potentiometer to an MCP3008, to control the brightness of an LED by turning the knob. We'll then add a second potentiometer and create an internet radio, using the two potentiometers to switch the station and adjust the volume.

You'll Need

- 1× solderless breadboard
- 1× MCP3008 ADC chip
- 2× potentiometers
- 1× LED
- 1× 330Ω resistor
- 7× pin-to-socket jumper wires
- 10× pin-to-pin jumper wires

Enable SPI

The analogue values from the ADC chip will be communicated to the Raspberry Pi using the SPI protocol. While this will work in GPIO Zero out of the box, you may get better results if you enable full SPI support. First, make sure the Python `spidev` package is installed (they should be installed by default). Open a terminal window and enter:

```
sudo apt install python3-spidev
```

Next, click the Raspberry Pi menu, choose Preferences, and open the Raspberry Pi Configuration tool. Next, enable SPI in the Interfaces tab. Click OK and reboot your Raspberry Pi. You can also use the command line tool by running `sudo raspi-config` from the command prompt, going to Interface Options, enabling SPI, and rebooting your Raspberry Pi.

Connect the ADC

As usual, you need to turn off the Raspberry Pi while creating the circuit. As you can see from Figure 9-1, there's quite a lot of wiring required to connect the MCP3008 ADC to the Raspberry Pi's GPIO pins.

First, place the MCP3008 in the middle of the breadboard, straddling its central groove. Now connect the jumper wires as in the diagram. Two go to the + power rail, connected to a 3V3 pin; two others are connected to a GND pin via the – rail. The four middle legs of the ADC are connected to GPIO 8 (CE0), 10 (MOSI), 9 (MISO), and 11 (SCLK).

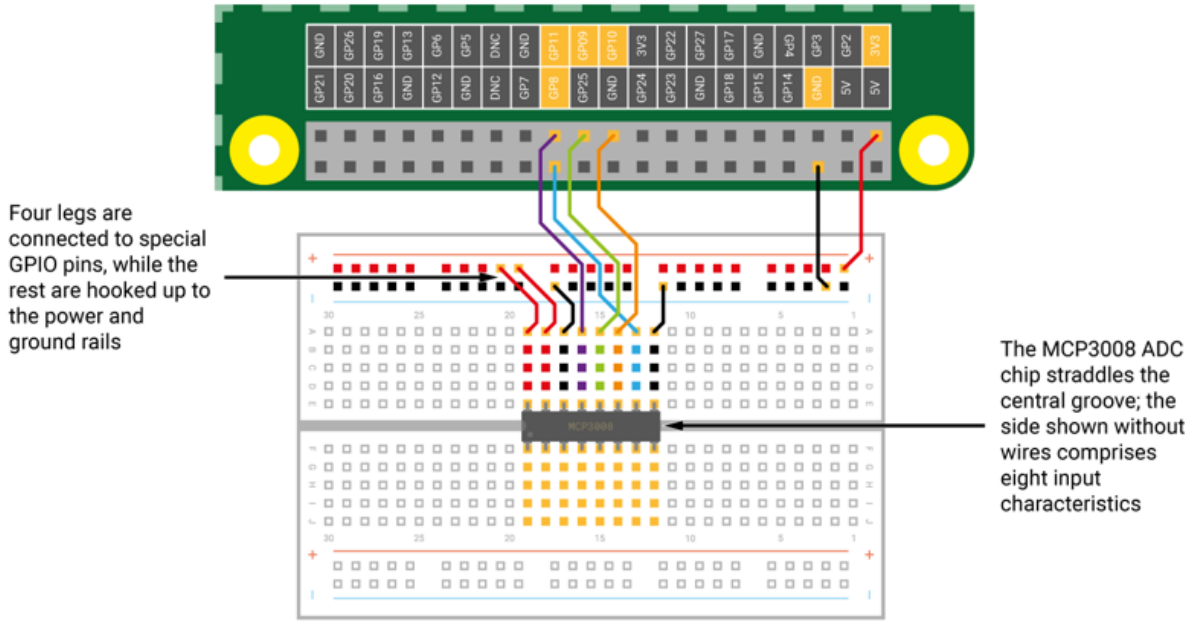


Figure 9-1: Wiring up the ADC on a breadboard

Read the value

With the ADC connected to the Raspberry Pi, you can wire devices to its eight input channels (numbered 0 to 7). Here, we'll connect the first (leftmost) potentiometer, which is a variable resistor: as you turn its rotary knob, the Raspberry Pi reads the voltage (from 0V to 3.3V). We can use this for control of other components, such as an LED. As shown in Figure 9-2, connect one outer leg of the potentiometer (bottom-left) to the + power rail, the other side to the – ground rail, and the middle leg to the first input of the MCP3008: channel 0.

We can now read the potentiometer's value in Python. Create a new file, then save the following code as test_pot.py, and run it.

```
from gpiozero import MCP3008
pot = MCP3008(channel=0)
while True:
  print(pot.value)
```

See “Light the LED” for an overview of editing and running code. At the top we import the `MCP3008` class from GPIO Zero, then set the `pot` variable to the ADC’s channel 0. A `while True:` loop then continuously displays the potentiometer’s value (from 0 to 1) on the screen; try turning it as the code runs, to see the number change. Press CTRL+C to exit.

Light an LED

Next, we’ll add an LED to the circuit as shown in Figure 9-2, connecting its longer (positive) leg to GPIO 21, and its shorter leg via a resistor to the – ground rail. Figure 9-3 shows the assembled project.

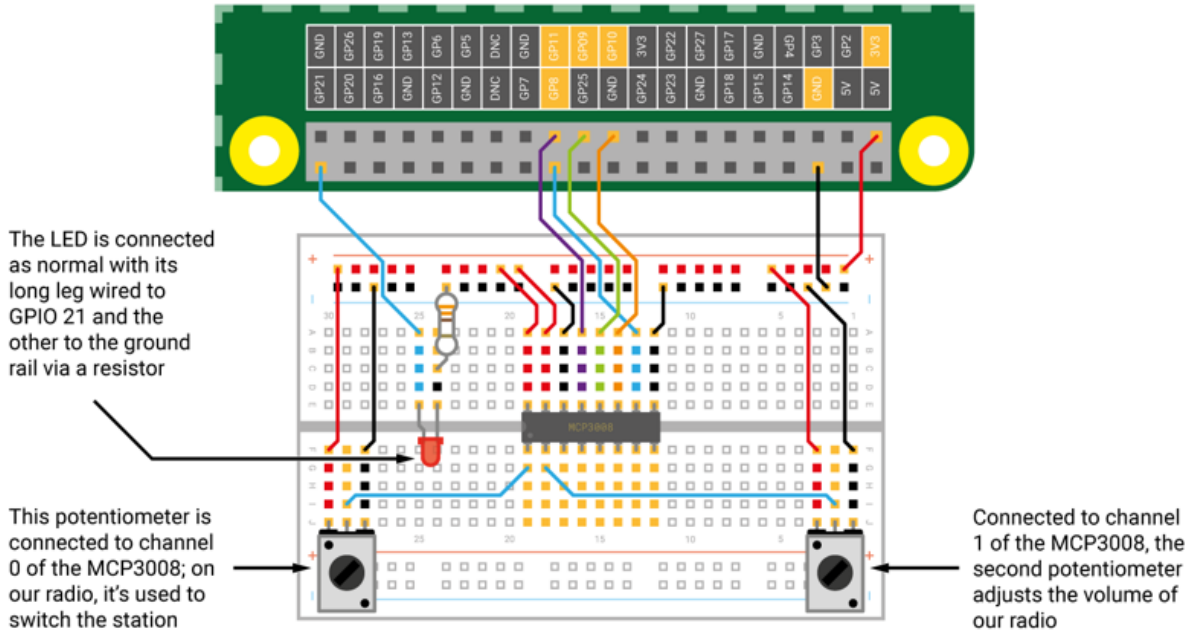


Figure 9-2: The LED and pots added to the breadboard

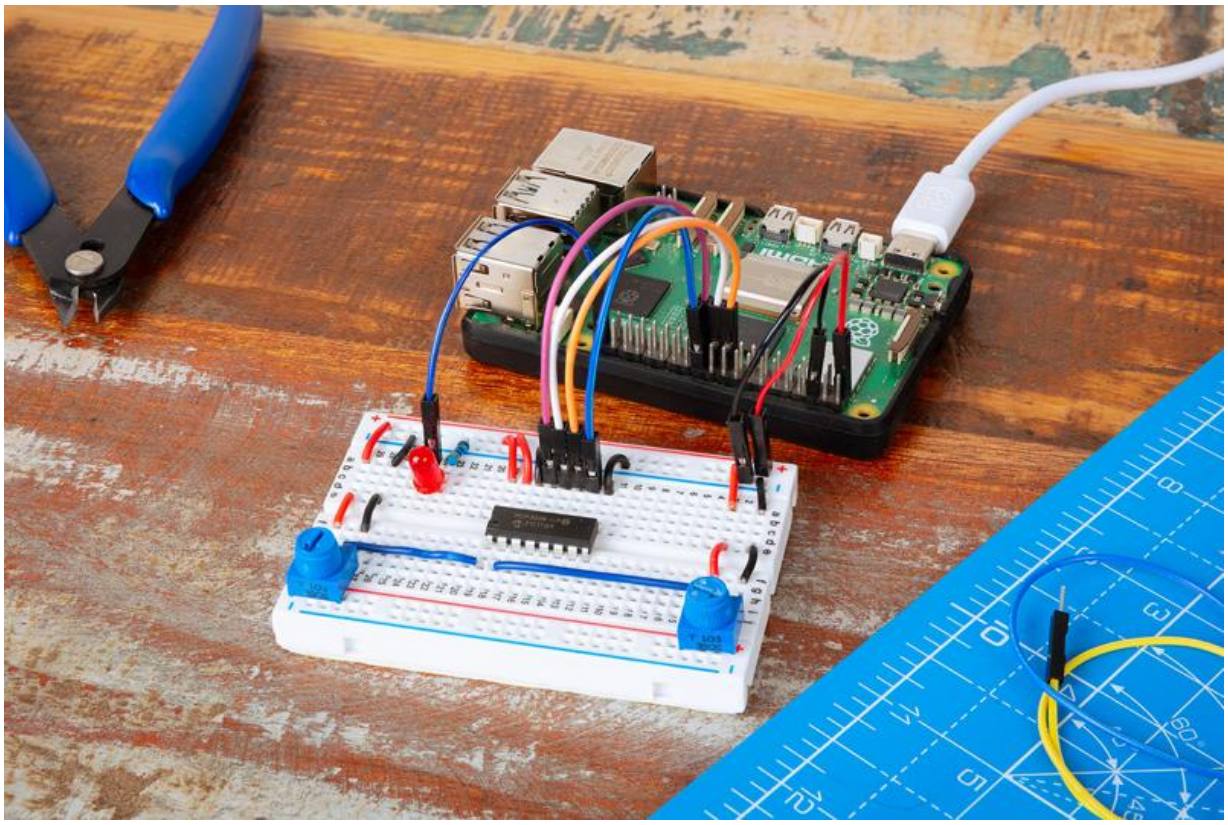


Figure 9-3: The assembled circuit

Create a new file, enter the following code, and save the program as source_values.py.

```
from gpiozero import MCP3008, PWMLED
from signal import pause

pot = MCP3008(0)
led = PWMLED(21)
led.source = pot.values
pause()
```

The code imports the `MCP3008` and `PWMLED` classes, as well as the `signal` module's `pause` function. The `MCP3008` class enables us to control the brightness of an LED using pulse-width modulation (PWM). We create a `PWMLED` object on GPIO 21, assigning it to the `led` variable. We assign our potentiometer to channel 0, as before. Finally, we use GPIO Zero's clever source and values system to pair the potentiometer with the LED, to

continuously set the latter's brightness level to the former's value. Run the code and turn the knob to adjust the LED's brightness (see Figure 9-2). Press CTRL+C to quit the program.

