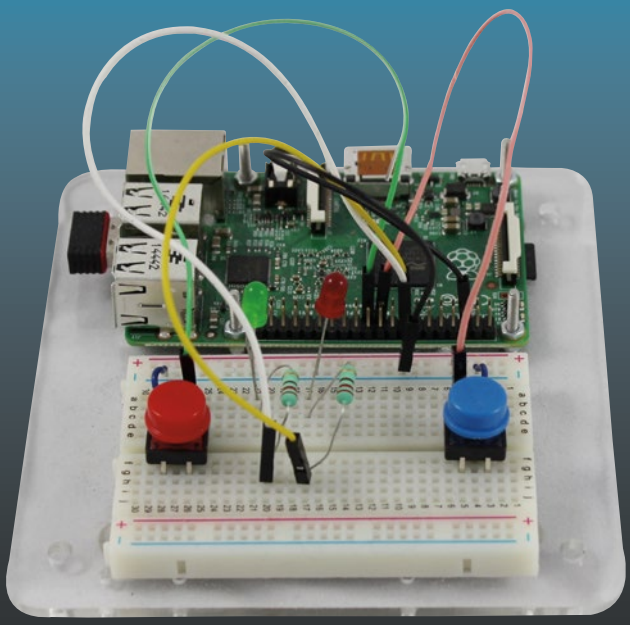
TECHNOLOGY IN ACTION™



Learn Electronics with Raspberry Pi

Physical Computing with Circuits, Sensors, Outputs, and Projects

Stewart Watkiss

Apress®

Learn Electronics with Raspberry Pi

Physical Computing with Circuits, Sensors, Outputs, and Projects

Stewart Watkiss

Apress®

Learn Electronics with Raspberry Pi: Physical Computing with Circuits, Sensors, Outputs, and Projects

Stewart Watkiss
Redditch
United Kingdom

ISBN-13 (pbk): 978-1-4842-1897-6 ISBN-13 (electronic): 978-1-4842-1898-3
DOI 10.1007/978-1-4842-1898-3

Library of Congress Control Number: 2016943841

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Chaim Krause
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484218976. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

For Amelia and Oliver

Contents at a Glance

Contents

About the Author

Stewart Watkiss has been a keen electronics hobbyist since the early 1990s. He first studied electronic engineering at Huddersfield Technical College and subsequently at the University of Hull, where he earned a master's degree in electronic engineering.

He then became more involved in the software side, studying the Linux operating system and programming. During this time he launched a web site (`www.penguintutor.com`) to teach Linux and to help those working toward Linux certification.

His interest in electronics was revitalized thanks in part to the Raspberry Pi. Stewart created a number of projects, some of which have been featured on the Raspberry Pi blog and *The MagPi* magazine. Stewart also volunteers as a STEM Ambassador, going into local schools to help support teachers and teach programming and physical computing to teachers and children.

About the Technical Reviewer

Chaim Krause presently lives in Leavenworth, Kansas, where the U.S. Army employs him as a simulation specialist. In his spare time, he likes to play PC games and occasionally develops his own. He has recently taken up the sport of golf to spend more time with his significant other, Ivana. Although he holds a bachelor's in political science from the University of Chicago, Chaim is an autodidact when it comes to computers, programming, and electronics. He wrote his first computer game in BASIC on a Tandy Model I Level I and stored the program on a cassette tape. Amateur radio introduced him to electronics, while the Arduino and the Raspberry Pi provided a medium to combine computing, programming, and electronics into one hobby.

Acknowledgments

My family has been very supportive during the writing of this book. Thank you to my wife Sarah for her support and especially to my children, Amelia and Oliver, who have been both a source of inspiration and enthusiastic testers of the games and activities.

I'd also like to thank the team behind the Raspberry Pi, including the Raspberry Pi foundation and the community that has grown around it. The Raspberry Pi has reinvigorated my love of electronics, making it possible to interact with hobby electronics projects in a way that I'd only dreamed about before. Raspberry Jams and community events have been a great way to meet the team behind the Raspberry Pi as well as other members of the community. All this has encouraged me to pursue it further.

There have been many other people have helped in the making of this book. Special thanks to the technical reviewer, Chaim Krause, who has tested all the projects. Michelle Lowman encouraged me to write this book and Mark Powers ensured that the book moved forward. I'd also like to thank Corbin Collins for his comments and suggestions and to the rest of the team at Apress.

Introduction

Learning computer programming is fun in itself, but when the computer is connected to external sensors and outputs, your programs can interact with the real world. This is known as *physical computing* and it opens up the opportunity to create some fun projects.

I am a big fan of learning by doing. It's much easier to learn when you get to make the projects rather than just read about what other people have been done. It's even better when those projects are fun. This book covers simple projects that you can do at home to make games, control toys, create your own films, or just have fun.

The Raspberry Pi computer is great for learning physical computing, thanks to special pins that provide access to ports on the processor. These 40 pins (26 on earlier versions) provide a simple way to extend computing into the physical world. The circuits in this book are adding sensors, outputs, and electronic circuits to a Raspberry Pi. With a little bit of programming, these can do some pretty amazing things.

We will start with some simple circuits, which can be controlled from Scratch, and then move up to Python and some more complicated circuits. By then end of the book, we will have covered enough so that you can start designing your own circuits.

Most of the circuits can be created by plugging wires into a solderless breadboard, but there are tips on how to solder, which opens the possibilities further. You'll learn how to design custom circuit boards and look at how you can use some of the common Raspberry Pi add-on boards and HATs.

Who Is This Book For?

This book is for anyone who wants to learn about electronics and have fun in the process. This book focuses on fun projects so is great for older children and young adults. My eight-year-old son helped with some of the easier projects so young children could have a go with adult help. While the fun aspect appeals to younger adults, there's no maximum age for having fun, so this is just as useful to adults of any age who want to learn about electronic circuits and connecting to the Raspberry Pi.

You don't need to know anything about electronic circuits before you start. Having a basic knowledge about computers and computer programming will be useful, but is not required and will be explained as we go. We'll be using Scratch and Python, as they are good programming languages for those learning programming, but the electronic circuits can be controlled using any programming language that can communicate with the GPIO ports.

How to Use This Book

As with many books, you can read this book from cover to cover, or you can jump straight to the project that you find most interesting. The book introduces new concepts and components in each chapter. The first few chapters start with simple circuits and work through to more complicated circuits. There is an explanation of each circuit as you go, so for the first few chapters you will find it most useful to follow in order, making the circuits as you go. Most of the circuits in these first few chapters are based on low-cost components that should be within the reach of most readers.

Most of the projects are within a single chapter, but a few need some of the concepts explained in later chapters and so are split between the chapters. The notes explain which chapter you will need to refer to for the rest of the project.

Some of the later projects do use more expensive components or add-on boards for the Raspberry Pi or are designed to interact with more expensive toys. Where possible, boards have been chosen to keep costs as low as possible. You may want to just read about how the circuit works or look for the suggestions on how these can be adapted for use with cheaper components or items you already own.

Creating the examples in this book should not be considered the end. I hope that the information in this book will provide you with the inspiration and knowledge to go on to learn more about electronics and design your own circuits.

After working through the projects, it's useful to have a summary of the components so that you can refer to the book when designing your own circuits. To make it easier to refer back, I've added a summary of some of the components in the appendix along with some extra technical details that are useful when designing your own circuits.

Is Soldering Required?

When I've talked to teachers and students about electronic projects, I often get asked the question of whether soldering is required. Unfortunately, I think that this is something that puts some people off from learning electronics. I don't think it should.

First, a lot can be learned through creating circuits that don't need any soldering. Many of the projects in the first few chapters, and in some of the subsequent chapters, are designed to be made without any soldering. These are typically using solderless breadboards, but some can also be made using crocodile clips or with an inexpensive crimp tool. There are, however, some components that are not suitable for use on a breadboard, or that need a small amount of soldering so that they can be used with a breadboard. In fact many "breadboard friendly" components may need headers to be soldered on to them first.

The second point I'd like to make is that soldering is not as hard, expensive, or dangerous as some people have been led to believe. Chapter 10 explains soldering and will hopefully dispel some of the myths surrounding it. If you are still uncertain, see if you have a local maker club or Hackspace where you can speak to someone experienced in soldering.

Buying a Raspberry Pi

If you are reading this book then there is a good chance that you already have a Raspberry Pi. Since the Raspberry Pi was first released in 2012 there have been several versions. Some have had only minor changes, but one of the bigger changes was increasing the size of the GPIO connectors from 26 to 40. This was introduced in the Raspberry Pi B+ and the larger connector has been used on all new models since, including the Pi Zero and Raspberry Pi 2 and 3 models. While most of the projects in this book use only the first 26 pins, some of them do need the additional pins. If you don't have a Raspberry Pi or only have an original version with 26 pins on the connector, I recommend buying a Raspberry Pi 2, which includes a quad-core processor, or a Raspberry Pi 3 with the 64-bit processor and built-in Wi-Fi and Bluetooth. You don't need the extra processor power for the projects in this book, but it does make it possible to use the computer for other things. After all, nobody ever says, "This computer is too fast!"

The official Raspberry Pi suppliers are listed on the Raspberry Pi web site, but they are also widely stocked by various electronics and hobbyist suppliers, so you shouldn't have a problem finding a supplier.

Buying the Components

In order to follow the instructions, you will need some electronic components. There is no single kit that will provide all the items required; it will depend on which circuits you decide to make as well as the suggested variations. Details of the main components required for each project are listed in Appendix A. One thing that is worth stocking up on is a variety of different resistors. You may want to buy either an E6 or E12 series resistor multi-pack (see Appendix C for an explanation of the resistor series).

Most of the components are fairly common and can be bought from any good electronics retailer. There are several retailers that are specifically geared toward makers. In the United States, there are companies such as Adafruit and Sparkfun, and in the UK, two popular suppliers are Maplin and CPC Farnell. You may also want to look at Raspberry Pi retailers, many of whom have an increasing range of electronic sensors and other components. In particular, Pimoroni has created several add-on boards and HATs specifically for the Raspberry Pi. There are also international electronic retailers such as RS and Farnell or many smaller independent suppliers located around the world.

One thing about electronic components in particular is that a device with an almost identical name may work differently. It may be possible to substitute a similar product, but when a specific component is required, I've tried to list the specific part number to help find the correct one. Watch out for codes that are almost the same but have different electrical properties.

Installing Raspbian

The official operating system for the Raspberry Pi is *Raspbian*. It is based on Debian Linux, but has been customized for the Raspberry Pi and comes with some additional software. The operating system needs to be loaded onto an SD card before it can run. The easiest way to install Raspbian is through the NOOBs installer. More details on installing NOOBs are available on the Raspberry Pi web site at `https://www.raspberrypi.org/help/noobs-setup/`.

The Raspbian image has been updated on a regular basis since the Raspberry Pi was launched. If you've had your Raspberry Pi for some time and haven't updated it for a while, now is a good time. For many of the projects in this book, you need a version that was released from at least November 2015. If you downloaded the image after November 2015, that is sufficient, but if you installed prior to that or used a pre-installed image that may not have been updated to that version then you will need to perform an update.

In the event of minor updates to the operating system, it is normally sufficient to run a `dist-upgrade` to update to the latest version of the installed software. To do this, launch the terminal under the Accessories category of the Start menu. Once you are in the terminal shell, enter the following commands:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

If the version is much older, or it has been a long time since your last update, then it is recommended that you download a new version of NOOBs from the Raspberry Pi web site. The new image can then be installed onto the SD card (this will involve formatting the SD card, so wipe off any data that is stored on it).

If you are unsure of your version, you can run the following command from the terminal:

```
uname -a
```

This provides a date that indicates the date of the kernel. If no date is shown, assume it is an old version and needs to be upgraded. To upgrade from an old version, you will first need to wipe the SD card using the SD formatter tool `https://www.sdcard.org/downloads/formatter_4/` and then copy the latest NOOBs files to the SD card.

Software Required

All the software required in this book is available for free and mostly available as open source software. This does include some libraries that have been created by others, and in those cases, links have been provided to the original location.

The source code used in the book can be typed in manually or downloaded from the Internet. For most short examples, I believe you can learn more through typing and experimentation, although downloads can be useful when trying to get a circuit working or when the amount of typing is enough to make your fingers ache. You can download the source code and media files as a ZIP file at:

https://github.com/penguintutor/learnelectronics/archive/master.zip

Unzip the `master.zip` file using the following commands:

```
unzip master.zip
mv learnelectronics-master learnelectronics
```

The first command unzips the file and the second command renames the directory to `learnelectronics`. The files are then contained in the relevant subdirectories.

You are free to use the software source code and circuit designs that I have provided in your own projects, but some of the accompanying files or suggested downloads may come under a different license. In particular, some of the files are provided under a Creative Commons license. Any additional copyright information is provided in the `LICENSE` file in the appropriate directory.

Safety Information

All the circuits in this book are designed to run at a low voltage and, as long as an appropriate power supply is used, they are safe to touch. To make these projects permanent may involve the use of power tools where the safety information related to the power tools needs to be followed.

Some of the projects use bright LED lights. Some people may be sensitive to high-frequency flashing lights, which in some circumstances may cause seizures. This is more likely if the flash rate is increased beyond that used in the supplied code. Please take this into consideration when modifying the source code, especially if you're using the lights in a public space. You should also avoid looking directly into any bright light, including the LED lights.

More Electronics

The projects in this book should be considered a starting point toward achieving more using electronics and the Raspberry Pi. At the end of each chapter there is a section that provides a summary of the key points in the chapter and suggestions on how the projects could be improved or ideas for related projects. These have been left as an exercise for the reader. I hope that this will result in future projects inspired from this book and I look forward to seeing some on the Internet in the future.

CHAPTER 1

Getting Started with Electronic Circuits

Most of this book involves connecting circuits to a Raspberry Pi, but before you actually plug anything into the Raspberry Pi, you will need a basic understanding of electronic circuits. This is going to be a gentle introduction, so if you already know how to build your own simple circuits and would like to jump straight in to connecting into the Raspberry Pi, feel free to jump to Chapter 2.

An *electronic circuit* combines individual electronic components to perform a specific function. This could be as simple as a light circuit in a torch/flashlight that turns on when the on switch is pressed or incredibly complex such as the circuitry inside a laptop computer. Electronic circuits are all built around the same principles.

The most basic principle is the concept that an electronic circuit must make a complete physical circuit. So for a circuit that includes a battery, there must be a complete path starting from the positive (+) side of the battery, through any components (such as a switch and buzzer), and then back to the negative (-) side of the battery. This is shown in the circuit in Figure 1-1.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1898-3_1) contains supplementary material, which is available to authorized users.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_1

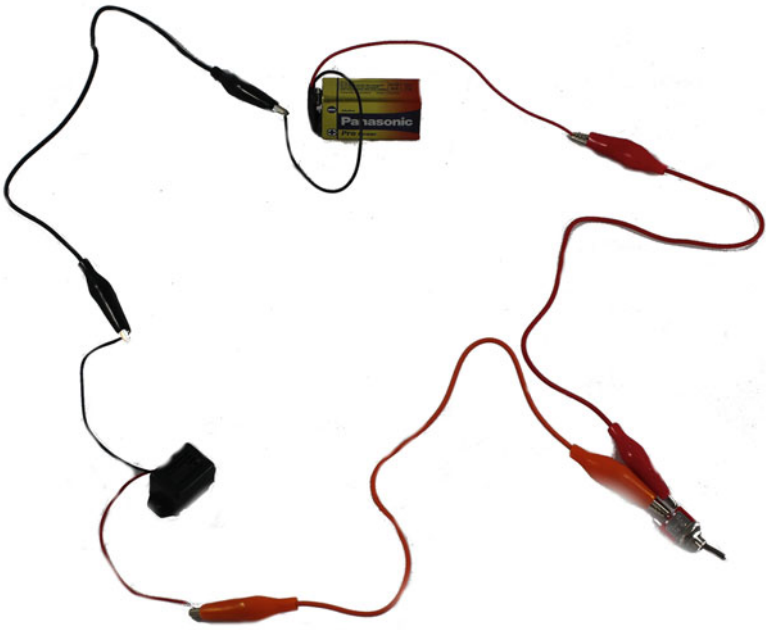


Figure 1-1. *Switch and buzzer circuit*

This is a simple circuit connected using crocodile clip leads. The circuit has a buzzer and a switch, which can turn the buzzer on and off. When the switch is closed (turned on), the circuit is complete, allowing current to flow around the circuit, and the buzzer will sound. When the switch is open (off), there is a gap between the connections inside the switch preventing the current flow and causing the buzzer to stop.

Obviously this is a very basic circuit, but it's also the basis of almost all the circuits you'll make. You will replace the mechanical switch with an electronic component and use different sensors to turn the switch on and off. You will also use different outputs, including LEDs and motors.

Voltage, Current, and Resistance

I'm going to keep the theory as simple as possible, but voltage, current, and resistance are some terms that I use throughout the book. Understanding how the circuit works and the math involved is going to be important by the time you get to the stage where you are designing your own circuits. I have avoided putting too much math into the projects, but there are some worked examples that you may find useful in future.

The *voltage* is the difference in energy between two terminals or points in a circuit. It is measured in volts indicated by a letter V. For example, you could have a 9V PP3 battery such as the one used in the buzzer circuit in Figure 1-1. The battery has a difference of 9 volts between its positive and negative terminals. We would consider the negative terminal to be at 0 volts and the positive terminal at 9 volts. The 0V connection is considered the ground terminal with voltages relative to that point. Although the battery is designed for 9V, the actual voltage may vary depending on how much charge is in the battery and what load is connected to it.

The *current* is the flow of electric charge around a circuit. Current is measured in *amperes*. This is normally abbreviated to amps and its value is indicated by a letter A. The current is related to the voltage: the higher the voltage of the power supply, the more current is able to flow through the circuit, although it also depends on what other components are in the circuit. Using conventional electric current, you say that the current flows from the positive to the negative terminal. In the electronic circuits you'll create, most currents will be small and so will normally be measured in *milliamps* (mA), where 1mA = 0.001A.

The electrical *resistance* is a measure of how difficult it is for the current to flow around a circuit. It is measured in *ohms*. The value ohms is represented by the Greek omega character (Ω). There is resistance in all components of a circuit, but you can normally disregard the small amount of resistance in a good conductor such as a wire. The resistors you will be using normally range from around two hundred ohms to several thousand ohms (kΩ).

Ohm's Law

When creating advanced circuits, some of the math can get quite complicated; fortunately, you don't need to do many calculations for most of the circuits in this book. However, there are still some basic calculations that you will need to do to. In particular for some of the circuits, you will need to work out a suitable resistor size to ensure that the current cannot damage any components, but is sufficient to allow the circuit to work.

To do this, you use a single formula that's almost certainly the most important formula used in electronics. It's also one of the simplest. This relationship was discovered by German scientist Georg Ohm and is known as Ohm's Law. The basic formula is this:

$$I = V/R$$

As you may expect, V represents voltage and R represents resistance (measured in ohms), but *I* is not so obvious. I is used to indicate current, based on the French phrase, "intensité de courant".[1]

So this formula says that to find the current through a circuit, divide the voltage by the resistance. This can be rearranged to find the voltage using this formula:

$$V = I \times R$$

To calculate the required resistor size, use:

$$R = V/I$$

An easy way to remember this is using the Ohm's Law triangle, shown in Figure 1-2.

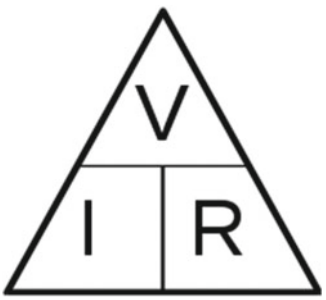


Figure 1-2. *Ohm's Law triangle*

[1]Much of the early research into electric current and magnetism was pioneered by the French scientist André-Marie Ampère. His surname is also used for the measure of current known as an ampere, which is usually abbreviated to amps.

To use the triangle, hide the value you want to calculate and read the remaining entries. So to find the voltage, you hide the letter V, leaving I and R. So multiply the current and resistance to find the voltage. To find the required resistor size, hide the letter R which leaves V above I. So you divide the voltage by the current to get the required resistor size.

Electrical Safety

Electricity can be dangerous. All the projects in this book are designed to work at low voltages up to 12V and, as long as an appropriate safe power supply (such as a wall wart or plug-in power supply) is used, there is little risk of electric shock. The same does not apply to the high voltage present in the main electricity supply.

In fact it's not the voltage that's dangerous but the amount of current that can flow through the body. Electric fences used for farm animals give a shock of several thousand volts, but although they give a nasty shock, they are considered safe for use near people as they are limited to short bursts of very low current (although should still be avoided particularly by children or those with heart conditions). The main electricity to your house is between about 100V and 250V (depending on which country you live in) and is very dangerous—it can supply enough current to be fatal. As a general rule to avoid any risk of electric shock, I recommend only working with circuits designed for 24V or less, unless you are 100% sure you know what you are doing.

It's not only electric shock that poses a risk. Even at much lower voltages, too much current can create a lot of heat and potentially start a fire. This is particularly important when using low voltage (12V) electrical lighting or car batteries, which can provide very high currents in the event of a short-circuit. I recommend only using power supplies with short-circuit and over-current protection and consider adding a fuse (this is explained later during the disco lights project).

■ **Caution** Do not try to connect any of these circuits to the main electricity in your home, except using the appropriate power supply adapter.

ANALOG VS. DIGITAL

The world we live in is varied. If we take sound as an example, we may use various words to describe the amount of sound something is making, from saying that someone is very quiet, or that the MP3 player is very loud, or even that a pneumatic drill is deafening. We don't normally know or care about the actual values of the sound (measured in decibels), but we do know if we want it to be louder or quieter. A computer however does not understand these terms. It only deals in actual values. In fact, at a most basic level, it only thinks of things being on and off, but it can compare against different levels to interpret this as a more accurate value.

An *analog circuit* can interpret any number of variations of the input. Perhaps one of the few remaining purely analog circuits you will find at home today is a simple amplifier built into a set of speakers. Here, as you turn the volume control clockwise, the volume smoothly increases compared to the input signal. Compare this to a modern TV, where you press the volume button on the remote control and the volume moves up a fixed amount, say between 1 and 40.

Most electronic circuits are now digital and in fact most include some kind of micro-processor, either a full computer such as a Raspberry Pi or a more basic micro-controller such as the ATMega micro-controllers used in the Arduino. The real world continues to be analog, so there is often an analog sensor or output and an element of conversion between analog and digital and vice versa.

Breadboard

Many of the circuits in this book are built on a solderless breadboard, sometimes called a *plugboard*. A breadboard is a good way of creating temporary circuits to allow testing prior to committing with solder. They consist of a plastic board with a matrix of holes. The holes are then connected in small rows so that components plugged into the same section are connected.

Breadboards are very easy to use and don't damage the components, so it's easy to change the circuit and experiment with different components. If you don't want the circuit any more, then the components can be removed and both the breadboard and the components can be used again for another circuit. Integrated circuits (ICs) can be also be inserted and wired to other components. To connect wires to a breadboard, you should use solid core wire or special jumper wires that have a solid end that can be plugged into the breadboard. The alternative type of wire is known as *multi-stranded wire* and it's more flexible and so more popular with soldered circuits, but it doesn't plug into the breadboard properly.

Breadboards are available in a variety of sizes, from small ones with 170 holes to large boards with multiple breadboards mounted onto a single panel. You can also connect multiple boards together, which slot together. Unfortunately, there is no standard for how the boards slot together, so this may work only if you're using the same manufacturer. A selection of different breadboards is shown in Figure 1-3.

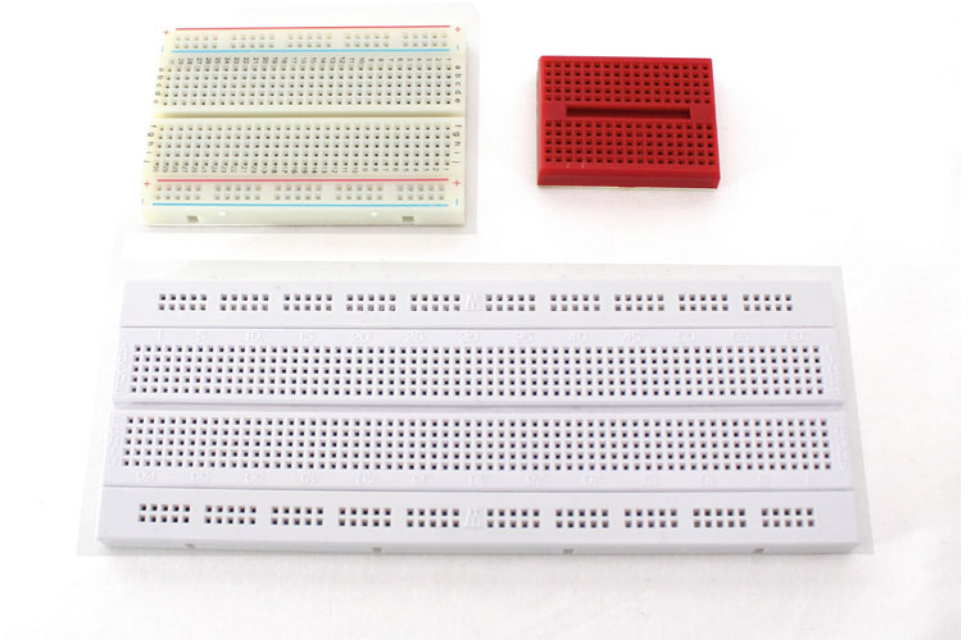


Figure 1-3. *A selection of different breadboards*

Each size of breadboard has a set of circumstances in which it works best. The smallest can be included in a small box, whereas the large one is great for larger circuits and includes connectors that are useful for plugging banana plugs from an external power supply.

For most of the circuits in this book, the half-size breadboard is an ideal size. It's about the same size as the Raspberry Pi and is a good compromise between the space taken up and the amount of space needed for connecting circuits. An example of the half-size breadboard layout is shown in Figure 1-4.