Add a second pot

If you haven't already, add a second potentiometer to our circuit as in Figure 9-2, with its middle leg connected to channel 1 of the MCP3008. We'll now use both potentiometers to control our LED's blink rate. In Thonny, create a new file, enter the following code and save it as two_pots.py.

```
from gpiozero import MCP3008, PWMLED

pot1 = MCP3008(0)
pot2 = MCP3008(1)
led = PWMLED(21)
while True:
  print(pot1.value, pot2.value)
  led.blink(on_time=pot1.value, off_time=pot2.value,
       n=1, background=False)
```

Here, we create two separate `pot1` and `pot2` variables, assigned to the ADC's channels 0 and 1 respectively. In a `while True:` loop, we then print the two values on the screen and make the LED blink, with its on and off times affected by our two potentiometers. Run the code and twist both knobs to see how it changes.

Install VLC

We'll use the same circuit to create a simple internet radio, with one potentiometer used to switch the station and the other to adjust the volume. If it's not installed by default, you'll need to install the VLC media player to be able to play M3U internet radio streams. You'll also want the `python3-alsaaudio` package to help control the system audio settings. Open a terminal window and enter:

sudo apt install vlc python3-alsaaudio

Make the radio

Create a new file, enter the code shown below, and save it as radio.py.

```
from gpiozero import MCP3008
from subprocess import Popen
import alsaaudio
import time

station_dial = MCP3008(0)
volume_dial = MCP3008(1)

url = "http://lstn.lv/bbcradio.m3u8?station={0}&bitrate={1}"
Music = url.format("bbc_6music", 96000)
Radio4 = url.format("bbc_radio_fourfm", 96000)
current_station = Radio4

vlc = None

def change_station(station):
  global current_station, vlc
  if station != current_station:
    if vlc is not None:
      vlc.terminate()
      vlc = None
    vlc = Popen(["cvlc", station])
    current_station = station

mixer = alsaaudio.Mixer()
while True:
  vol = int(65 + volume_dial.value * 35)
  mixer.setvolume(vol)
  if station_dial.value >= 0.5:
```

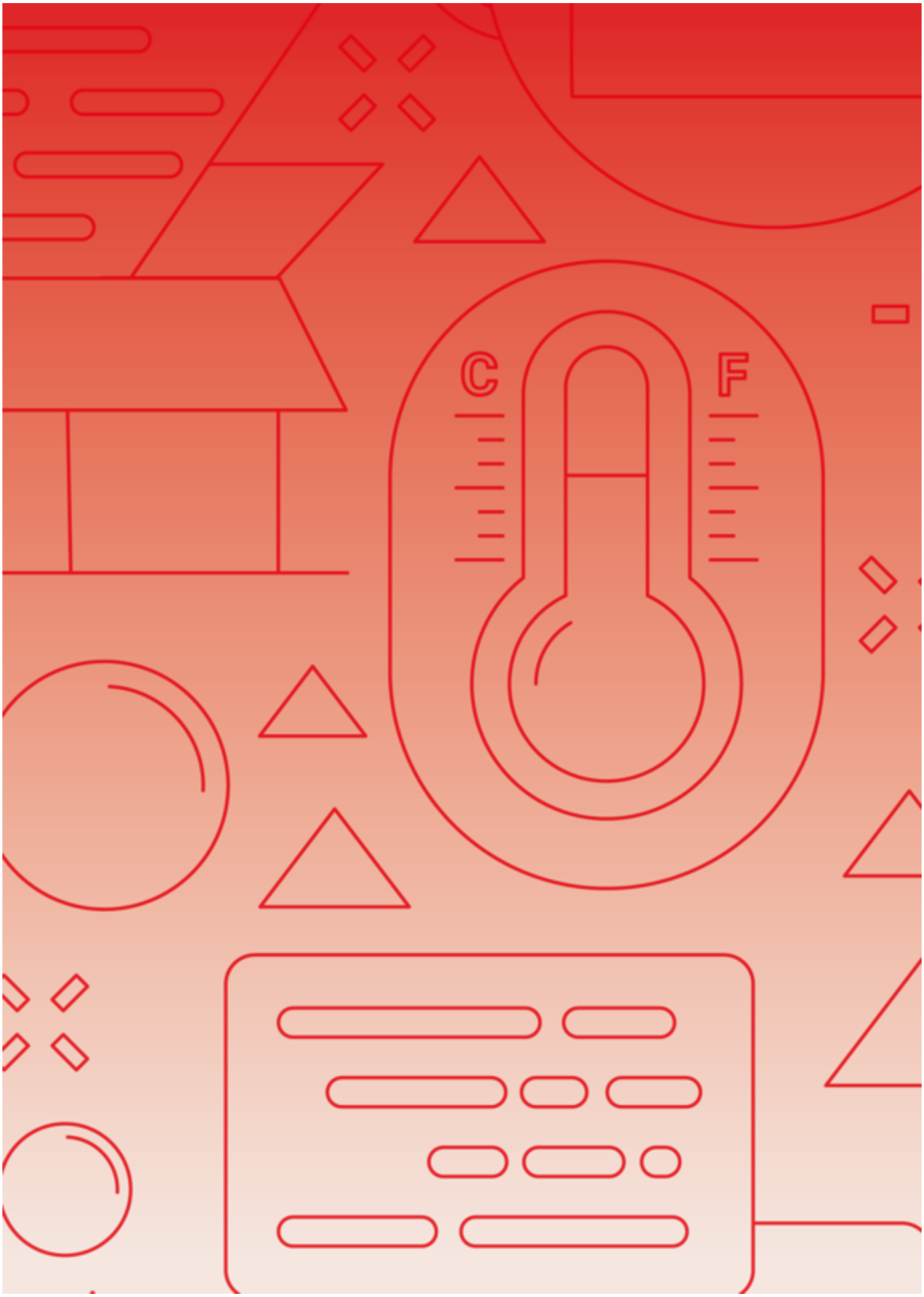
```
    station = Music
    change_station(station)
  elif station_dial.value < 0.5:
    station = Radio4
    change_station(station)
  time.sleep(0.1)
```

At the start, we import the MCP3008 class, along with Popen, alsaaudio, and time; Popen will enable us to start and stop VLC. We create variables for the station and volume dials, on ADC channels 0 and 1 respectively. We then assign variables to two radio stream URLs, for BBC 6 Music and BBC Radio 4, and set the current_station variable to the latter.

Next, we create a function called change_station which includes an if condition, so it only triggers when the station set by the first potentiometer position is different from the currently selected one (current_station). If so, it stops the current stream and starts playing the new one, before reassigning the current_station variable to it.

Finally, in a while True: loop, we set the audio volume to the value of the second potentiometer using the mixer; we've assigned a minimum value of 65%, but you can alter this. It then checks whether the first potentiometer is below or above 0.5 and calls the change_station function.

Run the code and try turning both potentiometers to switch the station and adjust the volume. To keep things simple, we've only used two radio stations in this example, but you could easily add more.


C
F

Chapter 10

Create an LED thermometer

Read a temperature sensor and display its readings on an LED bar graph

Continuing with the theme of analogue inputs, we'll use the MCP3008 analogue-to-digital converter (ADC) again and this time hook it up to a temperature sensor. We'll display the current temperature on the screen, then add some LEDs and use GPIO Zero's handy `LEDBarGraph` class to show the temperature in a bar graph style.

You'll Need

- 1× solderless breadboard
- 1× MCP3008 ADC chip
- 1× TMP36 temperature sensor
- 5× LEDs (red, yellow, green)
- 5× 330Ω resistor
- 1× 1µF ceramic capacitor
- 11× pin-to-socket jumper wires
- 8× pin-to-pin jumper wires

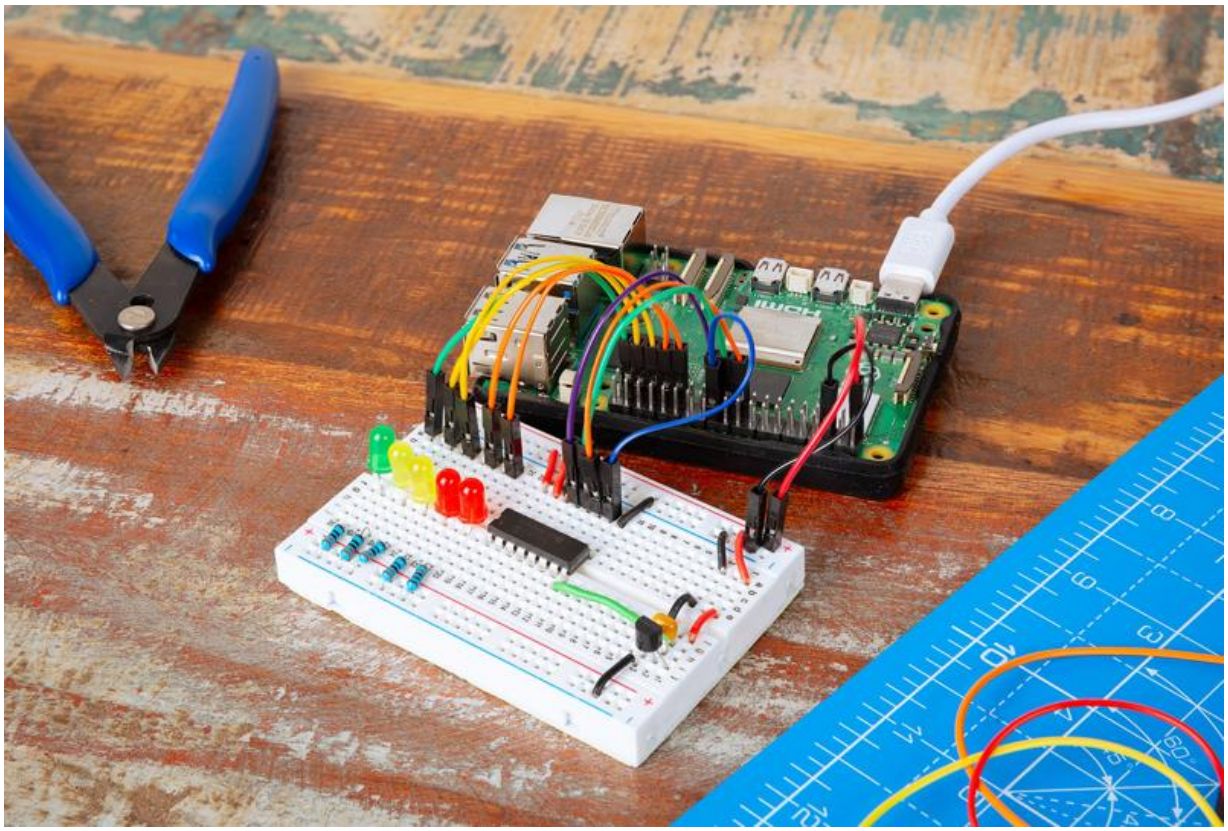


Figure 10-1: The LED thermometer

Enable SPI

As in *Chapter 9, Build an internet radio*, the analogue values from the ADC chip will be communicated to the Raspberry Pi using the SPI protocol. While this will work in GPIO Zero out of the box, you will get better results if you enable full SPI support. See “Enable SPI” for details.

Connect the ADC

If you still have the MCP3008 wired up from *Chapter 9, Build an internet radio*, leave it in place, straddling the central groove of the breadboard. As noted before, there’s quite a lot of wiring required; connect the jumper wires as in Figure 10-2. Two go to the + power rail, connected to a 3V3 pin; two others are connected to a GND pin via the – rail. Four legs of the ADC are connected to GPIO 8 (CE0), 10 (MOSI), 9 (MISO), and 11 (SCLK).

Add the sensor

Now that the ADC is connected to the Raspberry Pi, you can wire devices up to its eight input channels, numbered 0 to 7. Figure 10-2 shows how to connect a TMP36 analogue temperature sensor. It's vital that this is wired up correctly, otherwise it'll overheat. With its flat face towards you, the left-hand leg is for power, so connect this to the + power rail. The right-hand leg is connected to the – ground rail. Its middle leg is the output; here we're connecting it to channel 7 (the nearest one) of the MCP3008. Finally, to help stabilise the readings which might otherwise be erratic, we'll add a capacitor to link its output and ground legs.

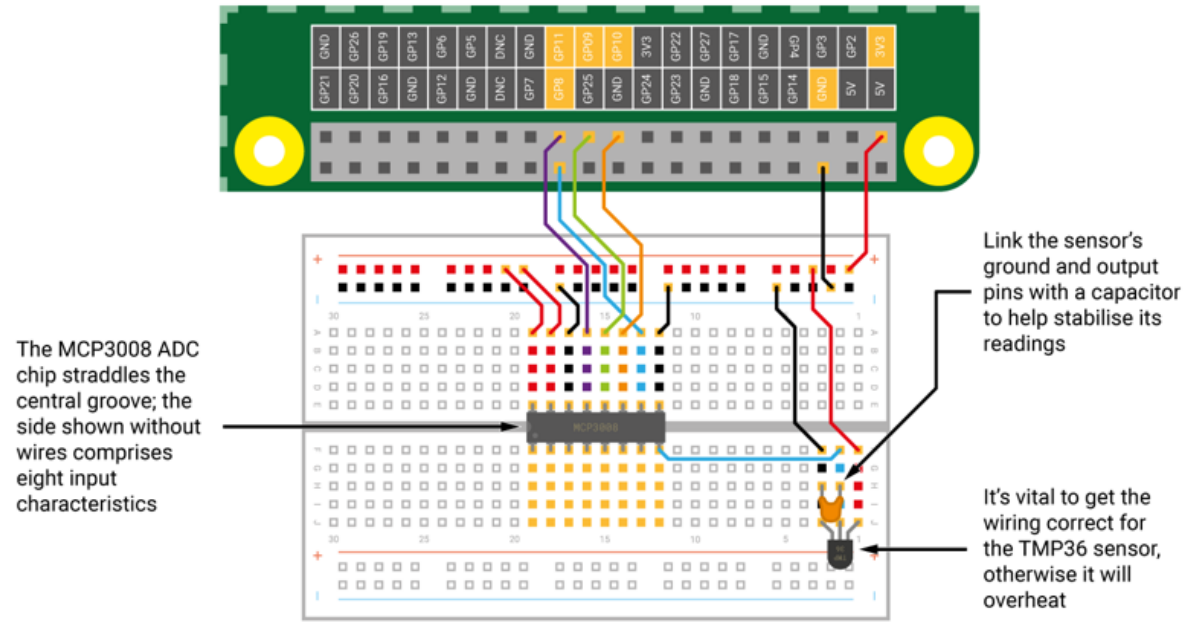


Figure 10-2: Wiring up the sensor

Take the temperature

We can now read the sensor's value in Python. Create a new file, then save the following code as test_tmp36.py, and run it (see "Light the LED" for an overview of editing and running code).

```
from gpiozero import MCP3008
```

```
from time import sleep


def convert_temp(gen):
  for value in gen:
    yield (value * 3.3 - 0.5) * 100

adc = MCP3008(channel=7)
for temp in convert_temp(adc.values):
  print(f"The temperature is {temp}C")
  sleep(1)
```

At the top we import the `MCP3008` class from GPIO Zero, then the `sleep` function from the `time` library. Next, we define a function that converts the sensor reading into degrees Celsius. We then set the `adc` variable to channel 7 of the MCP3008. Finally, we use a `for` loop to display the converted temperature, updating it every second. In this example, we're reading the `adc` object's `values` property, which is an endless source of sensor readings; the loop will continue until you stop the script. The values are fed to the `convert_temp` function, which is a *generator function*, a special type of function that can be used as the source of a `for` loop. Rather than using the `return` statement to return a single value, `convert_temp` uses the `yield` keyword, which causes Python to treat it as a generator rather than a regular function.

Note: If you've just been handling the sensor, it might take a little while to settle down to the ambient temperature.

LED bar graph

Next, we'll add our line of five LEDs to the circuit, as in Figure 10-3. From green to red, we've connected their longer legs to GPIO 26, 19, 13, 6, and 5. Create a new file, enter the following code, and save it as test_graph.py.

```
from gpiozero import LEDBarGraph
from time import sleep
graph = LEDBarGraph (26, 19, 13, 6, 5, pwm=True)
```

```
graph.value = 1/10
sleep(1)
graph.value = 3/10
sleep(1)
graph.value = -3/10
sleep(1)
graph.value = 9/10
sleep(1)
graph.value = 95/100
sleep(1)
graph.value = 0
```

At the start, we import the `LEDBarGraph` class from GPIO Zero; this will enable us to use the LEDs to display a bar graph, saving a lot of complex coding. We set the `graph` variable to our LEDs on the GPIO pins and enable PWM so that `LEDBarGraph` can use brightness levels to represent intermediate values, providing a more precise display. We then set `graph.value` to various fractions between 0 and 1 to light the relevant number of LEDs from green to red, including partially lit ones for precision. Note that if the value is negative, it will light the LEDs from the other end, red ones first.