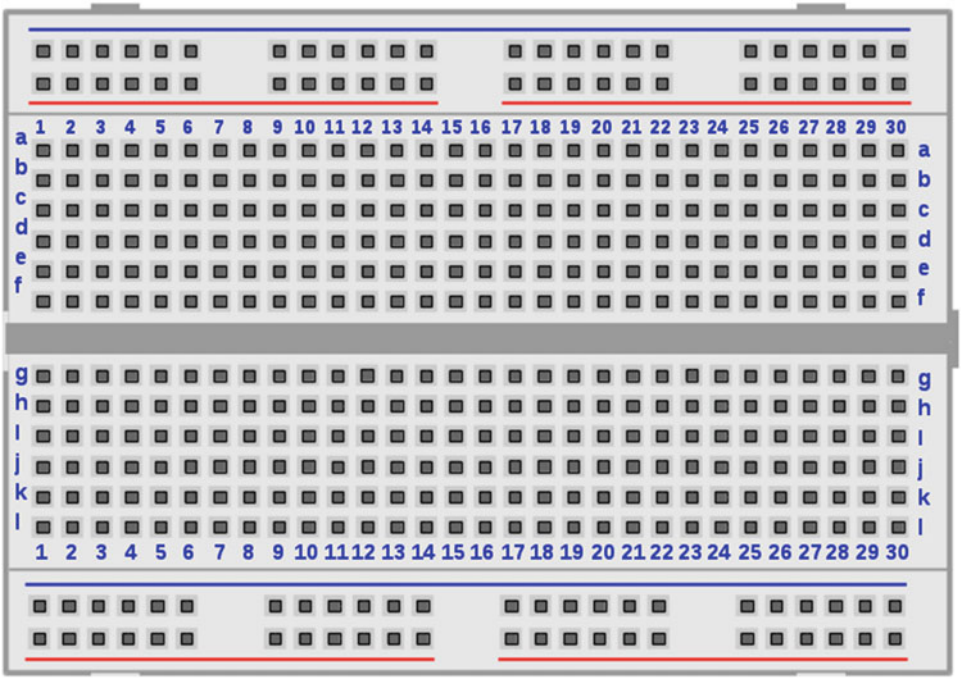


Figure 1-4. *A half-size breadboard*

The main central area consists of columns numbered 1 to 30 and rows from a to l. Each of the columns is connected with a break down the center. So for column 1 positions a to f are connected and then positions g to l. There are then two rows at the top and bottom of the breadboard, which depending on the manufacturer, may be included or as an optional extra. These rows are normally used for the main power rails, with the blue row used as ground as the red row used for the positive rail. Also note that on this example the red line covers 12 holes with there being a break in the line between the next 12. This indicates that there is also a break in the track at that point, so if you're using a single supply voltage, you may want to use a short wire to connect these together. This very much depends on the manufacturer, so you should check the ones that you have. It's frustrating trying to understand why your circuit isn't working and then finding out it's because that particular breadboard has a gap in the power rail. On the circuits in this book, I have assumed that there is no break in the power rails.

You may also notice that some breadboards have a slightly different number of pins (many have only 10 rows between a and j) and are numbered in a different direction. The actual positioning doesn't matter as long as the same pins are connected.

A useful addition is a mounting plate that allows a Raspberry Pi and a half-size breadboard next to each other. An example is shown in Figure 1-5. The mounting plate make it easier to wire the Raspberry Pi and breadboard together, as it means the wires are less likely to fall out. You could even make your own using an appropriately sized piece of plastic or thin wood.

Figure 1-5. *Raspberry Pi and a breadboard mounted together*

You also need a way to connect the Raspberry Pi and the breadboard. The Raspberry Pi has a male connector for the GPIO connector (explained in the next chapter), so a female connector is required. These are available as individual jumper wires, which go from male to female. A selection is shown in Figure 1-6.

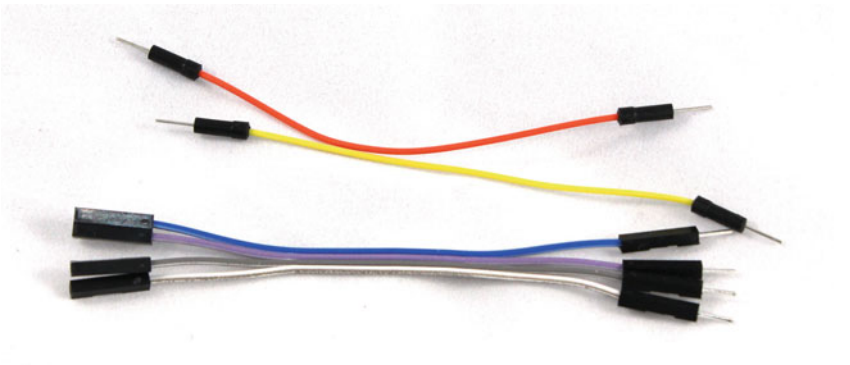


Figure 1-6. *Jumper wires—male-to-male and male-to-female*

An alternative is to use a *cobbler,* which allows a multi-way cable to connect to a breakout board that connects to the breadboard. This makes it easy to disconnect the Raspberry Pi if you need to use it for other purposes. The cobbler also provides a label against each pin, thus making it easier to know which port to connect to. It does however take up some of the space on the breadboard, especially if you're using a 40pin cobbler matching the larger GPIO connector on the newer versions of the Raspberry Pi. A typical cobbler and cable are shown in Figure 1-7.

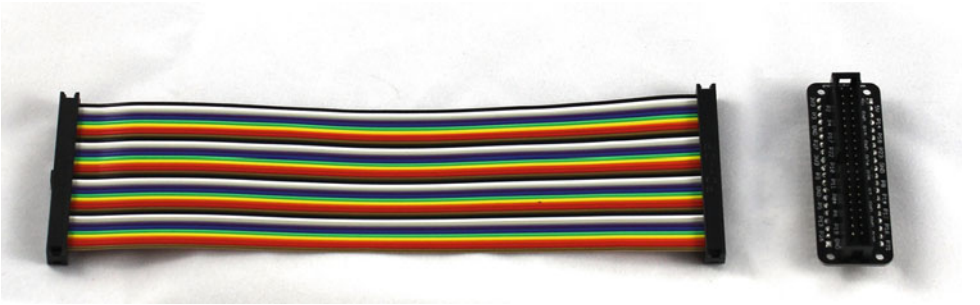


Figure 1-7. *Raspberry Pi cobbler (40pin)*

One disadvantage of using a breadboard is that wires or parts can be accidentally pulled out, so they are rarely used for a permanent circuit. Later, you will look at making more permanent circuits that can last much longer.

A First Breadboard Circuit

Your first circuit is a standalone circuit to get used to a complete circuit and for a first practice at using a breadboard. The breadboard layout is shown in Figure 1-8.

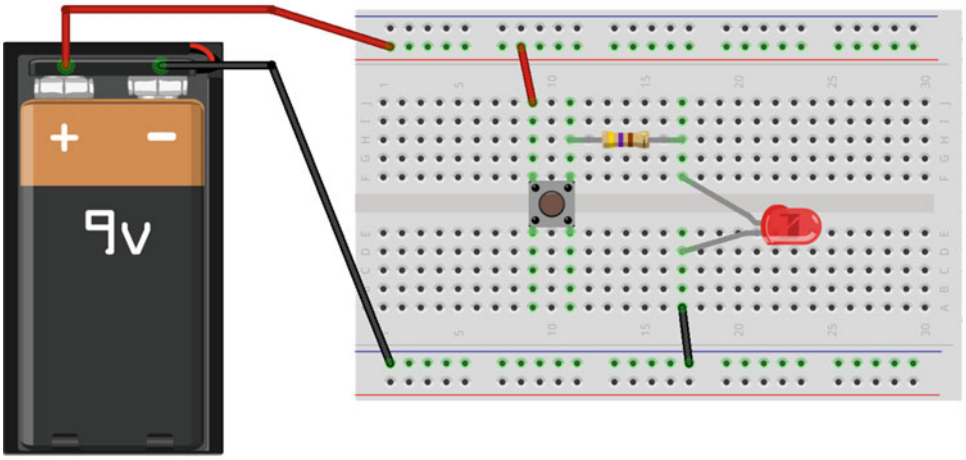


Figure 1-8. *Simple LED circuit*

Starting from the left in Figure 1-8 is a 9V PP3 battery. These can be connected using a push-on battery connector with leads. These normally have stranded wire, which as I said earlier, doesn't work so well with a breadboard. Some have the ends of the wire coated, which should connect to the breadboard. If you are not able to connect it to the breadboard directly, you can use wires with crocodile clips instead.

The next component is a miniature pushbutton switch. This should be a single-pole, single-throw type (often known by its initials, SPST). This means that there is a single switch inside and that switch can change between two states (in this case, on and off). This is a push-to-make switch, which means that when the button is pressed, the switch contacts are connected.

Only two connections are needed for an SPST switch, but this particular switch has four. Each pair is interconnected on the left and right of the switch, respectively. This is shown in Figure 1-9.

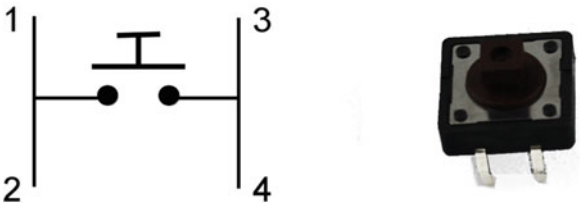


Figure 1-9. *SPST push-to-make switch*

As you can see in Figure 1-9, pins 1 and 2 are connected, as are pins 3 and 4. Each side is connected when the button is pressed. In this circuit, we are connecting the positive supply to pin 1 and then taking the output from pin 3.

The next component is a resistor. This example uses a 470Ω, which is indicated by the different colored strips around the body (yellow, violet, brown, and then gold). The resistor is used to reduce the amount of current that can flow through the LED, which would otherwise almost certainly damage the LED.

The final component is a light-emitting diode (LED). This must be connected a specific way around. The anode connects toward the positive end of the supply (connecting to the resistor) and the cathode to the ground connection and on to the negative end of the battery.

You can tell which end is the *anode* (the positive terminal), as it normally has a longer lead. Failing that, there is normally a flat area on the plastic casing that indicates the *cathode* (the negative terminal). If all else fails, then this is a simple circuit where it would be safe to temporarily connect it either way around and if it doesn't work then try the other way around. This is one of the advantages to using the breadboard.

Once you have connected the components and the wires, pressing the button should cause the LED to light and releasing the button will turn it off again.

Calculating the Resistor Value

Earlier I said that the resistor value was 470Ω, but I did not explain how that value is determined.

To calculate the resistor value, you first need to know the current you want through the LED, which is usually available from the supplier or from a data sheet. In this case, you are looking at a current of around 15mA to light the LED. You also need to know the voltage dropped across the LED, which is typically around 2V for a red LED.

Once you know these, you know that there will be 7V across the resistor (9V from the battery minus 2V across the LED) and that you want to limit the current to around 15mA.

Using Ohm's Law, the resistance = V ÷ I, which works out to 7 ÷ 0.015 = 467Ω.

The nearest value of resistor is 470Ω.

Static-Sensitive Devices

You may already know that you can create static electricity by rubbing a balloon through your hair, or using a comb to pick up bits of paper. You can also create static electricity by walking on a carpeted floor. Although harmless to us, that same static electricity can cause permanent damage to some electronic components. Static-sensitive components are often supplied in special bags like the one shown in Figure 1-10, but not always.

Figure 1-10. *Warning symbol for a static-sensitive component*

It is worth getting an anti-static wrist strap and connecting it to a suitable ground or earthing connection. A wrist strap and ground plug are shown in Figure 1-11. If you don't have a wrist strap, you can discharge static electricity by touching a metal object that is connected to an electrical ground connection. This could be an earthed radiator or the outside case of the main electrical equipment. Obviously, you should only touch the outside of the electrical equipment and never open up main electrical equipment in search of a ground connection.

Figure 1-11. *Anti-static wrist strap*

More Circuits

This chapter looked at what makes a circuit and the importance of connecting a complete circuit, which goes from the positive end of the power supply all the way back to the negative side.

You briefly looked at Ohm's Law, which will come in useful later. You then built your first standalone circuit using a breadboard and a few basic components.

To experiment further, you could try increasing the size of the resistor and watch the effect it has on the brightness of the LED or try swapping the LED and resistor with a buzzer (use the same voltage buzzer as the battery) for a noisy circuit instead.

In the next chapter, you will explore the Raspberry Pi and in particular the GPIO ports that allow you to add electronic circuits to the Raspberry Pi.

CHAPTER 2

All About Raspberry Pi

It's time to pull out the Raspberry Pi and get started. This chapter looks at the Raspberry Pi from both the hardware and software points of view. This chapter concentrates on the GPIO ports and the programming tools that will be useful for later chapters. It goes beyond the essentials, as a wider understanding of Linux will help with future projects involving the Raspberry Pi.

Introduction to the Raspberry Pi

The Raspberry Pi is a low-cost computer designed as a way to learn programming. It also makes a great platform for hobbyists looking for a computer to use with electronic projects. Although the price is obviously appealing, arguably more important are the two rows of pins that can be used to connect electronics to the Raspberry Pi. These pins are collectively known as the *GPIO pins* and provide a way to interface the computer to homemade electronic circuits that wasn't available before. As a result, there are just as many adult makers who want a computer that can be used to create hardware projects.

There are now several versions of the Raspberry Pi. The original model A and B versions included 26 GPIO pins (17 of which can be used for inputs and outputs), but the newer models (including the Pi Zero, A+, B+, and the Raspberry Pi 2 and 3) increase the number of pins to 40 (28 of which can be used for inputs and outputs). The extra pins allow for more complex circuits and additional add-on boards known as HATs (Hardware Attached on Top). There is one more version, called the Raspberry Pi compute module. This is designed to be incorporated directly into commercial circuits and is beyond the scope of this book.

The photo in Figure 2-1 shows three models of the Raspberry Pi—The Pi Zero, a model A+, and a Raspberry Pi 2. The Raspberry Pi 3 adds built-in wireless and Bluetooth connectivity.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_2

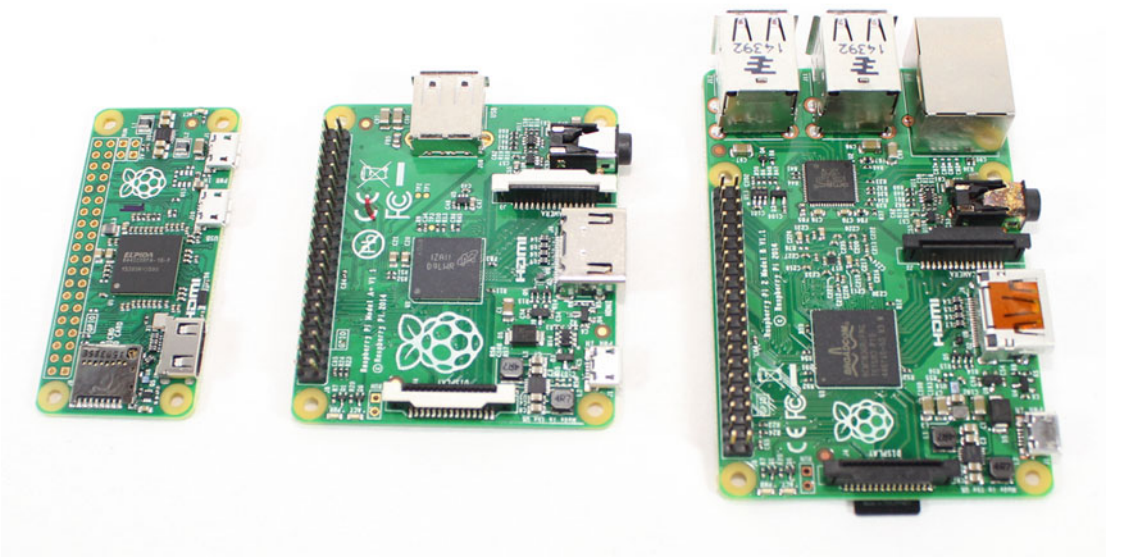


Figure 2-1. *Different models of the Raspberry Pi (from left): Pi Zero, A+, and Raspberry Pi 2*

In Figure 2-1, the GPIO connector is in the top-right of each of the computers. As you can see, the Pi Zero (left) does not include the GPIO header pins, which are soldered on the other versions. You can solder on your own headers, or wires can be soldered directly to the board. More details on learning to solder are in Chapter 10.

If you have an older Raspberry Pi, such as a model A or model B, you will still be able to work through most of the projects in the first few chapters, but you will need the additional pins if you want to try some of the projects in the later chapters. You should also be aware that the pin layout for some of the pins is different on the older models. More on this later in this chapter.

Raspberry Pi GPIO Ports

As mentioned previously, the GPIO ports on the Raspberry Pi are a real game-changer. They make it trivial to connect simple electronic circuits to the Raspberry Pi.

GPIO stands for General Purpose Input Output. It is a common term used to refer to ports on a processor that can be used either as inputs or outputs. The GPIO pins on the Raspberry Pi are connected directly to the GPIO ports on the processor. The processor runs at 3.3V and as such the GPIO ports are designed for 3.3V. This is less than the 5V commonly used by some electronic circuits and a 5V connected as an input to one of the GPIO pins could damage the Raspberry Pi. The output of the GPIO is only able to provide a current of up to 16mA.

■ **Caution** The GPIO ports do not include any built-in protection. Giving an input that is above 3.3V, or drawing too much current from an output, can permanently damage the Raspberry Pi.

Most of the GPIO pins can be used for normal input/output as well as some having alternative functions, such as I2C and PWM.

There have been some changes to the GPIO port numbers used on the GPIO pins. In September 2012, the revision 2 boards were released and they changed some of the pin allocations. The model B+ and A+ boards (released July 2014 and November 2014 respectively) added 14 new pins, thereby increasing the total pins from 26 to 40. The Raspberry Pi 2 has the same port allocations as the A+ and B+ models, as does the Pi Zero, although there are no header pins soldered on to the Pi Zero.

To check which version of Raspberry Pi you have, you can run this command:

```
cat /proc/cpuinfo
```

The revision numbers are shown as hexadecimal values. These are 0002 and 0003 for Raspberry Pi model B revision 1; 0004 to 000f for Raspberry Pi model A and B; 0010, 0012, and 0013 for A+ and B+; and a01041 and upward for Raspberry Pi 2.

A summary of the main pin allocations for the different versions is listed in Figure 2-2.

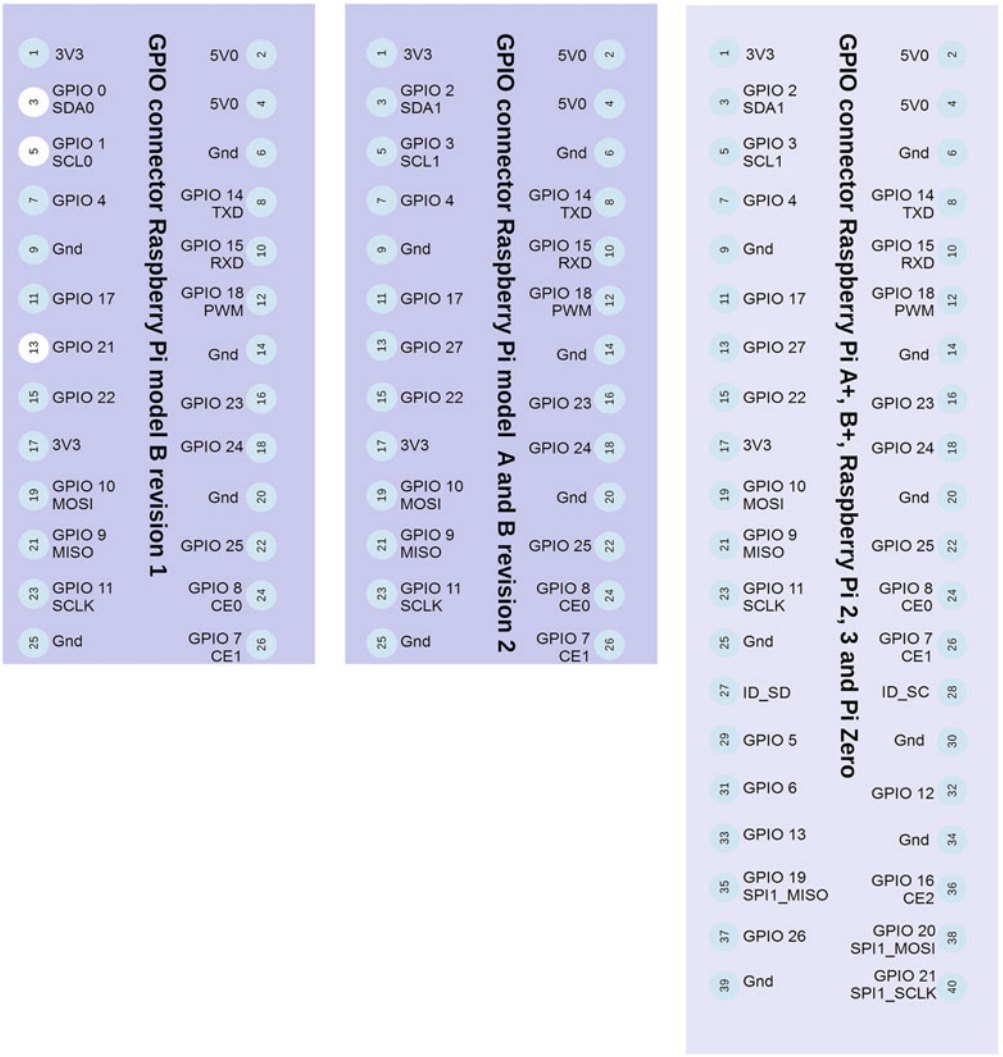


Figure 2-2. *Pin layouts for the GPIO for various models of the Raspberry Pi*

The GPIO connector is in one corner of the Raspberry Pi. If you position the Raspberry Pi with the SD card on the top edge, as shown in Figure 2-1, the pins are numbered with 1 on the top left, reading left to right, then top to bottom. This is shown in Figure 2-2, which shows the GPIO number and alternative function for many of the pins. The highlighted pins on the revision 1 boards are the ones that changed on the more recent versions.

A summary of the ports and common alternative functions is provided in the table in Appendix D. This is based on the B+. These are the same for the Model B revision 2 and the model A (although only up to pin 26 for the model A and model B).

You will look at some of the alternative functions in further details later in this book.

Serial Communications/UART

This is a way of sending information between two different devices. The data is sent one bit of data at time. There are multiple ways of sending data serially, but this is specifically about using the Raspberry Pi as a Linux serial device. This is how terminals were connected to computers in the "olden days" (common in universities and libraries during the 1980s) and was used to log in to remote computers using modems (and is still used in this way by some companies to provide remote access to networking equipment if the main network connection stops working). It's also used to communicate with the Arduino and with other microprocessor platforms.

There are two common ways that the Raspberry Pi can communicate as a serial device. One is to use a device connected to the USB port using the Linux USB serial driver (such as an Arduino, micro:bit, or a Bluetooth connection).

The other is to use pins from the GPIO port that are connected to a UART (Universal Asynchronous Receiver/Transmitter) within the processor. The UART pins can be wired directly to another serial device, which provides an alternative way to connect to an Arduino or similar device. The GPIO pins are physical pins 8 and 10, and they connect to transmit (TXD) and receive (RXD). By default, these pins are used for the Linux console, so they may periodically send error messages from the operating system. This can be useful to identify problems when the Raspberry Pi is not connected to a screen, but the console may need to be disabled or redirected if those pins are used for another purpose.

I²C: Inter-Integrated Circuit

Also known as I2C, this is another serial communication protocol that is used to communicate with peripherals. It works as a master/slave relationship. Typically the Raspberry Pi will act as a master and communicate with peripherals such as sensors or displays.

The latter Raspberry Pi models use I2C channel 1, which use GPIO pins 3 and 5. These are the SDA1, SCL1 ports, respectively. The earlier revision 1 model B Raspberry Pi computers used the same pins but with I2C channel 0 instead.

SPI: Serial Peripheral Interface Bus

SPI is another form of serial communication for communicating with peripherals. It has a higher bandwidth than I2C and is commonly used for connecting to SD cards and other peripherals. A physical difference between SPI and I2C is that, while I2C uses only two connections, SPI needs at least four and in some cases more.

The connections are SCLK (serial clock), MOSI (master output, slave input), MISO (master input, slave output) and SS (slave select). These are pins 23, 19, 21, and 24. The slave select is labeled as CE0 on the GPIO output, which means chip enable (and can be considered the same as slave select). This is used to determine which of multiple slave devices the master is communicating with. There is also a second slave CE1 on port 26, which allows an additional slave device to be connected.

There is also a second SPI connection on the more recent versions using pins 40, 38, 35, and 36.

PWM: Pulse Width Modulation

Pulse Width Modulation is a technique used to provide an analog output from a digital pin. This works by turning the output on for a set period of time and then off for a period of time. By varying the "width" of the on and off times, an equivalent average voltage can be calculated. This is useful for varying the brightness of an LED connected to a digital pin or varying the speed of an analog motor.

It also used as a way of sending information, which is popular with infrared remote controls that use PWM to send the relevant codes to turn your TV on and off. You will see this in action in Chapter 5 when you build an infrared transmitter and receiver.

Other pins can be used for PWM by creating an equivalent signal from switching the appropriate pin on and off at the right time using software, but the PWM output on pin 12 is supported by the hardware so is more accurate and less taxing on the processor. This will be useful in Chapter 4, where you will use the hardware PWM output to control NeoPixels.

Getting Started with Raspbian Linux

The official operating system for the Raspberry Pi is Raspbian, based on Debian Linux. In the recent versions of the Raspbian image, the Raspberry Pi boots straight into the desktop when started, shown in Figure 2-3. The desktop may look a little different from other operating systems you use, but whatever operating system you normally use, it shouldn't be too different.