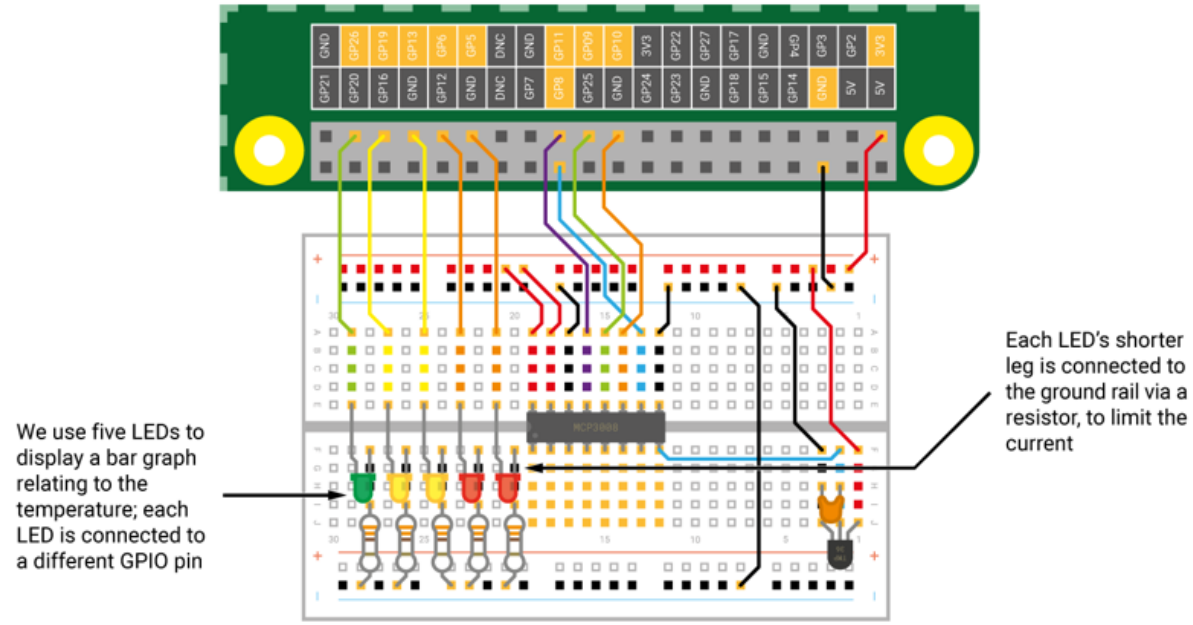


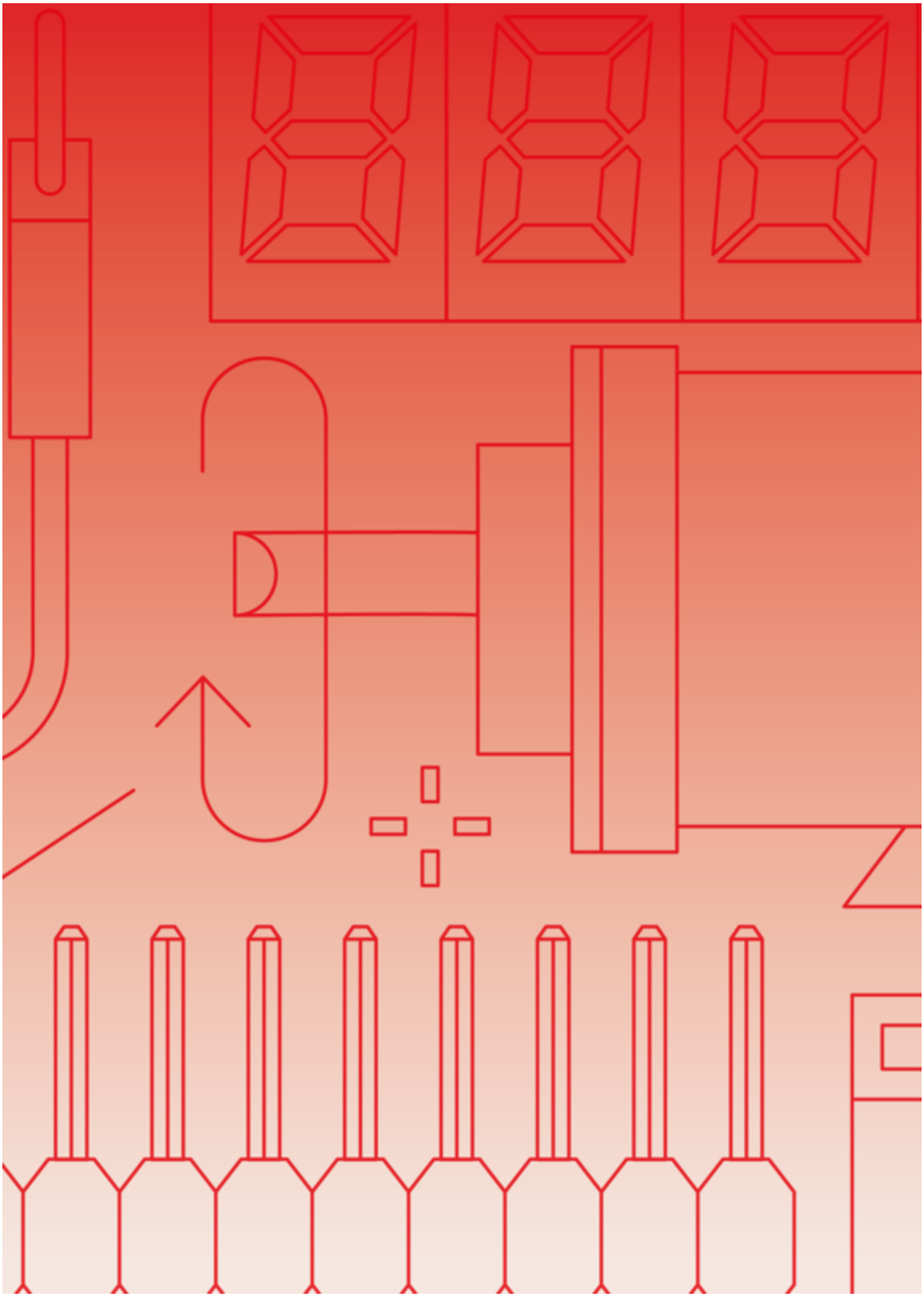
Figure 10-3: Adding the LEDs

Display the temperature

So, we've got our temperature sensor and LED bar graph set up; let's combine them to display the temperature on the LED bar graph. In Thonny, create a new file, enter the code below and save it as graph_temp.py.

```
from gpiozero import MCP3008, LEDBarGraph
from time import sleep

def convert_temp(gen):
  for value in gen:
    yield (value * 3.3 - 0.5) * 100

adc = MCP3008(channel=7)
graph = LEDBarGraph (26, 19, 13, 6, 5, pwm=True)
for temp in convert_temp(adc.values):
  bars = temp / 40
  graph.value = bars
  sleep(1)
```

At the top, we import GPIO Zero's `MCP3008` and `LEDBarGraph` classes, along with the `sleep` function from the `time` library. As in our original code, we then define a function to convert the temperature sensor's readings to degrees Celsius. We set the `adc` variable to channel 7 of the MCP3008 and `graph` to our LEDs' GPIO pins, setting PWM to `true`. Finally, in our `for` loop, we add a `bars` variable to determine how many LEDs are lit in the bar graph. In this example, we've divided `temp` by 40, which is around the maximum temperature for the UK, so if it gets to 40°C, all the LEDs should light up fully. Naturally, you can adjust this number to suit your own location's climate. When ready, run the code and see those LEDs light up to show the current temperature.



Chapter 11

Build a GPIO Zero robot

Control DC motors with GPIO Zero and build a robot

Raspberry Pi robotics is a popular pastime; you can scratch the surface with a basic robot that moves in a predetermined pattern, but you can also go much deeper and build fully autonomous bots. As with many other components, GPIO Zero makes it much simpler to build your own robots especially with its `Motor` and `Robot` classes. We'll take a quick look at those before showing you how you can extend GPIO Zero to build bots with components that aren't supported out-of-the-box.

You'll Need

- 1× 3D-printed or off-the-shelf robot chassis and wheels, or some craft materials
- 1×HC-SR04 ultrasonic sensor
- 2× resistors: 1× 510Ω, 1 or 2× 1kΩ
- 2× 28BYJ-48 stepper motors & ULN2003A driver boards
- 1× solderless breadboard
- 1× half-size solderless breadboard
- Mobile power bank
- Various jumper wires

Connect DC motors

Before we get into this chapter's build, let's look at what GPIO Zero supports without modification. To enable the Raspberry Pi to control DC motors, you'll need some kind of motor driver board; H-bridge controllers like Texas Instruments' DRV8833 work quite well.

You must never connect motors directly to the Raspberry Pi, as this is guaranteed to destroy your Raspberry Pi unless you are lucky enough to have forgotten to wire it up to power. In a typical configuration, you'd connect a motor driver board to the appropriate GPIO pins and connect the motor's two wires to terminals provided on the driver board. The CamJam EduKit 3 is a fantastic all-in-one kit that includes a driver board, motors, a battery box, a distance sensor, and a line-following sensor. You reuse its packaging to build the robot, and best of all, the example code is based on GPIO Zero! For more information, see rpimag.co/edukit3.

Run a motor

GPIO Zero includes a `Motor` class for running bidirectional motors connected via an H-bridge motor driver circuit. Here's an example of how you'd move a motor forward and backward with GPIO 8 connected to an H-bridge's forward input, and GPIO 7 connected to its backward input:

```
from gpiozero import Motor
from time import sleep
motor = Motor(forward=8, backward=7)
while True:
  motor.forward()
  sleep(5)
  motor.backward()
 sleep(5)
```

At the top, we import the `Motor` class from GPIO Zero, along with `sleep` from `time`. We then set the `motor` variable to a `Motor` class on the GPIO pins connected to our motor (GPIO 8 and 7). When we call `motor.forward`, the motor moves forward. Within the brackets, we can add a speed between 0 and 1 (the

default). Similarly, `motor.backward` moves it backward, while `motor.stop` will stop a running motor.

Move a robot

While you can control your motors individually using the `Motor` class, GPIO Zero also includes the `Robot` class for controlling a two-wheeled robot. Here's how you could control a two-wheeled robot with the motor for the left wheel connected to GPIO 8 and 7, and the motor for the right connected to GPIO 10 and 9:

```
from gpiozero import Robot
from time import sleep
robot = Robot(left=(8, 7), right=(10, 9))
for i in range(4):
  robot.forward()
  sleep(1)
  robot.right()
  sleep(0.2)
```

At the top, we import the `Robot` class from GPIO Zero, along with `sleep` from the `time` library. We then set the `robot` object to a `Robot` that has the GPIO pins set for the left and right motors. We can then run various commands to control it, including telling it to spin left or right. In this example, we're using a `for` loop with forward and turn right directions to make it drive around in a square pattern; adjust the `sleep` values to determine the square size. Try altering the directions to make different patterns.

Note

To save connecting your robot's Raspberry Pi to a keyboard and display when you need to modify code, SSH into it from another computer, tablet, or smartphone that's connected to the same wireless network.

Build a ZeroBot

Next, we'll show you how to build a ZeroBot based on a Raspberry Pi Zero and two stepper motors. The 28BYJ-48 is a cheap but versatile stepper motor that can normally be bought with a ULN2003A driver board for under £4. Stepper motors can be programmed to move in discrete steps, rather than just turned on/off like other motors. They also have a lot more torque, so if you need to build a bot that can drive over obstacles, or even cross from a bare floor to a carpet without flipping over, you'll be pleased with stepper motors. They may not be as fast as some other motors, but they will get your robot from point A to point B.

Using the Raspberry Pi Zero, you'll be able to control the speed and positioning of the motors very accurately. To cause the motor to rotate, you provide a sequence of 'high' and 'low' levels to each of the four inputs. The direction can then be reversed by reversing the sequence. In the case of the 28BYJ-48, there are four- step and eight-step sequences. The four-step is faster, but the torque is lower.

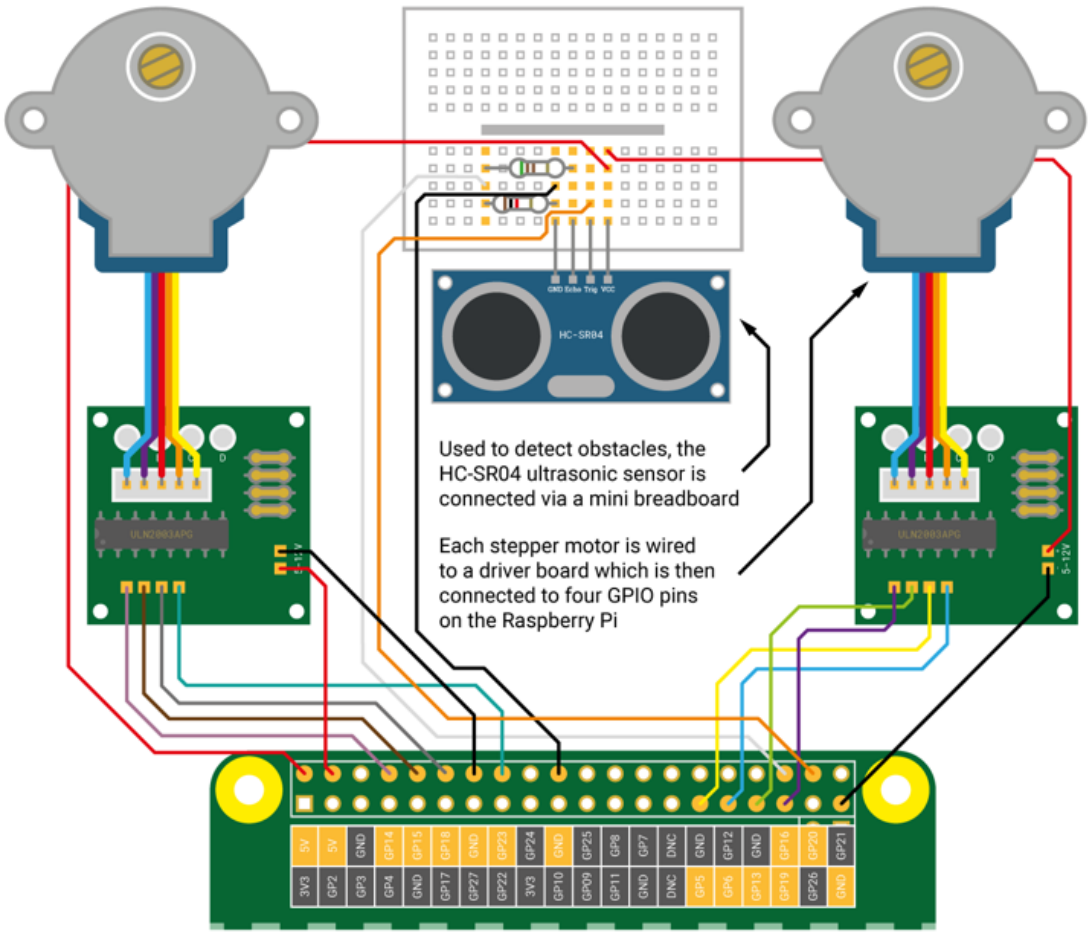


Figure 11-1: Wiring the ZeroBot

Power down your Raspberry Pi, disconnect it from power, and wire everything up as shown in Figure 11-1. Each motor has a connector block at the end of its coloured wires that slots into the white header on the ULN2003A. The GPIO pins controlling that motor connect to the four input pins below the IC, while the 5V power and ground connections go to the bottom two pins on the right.

Eyes to see

We'll give our ZeroBot some simple 'eyes' that allow it to detect obstacles, courtesy of the HC-SR04 ultrasonic sensor. This has four pins, including ground (GND) and 5V supply (VCC). Using Python, you can tell the Raspberry Pi to send an input signal to the Trigger Pulse Input (TRIG) by

setting a GPIO pin's output to HIGH. This will cause the sensor to send out an ultrasonic pulse to bounce off nearby objects. The sensor detects these reflections, measures the time between the trigger and returned pulse, and then sets a 5V signal on the Echo Pulse Output (ECHO) pin. Python code can measure the time between output and return pulses. Connect the HC-SR04 as shown. Its ECHO output is rated at 5V, which will damage the Raspberry Pi. To reduce this to 3V, use two resistors to create a simple voltage divider circuit, as shown in the diagram.

Once you have all your components connected, you can test the code on a bench before building the full robot. Point the 'eyes' away from you and run the code shown next. The red LEDs on the ULN2003 board should flash and both motors should start turning. Our example has the bot move in a square. Check that the motors behave accordingly then rerun the code, but this time place your hand a couple of centimetres in front of the HC-SR04 and check that everything stops.

At the time of this writing, GPIO Zero didn't include a class for stepper motors, but some community members have built their own. One of our favourites is Glenn Fabia's Stepper class, which you can find at rpimag.co/gzstepper. Download the gpiostepper.py file and put it in the same directory as the code you're about to run. Create a new program, enter the following code, and save it as zerobot.py.

```
from gpiozero import DistanceSensor
from gpiostepper import Stepper

sensor = DistanceSensor(echo=16, trigger=20)

# Steps per revolution and gear ratio
# from the 28BYJ-48 datasheet.
num_steps = 32
gear_ratio = 64
geared_steps = num_steps * gear_ratio

step_motor_l = Stepper(motor_pins=[14, 15, 18, 23],
```

```
                number_of_steps=num_steps)
step_motor_r = Stepper(motor_pins=[19, 13, 5, 6],
                number_of_steps=num_steps)

def move(direction='F', ctr=geared_steps):
  step_dir = 1 # 1 for fwd, -1 for back
  if direction == 'B':
    step_dir = -step_dir

  while ctr > 0:
    if sensor.distance > .1: # move if no obstacles
      # F=fwd, B=back, L=left, R=right
      if direction in ['L', 'B', 'F']:
        step_motor_l.step(step_dir) # Left wheel only
      if direction in ['R', 'B', 'F']:
        step_motor_r.step(-step_dir) # Right wheel only
      ctr -= 1

for i in range(4): # Draw a right-handed square
  move("F", geared_steps*2) # move forward two revolutions
  move("R", geared_steps)  # Turn right 1/2 revolution
```

At the top of the program, we import the `DistanceSensor` class from `gpiozero` and the `Stepper` class from `gpiostepper`. Have a look at the gpiostepper.py source to see how GPIO Zero devices are implemented. Next, we define the number of steps per revolution, taking into account the gear ratio inside the stepper motor. After that, we instantiate a left and right motor, and define a function called `move`, which moves the robot. You pass in a direction (F for forward, B for backward, L to turn left, and R to turn right) and number of steps. We set the step direction (`step_dir`) as `+1` when moving forward, and `-1` when moving backward. Next, we count down from the number of steps passed in, and so long as nothing is in the way of the distance sensor, we move forward. Because the wheels are oriented opposite each other, we reverse direction for the right-hand motor.

Back in the main body of the code, we run a loop four times. Each time through, we move forward through two full revolutions of the wheels, turn right one-half revolution, and then that's it. Your ZeroBot should have traced out a square with its movements!

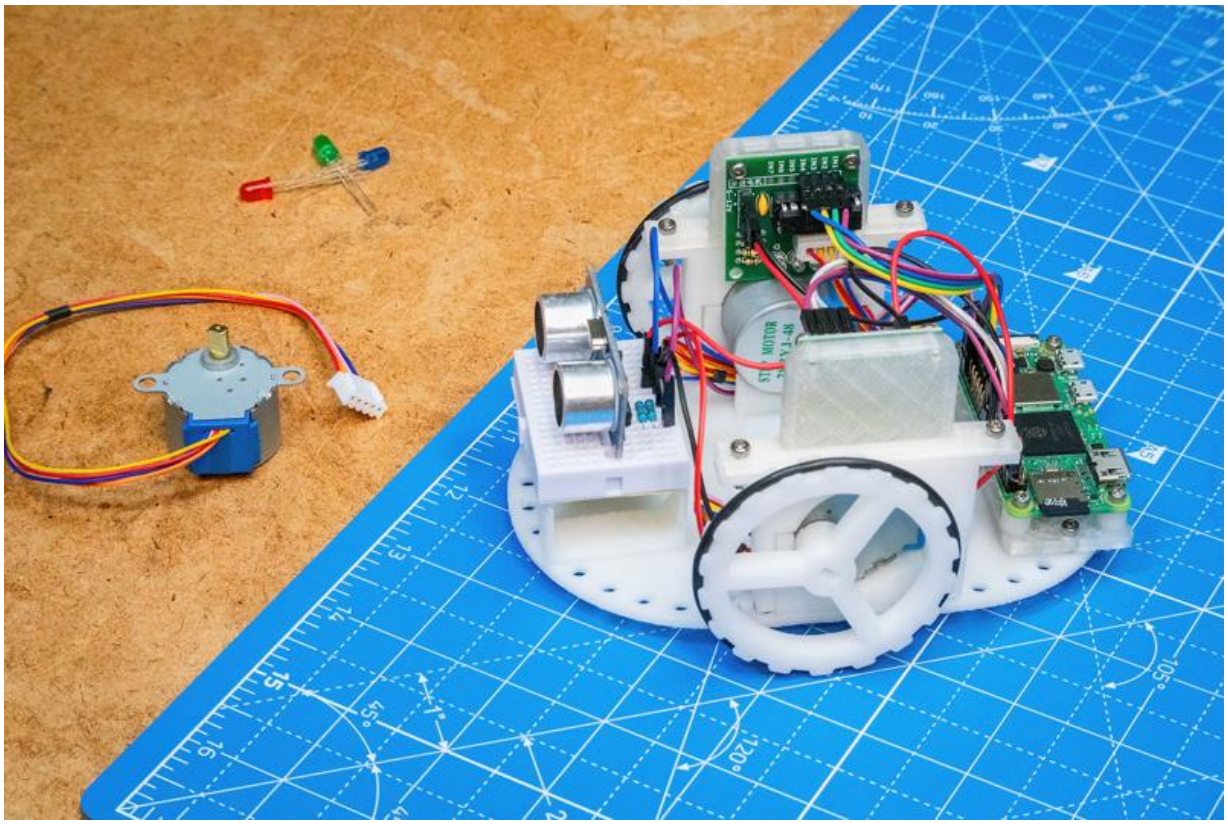


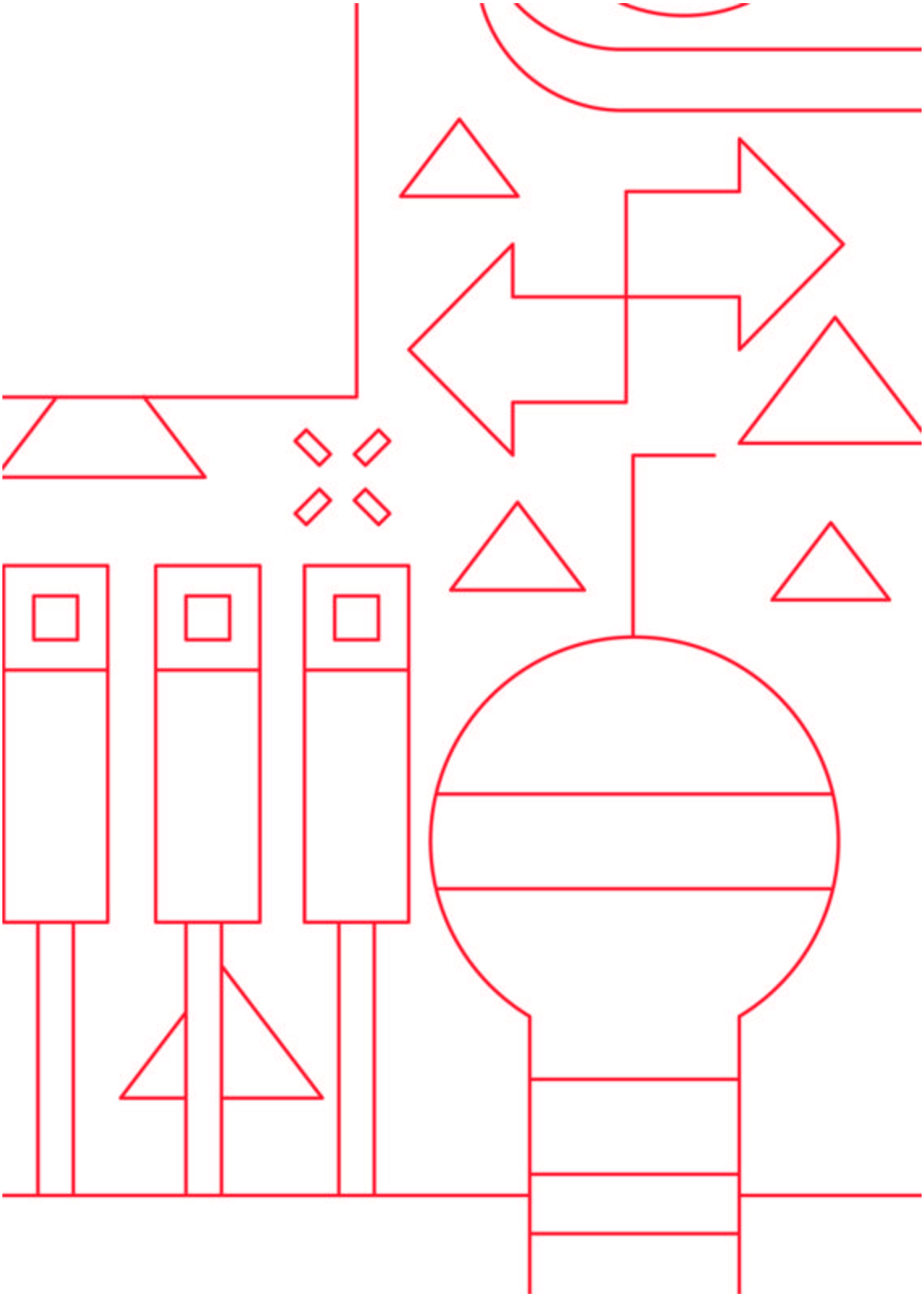
Figure 11-2: The diminutive ZeroBot features a Raspberry Pi Zero, two stepper motors, and a 3D-printed chassis

Give it a body

Now it's time to give the bot a body. If you have access to a 3D printer, you can print the parts for the ZeroBot. This design fits together easily, although you do need to glue the chassis end-caps in place. Alternatively, you could construct a similar design using reasonably thick cardboard for the wheels and part of a plastic bottle as the main tubular chassis. Use more cardboard for the end-caps.