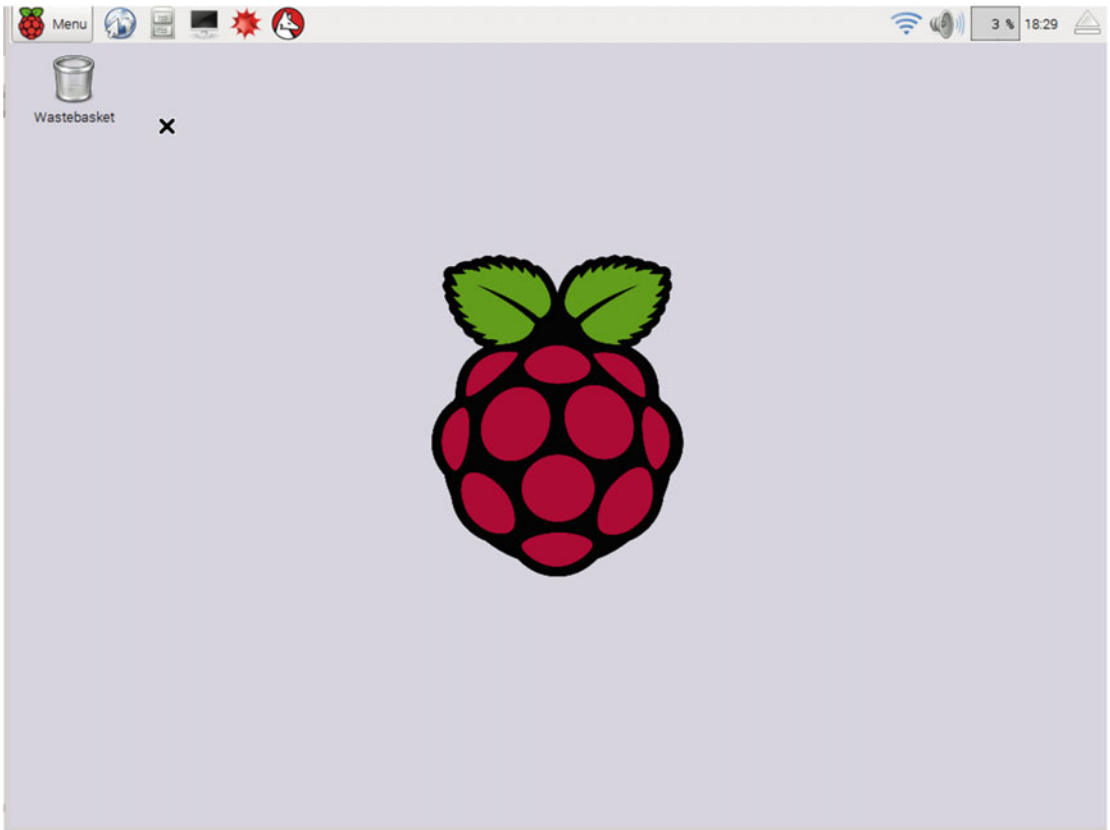


Figure 2-3. *Screenshot of the Raspbian desktop*

Here are some of the key pointers to navigating around the desktop:

- The application menu is in the top-left, as indicated by the Raspberry Pi logo.
- There are lots of applications included. Here is a small selection:
  - The programming menu includes a selection of different programming languages, including Scratch and Python, which you will be using throughout this book.
  - There is no need to pay for a word processor or other office application because LibreOffice is included.
  - The Epiphany web browser has been specially tweaked for the Raspberry Pi and works well even on the single-core versions.
  - There is a special version of Minecraft for the Raspberry Pi with an easy-to-use Python interface. You will be using it in Chapter 9.
  - You can adjust some of the desktop settings in the Preferences menu.
  - The Raspberry Pi configuration application provides a graphical alternative to `raspi-config`. This provides a tool for changing some of the options for the Raspberry Pi.
- The terminal application provides a way to issue commands using a text interface. This provides a very efficient way to run commands and automate certain features of the Raspberry Pi.
- Using the icons on the top-right of the screen, you can do the following:
  - Configure the network connection (including WiFi).
  - Change the volume.
  - See how hard the processor is working (as a percentage).
  - See the time.
  - Safely remove removable storage devices.

If you need more information about any of these, hover your mouse pointer above them to see any hints, left-click for typical configuration changes, or right-click for further configuration options. For example, hovering over the volume icon tells you the current setting, a left-click lets you to change the volume, and a right-click lets you to change whether the audio is sent to the TV (HDMI output) or through the audio connector (3.5mm jack).

Connecting to the Raspberry Pi Using the Network

Although it's possible to follow the projects in this book using a screen, keyboard, and mouse connected to the Raspberry Pi, it is often useful to connect from a remote computer. This could be another Raspberry Pi, or a desktop or laptop computer. You will use two methods of connecting here: ssh and vnc.

Before you can connect, you first need to know the IP address that has been allocated to the Raspberry Pi. In most cases this will be a dynamic IP address provided by your home router, but it could be set manually instead. If you have connected the Raspberry Pi to the router using an Ethernet cable, then this should have been allocated already, but if you are using WiFi then you will need to enter your WiFi network details through the WiFi configuration tool (top-right side of the screen).

Once you have connected to a network, hovering over the WiFi icon will display the IP address assigned. Figure 2-4 shows the IP address allocated to my Raspberry Pi, which is `192.168.0.109`.

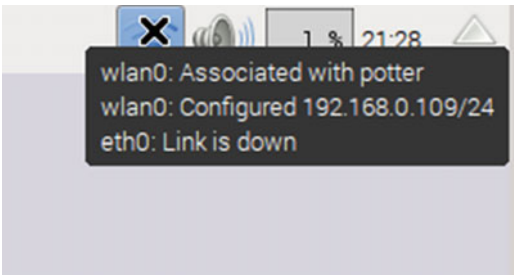


Figure 2-4. *Screenshot of WiFi status message*

Alternatively you can obtain the IP address on the command line by launching the Terminal application and entering this command:

```
ip addr
```

Secure Shell (ssh)

Secure Shell (often referred to as ssh) is a server that allows you to run instructions on the Raspberry Pi from a remote computer. It doesn't provide a graphical interface, but allows you to type instructions using a keyboard and receive a response through a display terminal.

The ssh server is installed by default on the Raspberry so it's simply a case of entering the IP address into an ssh client on the same network and it is possible to log in using the normal username and password (by default, username `pi` and password `raspberry`).

For Linux and Mac OS X, an ssh client is available directly from the command line. For Windows, you need to download an ssh client, such as the open source PuTTY, available at `http://www.chiark.greenend.org.uk/~sgtatham/putty/`. Various graphical ssh clients, including PuTTY, are also available for Linux and Mac OS, if preferred.

To connect using a terminal client on a Mac or Linux, prefix the IP address with the username and the `@` character.

```
ssh pi@192.168.0.109
```

From the graphical clients, such as PuTTY, enter the IP address and username in the appropriate fields, as shown in Figure 2-5.

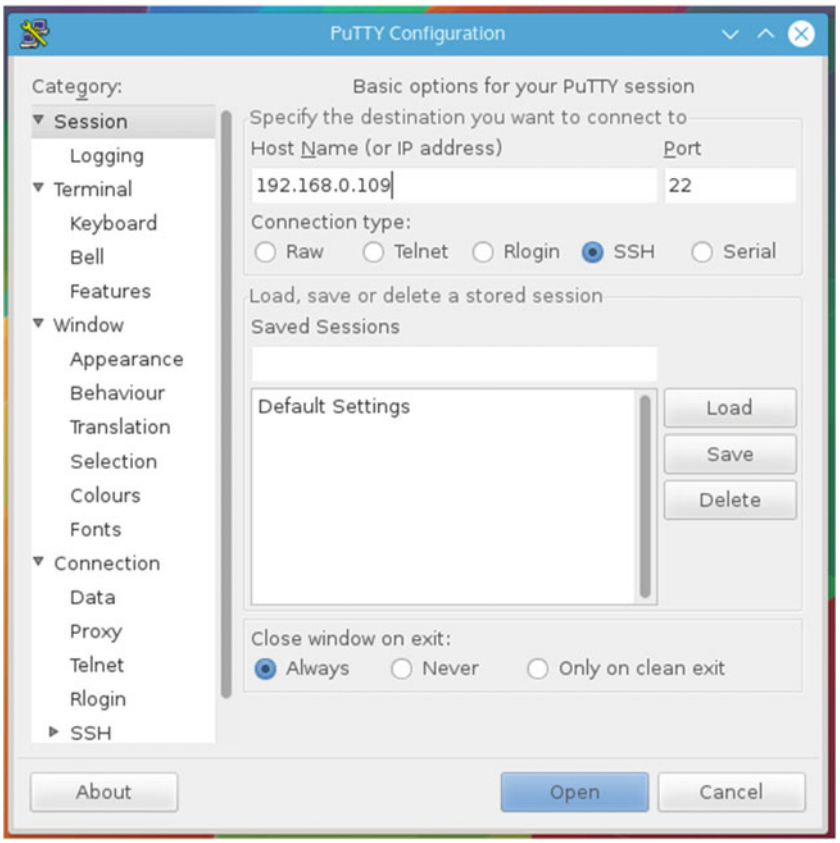


Figure 2-5. *PuTTY application, an ssh client*

Remote Desktop Using VNC (TightVNC)

Having access to a remote shell through ssh is a good way of running command-line programs, but a graphical interface can be useful at times. In fact most of the screenshots in this book were taken by connecting using VNC from my laptop.

To achieve this, you need to install TightVNC on the Raspberry Pi. Rather than give access to the existing screen, this starts a new instance of the desktop that you can control access remotely.

Install the server software from the repositories:

```
sudo apt-get install tightvncserver
```

After installing, run `tightvncserver` to start the server as the current user. The first time you run the server, it will prompt you to set a password. This is the password you use when connecting remotely. There is no need to create a *view only* password.

When the server starts, it will tell you which virtual desktop has been started. This will normally be session 1:

```
New 'X' desktop is raspberrypi:1
```

You connect to this using `:1` at the end of the IP address in the client.

You can run multiple instances. Each time you start `tightvncserver` it uses the next available desktop, but in most cases you will just need one. You can leave it as that, which means that the remote desktop will only be available if you log in and start it first, or you can set it up to run whenever the Raspberry Pi is booted by following the instructions at http://www.penguintutor.com/linux/tightvnc.

You also need to install the TightVNC client (or one of the alternative VNC clients) on your local computer. For Linux operating systems, a client may be installed already; for Windows, there is a native client available; and for Mac OS X, the Java client can be used. These are available from http://www.tightvnc.com. Figure 2-6 shows how to connect using the TightVNC viewer on Windows 8.

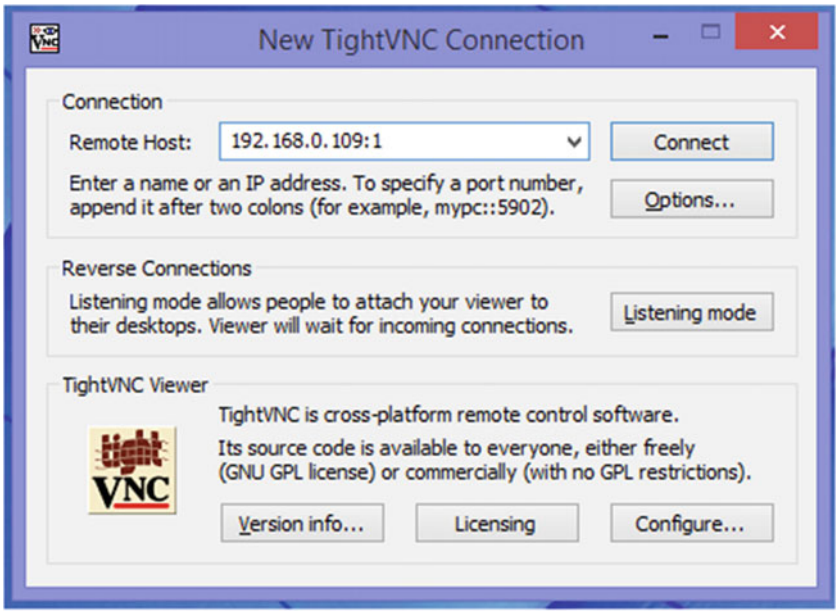


Figure 2-6. *Connect remotely using TightVNC*

Once you're connected, you can run most software through the client as though using a mouse and keyboard on the Raspberry Pi. There are some limitations to what TightVNC can be used for—in particular, it is not possible to run Minecraft over a TightVNC session. For most applications, though, TightVNC does provide a useful way to run programs remotely.

More Raspberry Pi

This chapter looked at the different versions of the Raspberry Pi. You then looked at the GPIO interface pins and read an introduction to Raspbian, the Linux-based operating system that runs on the Raspberry Pi.

There are lots of other resources for learning Linux, including another Apress book entitled *Learn Raspberry Pi with Linux*, written by Peter Membrey and David Hows.

You may also want to spend some time exploring the installed software and games such as Minecraft.

This is the end of the initial theory. The next chapters involve hands-on electronics connecting to the Raspberry Pi. In Chapter 3, you connect your first circuit to the Raspberry Pi and learn to access the GPIO pins through the Scratch programming language.

CHAPTER 3

Starting with the Basics: Programming with Scratch

Scratch is a visual based programming language. It was designed at MIT for teaching programming to children, but has become popular with young and old programmers. Some universities even use it as an introductory language for new students.

The latest version of Scratch, version 2, uses a web based interface. Unfortunately it is based on Adobe Flash and as such will not run on the Raspberry Pi. The Raspberry Pi instead includes an older version, based on version 1.4. The layout is a little different from version 2, but the functionality is mostly the same.

Introduction to Scratch

If you are not already familiar with Scratch, here is a quick introduction. Scratch is available from the programming menu in Raspbian and, when running, looks like Figure 3-1. If you have used version 2 already, it should look familiar, but you will notice that the stage area for Raspberry Pi is on the right, whereas it is on the left in version 2.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_3

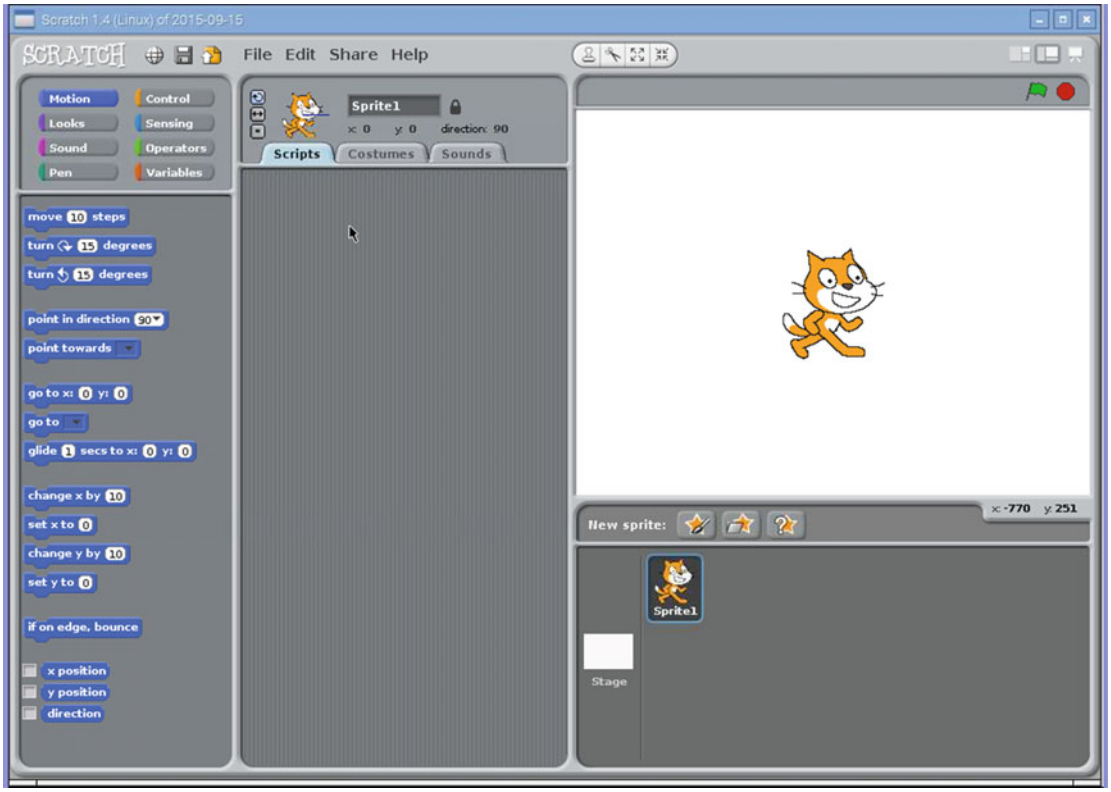


Figure 3-1. *Scratch on the Raspberry Pi*

The application is split into four main sections. In Figure 3-2, I shaded the main areas and labeled them as A, B, C, and D.

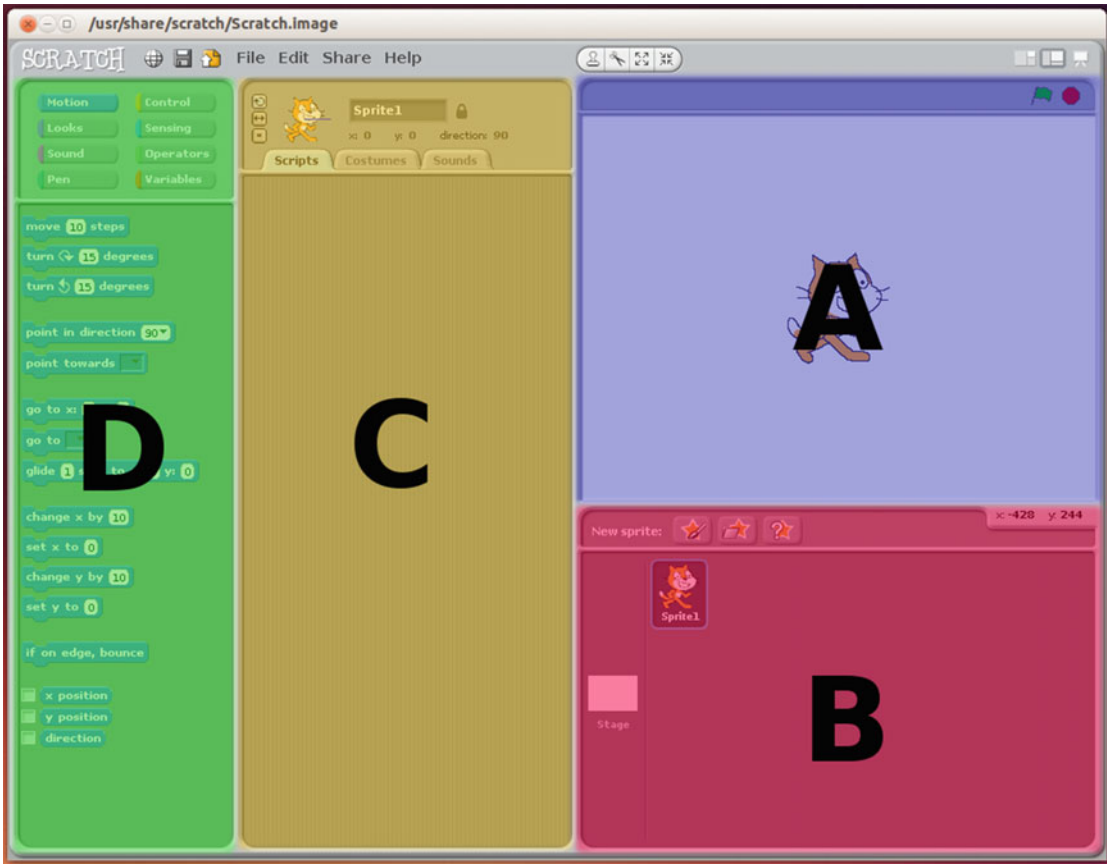


Figure 3-2. *The four main areas of Scratch*

The most important part of the Scratch application is the stage area (section A). This is a representation of the screen when the program is running. The look of the game will be designed on the stage area and it's also where you can see the program run. There is also a *green flag,* which is used to start the program running, and a *red stop sign,* used to stop the program.

If the screen is a stage, then the sprites are the actors, and props and lights are used to create the show. These are stored in the sprite list area (section B). There are three icons for creating a new sprite, depending on whether you want to draw your own sprite, choose an existing file (many are included with Scratch), or get a random sprite. In Scratch, a sprite can have its own costumes, scripts, and sounds associated with it. The stage is considered a special type of sprite and is always shown in the left part of the sprite's list area.

When you want to change a sprite, you load it into the sprite edit area (section C). Sprites are loaded into this section by clicking on them in the sprite list, where they will then be shown with a border. There are three tabs in the sprite area called Scripts, Costumes, and Sounds.

In Scratch the program code is grouped together into a script and is edited on the Scripts tab. It is fairly common to have several scripts associated with each sprite. The stage can also have its own scripts, which is often used to control the sprites or the stage background.

The Costumes tab is used to change the look of a sprite. It does not need to be a different form of clothing (although that is one use of a costume), it could be used to represent a different object or a completely different look, such as a firework rocket that changes to a burst of light when it explodes. When editing the stage, the Costumes tab is replaced with a Background tab, which can be used to change the scene. There's also a Sound tab to add sound to the sprite.

The code blocks area (D) holds the blocks of code that can be built together to make the scripts. Each block of code is shaped with interlocking tabs, which can be connected with other appropriate blocks of code.

The blocks of code are grouped into eight sections and are color-coded along with the code groups. Drag the appropriate code block from this area to the scripts area for the sprite. If you position the block close to another block then they will snap into place.

Scratch with GPIO Support

The version used on the Raspberry Pi has been modified to provide a way to control the GPIO pins through a built-in GPIO server. This was added in September 2015. As you can see from the title on the screenshot in Figure 3-1 that the date is included in the title bar (in ISO date format). If Scratch is showing an older version, you may need to upgrade the version of Raspbian (see the Introduction).

The GPIO server waits for broadcast messages from Scratch and turns output pins on and off appropriately. It also handles input to appropriate pins on the GPIO port and sends broadcast signals to Scratch when the status changes. You will need to start the GPIO server using the Start GPIO Server menu option from the Edit menu, as shown in Figure 3-3.

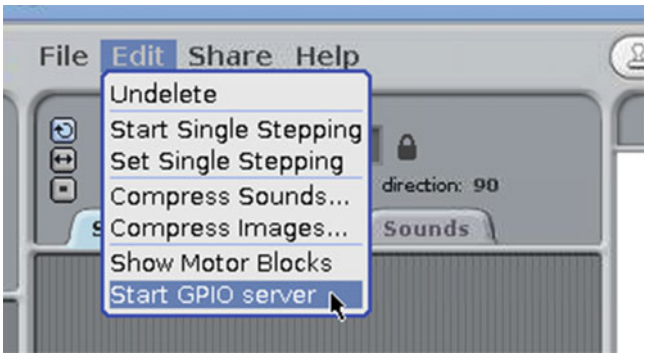


Figure 3-3. *Start GPIO Server option on recent versions of Scratch on the Raspberry Pi*

Controlling an LED Using Scratch GPIO Server

Before creating your program in Scratch, you need to create your first Raspberry Pi electronic circuit. You will start with a very simple LED circuit that allows you to turn a light on and off, depending on events in a Scratch program. You'll then expand this to make your first game.

To understand the circuit, you first need to understand two components—the LED and the resistor. You've read about these in Chapter 2, but I've included a more detailed explanation here.

Light Emitting Diode (LED)

An LED is a component that gives out a light when an electric current passes through it. The LED is very efficient and so you will often see them used in battery-operated light-up toys. The fact that it doesn't use much energy is important because the amount of current you can supply from the GPIO pins is very small. If you instead used a light bulb such as those you may have in an older flashlight, you would need a more complex circuit or risk damage to the Raspberry Pi.

An important thing about an LED is that it needs to be inserted the correct way around in the circuit. This is due to the *diode* part of its name, which is a component that acts similar to a one-way valve, allowing current to flow only in one direction. The diode has an anode, which should be connected to the positive end of the connection, and a cathode, which goes to the other side.

You can tell which end is the anode (the positive terminal) because it normally has a longer lead. If that's not obvious (such as if the lead has been cut short) then there is normally a flat area on the plastic casing that indicates the cathode (negative terminal). This is shown in Figure 3-4.

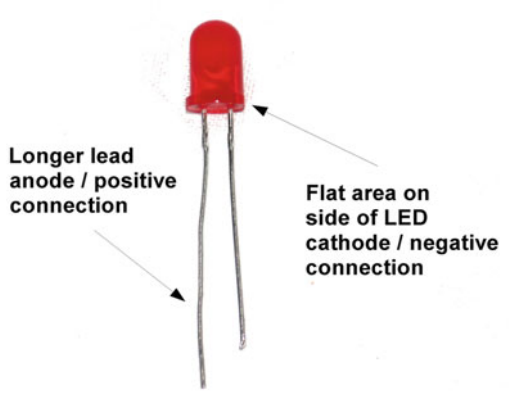


Figure 3-4. *A typical LED*

Resistor

A resistor acts to reduce the current flowing through a circuit. This is particularly important to protect a component from being damaged because too much current is flowing through.

The size of the resistor is measured in ohms (Ω), which is marked on the side of the resistor using a color code. You can see a photo of a typical resistor in Figure 3-5 and details of the color-coded markings are provided in Appendix C.

Figure 3-5. *A 220Ω resistor*

In this circuit, the resistor is required to protect the GPIO port from too high an output current and to protect the LED from excessive current. To calculate the value of the resistor, you first need to know the voltage across the resistor and the current you want to limit it to.

The voltage of the GPIO output is 3.3V. Approximately 2V of this is across the LED, so there is approximately 1.3V across the resistor.

Although the maximum current that the GPIO can provide is 16mA, around 8mA is sufficient to light the LED to show whether something is on or off. To calculate the resistor, divide the voltage by the current (see Chapter 1). The resistor size should therefore be 1.3 ÷ 0.008, which is 162Ω. Resistors come in standard values. In this case I've selected the next highest in the E6 series of resistors, which is 220Ω. If you have a set of E12 series resistors, you could use a 180Ω resistor, but either will do. The color of the resistor is:

- Red (2)
- Red (2)
- Brown (x 10)
- Gold

Due to differences in manufacturing, the resistor is not necessarily going to be exactly the same as the specified size. The fourth entry is shown as gold, which gives you the tolerance rating (how close the resistor must be to the specified value)—in this case 5%. Taking the tolerance into consideration, the actual resistor will be between 209Ω and 231Ω.