Put your mobile power bank at the bottom of the chassis tube, then attach the motors to the end-caps with screws. Next, place the ULN2003A boards on

top of the power bank, and then sit the breadboard with the HC-SR04 ‘eyes’ on top. Finally, slot the Raspberry Pi Zero in at the back. All nice and cosy, and ready to roll!



Appendix A

Output devices

GPIO Zero includes a range of classes that make it easy to control output components such as LEDs, buzzers, and motors

This appendix is a reference for the output device classes we used or mentioned in this book. For a complete list, see rpimag.co/gzOutput.

LED

```
gpiozero.LED(pin, active_high=True, initial_value=False)
```

Turns an LED on and off. Connect the LED's longer leg (anode) to a GPIO pin, and the other leg (cathode) to GND via a current-limiting resistor. The following lights an LED connected to GPIO 17:

```
from gpiozero import LED
led = LED(17)
led.on()
```

Methods:

on()

Turn the device on.

off()

Turn the device off.

blink(on_time=1, off_time=1, n=None, background=True)

Make the device turn on and off repeatedly.

toggle()

Reverse the state of the device; if on, it'll turn off, and vice versa.

Properties:

is_lit

Returns True if the device is currently active, and False otherwise.

pin

The GPIO pin that the device is connected to.

value

The state of the LED (1 for active, 0 for inactive).

PWMLED

```
gpiozero.PWMLED(pin, active_high=True,
        initial_value=0, frequency=100)
```

Lights an LED with variable brightness. As before, use a resistor to limit the current. This will light an LED connected to GPIO 17 at half brightness:

```
from gpiozero import PWMLED
led = PWMLED(17)
led.value = 0.5
```

Methods:

on()

Turn the device on.

off()

Turn the device off.

`blink(on_time=1, off_time=1, fade_in_time=0, fade_out_time=0, n=None, background=True)`

Make the device turn on and off repeatedly.

`pulse(fade_in_time=1, fade_out_time=1, n=None, background=True)`

Make the device fade in and out repeatedly.

`toggle()`

Toggle the state of the device. If it's currently off (`value` is 0.0), this changes it to fully on (1.0). If the `value` is 0.1, this will toggle it to 0.9, and so on.

Properties:

`is_lit`

Returns True if the device is currently active, and False otherwise.

`pin`

The GPIO pin that the device is connected to.

`value`

The duty cycle of the PWM device, from 0.0 (off) to 1.0 (fully on).

RGBLED

```
gpiozero.RGBLED(red, green, blue, active_high=True,
        initial_value=(0, 0, 0), pwm=True)
```

As shown in *Chapter 5, Measure CPU usage with an RGB LED*, this class is used to light a full-colour LED (composed of red, green, and blue LEDs). Connect its longest leg (cathode) to GND, and the other to GPIO pins via resistors (or use one on the cathode). The following code will make the LED purple:

```
from gpiozero import RGBLED
```

```
led = RGBLED(2, 3, 4)
led.color = (1, 0, 1)
```

Methods:

on()

Turn the device on: equivalent to setting the colour to white (1, 1, 1).

off()

Turn the device off: equivalent to setting the colour to none (0, 0, 0).

```
blink(on_time=1, off_time=1, fade_in_time=0,
    fade_out_time=0, on_color=(1, 1, 1),
    off_color=(0, 0, 0), n=None, background=True)
```

Make the device turn on and off repeatedly.

```
pulse(fade_in_time=1, fade_out_time=1, on_color=(1, 1, 1),
    off_color=(0, 0, 0), n=None, background=True)
```

Make the device fade in and out repeatedly.

toggle()

Toggles the LED. If it's off (value is (0, 0, 0)), this changes it to fully on (1, 1, 1). If it's set to a specific colour, this inverts it.

Properties

value

The colour as (red, green, blue). If pwm=True, each value is between 0 and 1 (otherwise 0 *or* 1). For example, purple is (1, 0, 1) and orange is (1, 0.5, 0). The red, green, and blue properties are also available.

is_lit

True if the LED is active (value is not (0, 0, 0)) otherwise False.

Buzzer

```
gpiozero.Buzzer(pin, active_high=True,
        initial_value=False)
```

This class is used to control a piezo buzzer. This example will sound a buzzer connected to GPIO pin 3:

```
from gpiozero import Buzzer
bz = Buzzer(3)
bz.on()
```

Methods:

on()

Turn the device on.

off()

Turn the device off.

beep(on_time=1, off_time=1, n=None, background=True)

Make the device turn on and off repeatedly.

toggle()

Reverse the state of the device; if on, it'll turn off, and vice versa.

Properties:

is_active

Returns True if the device is currently active, and False otherwise.

pin

The GPIO pin that the device is connected to.

Motor

```
gpiozero.Motor(forward, backward, pwm=True)
```

Drives a motor connected via an H-bridge motor controller. This code turns a motor (connected to GPIO 17 and 18) 'forward':

```
from gpiozero import Motor
motor = Motor(17, 18)
motor.forward()
```

Methods:

backward(speed=1)

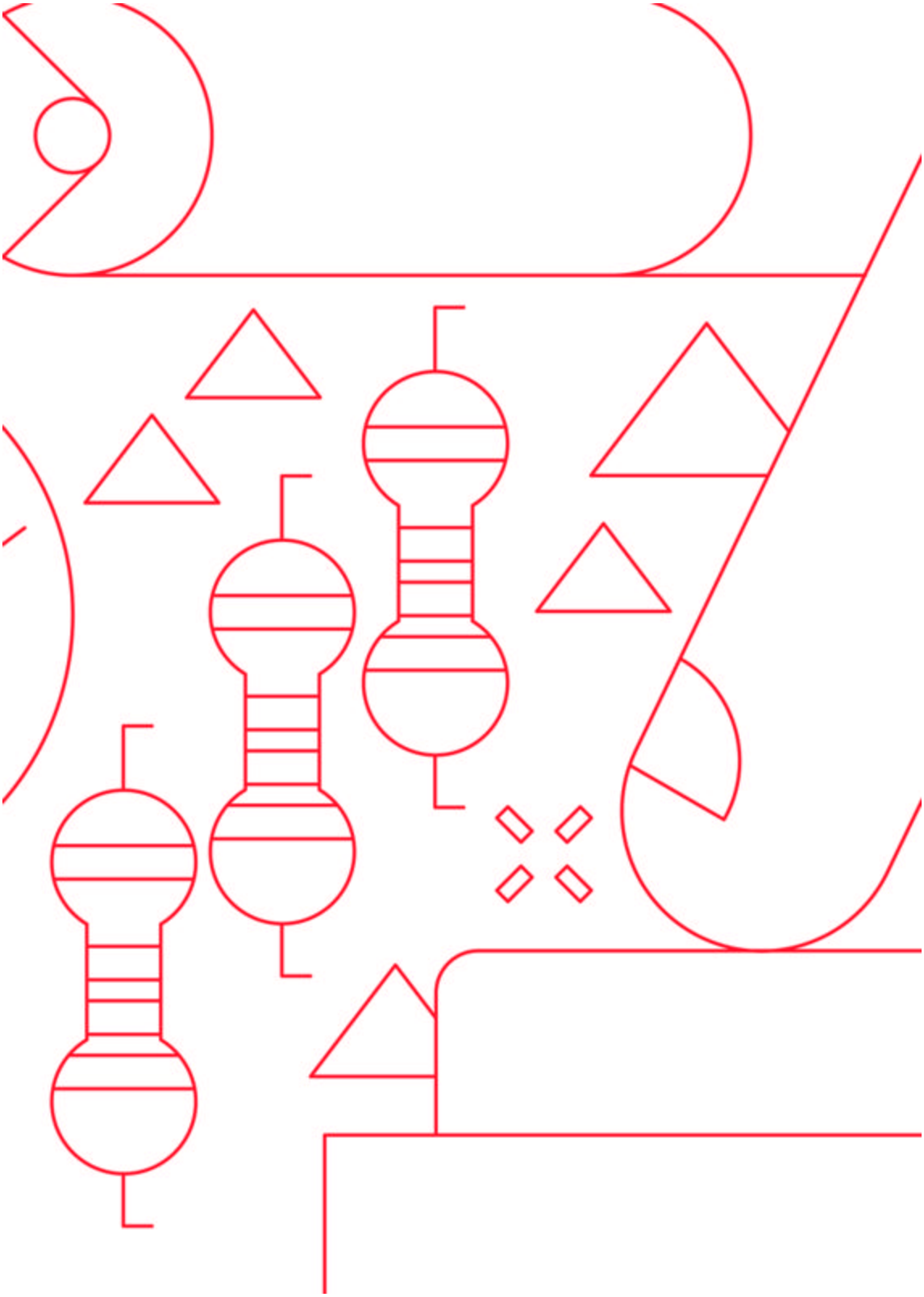
Reverse the motor. Speed is between 0 and 1 (if pwm=True), otherwise 0 or 1.

forward(speed=1)

Drive the motor forward.

stop()

Stop the motor.



Appendix B

Input devices

The GPIO Zero module includes a range of classes that make it easy to obtain values from input devices such as buttons and sensors

This appendix is a reference for the input device classes we used or mentioned in this book. For a complete list, see rpimag.co/gzInput.

Button

```
gpiozero.Button(pin, pull_up=True, bounce_time=None)
```

Use this class with a simple push button or switch. The following example will print a line of text when a button connected to GPIO 4 is pressed:

```
from gpiozero import Button
button = Button(4)
button.wait_for_press()
print("The button was pressed!")
```

Methods:

wait_for_press(timeout=None)

Pause the script until the button is pressed or the timeout (in seconds) is reached.

wait_for_release(timeout=None)

Pause until the button is released or the timeout is reached.

Events:

when_pressed

The function to run when the device goes from inactive to active.

when_released

The function to run when the device goes from active to inactive.

when_held

The function to run when the device has remained active for hold_time seconds.

Properties:

hold_time

The length of time (in seconds) to wait after the device is activated, until executing the when_held handler. If hold_repeat is True, this also sets the length of time between calls to when_held.

hold_repeat

If True, when_held will be executed repeatedly with hold_time seconds between each call.

held_time

The length of time (in seconds) that the device has been held for.

is_held

When True, the device has been active for at least hold_time seconds.

is_pressed

Returns True if the device is currently active, and False otherwise.

pin

The GPIO pin that the device is connected to.

pull_up

If True, the device uses a pull-up resistor to set the GPIO pin 'HIGH' by default.

value

1 if the button is pressed, otherwise 0.

LineSensor

`gpiozero.LineSensor(pin)`

This class is used to read a single pin line sensor, like the TCRT5000 found in the CamJam EduKit #3. The following example will print a line of text indicating when the sensor (with its output connected to GPIO 4) detects a line, or stops detecting one:

```
from gpiozero import LineSensor
from signal import pause
sensor = LineSensor(4)
sensor.when_line = lambda: print('Line detected')
sensor.when_no_line = lambda: print('No line detected')
pause()
```

Methods:

wait_for_line(timeout=None)

Pause the script until the device is deactivated, or the timeout (in seconds) is reached.

wait_for_no_line(timeout=None)

Pause the script until the device is activated, or the timeout (in seconds) is reached.