Connecting the LED to the Raspberry Pi

For this circuit, you will use GPIO port 22, which is physically pin 15 on the GPIO connector (see the GPIO diagram in Chapter 2 for more details). The resistor connects to the GPIO port and it then connects through the LED to the ground connection. The breadboard layout diagram is shown in Figure 3-6.

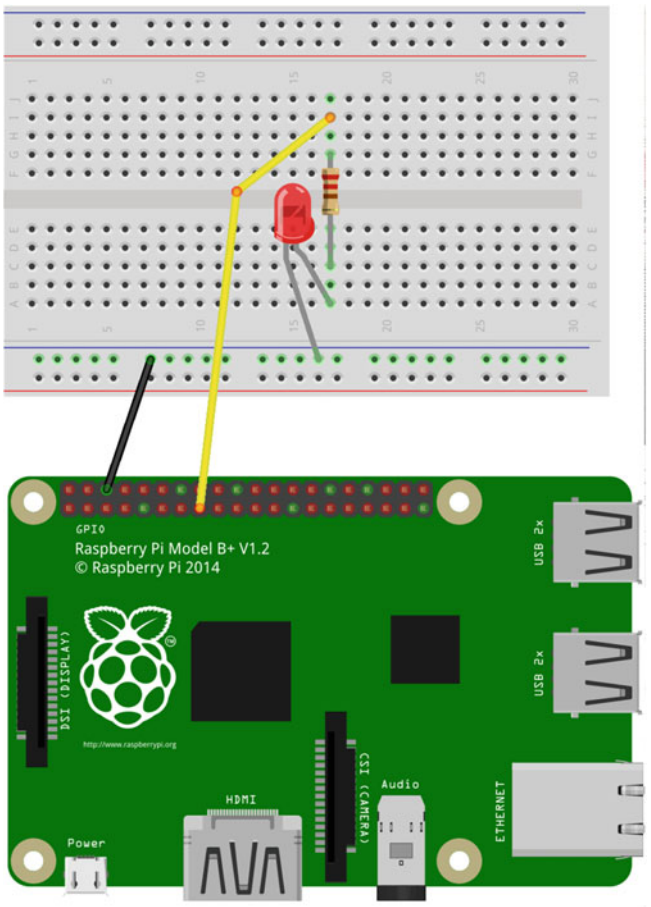


Figure 3-6. *LED circuit for Scratch GPIO*

Remember to make sure that the LED is the correct way. Looking at the diagram in Figure 3-6, the longer wire should be the one nearest the top and in the same block as the resistor.

Now move on to having Scratch turn the LED on and off. First start the GPIO server using the menu option shown in Figure 3-3. Using a new Scratch program, add the code blocks to the Scripts tab of the default sprite (`sprite1`, which is the Scratch cat). This is shown in Figure 3-7.

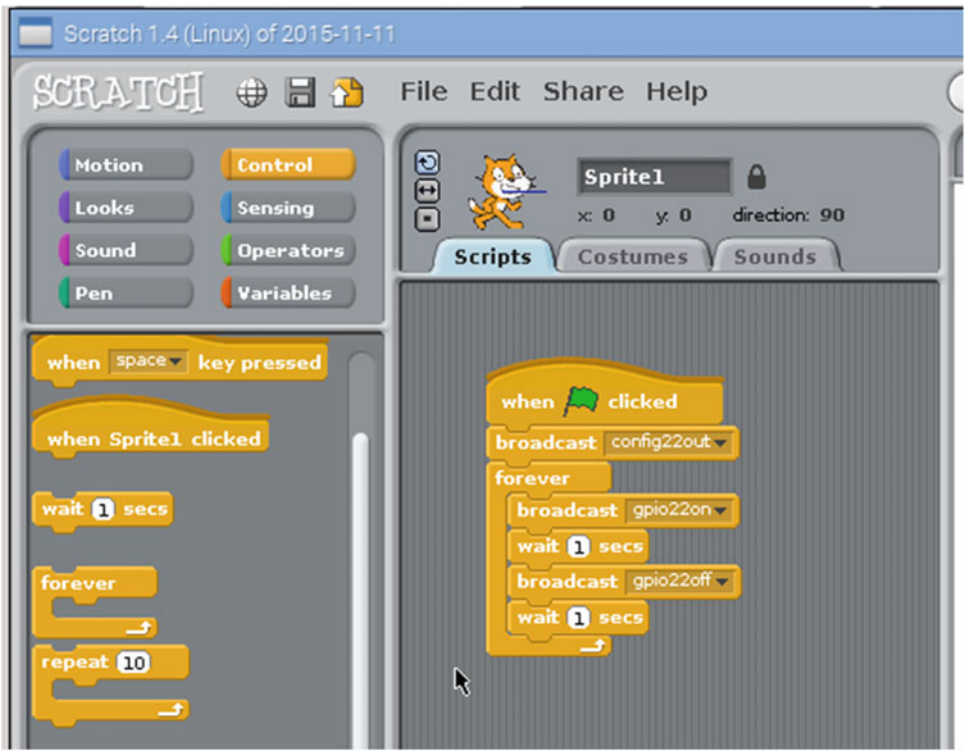


Figure 3-7. *Initial LED script*

All the blocks are from the Control scripts. These are all orange-colored, the same as the Control scripts tab. This indicates that these are all control commands, which control the flow of the program, or as in the case of the broadcast messages, are used to send messages to another program (the GPIO server). The broadcast tells the GPIO server when to turn the pins on and off.

The green flag at the top of this script indicates that this script runs whenever the green flag is clicked to start the program.

The first `broadcast` command is a `config` command.

This command can be split into three parts:

- `config` indicates that this is a configuration request
- `22` makes the changes to GPIO port 22
- `out` sets the port as an output

Remember that GPIO port 22 is the port that is connected to GPIO 22 on the processor. This is not the same as the physical pin number. The physical pin number that you have connected to is 15.

You then have a `forever` loop, which as the name implies, runs the code forever. Within the `forever` loop, there are `broadcast` commands that tell the GPIO server to turn the pin on and off and `wait` statements that cause the program to pause for a period of time.

The command to turn the GPIO port on, setting the output to +3.3V, is to `broadcast gpio22on`.

To turn it off again, setting the output to 0V, use `broadcast gpio22off`.

Again, there are three parts to this command:

- `gpio` indicates that this should be issued to the GPIO port
- `22` makes the changes to GPIO port 22
- `on/off` sets the port to +3.3V (on) or 0V (off)

When you click on the green flag, the code will run. It will configure the port and then alternate between the LED on and the LED off with a delay of one second between turning the LED on and off.

Adding an Input to the Scratch GPIO Program

Using the LED circuit (Figure 3-6) as a starting point, you will now add an input to the Raspberry Pi. Inputs can take many different forms. In this case, you'll be using a simple switch.

A *switch* is used to create a break in a circuit or join two parts of a circuit together depending on the position. There are different types of switches depending on how the switch is being used. The one you will use here is known as a push-to-make pushbutton switch. When the button is pressed, it pushes the conductors together inside the switch, thus completing the circuit. This is similar to a door bell push-switch, which completes the circuit causing the door bell to sound.

The type used here is designed to be installed onto a printed circuit board or a breadboard. It has four pins, although the pins on each side are connected together with just a single switch in the package. The ones I have used include clip-on button caps, which makes them stand out a bit more. The caps are not necessary and only fit on certain switches.

A photo of the switch along with its circuit symbol is shown in Figure 3-8.

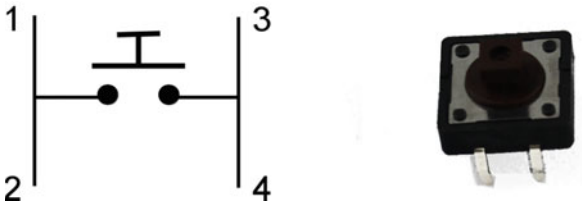


Figure 3-8. *Pushbutton switch and circuit symbol showing connections*

Using a Switch as a Digital Input

The GPIO port needs to be sent a digital on or off signal, which should be +3.3V for on and 0V for off. At first thought, you may consider using the switch to connect to the positive supply and rely on the pin being disconnected for a low 0V signal. Unfortunately that is unlikely to work, as when the power is disconnected,

the input will be considered floating. When an input is floating, the state is unpredictable and will often change between high and low.

Instead you need to connect the input to the positive voltage so that the supply is high when the switch is not pressed. The switch then connects the pin to the 0V supply to bring it low when pressed. A fairly high value resistor (usually in the tens of kΩ) is required for the connection to the positive rail to prevent a short-circuit. The Raspberry Pi has internal pull-up resistors that can be used rather than needing to add an external resistor[1]. The pull-up resistors can be enabled using the broadcast message `config<GPIOport>inpullup` and inserting the appropriate GPIO port number.

Adding the Switch to the Circuit

The diagram in Figure 3-9 shows the LED circuit you used previously with the switch added. The switch is connected to GPIO port 10 (physical pin 19) and to the 0V ground pin on the Raspberry Pi.

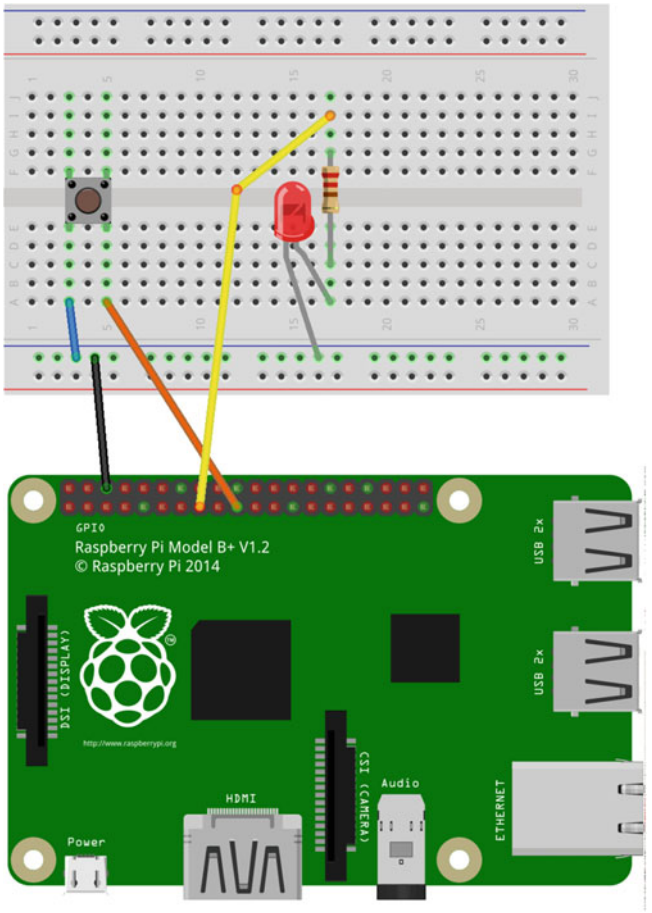


Figure 3-9. *Switch and LED circuit*

[1]It's actually possible to use this the other way around, using a pull-down resistor to represent a low input when the switch is not pressed and connecting to the positive supply when the switch is pressed. The use of a pull-up resistor is more common and is the default for the Raspberry Pi.

Next you need to add code to `Sprite1` to configure the GPIO port and detect when the button is pressed. The updated code is shown in Figure 3-10.

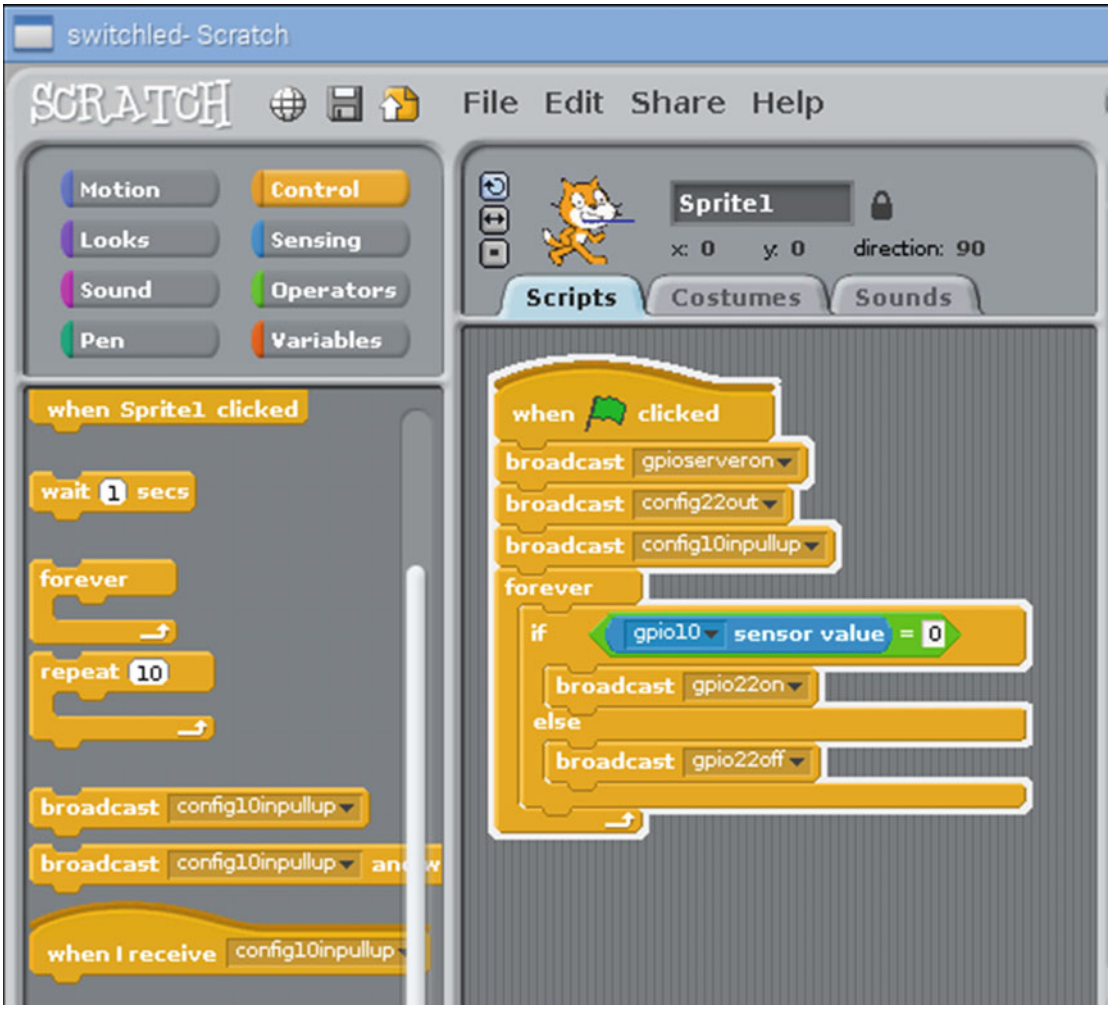


Figure 3-10. *Switch and LED Scratch code*

I added a new block at the start, which turns the GPIO server on rather than having to configure it through the menu. GPIO port 10 has been configured as an input with a pull-up resistor.

Within the `forever` loop, the sensor value block is used to check on the status of the switch. If it has value 1 then the switch is not pressed and the internal pull-up resistor is pulling the value up high. When the switch is pressed, the input is pulled low and the sensor value will be equal to 0.

The sensor value block is found in the Sensing section, but it will not show the GPIO value until the port has been enabled. You will therefore need to add the `broadcast config10pullup` entry to run that part of the code before adding the sensing block. This can be achieved by running in the usual way with the green flag to start and the red flag to stop, or by double-clicking on the relevant block of code.

When the code has been entered and the program runs, the LED will be initially off, turning on when the button switch is pressed and off when it's released. This is effectively performing the same as the LED circuit in Chapter 1, but now using an entire computer where the switch was able to handle this on its own. I have done this to introduce the way that inputs and outputs are handled in Scratch GPIO. You will now be adding a few more components and creating a game in Scratch using this circuit.

Robot Soccer

The reason for creating the circuit so far is so you can use it to make your own game. This game is called *Robot Soccer*. The basic idea is to have a robot goalkeeper that can move from side to side to catch the ball. This is a book about the electronics, so the programming is kept fairly simple with lots of opportunity for improvement or to change to your favorite sport or other activity. The goal fills the full screen so if the robot reaches the ball, that is a save, and if the robot misses it, that's a goal. When complete, the game will look like Figure 3-11.

Figure 3-11. *Robot Soccer game*

To be able to move left and right, you need to add a few more components. In fact, you are going to duplicate the circuit so that instead of one switch and one LED, you have two switches and two LEDs. The existing LED is red to signify that the robot missed the ball, so the new one you will add will be green to signify that the robot reached the ball. The green LED needs to be connected through an appropriate resistor to GPIO port 23 (physical pin 16) and the second switch connected to GPIO port 9 (physical pin 21). The other connection from the switches and LEDs will connect to the ground, which will be used for circuits going forward unless otherwise stated. This is shown in Figure 3-12.

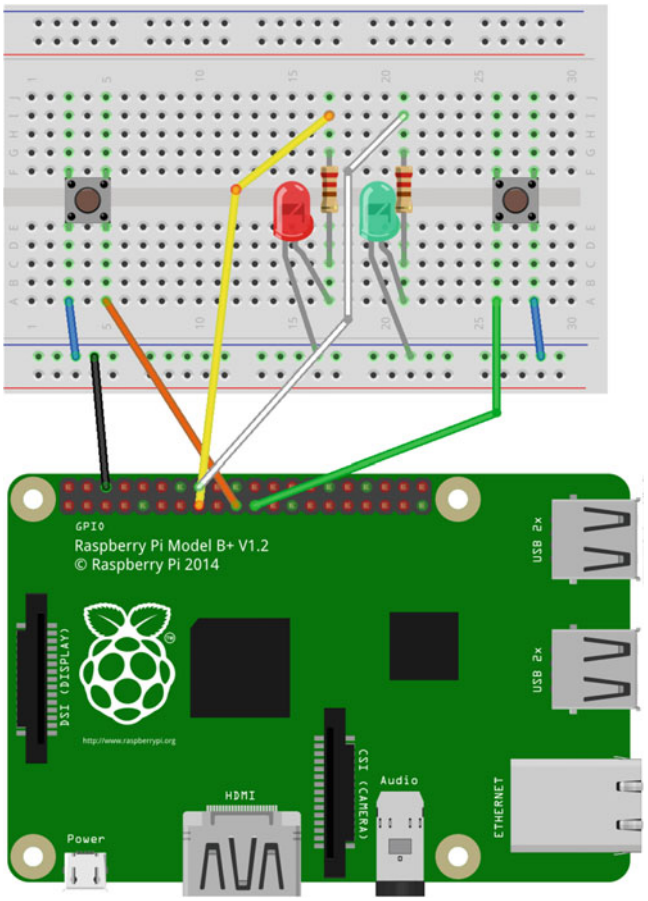


Figure 3-12. *Circuit for Robot Soccer*

Rather than editing the previous Scratch project, let's start a new one. Clicking File ➤ New creates a new project with the Scratch cat sprite. Right-click on the cat sprite and delete it. From the stage click, import and find the `playing-field` in the `Sports` folder. This is shown in Figure 3-13.

Figure 3-13. *Adding the background for Robot Soccer*

Add the setup code in Figure 3-14 to the Stage Scripts tab. The `config` commands are the same as you used previously in the switch and LED circuits, but with extra entries for the additional switch and LED. It includes the new broadcast for `gpioserveron`, which turns on the GPIO server rather than having to start it from the menu.

Figure 3-14. *Setup code for Robot Soccer game added to the Stage Scripts tab*

You now need to create the robot sprite. From the sprite section, choose New Sprite from a File. In the `Fantasy` folder, choose `robot3`. This is shown in Figure 3-15. You should then rename the sprite robot so that you can access it later. This is done by clicking on the name in the top of the Sprite area and changing it to `robot`.

Figure 3-15. *Adding a new sprite using the robot3 sprite file*

There are three code blocks that need to be added to the Scripts tab of the robot sprite, as shown in Figure 3-16.

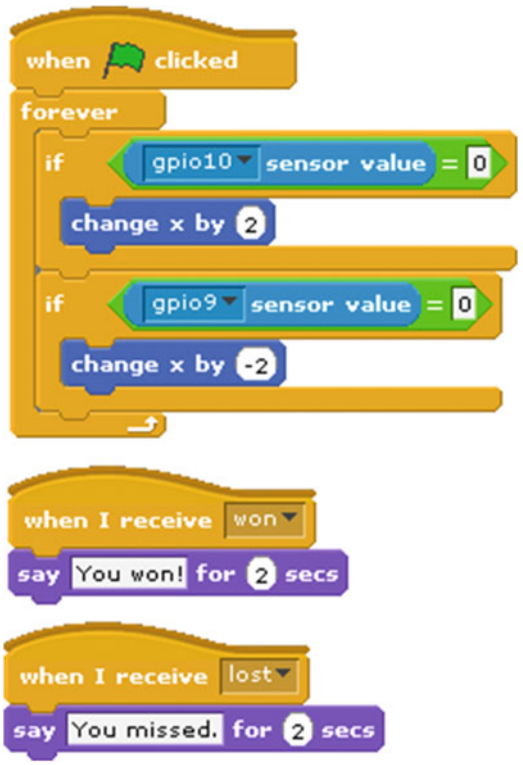


Figure 3-16. *Code to add to the Scripts tab of the robot sprite*

The main part of the code on the robot sprite is a `forever` loop checking for the status of the two GPIO pins 9 and 10. If the button is pressed then it increases or decreases the x position of the sprite, which makes the robot move left and right as appropriate. One thing that you may notice is that based on Figure 3-12, the switches are the wrong way around. You will turn the circuit upside down when you come to play the game. The other two code blocks receive a broadcast message and use the `say` command to show a message. The broadcast message is sent by the final sprite, which handles the rest of the logic.

The final sprite that needs to be added is the ball. This can be added from the file `soccer1` in the `Things` folder, as shown in Figure 3-17.

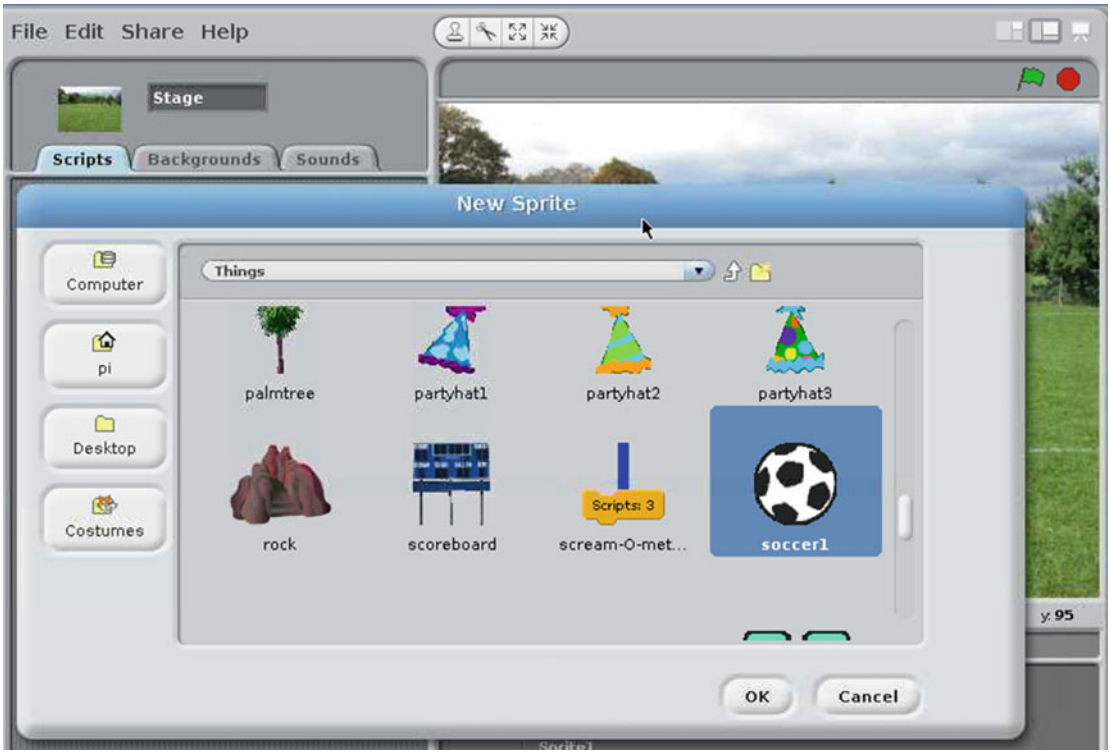


Figure 3-17. *Adding the ball sprite*

The code to add to the ball sprite is listed in Figure 3-18.

Figure 3-18. *Code for the ball sprite*

This code is a little more complicated than the previous code snippets, so I'll go into it in more detail. The first thing the code does is set the size of the sprite to 1% and position it at the center back. This will make the ball small so it looks like it's off in the distance. The Set Size block comes from the Looks blocks and the Go To block comes from the Motion blocks.

You now need to create a new variable called `direction`. The variable will hold a number that determines how far to the left (negative number) or right (positive number) the ball will move. To add the variable, select the Variables code blocks and then click Make a Variable (see Figure 3-19). When adding the variable, click For this Sprite Only, as the variable relates to this particular ball sprite.

Figure 3-19. *Creating a variable in Scratch*

Next you can set the variable to a random number by using the Pick Random option from the Operators code blocks.

The following block is a repeat block that will run the code to move and change the size of the ball ten times. The x direction changes based on the direction variable. If the random number happened to be a large number (up to the maximum of 20), the ball will go all the way over to the right; if it's a small number then it will be near the center of the screen. If it's a large negative number then it will go to the left. The y position has been changed slightly to make it appear as though the ball is rolling down the field. You can also make the ball increase in size as though it is getting closer to the robot.