Events:

when_line

The function to run when the device goes from active to inactive.

when_no_line

The function to run when the device goes from inactive to active.

Properties:

pin

The GPIO pin that the device's output is connected to.

value

1 if a line was detected, otherwise 0.

Motion Sensor

```
gpiozero.MotionSensor(pin, queue_len=1,
            sample_rate=10, threshold=0.5,
            partial=False)
```

As shown in *Chapter 6, Make a motion-sensing alarm*, this class is used with a passive infrared (PIR) sensor. The following example will print a line of text when motion is detected by the sensor (with its middle output leg connected to GPIO pin 4):

```
from gpiozero import MotionSensor
pir = MotionSensor(4)
pir.wait_for_motion()
print("Motion detected!")
```

Methods:

wait_for_motion(timeout=None)

Pause the script until the device is activated, or the timeout (in seconds) is reached.

wait_for_no_motion(timeout=None)

Pause the script until the device is deactivated, or the timeout (in seconds) is reached.

Events:

motion_detected

Returns True if the device is currently active, and False otherwise.

when_motion

The function to run when the device goes from inactive to active.

when_no_motion

The function to run when the device goes from active to inactive.

Properties:

pin

The GPIO pin that the device's output is connected to.

value

1 when motion has been detected, otherwise 0.

Light Sensor

```
gpiozero.LightSensor(pin, queue_len=5,
            charge_time_limit=0.01,
            threshold=0.1, partial=False)
```

As shown in *Chapter 8, Make a laser tripwire*. Connect one leg of the LDR to the 3V3 pin; connect one leg of a 1µF capacitor to a ground pin; connect the other leg of the LDR and the other leg of the capacitor to the same GPIO pin. This class repeatedly discharges the capacitor, then times the duration it takes to charge, which will vary according to the light falling on the LDR. The following code will print a line of text when light is detected:

```
from gpiozero import LightSensor
ldr = LightSensor(18)
ldr.wait_for_light()
print("Light detected!")
```

Methods:

wait_for_dark(timeout=None)

Pause the script until the device is deactivated, or the timeout (in seconds) is reached.

wait_for_light(timeout=None)

Pause the script until the device is activated, or the timeout (in seconds) is reached.

Events:

light_detected

Returns True if the device is currently active, and False otherwise.

when_dark

The function to run when the device goes from active to inactive.

when_light

The function to run when the device goes from inactive to active.

Properties:

pin

The GPIO pin that the device is connected to.

value

A value between 0 (dark) and 1 (light).

Distance Sensor

```
gpiozero.DistanceSensor(echo, trigger, queue_len=30,
                        max_distance=1, threshold_distance=0.3,
                        partial=False)
```

As shown in *Chapter 7, Make a range finder*, this class is used with a standard HC-SR04 ultrasonic distance sensor. Note: to avoid damaging your Raspberry Pi, you'll need to use a voltage divider on the breadboard to reduce the sensor's output (ECHO pin) from 5V to 3.3V. The following example will periodically report the distance measured by the sensor in cm (with the TRIG pin connected to GPIO 17, and ECHO pin to GPIO 18):

```
from gpiozero import DistanceSensor
from time import sleep
sensor = DistanceSensor(echo=18, trigger=17)
while True:
  print('Distance: ', sensor.distance * 100)
  sleep(1)
```

Methods:

wait_for_in_range(timeout=None)

Pause the script until the device is deactivated, or the timeout is reached.

wait_for_out_of_range(timeout=None)

Pause the script until the device is activated, or the timeout is reached.

Events:

`when_in_range`

The function to run when the device goes from active to inactive.

`when_out_of_range`

The function to run when the device goes from inactive to active.

Properties:

`distance`

Returns the current distance measured by the sensor in metres. Note that this property will have a value between 0 and `max_distance`.

`max_distance`

As specified in the class constructor, the maximum distance that the sensor will measure in metres.

`threshold_distance`

As specified in the class constructor, the distance (measured in metres) that will trigger the `when_in_range` and `when_out_of_range` events when crossed.

`echo`

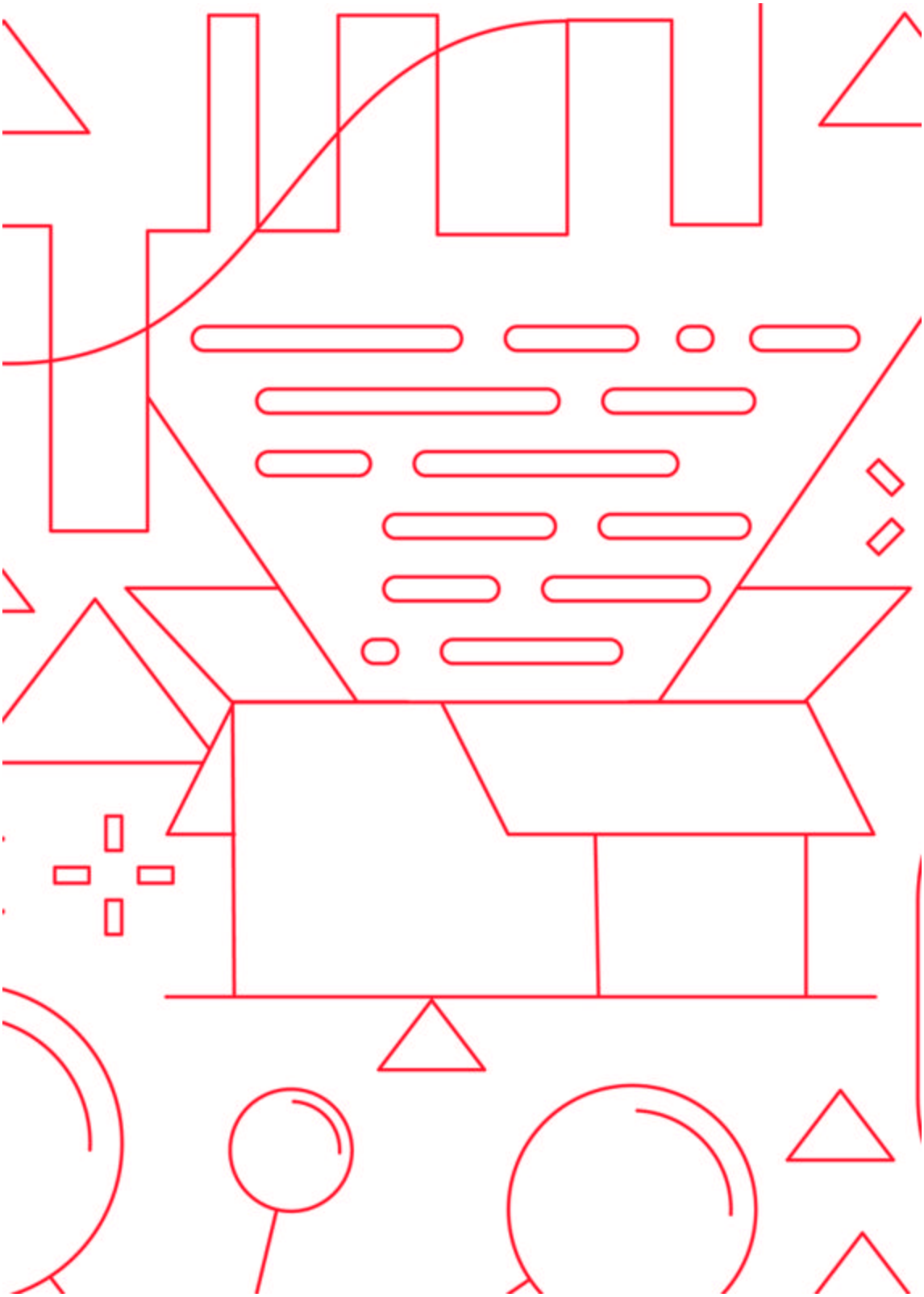
Returns the GPIO pin that the sensor's ECHO pin is connected to.

`trigger`

Returns the GPIO pin that the sensor's TRIG pin is connected to.

`value`

Returns 1 if the object is at or beyond `max_distance`, otherwise 0.



Appendix C

SPI devices

SPI (serial peripheral interface) is a mechanism allowing compatible devices to communicate with the Raspberry Pi. GPIO Zero provides some classes for devices, including a range of analogue-to-digital converters.

When constructing an SPI device, there are two schemes for specifying which pins it's connected to. You can pass these arguments as keyword arguments where you see `**spi_args` in the reference section that follows.

1. You can specify port and device keyword arguments. The `port` parameter must be 0; there's only one user-accessible hardware SPI interface on Raspberry Pi, using GPIO 11 as the clock pin, GPIO 10 as the MOSI pin, and GPIO 9 as the MISO pin. The device parameter must be 0 or 1. If device is 0, the select pin will be GPIO 8; if device is 1, the `select` pin will be GPIO 7.
2. Alternatively, you can specify `clock_pin`, `mosi_pin`, `miso_pin`, and `select_pin` keyword arguments. In this case, the pins can be any four GPIO pins. SPI devices can share clock, MOSI, and MISO pins, but not select pins; the GPIO Zero library will enforce this restriction.

You can't mix these two schemes, but you can omit any arguments from either scheme. The defaults are:

- `port` and `device` both default to 0.
- `clock_pin` defaults to 11, `mosi_pin` defaults to 10, `miso_pin` defaults to 9, and `select_pin` defaults to 8.

Analogue-to-Digital Converters (ADCs)

MCP3001

```
gpiozero.MCP3001(max_voltage=3.3, **spi_args)
```

MCP3002

```
gpiozero.MCP3002(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3004

```
gpiozero.MCP3004(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3008

```
gpiozero.MCP3008(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3201

```
gpiozero.MCP3201(max_voltage=3.3, **spi_args)
```

MCP3202

```
gpiozero.MCP3202(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3204

```
gpiozero.MCP3204(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3208

```
gpiozero.MCP3208(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3301

```
gpiozero.MCP3301(max_voltage=3.3, **spi_args)
```

MCP3302

```
gpiozero.MCP3302(channel=0, differential=False,
    max_voltage=3.3, **spi_args)
```

MCP3304

```
gpiozero.MCP3304(channel=0, differential=False,
                 max_voltage=3.3, **spi_args)
```

GPIO Zero supports a range of ADC chips, with varying numbers of bits (from 10-bit to 13-bit) and channels (1 to 8). As shown in *Chapter 9, Build an internet radio* and *Chapter 10, Create an LED thermometer*, numerous jumper wires are required to connect the ADC via a breadboard to the Raspberry Pi.