There is also a Wait Code block from the Control blocks, which slows the game down. Otherwise the ball shoot toward you as though it was blasted from a cannon.

Once that has repeated 10 times then the `repeat` loop will be finished and it progresses to the `if` block. This tests to see if the ball sprite is touching the robot sprite. If you renamed the robot sprite when you first created it, then the robot option should be available on the pull-down; otherwise, it will be whatever your sprite is called.

The code then sends two broadcasts. If the ball was touching, the robot managed to get to the ball, so it broadcasts `gpio23on`, which will turn the green LED and send a broadcast called "`won`" which will trigger the robot to say "`You won!`" If the robot did not get to the ball, then it will turn the red led on (`gpio22on`) and trigger the "`You missed!`" message.

The code then waits for two seconds before switching the LEDs off again. Note that it turns off both LEDs even though one is already off. Obviously turning something off when it's already off will not do anything. You can instead add a variable and an additional `if` statement so you only send the message to the LED that is switched on; it's a matter of personal choice which you prefer.

Playing Robot Soccer

Now that it's complete, you can play the game. The game has been designed to be mounted on a board that's turned around so that the breadboard and button are on the bottom. I used a Raspberry Pi breadboard mount, but you could use a thin piece of wood or a thick card.

The finished project is shown in Figure 3-20.

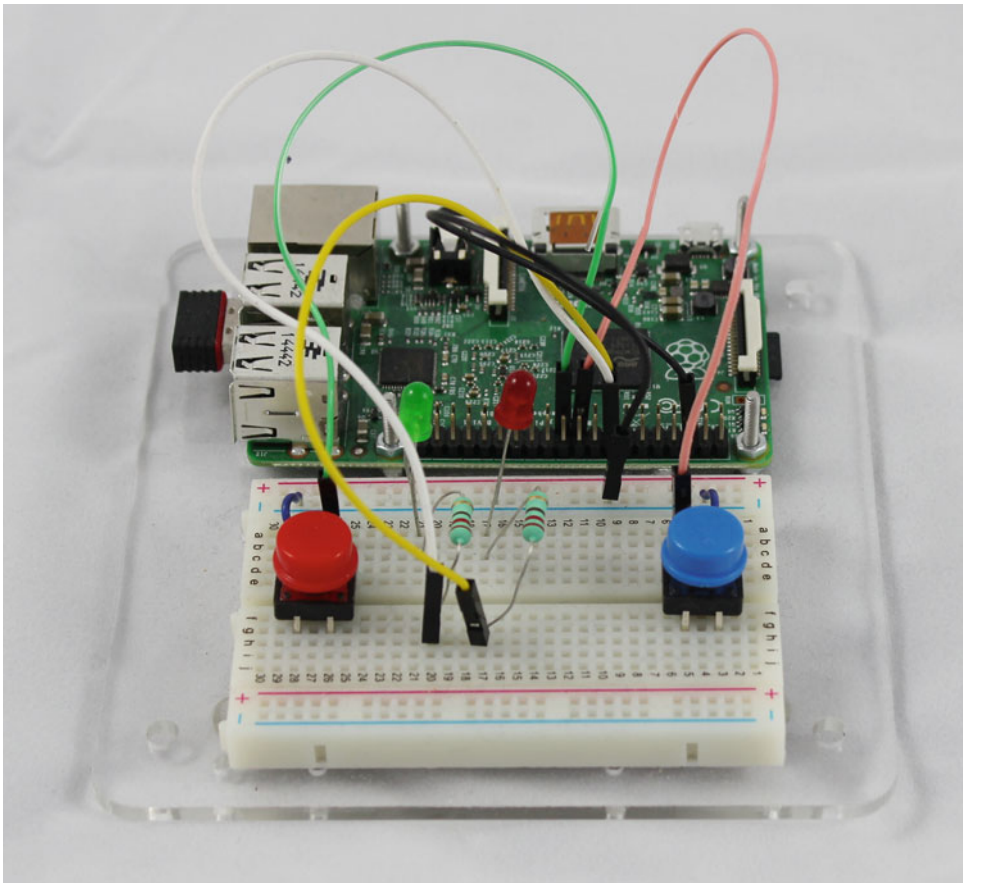


Figure 3-20. *Finished controller for Scratch Robot Soccer*

Click on the green flag in the top-right side of the Scratch stage to start. Press the left and right buttons to move the robot from side to side to stop the ball. The green LED will light up if you win and the red LED will light up if you lose. Click the green flag to start again.

Mars Lander Arcade Game

In this game, you are going to join the exploration of space by landing on Mars. This will be a arcade-style game controlled by a joystick. This will provide further insight into how Scratch can be controlled using GPIO inputs.

Making an Enclosure

In this project, you will create your own case with joystick and buttons. This is going to be something in-between a games console and an arcade games machine. In electronics terms, a box that holds an electronic circuits is known as an *enclosure*. You can go out and buy a mini-arcade game cabinet for the Raspberry Pi, but they are very expensive. In this project, you will see how to make a cheaper version that plugs into any HDMI TV.

Even though you are not using a full arcade cabinet, you will still need a box to house this. I used a Really Useful Box, which is a strong plastic box. You can use any suitably sized box, but it's worth paying a little extra for a strong box. As the box is made out of plastic, it's easy to make holes for the switches and for the connections to the Raspberry Pi. You will need a drill that matches the size of the switches and to file down the holes for the HDMI output and the power connector. The joystick and buttons are mounted to the top of the enclosure. The finished enclosure with joystick and buttons is shown in Figure 3-21.

Figure 3-21. *Enclosure with joystick for the arcade game*

Inside the enclosure, the Raspberry Pi should be mounted to one side of the box with a hole for the power connector, HDMI connection, and audio output. The USB ports will be inside the box, which can be used for wireless devices, but it's also possible to mount a panel-mount USB connector to allow external devices to be connected.

The photo in Figure 3-22 shows the inside of the joystick enclosure. This is shown the opposite way around compared to the previous photo, as the cable lengths of the switches make it difficult to open in the other direction. I used a breadboard and cobbler to make it easier to connect the wiring, although the wires could be connected directly to the GPIO port with an appropriate connector.

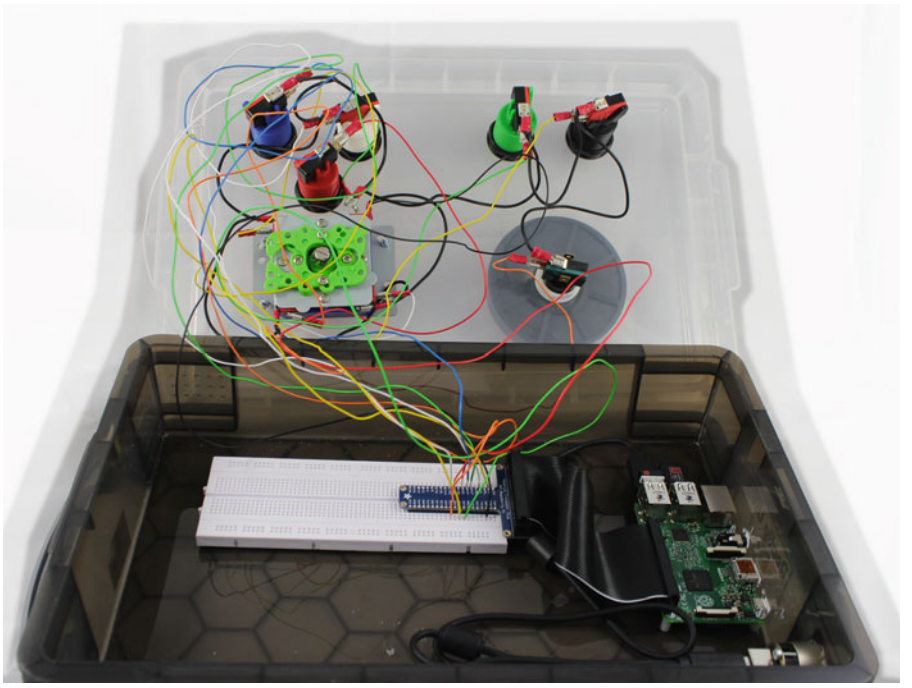


Figure 3-22. *Inside the joystick enclosure*

Note that the wiring looks very messy. This is designed as an initial prototype using wires that are extra long to make it easier to lift the lid and make changes to both the wiring and the Raspberry Pi. Even with these long wires, it is very easy for the wire to get pulled out of the breadboard. For a more professional and reliable connection, replace the solid core wires with a single ribbon cable or with flexible wires cable tied together, which would run in a neat bundle.

Adding Joystick and Switches

For this game you need a four-way joystick controller. There are two kinds of joysticks: digital ones that have a switch indicating which direction the joystick is pushed and analog ones, whose resistance changes as the joystick moves. The one you need for this project is a digital one that has four micro-switches. The micro-switches are pressed into the closed position as the joystick moves. This type of joystick can normally be found by searching for micro-switch joysticks from your favorite electronics supplier.

A *micro-switch* is a small switch similar in action to the pushbutton switches, but designed to be pushed by a mechanical device. These are often found in safety guards on machines or incorporated into other devices such as the joystick and the button switches.

I also used a jumbo switch and five more arcade-style button switches. These all use micro-switches with a plastic button to provide a large button to press. You will only be using one of the buttons for this project, but more will be useful in Chapter 9, when you will use the additional buttons to interact with Minecraft.

Wiring the Switches

The circuit for this project is fairly simple. Each switch will be connected between a GPIO input port and ground. This is the same way that the switches were connected in the previous project, using the pull-up resistors within the Raspberry Pi.

To make it easy to change in the future and to avoid the need to solder, I used a cobbler to take the pins from the Raspberry Pi to a breadboard. I then used solid core wire, crimped together onto crimp female spade connectors (the wire had to be doubled up to make a secure contact). This involves putting the wire into a metal barrel and then squeezing the barrel using a crimp tool. The crimp squeezes the barrel, which holds the wire in place and forms an electrical connection. The use of the crimps (which connect directly to the switches) avoids the need for soldering and also make it easier to reuse the switches. The terminals are connected using a crimp tool such as the one shown in Figure 3-23.

Figure 3-23. *Crimp tool with female blade connectors*

All the switches need to have one side of the switch connected to ground. This is done by having a wire from the ground connection on pin 6, which is then daisy-chained to one side of each switch. The normally-open terminal of each switch is then connected to the appropriate GPIO port, as shown in Figure 3-24. The color in brackets is the color of the wire used to connect the switch back to the GPIO port; this is for convenience to make it easier to identify each wire.

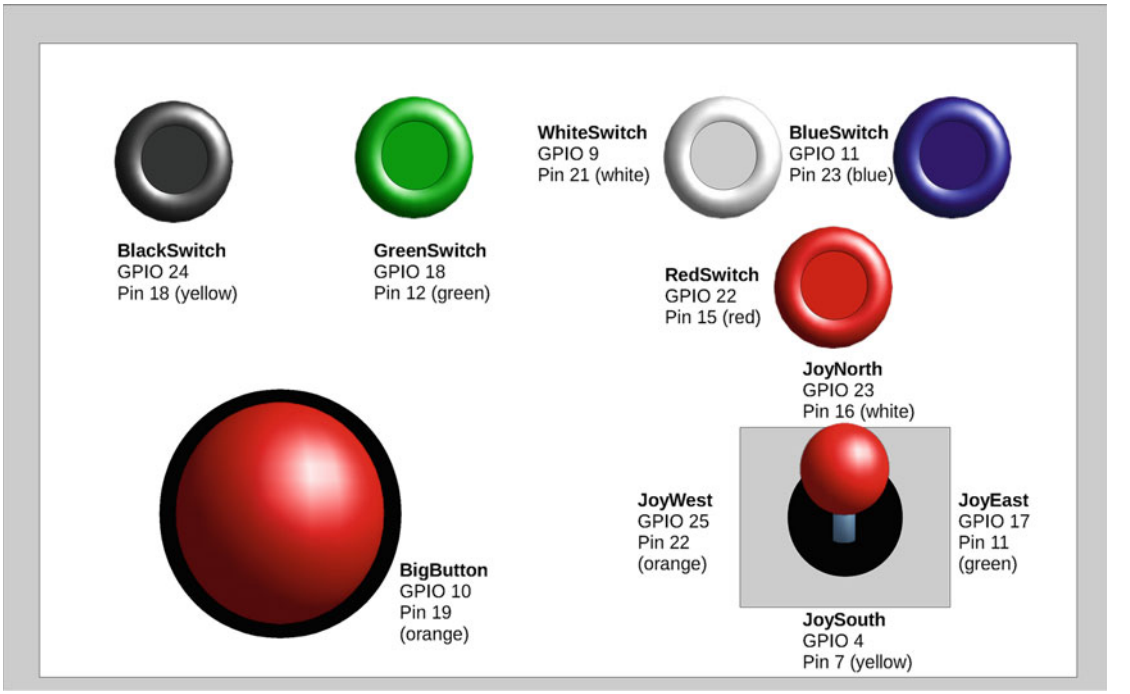


Figure 3-24. *Wiring details for the Joystick and arcade buttons*

One point to note about the wiring is that the switches on the joystick are physically located in the opposite positions than those shown in the diagram. The diagram is based on the direction of the joystick. When the joystick is pushed forward (JoyNorth), then the bottom of the joystick pushes in the opposite direction. So looking at Figure 3-24, the switch for JoyNorth is actually at the bottom of the joystick and the switch for JoySouth is at the top of the joystick. The same applies for JoyEast and JoyWest, where the direction of movement pushes the switch at the opposite side.

Creating the Game

You'll now see how to create the game in Scratch. Start a new Scratch project and delete the default cat sprite (right-click on the sprite and choose Delete). You will start by adding your own background and using that to set the GPIO ports as inputs.

Click on the Stage icon, then click on Backgrounds and Import. If you have downloaded the files to accompany this book (see the Introduction), the background is located in `/home/pi/learnelectronics/scratch/media/mars.jpg`. This is shown in Figure 3-25. Alternatively, you can use the `Moon` background in the `Nature` folder.

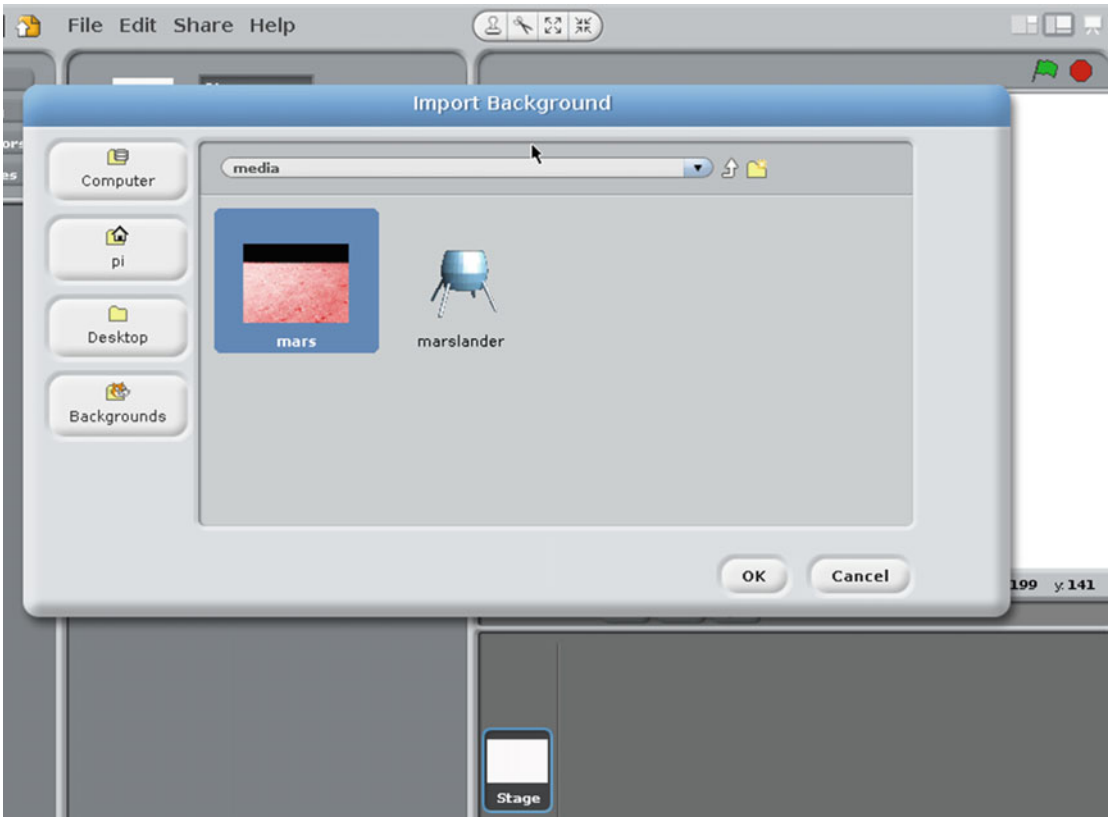


Figure 3-25. *Adding the Mars background in Scratch*

Now add the script listed in Figure 3-26 to the Scripts tab of the stage.

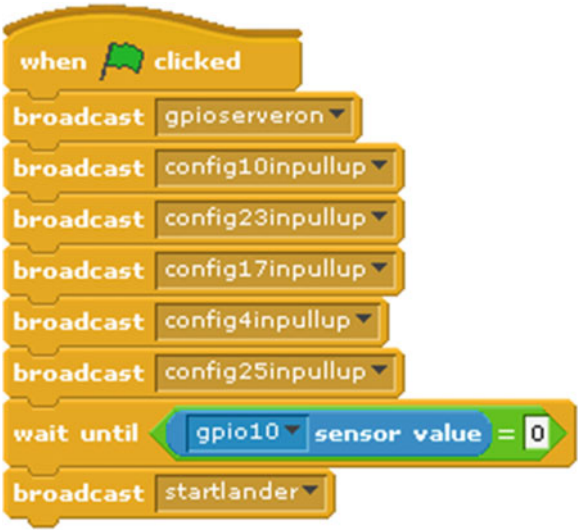


Figure 3-26. *Script for the Mars Lander stage*

The `config broadcast` messages are used to set the appropriate GPIO ports as outputs. The ports configured are in order: the large red button (used to start the game) and the joystick buttons—North, East, South, and West.

The script then waits until the large red button is pressed. Pressing the button pulls the input down to ground and hence changes the value of the sensor to 0. It then sends a broadcast to the `marslander` sprite to tell it to start the game.

After adding the code to the stage script, run it by clicking on the green flag on the top right of the stage. You haven't finished the game, so you won't be able to play anything yet, but running the broadcast messages will update the Scratch sensor blocks with the available inputs, making adding the controls easier.

Next you will add the craft that is going to land on Mars. If you have downloaded the files to accompany the book, it is available at: `/home/pi/learnelectronics/scratch/media/marslander.jpg`. Alternatively, you can choose another sprite from those provided with Scratch.

The image is of a basic lander that I created in the open source 3D modeling software called Blender.

Use the Choose New Sprite from File option and add the sprite, then rename the sprite to `marslander` using the sprite name box, as shown in the top of Figure 3-27.

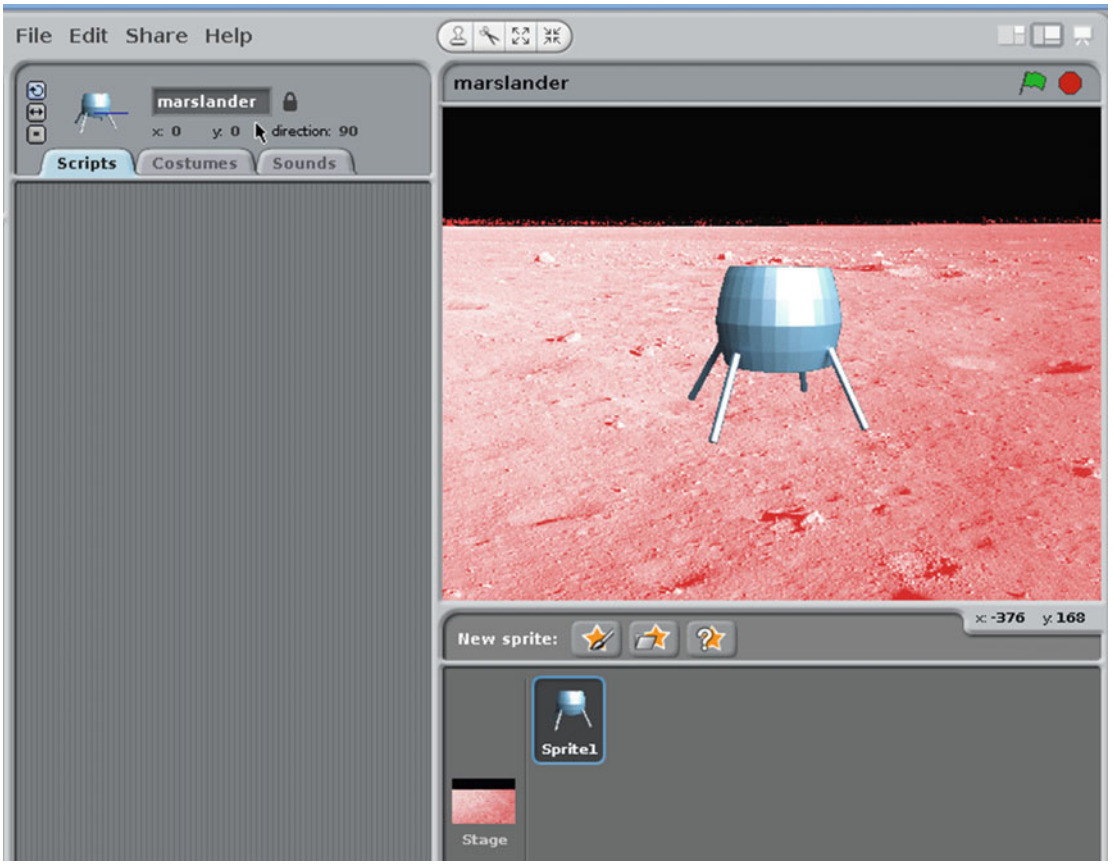


Figure 3-27. *Renaming the marslander sprite*

Now add the two scripts listed in Figure 3-28 to the Scripts tab of the `marslander` sprite.

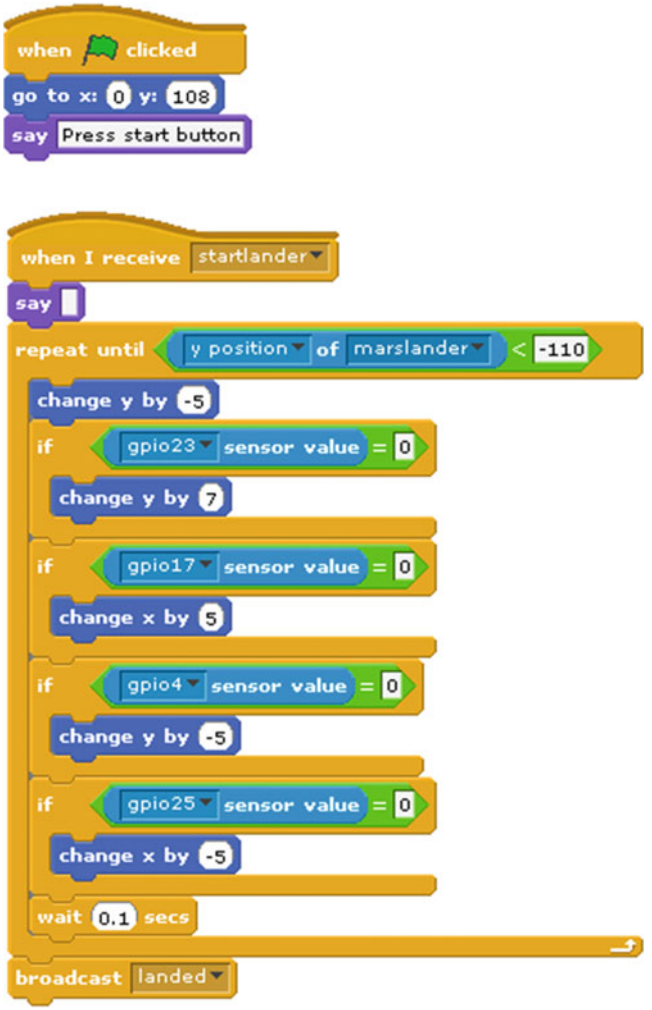


Figure 3-28. *Scripts for the marslander sprite*

This is quite a page-full, but broken down it shouldn't be too difficult to follow. The first script consists of three blocks of code. It starts with the green flag, so this part of the code will run when the user presses the green start program flag. It then positions the landing craft to the top center of the screen and gives a message that the start button (big red button) needs to be pressed to start the game.

The next script uses a "`When I receive broadcast`" block. This means that this block of code won't start running until after the start button switch has been pressed and sent its broadcast message. The `say` code block has no text in it, which is used to remove the start button message from the sprite. Most of the code is then within a loop, which continues running until the `marslander` sprite reaches the bottom of the screen.

In the loop, the sprite is moved down by five pixels[2] and then it checks each of the buttons in turn to see if any have been pressed. Depending on which switches are pressed, the lander moves in the appropriate direction. If more than one button is pressed, such as if the joystick is between two positions. then both buttons are pressed and the sprite will move appropriately. Once the lander reaches the bottom of the screen, it exits the loop and notifies the `landingpad` sprite through a broadcast message.

The final sprite is a landing pad for the lander to land on. The landing pad sprite will be created by drawing it by hand. Use the Paint New Sprite option from the sprite area. This will launch the Sprite Editor window, where you can create your own landing pad. As you can see in Figure 3-29, I created a fairly plain black rectangle, but feel free to add a bit more creativity and create your own personalized landing pad.

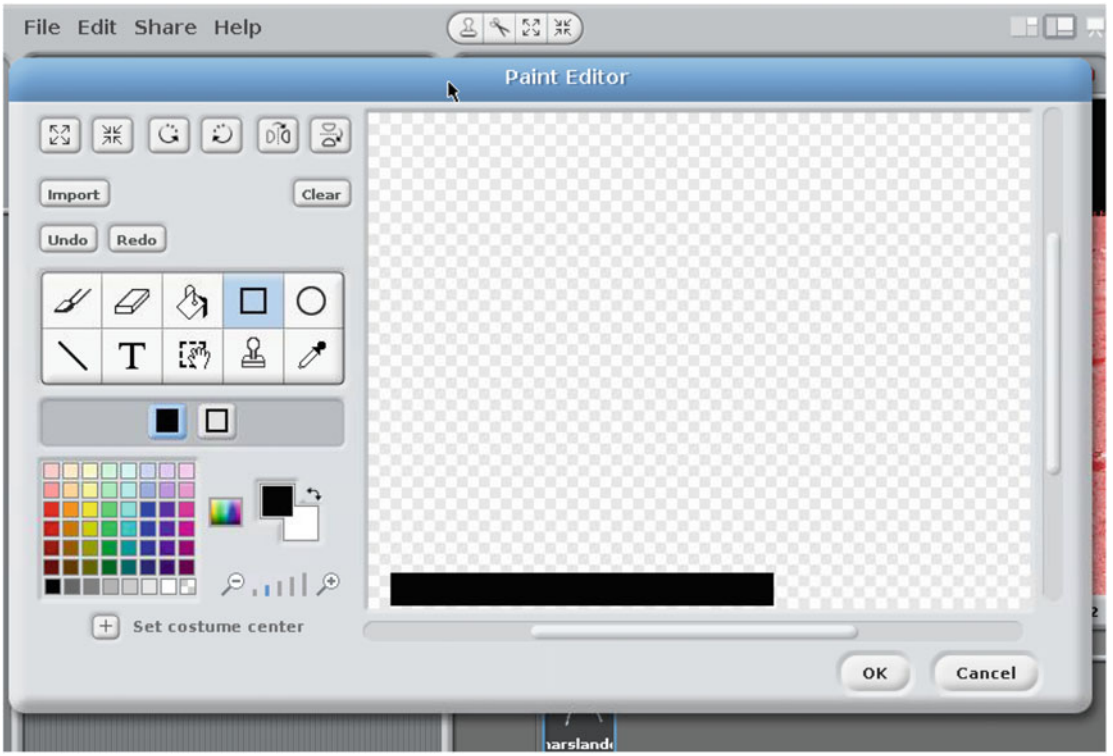


Figure 3-29. *Paint a new sprite for the landing pad*

Once you've completed your landing pad, click OK to add it to the stage. The landing pad should be about the same size as the `marslander` sprite. You can use the grow and shrink buttons, near the top of the screen, to change the size of the sprite if needed. This is shown in Figure 3-30.

[2]I have used the word *pixels* to represent the virtual pixels used by the Scratch stage. The actual number of pixels on the screen will be higher.

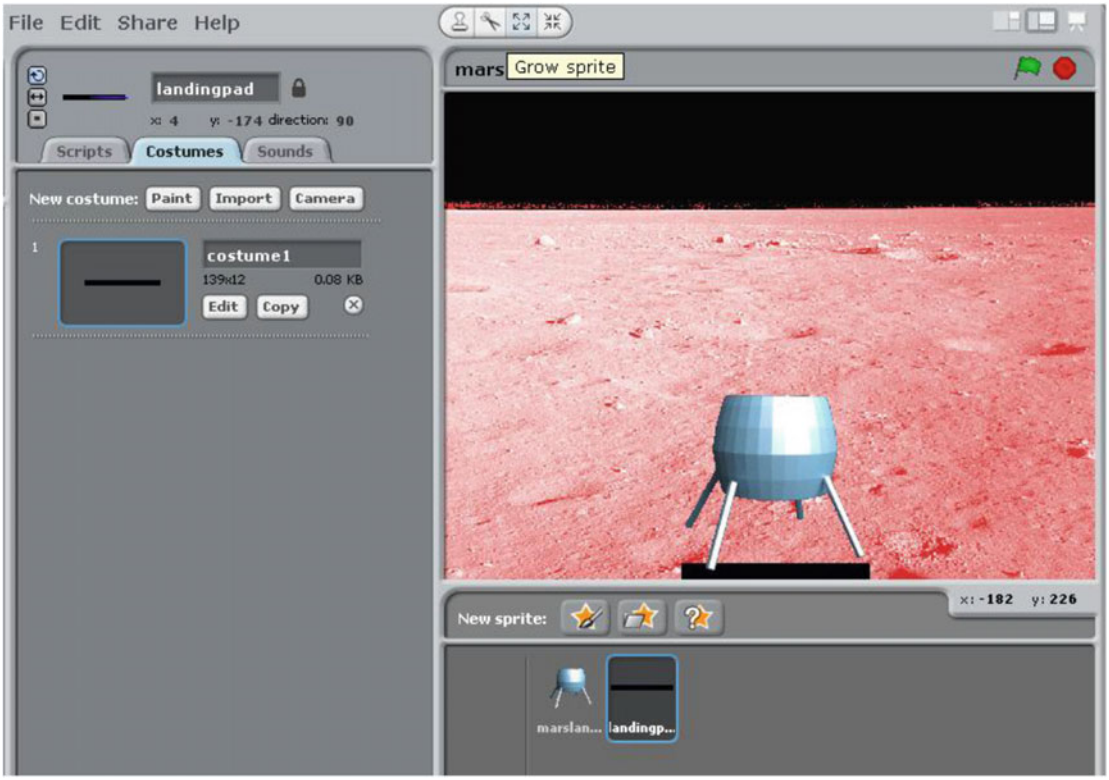


Figure 3-30. *Resizing the landing pad*

Once you've created the `landingpad` sprite, you will need two scripts, which should be added to the `landingpad` scripts area. The first is to set the position of the landing pad and the second is to detect the position of the lander. The first of these is shown in Figure 3-31.

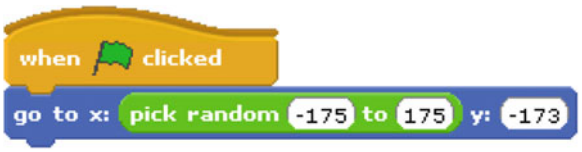


Figure 3-31. *First script for the landingpad sprite*

This script runs when the game starts and positions the sprite at a random position near the bottom of the screen. The y position is fixed at -173. It chooses a random x position. The x position is based on the center of the sprite so the range (-175 to 175) is quite a bit less than the total width of the screen to allow for the size of the landing pad. The second script is shown in Figure 3-32.

Figure 3-32. *Second script for the landingpad sprite*

At first glance, this looks a bit scary. The `if` statement checks several different sensor variables and forms a long line. If you break it down, it should be fairly straightforward to understand.

This script is triggered by the `marslander` sprite, which sends a broadcast message called "`landed`" when it is positioned at the bottom of the screen. The long `if` statement then checks the position of the lander in relation to the landing pad. It's not possible to check if the position is the same as the landing pad, as that would match only if they were exactly the same, which is pretty unlikely. Instead, you need to check within a reasonable range. The code therefore checks that both the following conditions are met:

```
x position of marslander < x position of landingpad + 15
```

and

```
x position of marslander > x position of landingpad - 15
```

The first line checks to see if it is at least far enough to the left. It does this by checking that the x position of the `marslander` Scratch is to the left of `landingpad`, plus `15`, which provides a bit of leniency in the game. The second comparison does the same for checking whether the lander is far enough to the right. If both of these conditions are met, then the lander is in the middle of the landing pad, +/- 15 pixels.

The appropriate message is shown on the sprite. It shows a `success` message if it landed on the pad; otherwise, it sends a message that the landing pad was missed.

Playing the Game

To play the game, press the green flag on the top of the stage. Then press the big red button on the controller to start the game. Move the joystick left and right to move the lander. Pushing the joystick up will cause the lander to move up slowly and pushing it down will increase the speed that the lander moves down. Landing the craft in the center of the landing pad is a win.

The difficulty of the game is based on the speed and the accuracy of the landing pad position. The speed can be changed by changing the value of the `delay` block. The accuracy that the lander needs to be positioned is based on the value used in the `if` statement. In the default code this is 15, reducing this will require the lander to be positioned more accurately. Increasing it will allow the lander to be farther from the center of the landing pad.

More Games

This chapter looked at simple circuits using LED outputs and switches as inputs to the Raspberry Pi. Both circuits are a form of game controller used to create two games in Scratch.

These games can be changed to create your own personal game. If you don't like soccer then change it into a tennis game, or if you prefer to remain within the earth's atmosphere, change the space craft into a hot air balloon landing in Central Park.

You could also add some realism to the game by having the descent speed up as though gravity was pulling it down. Or if you're creating a hot air balloon game, you could add wind to blow the balloon off course.

In the next chapter, you'll look at using Python instead of Scratch and controlling some bigger and brighter outputs. You will also reuse the arcade joystick in Chapter 9 when you learn to use it to control Minecraft.

CHAPTER 4

Using Python for Input and Output: GPIO Zero

This chapter is going to look at lights. Not just any old boring desk lamp, though, the chapter looks at how lights can be controlled by the Raspberry Pi. You are going to connect some fun disco lights and projects using bright LEDS. In the process, you will learn about the most important electronic component, the transistor. You will use some different types of transistors to switch lights on that need more current than the Raspberry Pi can provide.

In Chapter 3, the projects used the graphical programming environment Scratch, but from here on you are going to be using Python, which is a text-based programming language. There are some rules about writing programs that need to be followed, but once you get the hang of that, it's not much more difficult than using the code blocks from Scratch. Later in the book, you will create a graphical program for one of the projects.

In this chapter you will be using the Python GPIO Zero module. This is a fairly new module that greatly simplifies controlling the GPIO ports using Python. You need a recent version of Raspbian installed on your Raspberry Pi. For more details about upgrading, see the introduction.

Power Supplies

First let's look at the power supply that you will use for this. So far, you have relied on the output from the Raspberry Pi to drive the electronic circuits. This is okay for the simple low-power circuits you have used so far, but is not sufficient for some of the brighter lights you are going to look at in this chapter.

Next you will look at a few different options that can be used to provide power for more powerful LED lights and for other circuits that may be connected to the Raspberry Pi. You will be looking at DC circuits at first, but will learn about the AC power supplies later in the chapter. DC stands for direct current, which has a positive and negative terminal and where the current flows in one direction, from the positive to the negative terminal. AC power is alternating current, where the direction of the current changes so that half the time it flows in one direction and the other half it flows in the other direction. In most AC supplies, the current changes direction around 50 to 60 times per second. Almost all the electronic circuits in this book are based on DC, but the main electricity coming into your home is AC. The power supplies in the following sections assume that you want a DC current to power the circuit.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_4

+5V from the Raspberry Pi GPIO

If you look at the pin layout for the GPIO pins of the Raspberry Pi, then you will see that some of the pins are labeled as 5V. As I have already said the outputs from the GPIO pins are 3.3V, so what's this all about? The 5V connection on the GPIO connector is a fixed 5V power supply that can be used to power a low-power circuit from the Raspberry Pi. In fact, you could even connect an external 5V supply to that pin and use that to power the Raspberry Pi. I won't be using this 5V supply in this chapter as the amount of current that can be taken from this supply is limited, especially on the older versions of the Raspberry Pi, but it could be used to power low-power electronic circuits, which you will look at later in the book.

Another USB Power Supply

A good form of power is through another USB power supply/charger separate from the one used to power the Raspberry Pi. A portable cell phone charger can be used—these are readily available and fairly inexpensive. You can buy a USB power break-out connector, which allows you to connect power to a breadboard. The one shown in Figure 4-1 is suitable for connecting to a micro-USB power supply. It is also possible to get ones that connect to standard USB connectors.

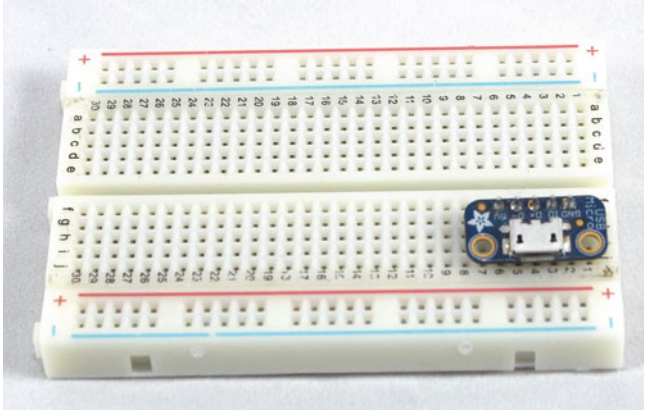


Figure 4-1. *A USB power breakout connector*

You will need to check the labeling on the power supply to ensure it provides sufficient current for the circuit. A good quality supply will provide around 1 to 1.5A at 5V. I recommend buying a good quality power supply from a reputable supplier as there have been incidents of problems with supplies by less scrupulous manufacturers. The official Raspberry Pi power supply (Figure 4-2) is a good quality power supply made by Stontronics. The plug includes adapters for the power sockets in several different countries. It provides 5V up to a maximum of 2A.

Figure 4-2. *USB power supply*

Other External Power Supplies

The USB power supplies are good if you have a 5V circuit with modest power requirements, but if you have a circuit that runs at a different voltage or needs more power than the USB power supply can provide, you will need to use a different power supply.

These are available built into plugs (sometimes called wall warts) or as a power brick that connects to the plug through a lead. Most of these have a fixed voltage output, although it is possible to get ones that have a selector for different output voltages. I have a variety of power supplies for different purposes and will be mainly using a 5V and a 12V power brick for projects in this book. A 5V power brick that will be useful for some of the projects in this book is shown in Figure 4-3.

Figure 4-3. *5V DC power adapter*

Power bricks normally come with barrel jack at the end for plugging into equipment. A common connector is known a 2.1mm barrel jack and it has a 2.1mm pin and a 5.5mm sleeve. The center of the jack (sometimes known as the pin or tip) is often connected to the positive supply and the outside of the barrel (sometimes called the sleeve) is connected to the negative. This is not the case for all supplies though, so check the manufacturer's datasheet before connecting. The image in Figure 4-4 is a DC power supply showing that it is a positive center supply.

Figure 4-4. *DC power adapter center positive indicator*

You also need to check that it is a DC supply, as they are also available as AC power supplies. When choosing a power supply, you will need to ensure that the voltage matches the required voltage and that it is capable of supplying at least as much current as is required. There is no harm in connecting a power supply with a higher current rating than is required, as the power supply will only provide the current that the circuit requires, but you should not use a power supply with a higher voltage unless you know that it is within the tolerance of circuit power supply requirements.

You can get a female connector for mounting into an equipment enclosure or to connect to a trailing lead. These normally require soldering, but you can also buy a female connector to screw terminal, which is particularly useful for connecting to a breadboard when testing a circuit. Figure 4-5 shows an example of a DC power adapter to screw terminal.

Figure 4-5. *DC power adapter to screw terminal*

As with the USB power supply, I recommend only using a power brick from a reputable supplier that meets the appropriate safety requirements. The output power supply from a power brick is normally low voltage and safe from electric shock. (typically 5 to 12V). A good-quality power supply should also include short-circuit overload protection, but you should not rely on this protection. Ensure that any circuits connected to these supplies do not draw more than the specified maximum current. As an extra precaution you may want to include a fuse in your circuit, which I will explain later with the disco lights.

Main Power Supply

It is possible to create your own low-voltage power supply by building a power supply circuit that connects directly to the main supply. I do not recommend this unless you know what you are doing.

The main power supply in your home is dangerous. If you come into contact with a bare connection, it can kill you through electrocution, or overloading a supply or wire could cause a fire. Unless you know what you are doing, you should only connect equipment to the main power supply that already includes the appropriate insulation and protection (such as a wall wart).

■ **Caution** Do not attempt to connect a homemade circuit directly to the main electricity supply unless you know what you are doing—it is not worth the risk!

Batteries

I discuss using batteries to power the Raspberry Pi in more detail in Chapter 8. But if it's just an external circuit that is being used, and the power requirements are quite low, batteries can be a good way of powering a circuit. An obvious advantage of batteries is that they are portable and so make it easier to move your project around, but do make sure that the batteries have sufficient charge. If the circuit isn't working or behaves differently from how you expect, it could be because the batteries need replacing or recharging.

Brighter LEDs with a Transistor

In Chapter 3, you created some simple circuits that had LEDs and a resistor connected directly to the Raspberry Pi GPIO ports. This worked well with a standard LED that uses only a small current (around 10mA is usually sufficient), but when swapping for larger or brighter LEDs, the current needed to light the LED is going to be higher. The maximum current that can be drawn from an individual Raspberry Pi GPIO port is around 16mA. It will be even less if you have lots of devices connected.

The LED I am going to use first is a 10mm white LED. This is a bright LED that you want to drive at around 20mA to give off a good amount of light. This is more than the GPIO port can supply, so now it's time to look at adding an electronic component that can increase the amount of current flowing (amplify the output). The component you will use is a transistor.

■ **Note** When referring to digital signals, we won't necessarily know the actual voltage, which depends on the individual components and the voltage that the circuit is supplied with. Instead we use the terms high and low. A high logic level indicates that the voltage would be near to the positive voltage of the circuit (3.3V for the Raspberry Pi GPIO) and low logic level means that it is near to 0V or ground. These are also sometimes referred to as *on* for a high signal and *off* for a low signal.

Transistor

The transistor is a semiconductor device, which means that under some conditions it allows current to flow (acts as a conductor) and in other conditions restricts the flow of current. It is this property that is important in electronics and is the basis of most electronic circuits, including computer processors.

The first transistor you will look at is known as a bipolar transistor. Specifically this will be an NPN bipolar transistor (the NPN relates to how the transistor is made and as a result how it operates). The transistor has three connections known as the collector, base, and emitter (represented by the letters C, B, and E on the figures). When a small current flows between the base and emitter, it allows a much larger current to flow between the collector and the emitter. The transistor is an analog component, so varying the base current changes the corresponding collector current. By providing an appropriate signal, the transistor can act as a digital switch. This is achieved by working at the two extremes of the transistor. The first condition is where it is switched off, not allowing any current to flow through the collector. The other extreme is where it is switched fully on, allowing a large current to flow through the collector. This is referred to as using it as a transistor switch. The full on condition of the transistor is called the saturation region. A circuit diagram of the transistor switch is shown in Figure 4-6.

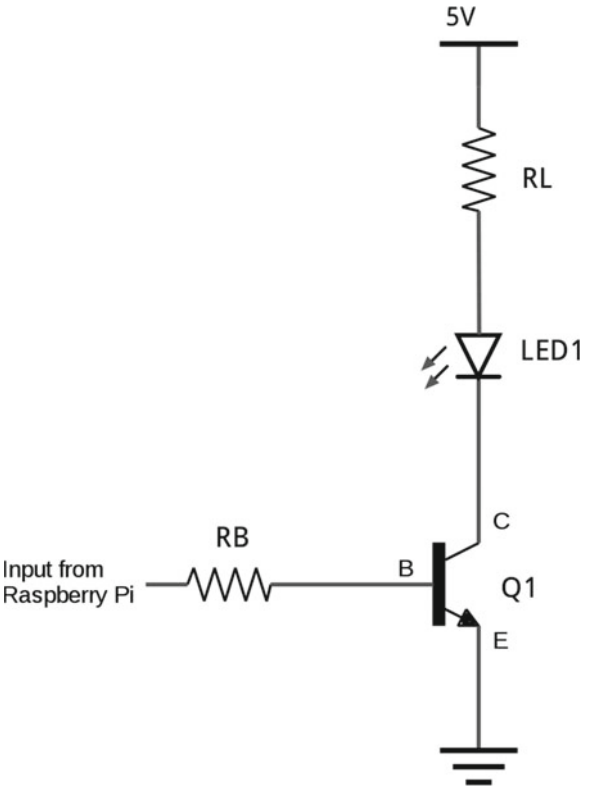


Figure 4-6. *Circuit diagram for a transistor switch*

You will notice that Figure 4-6 looks very different from the earlier circuits. Previously, you had a breadboard layout that showed the physical appearance of the components and wiring. This figure shows the components as symbols rather than showing their physical appearance and shows the connecting wires as straight lines. This is known as a circuit diagram or schematic. You will look at circuit diagrams in more detail later in this chapter—for now we will use the figure to show how the components are connected.

The circuit shows two resistors labeled RB and RL, an LED labeled LED1, and a transistor labeled Q1. The connection from the Raspberry Pi GPIO port goes through a resistor RB and connects to the base of the transistor. When the GPIO port is switched on, a current flows through that resistor and between the base and emitter of the transistor. This current switches the transistor on, allowing a larger current to flow

through resistor RL, the LED, and then from the collector to the emitter of the transistor. The power supply used for the LED is from a 5V supply independent of the Raspberry Pi power supply, although the ground (0V connections) needs to be connected together.

When using a transistor as a switch, the main characteristics of the transistor that need to be considered are the maximum current that it can switch, which is referred to as the collector current (usually provided as I_C), and the current gain of the transistor, referred to as h_{FE}. You also need to know about the voltage between the base and emitter to work out appropriate resistors, although that is not necessarily a characteristic that is used to decide on which transistor to use. You can find out lots of information about electronic components from their datasheets. A detailed explanation of datasheets is provided in Chapter 11.

You could use a number of different transistors for this circuit, but I am looking at two different types: the 2N2222 (in particular, the P2N2222A[1]) and the BC548. These are both common transistors used for hobby projects. You will use only one of these for calculating the appropriate resistors, but it will be possible to use either in the circuit depending on which component you are able to buy. It will also provide a useful comparison of different transistors, which can be useful when designing your own circuits in the future.

First, check the suitability of these transistors. The maximum collector current for the 2N2222 is 600mA and the BC548 is 100mA. So both of these are easily able to support the 20mA that you need to control the bright LED.

The *gain* is a measure of how much more current can flow through the collector compared to the current through the base of the transistor. The actual value depends on a number of factors and conditions at the time of manufacture. In the case of the 2N2222, it is an amplification factor of about 75 and for the BC548 about 110. If you take the lowest of these, it means that either transistor can switch 75 times the current through the collector compared to the base current. The 2N2222A is used for the rest of these calculations, but due to the similarity between these two transistors, either transistor can be used in this particular circuit.

Calculating the Resistor Sizes

You need to calculate the size of the two resistors. First let's look at RL, which will protect both the LED and the transistor from having too much current through them. If too much current flows through the LED, not only is it a waste of energy, it could result in permanent damage.

To work out the resistor size, you need to know the voltage dropped across the resistor. The voltage will be the supply voltage minus the voltage dropped across both the LED and the transistor. The voltage drop across the transistor varies depending on the manufacture and the current flowing through it. The voltage between the collector and the emitter is referred to as $V_{CE(sat)}$ on the transistor specification. The value is approximately 250mV, which is the value you will use.

To summarize, you have the following values:

- Supply voltage: 5V
- Voltage across the LED: 3.3V
- Voltage across the transistor: 250mV

The voltage across the resistor RL is therefore 5V – 3.55V = 1.45V.

To calculate the resistor value, you use Ohm's Law, which you saw in Chapter 1. To calculate resistance then, you use V/I:

$$1.45V/0.02A = 72.5\Omega$$

[1]I will refer to this as 2N2222, but there are differences between the versions of this transistor. The specifications of P2N2222A has been used.

Using the next value up in the E12 series of resistors is 82Ω.

You can now work out the actual current using the selected resistor to check that the current is appropriate. Using Ohm's Law, the current is V/R. This works out as 1.45/82 = 18mA.

Alternatively, you could have used a 68Ω resistor, which would have worked out to 21mA. Again this would have been okay because it would still have been within the 30mA maximum that the LED could use. This is a fairly common occurrence within digital circuits where it is possible to use components whose values are near the desired value.

To work out the size for RB, you need to work out the current you want to flow into the base of the transistor and then choose an appropriate resistor. The current you need at the base is worked out using the collector current divided by the gain (h_{FE}). Use the h_{FE} from the 2N2222A as this has the lowest gain. This gives 20mA/75 = 0.3mA.

You normally use a factor of 10 to ensure that you are well above the required base current. This ensures that you are in the saturation region where the power lost within the transistor will be at its minimum. Multiplying by a factor of 10 gives you 3mA, which is the current you need at the base of the transistor.

The following is now needed to calculate the value for the base resistor:

- Voltage from the GPIO pin: 3.3V
- Voltage drop across the transistor: (V_{BE}) 0.7V

Voltage across the resistor will be the GPIO on voltage minus the voltage dropped in the transistor. 3.3V – 0.7V = 2.6V.

You require 3mA, so using Ohm's Law to calculate the resistor gives 2.6V/0.003A = 866ohms. The nearest value is 1kohm using the next higher value or 820ohm for the next lowest. I have used 1kohm, which gives around 2.6mA to the base of the transistor. Although the desired value is 3mA, this included a factor of 10, so we will still be well within the saturation region.

So the values used are:

- RL = 82Ω
- RB = 1kΩ

The resulting circuit layout on a breadboard is shown in Figure 4-7.