Methods:

channel

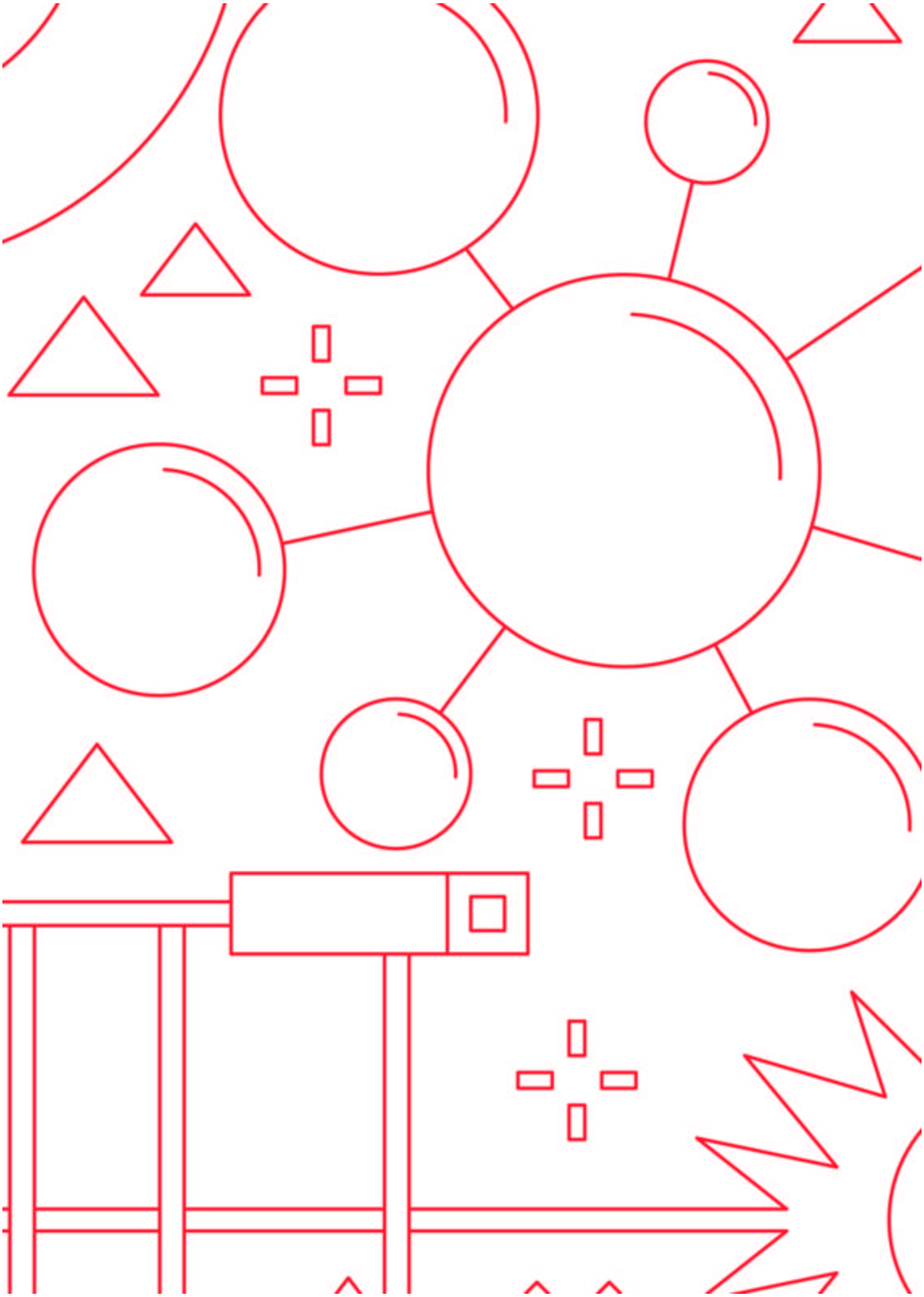
The channel to read data from. The MCP3008/3208/3304 have 8 channels (0-7), while the MCP3004/3204/3302 have 4 channels (0-3), the MCP3002/3202 have 2 channels (0-1), and the MCP3001/3201/3301 only have 1 channel The MCP3301 always operates in differential mode between its two channels, and the output value is scaled from -1 to +1.

differential

If True, the device is operated in pseudo-differential mode. In this mode, one channel (specified by the channel attribute) is read relative to the value of a second channel, informed by the chip's design.

value

The current value read from the device, scaled to a value between 0 and 1 (or -1 to +1 for devices operating in differential mode).



Appendix D

Boards, accessories, and system devices

To make things even easier, GPIO Zero provides extra support for a range of add-on devices, component collections, and even internal hardware

This appendix is a reference for the board, accessory, and internal device classes we used or mentioned in this book. For a complete list, see rpimag.co/gzAccessories and rpimag.co/gzInternal.

LEDBoard

```
gpiozero.LEDBoard(*pins, pwm=False, active_high=True,
          initial_value=False, **named_pins)
```

This class enables you to control a generic LED board or collection of LEDs. The following example turns on all the LEDs on a board containing five LEDs attached to GPIO pins 2 through 6:

```
from gpiozero import LEDBoard
leds = LEDBoard(2, 3, 4, 5, 6)
leds.on()
```

Methods:

on(*args)

Turn all the output devices on. You can specify the index (starting at 0) of the LEDs you wish to turn on. 0 will turn on the first, -1 will turn on the last. You can pass more than one index.

`off(*args)`

Turn all the output devices off. You can also specify one or more LEDs by their index order.

```
blink(on_time=1, off_time=1, fade_in_time=0,
    fade_out_time=0, n=None, background=True)
```

Make all the LEDs turn on and off repeatedly.

```
pulse(fade_in_time=1, fade_out_time=1,
    n=None, background=True)
```

Make the device fade in and out repeatedly.

`toggle(*args)`

Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on. You can also specify one or more LEDs by their index order.

Properties:

`leds`

All LEDs contained in this collection (and all sub-collections).

`source`

The iterable to use as a source of values for `value`.

`source_delay`

The delay (measured in seconds) in the loop used to read values from `source`. Defaults to 0.01 seconds.

`value`

A value for each LED in the board. This property can also be set to update the state of all LEDs.

`values`

An infinite iterator of values read from `value`.

LEDBarGraph

```
gpiozero.LEDBarGraph(*pins, initial_value=0)
```

As shown in *Chapter 10, Create an LED thermometer*, this is a class for controlling a line of LEDs to represent a bar graph. Positive values (0 to 1) light the LEDs from first to last. Negative values (-1 to 0) light the LEDs from last to first. The following example demonstrates turning on the first two and last two LEDs in a board containing five LEDs attached to GPIOs 2 through 6:

```
from gpiozero import LEDBarGraph
from time import sleep
graph = LEDBarGraph(26, 19, 13, 6, 5, pwm=True)
graph.value = 2/5 # Light the first two LEDs only
sleep(1)
graph.value = -2/5 # Light the last two LEDs only
sleep(1)
graph.off()
```

Methods:

on()

Turn all the output devices on.

off()

Turn all the output devices off.

toggle()

Toggle all the output devices. For each device, if it's on, turn it off; if it's off, turn it on.

Properties

lit_count

The number of LEDs that are lit up. If the LEDs were lit from last to first, this number may be negative.

leds

All LEDs contained in this collection (and all sub-collections).

source

The iterable to use as a source of values for value.

source_delay

The delay (measured in seconds) in the loop used to read values from source. Defaults to 0.01 seconds.

value

A value for each LED. This property can also be set to update the state of all LEDs in the graph. To light a particular number of LEDs, simply divide that number by the total number of LEDs.

values

An infinite iterator of values read from value.

Robot

```
gpiozero.Robot(left=None, right=None)
```

Designed to control a generic dual-motor robot (as seen in *Chapter 11, Build a GPIO Zero robot*), this class is constructed with two tuples representing the forward and backward pins of the left and right controllers respectively. For example, if the left motor's controller is connected to GPIOs 4 and 14, while the right motor's controller is connected to GPIOs 17 and 18, then the following example will drive the robot forward:

```
from gpiozero import Robot
robot = Robot(left=(4, 14), right=(17, 18))
```

```
robot.forward()
```

Methods:

forward(speed=1, curve_left=0, curve_right=0)

Drive the robot forward by running both motors forward.

backward(speed=1, curve_left=0, curve_right=0)

Drive the robot backward by running both motors backward.

left(speed=1)

Make the robot turn left by running the right motor forward and left motor backward.

right(speed=1)

Make the robot turn right by running the left motor forward and right motor backward.

reverse()

Reverse the robot's current motor directions. If the robot is currently running full speed forward, it will run full speed backward. If the robot is turning left at half speed, it will turn right at half speed. If the robot is currently stopped, it will remain stopped.

stop()

Stop the robot.

Properties:

source

The iterable to use as a source of values for value.

source_delay

The delay (measured in seconds) in the loop used to read values from source. Defaults to 0.01 seconds.

value

Represents the motion of the robot as (left_motor_speed, right_motor_speed), with (-1, -1) representing full speed backward, (1, 1) representing full speed forward, and (0, 0) representing stopped.

values

An infinite iterator of values read from value.

TimeOfDay

```
gpiozero.TimeOfDay(start_time, end_time,
          utc=True, event_delay=10)
```

This device becomes active when your Raspberry Pi's system clock falls between start_time and end_time. Here's how you could turn on an LED connected to GPIO 2 between 6 and 7am:

```
from gpiozero import LED, TimeOfDay
from datetime import time
from signal import pause

led = LED(2)
alarm_time = TimeOfDay(time(6), time(7), utc=False)

alarm_time.when_activated = led.on
alarm_time.when_deactivated = led.off
pause()
```

Events:

when_activated

The function to call when `start_time` is reached.

`when_deactivated`

The function to call when `end_time` is reached.

Properties:

`start_time`

The device will activate after this time.

`end_time`

The device will become inactive after this time.

`utc`

If True, the device is using UTC time; if False, then it's using the local time zone.

`value`

This will be 1 when the Raspberry Pi system clock is between the start and end time, 0 otherwise.

CPUTemperature

```
gpiozero.CPUTemperature(sensor_file=
        "/sys/class/thermal/thermal_zone0/temp",
        min_temp=0.0, max_temp=1.1,
        threshold=80.0, event_delay=5)
```

This device represents the current CPU temperature. The following code will light an LED bar graph to indicate where the CPU temperature falls along the range of 20 and 90 C:

```
from gpiozero import LEDBarGraph, CPUTemperature
from signal import pause
```

```
cputemp = CPUTemperature(min_temp=20, max_temp=90)
graph = LEDBarGraph (26, 19, 13, 6, 5, pwm=True)
graph.source = cputemp
pause()
```

Events:

when_activated

The function to call when the temperature reaches threshold.

when_deactivated

The function to call when the temperature falls below threshold.

Properties:

is_active

True when the temperature has exceeded threshold.

temperature

CPU Temperature in degrees Celsius.

value

A value between 0.0 and 1.0 that represents the CPU temperature between min_temp and max_temp. If the current temperature is less than min_temp, this value will be negative.

DiskUsage

```
gpiozero. DiskUsage (filesystem="/", threshold=90.0,
            event_delay=30)
```

This device represents available disk space. The following code illuminates an LED bar graph based on how full the `/` filesystem is:

```
from gpiozero import LEDBarGraph, DiskUsage
from signal import pause

disk = DiskUsage()

graph = LEDBarGraph(26, 19, 13, 6, 5, pwm=True)
graph.source = disk
pause()
```

Events:

when_activated

The function to call when the disk usage has exceeded threshold.

when_deactivated

The function to call when the disk usage has fallen below threshold.

Properties:

is_active

Returns True if the disk usage has exceeded threshold.

usage

Current disk usage as a percentage of total disk space.

value

The current disk usage as a value between 0.0 and 0.1.

Simple electronics with GPIO Zero