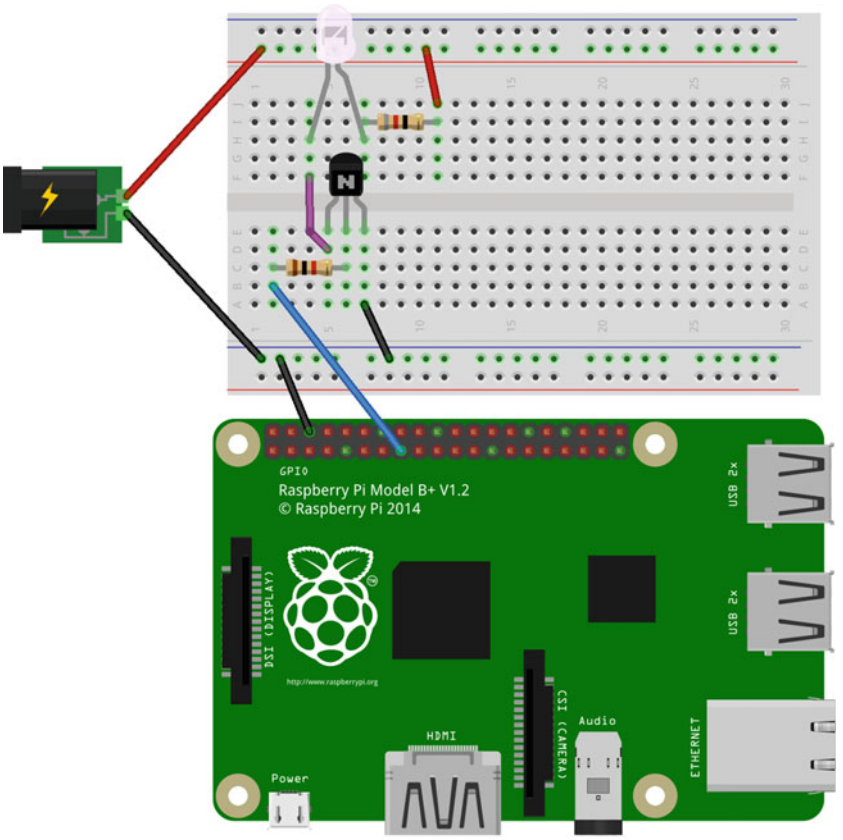


Figure 4-7. *Breadboard layout for the bright LED circuit*

You will see that I have used an external power connector in this circuit. You could use any of the other power supplies discussed earlier. This needs to be connected the correct way, with the red wire going to the top of the breadboard being from the positive supply of the power connector. Also note the orientation of the transistor, which is connected with the flat part of the transistor facing forward in Figure 4-7. The pin layout for the 2N2222 and the BC548 is the same, although that is not the case for all transistors. You will also need to ensure that the LED is connected correctly.

This uses the same GPIO port that you used in the first Scratch LED circuit, which is GPIO port 22 (physical pin 15). If you want to try this as a brighter version of the LED circuit, you can run the code from Chapter 3 (see Figure 3-7). Next, you will learn to develop some code in Python, which is a more powerful programming language.

Introduction to Python

Python is a popular programming language that is used throughout education and industry. Python, a text-based language, involves writing the code that is run by the Python interpreter. The examples used here will be run from the command line, but you will create a GUI (graphical user interface) application in Chapter 6.

There isn't room to provide a full guide to Python in this book, but this chapter covers some of the basics to get you started. If you haven't done any programming in Python, you may want to check out one of

the many books dedicated to teaching Python programming, such as Python Programming Fundamentals by Kent D. Lee (Apress, 2015).

There are two versions of Python installed on the Raspberry Pi. The first is version 2.7, which is that last version of the "old" Python, and Python 3, a newer version that is not compatible with the earlier version. There is a lot of software that has been written that will run with both versions, but code written for one version may not work with the other. I encourage using Python 3 because that is clearly where the future of Python lies and that's what the examples in the majority of this book use. At the time of writing the latest version on the Raspberry Pi is 3.4.2. Future versions are expected to maintain compatibility with version 3 for the foreseeable future.

As mentioned, Python is a text-based programming language. If you are used to a block-based programming language like Scratch, this can be a big jump at first, but it shouldn't take you long to get used to it. Once you've learned the basic rules, the additional flexibility will increase the scope of what you can achieve.

To get started with Python, I recommend using its IDLE (Integrated Development Environment). You can then go on to develop using other editors in the future, but IDLE includes some features that let you run your code directly. IDLE can be launched from the start menu, under Programming, as shown in Figure 4-8.

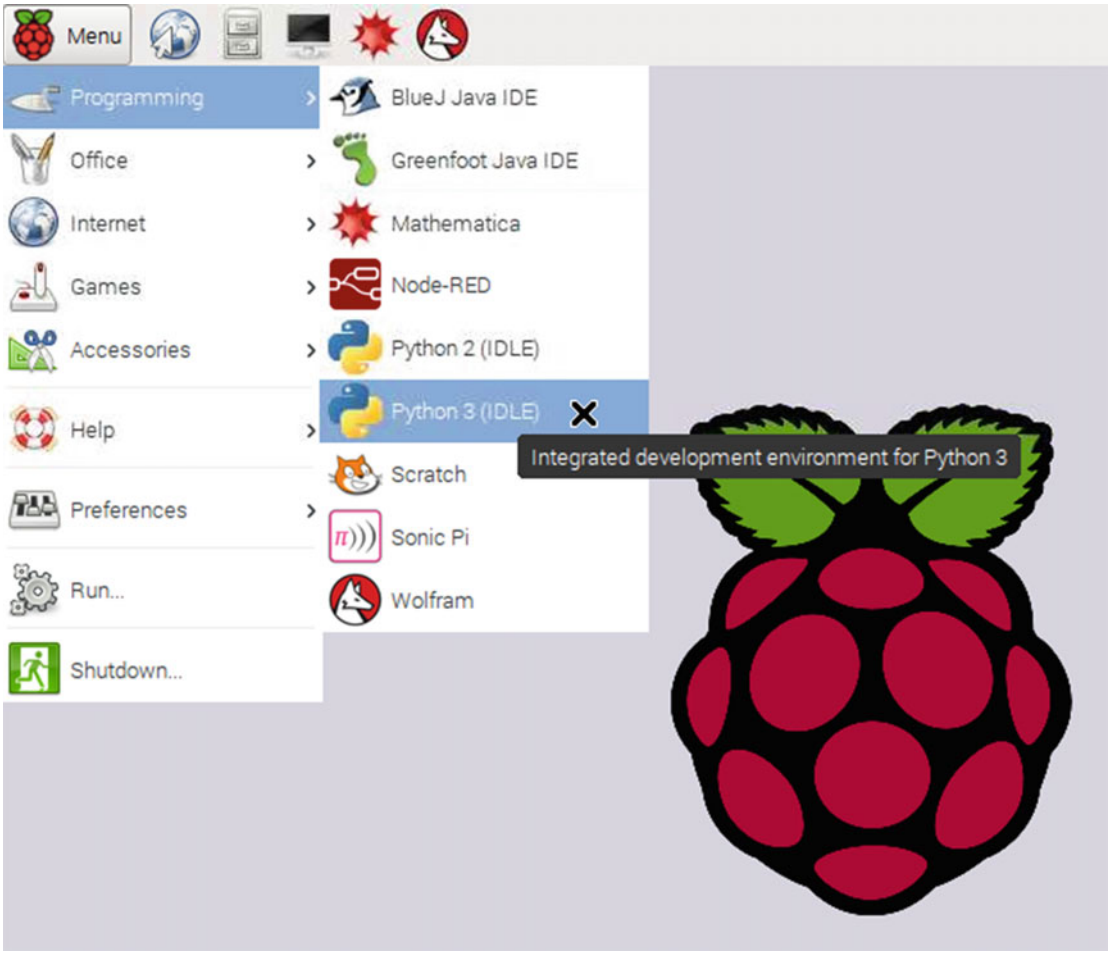


Figure 4-8. *Launching Python 3 IDLE*

Launching IDLE for Python 3 opens the Python shell. This is an interactive environment where you can enter commands that will run directly. This can be useful to test a short snippet of code or to understand what a certain function does, but for most programs we will want to run the IDLE text editor. Choose File ➤ New File to launch the text editor window. I suggest leaving both windows open and visible when programming, as the shell window will be used to test your program. Figure 4-9 shows the Python shell on the left and the programming text editor on the right.

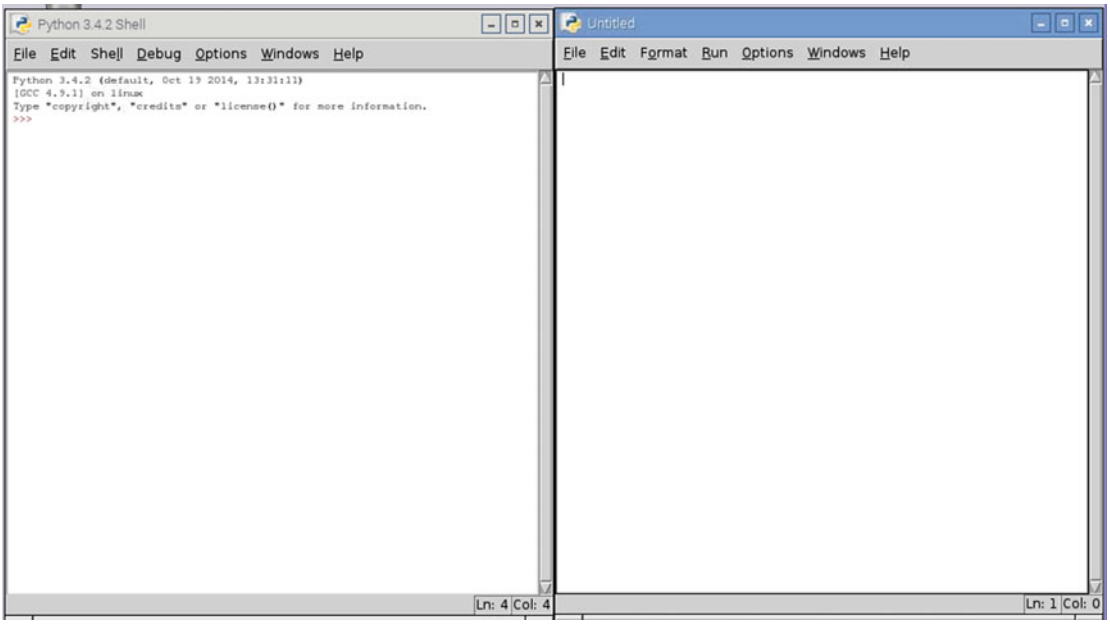


Figure 4-9. *Python shell and text editor*

The advantage of having both of these open is that code can be entered into the text editor, saved, and then tested by using Run Module from the Run menu. Any output or error messages will be displayed in the Python shell window.

You write commands in the text editor using a similar language to what you have already seen in Scratch. There are a few syntax rules that need to be followed.

The first is that indentation and white spaces count. This is one of the most noticeable things in Python as very few other languages are as strict in enforcing rules about spaces. In most other languages, such as C and Java, any spaces or tabs at the beginning of a line are ignored. In Python they are used to determine whether you are inside an `if` statement, loop, or function. Python 3 is pretty fussy about whether they are tabs or spaces as well. By default, IDLE replaces each tab with four spaces, which is a good way to deal with the spaces. Python will ignore any completely blank lines (that have only whitespace characters).

Another thing about Python (and most other programming languages) is that it is case-sensitive. This means that if you call a variable `MyVariable`, but then refer to it later as `myvariable`, you will not be referring to the same variable and will likely cause the application to give an error message or even crash.

Any line that ends with backslash will continue to the next line. So a command like

```
sum = value1 + \
value2 + \
value3
```

will add all three entries together even though they are split across multiple lines. It is very rare that I use this feature in the code I write, but it is useful in a book where there is a limited page width to fit the code in.

Finally, for now, the # sign is used to mark something as a comment, which is ignored when the program runs. This is very useful when you need to maintain your code in the future, because although it may be fresh in your mind when writing, code without comments can be a challenge to understand if you need to make changes to your code in the future.

I introduce more elements of Python programming in Chapter 6.

Getting Started with Python GPIO Zero

Controlling the GPIO ports involves some fairly complex code that is specific to the Raspberry Pi. Fortunately this has already been written for you and made easy to use thanks to the GPIO Zero Python module. The module actually uses another module called `RPi.GPIO`, which could be used by itself instead, but GPIO Zero makes it much easier to use. The list of rules used to communicate with the GPIO Zero code is known as an Application Program Interface or API. I won't go into the full details, but a summary of the API is discussed next.

The module needs to be loaded from within the Python programming using `import gpiozero`. To import a specific set of instructions, use `from gpiozero import <name>`. The *<name>* should be replaced with the name of the feature you want to import (such as LED) or with * to mean import everything in that module. If the `import` uses the first format (without the `from` instruction), all references to the module need to be prefixed with `gpiozero`. For example, to refer to an LED, you need to use `gpiozero.LED`. If the code uses the latter `from` format, there is no need to include the reference to `gpiozero`.

There is a list of supported devices at http://pythonhosted.org/gpiozero/.

Here is a simple program using GPIO Zero that will flash an LED on and off. Note that although the name is LED, this will just turn a GPIO pin on and off. The LED function can therefore also be used to control some other circuit that needs a simple on and off output and so will work with the transistor switch circuit. You could instead replace LED with `OutputDevice` or `GenericOutputDevice`, but the code looks simpler if you stick with LED:

```
from gpiozero import LED
import time

LED_PIN = 22
led = LED(LED_PIN)

print ("on")
led.on()
time.sleep(1)
print ("off")
led.off()
time.sleep(1)
print ("on")
led.on()
time.sleep(1)
print ("off")
led.off()
time.sleep(1)
```

In IDLE, create a new file and enter the previous code. Save it as a file called `flashled.py` and then choose Run Module from the Run menu. This is shown in Figure 4-10.

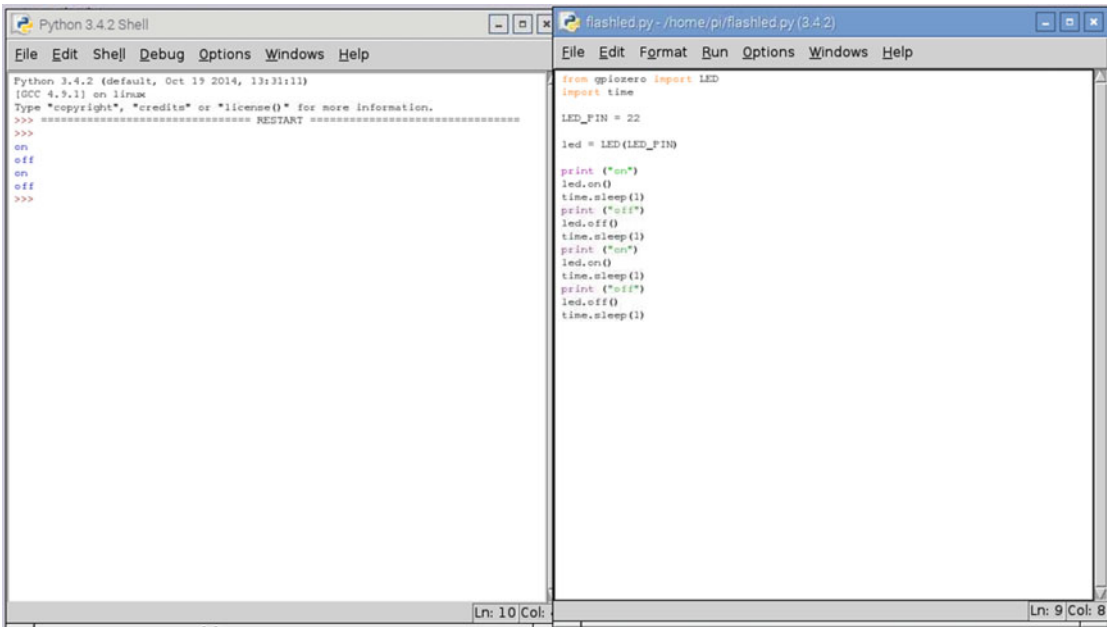


Figure 4-10. *Running the GPIO Zero LED code*

This code is also available for downloading. The `flashled.py` file is in the `gpiozero` folder.

In Figure 4-10, the code is shown in the editor on the right. The output from running the program is shown on the left. Obviously the main output is in the form of the LED flashing on and off, but I have also included some `print` statements so that it shows the expected status on the screen. The LED should flash twice, coming on for one second and then off for one second each time around the cycle. If the LED does not flash as expected, check that the wiring matches Figure 4-7 and, in particular, check that the transistor and LED are correctly connected.

The code starts by importing two modules. The first is `gpiozero` and the second is the `time` module, which is used to add a delay between the instructions. When `gpiozero` is imported, it uses the `from` format, which means that whenever you want to refer to the LED, you can just call it LED. The `time` import just uses the `import time` instruction and so whenever you want to use anything from within the `time` module, it needs to be prefixed with `time` to tell Python to look in the `time` module. For example, you can use `time.sleep`, which means use the `sleep` function inside the `time` module.

The next entry creates a variable called `LED_PIN` and stores the value 22. This is to store the GPIO port you are using. It's usually a good idea to list the LED pin assignments at the beginning of a program so that if at a later date you change which port the circuit is connected to then you only need to change it in one place. You can then refer to `LED_PIN` throughout the code, which will be replaced by the value 22 when the program runs. You will note that this is all in capital letters, which is a convention used for constant values that won't change during the program. It will still work if you don't use capital letters, but this is a useful convention that others will understand and will also act a reminder to not change the value, as it is a constant.

You then create an LED object using LED from the `gpiozero` module. I have called it `led` (in lowercase), which is how it will be referred to when you want to turn its output on and off. This will set the port, entered within the brackets, as an output port and allow you to change the state of the output.

Next is the code that changes the status of the LED. First a `print` statement sends a message to the shell where the program is running. This is not necessary but can be helpful when testing because it tells you whether the LED should be on or off.

The lines `led.on()` and `led.off()` turn the output pin on and off as appropriate, and `time.sleep(1)` tells the computer to sleep (do nothing with this particular program) for one second.

Using a While Loop

The previous code relied on individual entries to turn the LED on and off each time. If you wanted the LED to keep flashing for a reasonable length of time, this would need a lot of repeated code. So instead you can change it to a `while` loop that will keep running. You will use a `while True` loop, which will run the loop forever. This is similar to the `forever` loop in Scratch. Here is the updated code:

```
from gpiozero import LED
import time

LED_PIN = 22

led = LED(LED_PIN)

while True:
    print ("on")
    led.on()
    time.sleep(1)
    print ("off")
    led.off()
    time.sleep(1)
```

The text within the `while` loop is indented. This tells Python that those instructions are within the `while` loop. If they were not indented, they would not be within the loop and so would not run since Python will never exit the loop. To stop the program from running, you need to be in the command shell and press Ctrl+C. This code is also in the source code for the book; it's called `flashled2.py`.

Circuit Diagram and Schematics

For most of the circuits so far, I have shown the breadboard layout. This is a good way to see how to wire up the circuit, but can make it quite difficult to see how the circuit works. For example, in Figure 4-7 you had a transistor on the breadboard but without knowing which pin of the transistor is the base, collector, and emitter, it's difficult to visualize how the transistor works within the circuit. It is also very difficult to show larger circuits using a breadboard layout as the wires crossing each other could end up looking like a bowl of spaghetti.

Instead, you need a diagram that shows how the components fit together without showing the actual physical positioning; as mentioned earlier, this is called a circuit diagram or a schematic. In a circuit diagram, the components in the circuit don't look like the physical components, but instead are replaced with circuit symbols. The circuit diagram shows the components as symbols connected with straight lines. These lines are generally replaced with wires or copper tracks on a PCB.

The best way to understand circuit diagrams is to look at some. The first, in Figure 4-11, shows the first LED circuit from Chapter 1 both as a circuit diagram and as the breadboard layout you used previously.

Figure 4-11. *Simple LED circuit as circuit diagram and breadboard layout*

As you can see, these look very different, although if you follow the lines around the circuits, you will see that the same components are used and that they are connected in the same order on both circuits.

Unfortunately, different people use different symbols when creating their diagrams, although sometimes there are only minor variations. There are a number of different standards that provide recommended symbols. The two most common standards are an international standard IEC 60617, commonly used in Europe, and the ANSI standard Y32.2, commonly used in the United States. One of the biggest differences between these symbols is the resistor. Figure 4-12 shows two different symbols for resistors—the one on the left is the U.S. ANSI symbol and the one on the right is from the international standard IEC 60617.

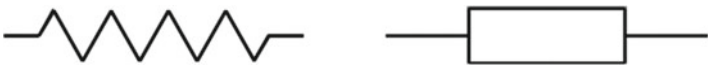


Figure 4-12. *The resistor circuit symbol using the U.S. and international standards*

A selection of different symbols for the LED are shown in Figure 4-13. Unlike the resistors, which are very different, the differences between these are much more subtle.

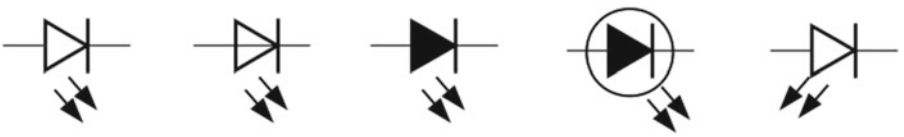


Figure 4-13. *Different symbols for the LED*

The variations shown in the LEDs are fairly common, with a few other components which may or may not be filled, or may have a circle around the symbol.

For this book, most of the symbols are the ones used by the Fritzing application (explored more in Chapter 8). The components in Fritzing are based on the U.S. circuit symbols, so we will be using the U.S. resistor symbol. I have made a few deviations for some components. In particular I have opted to use a specific symbol for a pushbutton switch, which differs from the generic switch symbol used by Fritzing. Some of the most common symbols that are used in this book are shown in Figure 4-14.

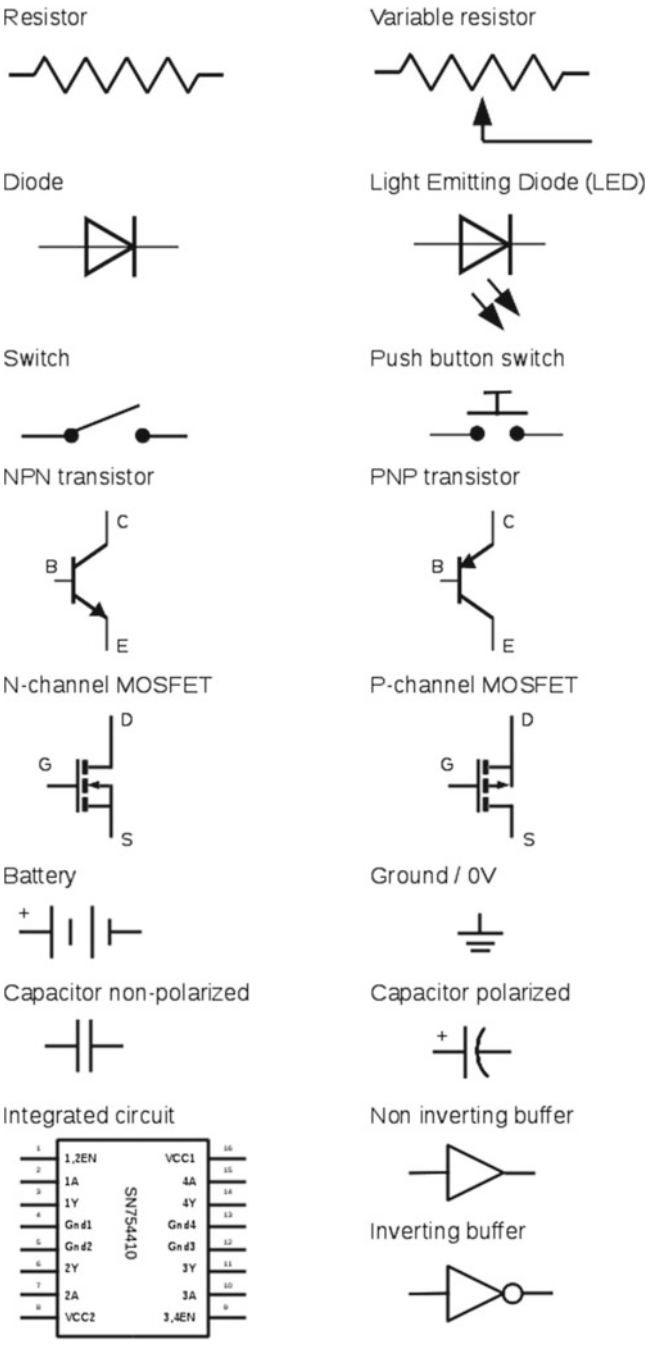


Figure 4-14. *Common circuit symbols*

Each of the symbols is briefly explained in the following bullets, with more details of the actual component provided in Appendix B.

- You have already seen the resistor throughout the book. The variable resistor adds a third connector, which can be moved along the length of the resistor, thus providing a different resistance value.
- A diode only allows current to flow in one direction. The symbol shows the direction of the current going from the positive end of the component (anode) on the left to the negative end (cathode), which is on the right. The LED (light emitting diode) is a specific version of the diode that gives out light when the current flows.
- The switch symbol can be used for any type of switch, although sometimes different symbols are used for different types of switches. The pushbutton switch symbol is sometimes used for switches that are closed only when the button is being pressed and then open when the button is released.
- You have already come across the *NPN transistor* in Figure 4-6. There is another variation, called the PNP transistor, which relies on the current flowing out of the base rather than into the base. The base (B), collector (C), and emitter (E) are shown on the symbol, but are not normally shown in a circuit diagram.
- The *MOSFET* is explained later in this chapter. The drain (D), gate (G), and source (S) are shown on the symbol, although these are not normally shown on a circuit diagram.
- A *battery* symbol is shown. Many diagrams will just show this as a power supply by indicating the voltage rather than specifically showing it as a battery. There is a longer line for the positive terminal on the battery. The + symbol for the positive terminal is sometimes included and sometimes not. The positive terminal will normally be at the top of the diagram.
- The *ground* symbol is used to indicate the ground 0V connection of a power supply. There may be multiple ground connections on a diagram. All the ground connections are considered to be connected, which can reduce the number of lines on the diagram.
- A *capacitor* is used to store electrical charge. They are available as non-polarized capacitors, which can be connected either way around, or polarized capacitors, which need to be connected the correct way around. It is important that polarized capacitors are connected the correct way around as otherwise they can explode.
- Two different ways of showing an *integrated circuit* are shown. The first shows the integrated circuit as a rectangle with each of the pins as connection into the symbol, the model of the integrated circuit is shown in the center. Usually the pins are shown in numerical order, although sometimes they are rearranged to fit in with the circuit. The pin numbers are often shown especially when they are not in numerical order. The pins function is shown inside the symbol using abbreviations. A common abbreviation shown in the diagram is EN, which indicates an enable pin that turns on the appropriate part of the circuit. For other abbreviations, you may need to look in the specification to find the meaning of the pins.
- The other way for an integrated circuit to be shown is by its function. The *non-inverting buffer* and *inverting buffer* are shown by their function. In this case, the power supply connections to the integrated circuit are not shown, but will still need to be physically connected. The circle at the end of the buffer indicates that it's inverting, which means that it will give a low output for a high input and a high output for a low input.

Other circuits and boards, including the Raspberry Pi, are shown in a similar way to the integrated circuits.

One more thing to be aware of is about how lines cross each other. In the past, it was common to have a small bridge symbol indicating where a wire crossed another rather than joining. This was not so easy for early CAD systems and so the convention changed and now two lines crossing are most commonly shown as two lines crossing at 90 degrees to each other. Where wires are connected, they are shown with a filled circle over the join, often referred to as a *dot*. The different types of crossing and connecting wires are shown in Figure 4-15. I have marked the first as my preferred style and the other as an alternative.

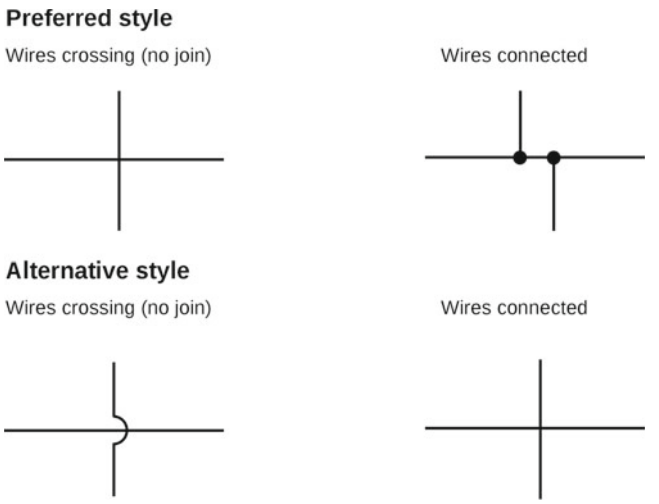


Figure 4-15. *Circuit diagram representation of crossing and connecting wires*

Note that the two different styles use the same straight crossed line to mean the opposite. It is the presence of either the dots or the bridge that indicates which style is being used.

The circuit diagram for the robot soccer game is shown in Figure 4-16. The Raspberry Pi is shown in the center of the diagram with the components around it. The switches have been placed on the same side as the GPIO ports that they connect to. The LEDs are both on the right even though the GPIO port that one of them is connected to is on the left. This makes it easier to see that the inputs are on the left and the outputs are on the right. The lines crossing at the top do not have a dot at the join, indicating that they cross and do not join. The positive power supply is shown at the top and the ground connection at the bottom.

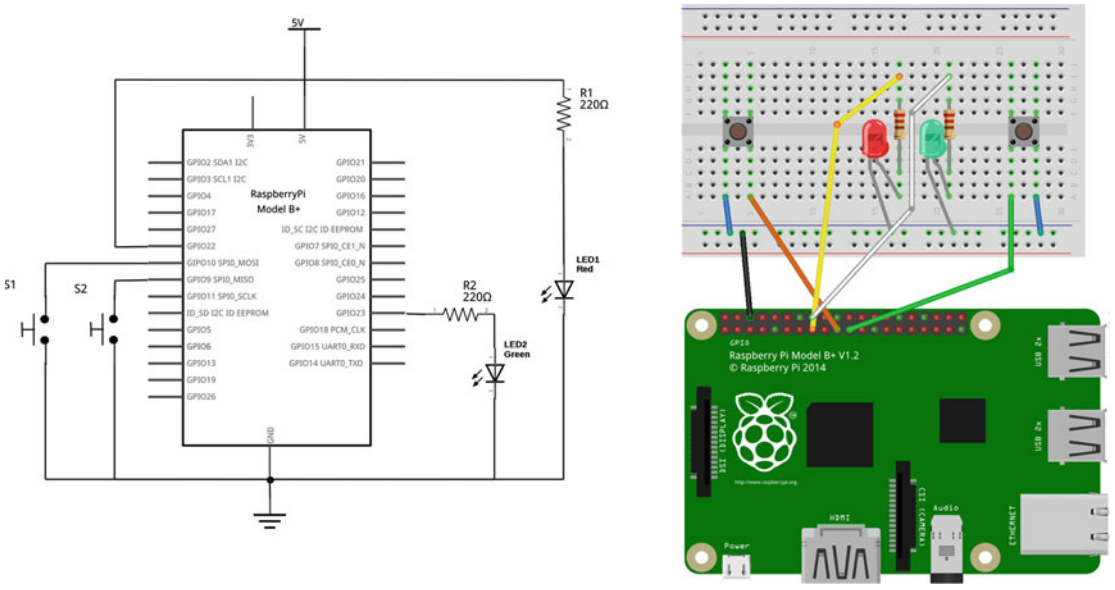


Figure 4-16. *Circuit diagram for the robot soccer game*

Brighter LEDs with Darlington Transistors

The transistor switch is useful to switch loads that need more current than the Raspberry Pi GPIO ports can provide, but it still has its limitations. For the next circuit, I have used a USB powered light, which provides much more light than the single LED. Opening up the light, I can see that there are actually 10 LEDs and their associated resistors, as shown in Figure 4-17.

Figure 4-17. *LED light with and without the top removed*

The USB light was bought from a discount retail shop rather than a specialist electronics supplier and as such, no technical information was provided. Connecting to a 5V USB power supply the light draws around 500mA, although when it does the supply voltage also drops to around 4V.

The BC548 is not able to switch to 500mA and so could not be used, but what about the 2N2222? The transistor can supply the 500mA current you need, but you need to take into consideration the amount of base current needed to switch the transistor into saturation.

In the previous LED example, you needed around approximately 3mA to switch the 20mA required for the bright LED, so to switch 500mA, you will need approximately 25 times that value, which is 75mA. That's much more than the 16mA that the GPIO port can provide.

If a transistor can increase the current then perhaps you can just add a second transistor to increase the current a second time. The combination of two transistors to increase the gain is known as a Darlington pair. The collector for each transistor is connected and the emitter from the first stage is used to provide the input to the second transistor's base. This is shown in Figure 4-18.

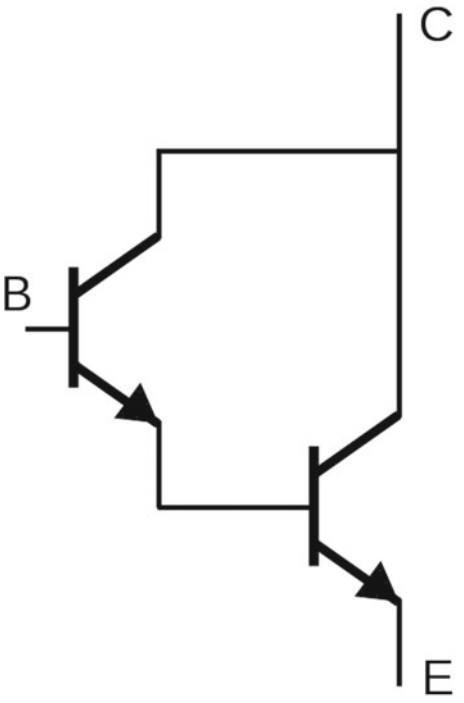


Figure 4-18. *Darlington pair*

You could make this up using two 2N2222 transistors, but a Darlington pair is also available in a single package usually known as a Darlington transistor. The particular model used in this circuit is the BD681. The specification of the BD681 gives the following values:

- V_{BE} is 1.5V
- V_{CE} (sat) is 1.5V
- h_{FE} (gain) is 750
- I_C (maximum collector current) is 4A

These values can be used in the same way as a single transistor. Note that the voltage required at the base and the voltage dropped across the collector and emitter are both noticeably higher than a single transistor, but so is the gain. The maximum collector current is also much higher. Darlington transistors typically do have a high maximum collector current, although you can also get power transistors that can switch this amount of current as well.

The circuit diagram for the Darlington transistor light is shown in Figure 4-19.

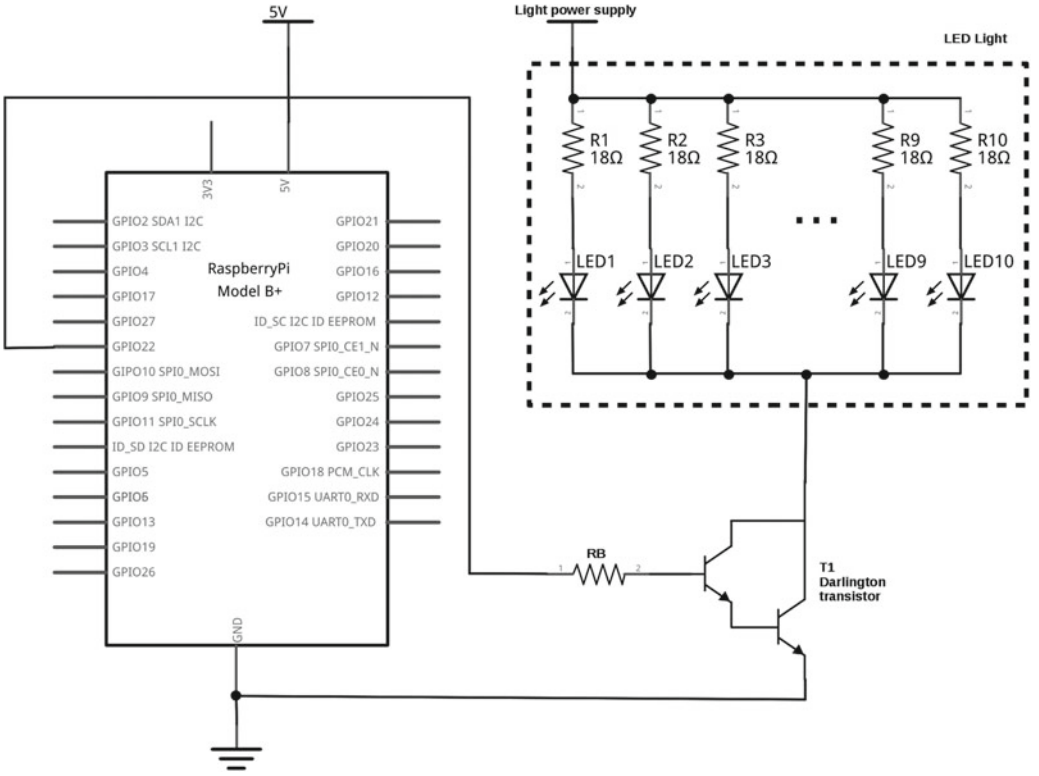


Figure 4-19. *Darlington light circuit*

For this circuit, I have shown the LEDs and resistors that are inside the light, using a dashed box. This is done to show which parts are within the USB light rather than in the circuit I created. Due to space, I have only shown five LEDs but there are actually 10 within the light.

There is a potential problem with this circuit. The resistors in the LED light will have been calculated assuming a 5V power supply connected to the light. But assuming you stick with a 5V supply, with a 1.5V drop across the Darlington transistor, you will be running the light at 3.5V for the total light. Let's perform some quick calculations to check this.

The resistors in the light are labeled 180, which indicates that these are 18Ω resistors (18×10^0). Assuming a 50mA current through each LED and associated resistor, this works out as 0.9V (18×0.05) dropped across the resistor. There is also some resistance in the USB lead of a little over 3Ω. Although 3Ω does not sound like it will make much difference, the lead carries the full 500mA total current, which works out at about 1.5V dropped within the lead. After rounding the figures, you can work out that the forward voltage of the LEDs is around 3A, which I verified using a multimeter. (Multimeters are explained in detail in Chapter 10.)

Based on the 1.5V voltage dropped across the Darlington transistor, will you still have enough to the light the LEDs? If not, what can you do about it?

It turns out that in practice this does still work, and there is only a small drop in the brightness. When switched using the Darlington transistor, due to the voltage dropped across the transistor, the current flowing through the LEDs is less than when it is connected direct to the power supply. This means that the voltage dropped within the lead and resistors will be lower and there is still sufficient current to light the LEDs. It appears that you can use the Darlington transistor. There is a risk that, due to the tolerance of the components used, it may not work for every LED light.

■ **Caution** When designing circuits for your own pleasure, you can often "get away" with working outside of the specified range. This is something you should be much more cautious of if you're designing a circuit for commercial purposes.

So how do you ensure that it will work?

You could remove the resistors altogether and replace them with a direct link or a zero ohm resistor, which would help bring you closer to the original current, as you would just have the resistance within the lead. This is generally a bad idea unless you are going to permanently attach the lights to the circuit. If you leave the USB connector on the end, there is a risk that someone could plug it into a computer or other power supply and without the resistors it could attempt to draw too much current, thus damaging either the USB light or the power supply.

The safe solution is to increase the power supply to accommodate for the voltage dropped in the Darlington transistor. If you replaced the 5V power supply with a 6.5V power supply (if one was available), this would allow the full 5V for the USB light. A more common power supply is 7.5V, but then you'd need to add a further resistor to reduce that by the extra 1V, which would again be a waste of energy.

As this works in practice using a 5V I've just worked all the calculations based on 5V. To calculate the appropriate resistors, you can use the same calculations as for the standard transistor switch circuit, but substitute the equivalent values for the Darlington transistor. As you already have the resistance for the LEDs, it is just RB that you need to calculate.

To work out the size for RB, you first need to work out the current you want to flow into the base of the transistor. The current you need at the base is the collector current divided by the gain (h_{FE}). This gives 500mA/750 = 0.7mA.

It's a good idea to use a factor of 10 to ensure that you are well above the required base current. This ensures that you are in the saturation region where the power lost in the transistor will be at its minimum.

Multiplying by a factor of 10 gives 7mA, which is the current you need at the base of the transistor. The voltage dropped across the base to emitter is 1.5V. This gives 3.3V – 1.5V = 1.8V dropped across the resistor.

Again using Ohm's Law to work out the resistance from the voltage and the current, you have:

$$R = V/I$$
$$1.8/0.007 = 257\Omega$$

A suitable resistor value (using the E6 series) is therefore 220Ω.

The BD681 Darlington Transistor comes in a different package than the transistors used previously. The pins are also in a different order. Figure 4-20 shows the pin layout for the BD681, shown with the label to the front.

Figure 4-20. *BD681 Darlington Transistor pinout*

This circuit uses the same GPIO port as used in the previous examples, so it will work with the existing code. Remember that GPIO port 22 is physically pin 15 and that the ground needs to be connected to an appropriate pin, such as pin 6.

Such a bright light is more useful as a time-delay light. With an appropriate delay, it can be used as a time delay nightlight or a really bright egg timer. The light will turn on when a switch is pressed and then off again after a set delay; for that you need an input, which you will add next.

Reading a Switch Input with Python GPIO Zero

I've added a pushbutton to the circuit using the same switch configuration used in Chapter 3. This gives the updated circuit diagram shown in Figure 4-21.

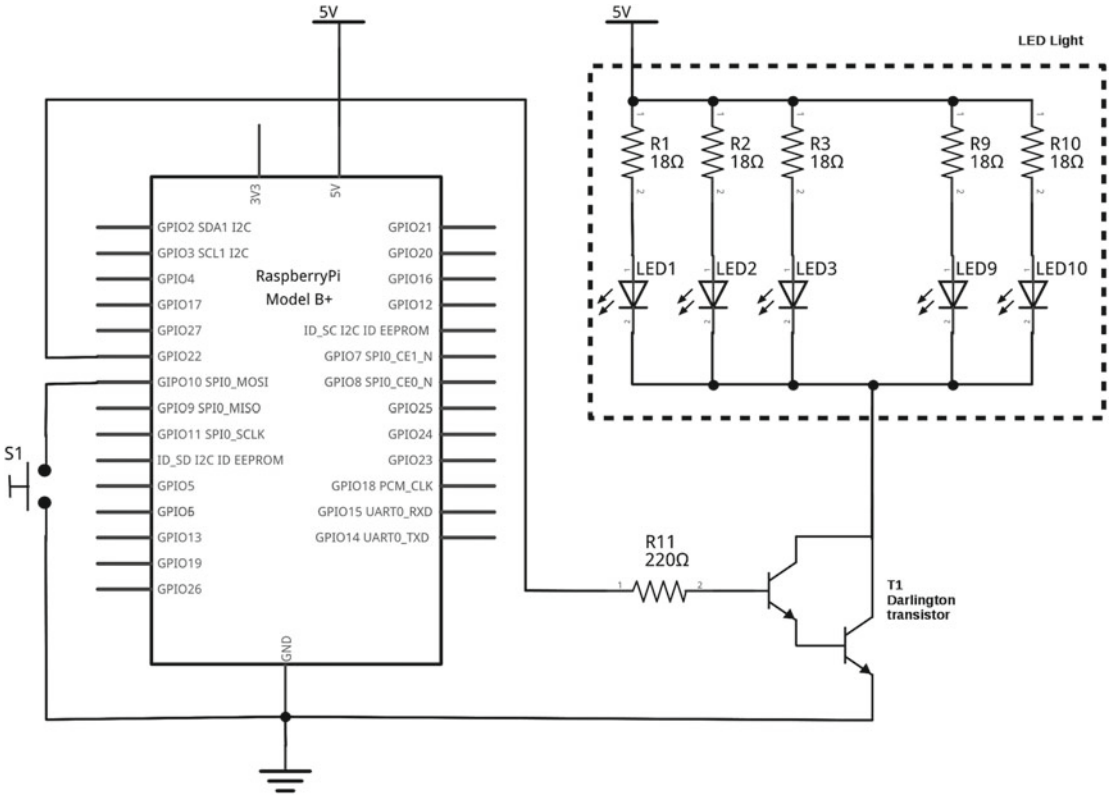


Figure 4-21. *Darlington light circuit with switch*

Detecting and handling the button is easy in the same way that Python GPIO Zero makes it easy to turn an LED on and off. The first thing you need to do is import `Button` from the `gpiozero` library. This can be done in the same was as done previously, with an additional line: `from gpiozero import Button`

To save on typing and make the code a little shorter, you can combine these into a single line, as:

```
from gpiozero import LED, Button
```

You then create a button as `button = Button(GPIO_Number)`, which is very similar to what you did with the LED. By default, GPIO Zero assumes that a button should have the pull-up resistors enabled, which is exactly what you want. If you wanted to use pull-down instead, you would add `pull_up=False` as a second parameter to the `Button` function.

There are a few different ways to then detect whether the button is pressed or not. The first checks for the `is_pressed` state of the button. If the button is pressed down, this gives a `True` value, if not then it gives a value of `False`. The way that you will use it here though is a `wait_for_press()` method, which pauses the execution of the code until the button has been pressed. You then turn on the LED and use `time.sleep()` to wait for a set delay period. The LED is turned off again and the `while` loop starts again. I've also added some comments (prefixed with a `#`) to explain what the constants are used for.

The complete code is as follows:

```
from gpiozero import LED, Button
import time

# GPIO port numbers for the LED and Button
LED_PIN = 22
BUTTON_PIN = 10
# Time to keep the light on in seconds
DELAY = 30

led = LED(LED_PIN)
button = Button(BUTTON_PIN)

while True:
    button.wait_for_press()
    led.on()
    time.sleep(DELAY)
    led.off()
```

This code is also available in the source code to accompany the book. The file is called `ledtimer` and it is stored in the `gpiozero` folder.

When it runs, this will wait until the button is pressed, turn the LED on for the set period of time (30 seconds), and then turn it back off again. If the button is held down or pressed again, it will turn the LED back on again for a further period. Press Ctrl+C to cancel and stop the program from running.

One of the disadvantages of the Darlington transistor is that the voltage drop between the collector and emitter is fairly high. This results in quite a bit of wasted energy in the transistor. In the next example, you will see an alternative that can be a more efficient way of switching bright lights.

Disco Lights with MOSFETs

Now to get even brighter! The LED light you used in the last project was around 2.5W (watts), but in this one you are going to look at 5W LEDs. Not only that, you are going to have four of them with 20W of power being controlled from the Raspberry Pi.

First, I explain *power*, which is a measure of the amount of energy being used. Chapter 3 looked at voltage, which indicates the amount of potential energy that available, and current, which is the flow of electrical charge around the circuit. If you combine these, then you know how much energy is being used, which is measured as a watt and denoted by the letter W. The formula for working out the amount of energy used in a DC circuit is

$$P = V \times I$$

where P is the amount of power in watts, V is the voltage in volts, and I is the current in amps.

The LED used previously was designed for a 5V supply and used 500mA (0.5A), so this gives 5 x 0.5 = 2.5W. This is the total power used by the light, which includes the power that is wasted in the resistors and the cable.

The lights I am using here are PAR 16 theatre spotlights, which I use as disco lights for a mobile DJ setup. The lights are shown in Figure 4-22, and they include different colored gels in front of the lights so that each light is a different color.

Figure 4-22. *PAR 16 spotlights used as multi-colored disco lights*

The PAR 16 spotlights are available as two different types. Some are designed for main electrical voltage using GU10 light bulbs, and others are designed for low 12V operation using MR16 light bulbs. The following circuit is intended for the 12V light bulbs only.

■ **Caution** The circuit used here is only suitable for low voltage (12V) bulbs. Do not attempt to use this circuit to control bulbs that are designed to be connected to the main electricity.

Besides choosing the correct spotlights, you will also need to ensure that you have appropriate bulbs. The bulbs need to be MR16 LED bulbs that can be powered using 12V DC. These spotlights are normally intended to be connected to a 12V AC (alternating current) supply rather than DC (direct current), but most MR16 LED bulbs will work with either AC or DC. You may want to check with the technical information for the LEDs that they are suitable for use with DC before purchasing them. The photo in Figure 4-23 shows some of the bulbs I have tried, which clearly say 12V AC/DC on the packaging, indicating they can work with either. You can't power halogen bulbs using this circuit because they need a lot more power, typically between 25W and 50W.

Figure 4-23. *MR 16 LED bulbs which work with 12V AC/DC*

Another thing you are going to need is a fairly powerful 12V DC power supply. The one that I am using is a replacement for an LCD monitor. It is a 12V DC power brick that's able to provide up to 10A, which is 120W. This is more than enough power. The power brick is shown in Figure 4-24.

Figure 4-24. *Power brick for the disco lights*

The one risk with using such a powerful power supply is what happens if there is a problem with the circuit? Most good-quality power supplies should have some kind of overcurrent or short-circuit protection, but it does mean that you are able to provide 120W of power into a circuit before that kicks in. I therefore recommend adding your own fuse or protection into the circuit. This should be rated above the maximum current that you expect to draw, but less than the total power output of the power supply. I use a polyfuse in my own setup, which is a self-resetting fuse. Effectively the fuse will break the circuit when the current is exceeded, but when left to cool will then reset and work again. The fuse is installed between the positive connection on the power supply and the rest of the circuit.

You now have the lights, the bulbs, and a power supply, so the next thing you need is an electronic circuit. The first thing you need to look at is the amount of power you are going to control. The LED packaging states 5W, but looking at the technical specification (which varies between manufacturers) then the current draw is actually about 625mA, which is 7.5W. This looks like the rating on the packaging is based on the amount of energy that is used to create the light and the rest is lost as heat energy. This is much less energy wasted than with an incandescent or halogen bulb, but you need to use the actual power used.

You could use the same Darlington transistor circuit used previously. At 625mA this is only a little more than the previous example where you had a current of 500mA, but as I mentioned, the Darlington transistor does have a relatively high voltage drop. Instead, you are going to see how to use a MOSFET transistor.

Although earlier electronic circuits were based around bipolar transistors, MOSFETs are now more commonly used. One of the main advantages of the MOSFET over the bipolar transistor is that it can be controlled with much smaller currents, which is more efficient, particularly for use within integrated circuits. In this case, the voltage drop is also lower and generally more efficient than the Darlington transistor. For this particular MOSFET and 5W LEDs, you are looking at about 0.2V dropped across the MOSFET, which for a 12V supply is negligible.

The MOSFET switch circuit is shown in Figure 4-25. This uses an IRL520 MOSFET transistor.

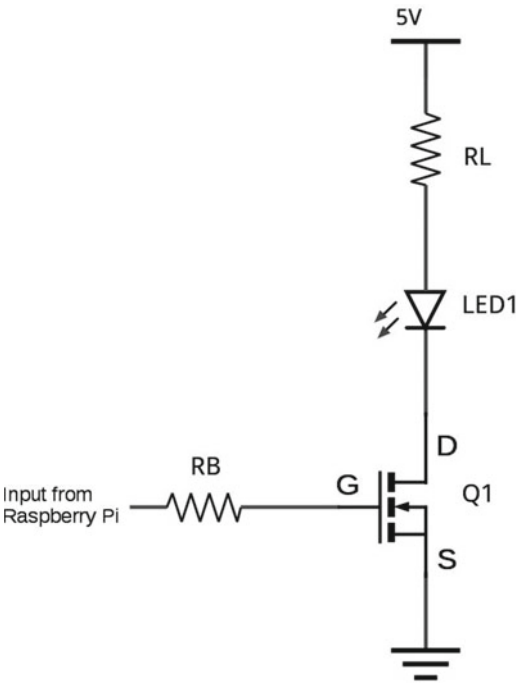


Figure 4-25. *MOSFET switch circuit*

You will see from the circuit diagram in Figure 4-25 that the terminals of the MOSFET are labeled G, D, and S. This refers to the *gate* (like the base), *drain* (like the collector), and *source* (like the emitter). Whereas the bipolar transistor was turned on by a current flowing into the base, the MOSFET instead needs a positive voltage at the gate to turn it on and a low voltage at the gate to turn it off. The resistance between the gate and source is very high and so only a tiny current will flow. You do not need to worry about any gain within the MOSFET, as setting the output high is enough to turn the MOSFET on.

In theory, the high internal resistance within the MOSFET would mean that you didn't need the resistor RG at all, but in reality the MOSFET behaves differently when it is first switched on, allowing a larger initial current to flow. You should provide a suitable resistor to reduce this rush of current.

This resistor just needs to be enough to protect the Raspberry Pi GPIO port, so assuming 3.3V output and 16mA maximum current, you have the resistance as 3.3/0.016 = 206Ω. So I have used a 220Ω resistor.

As the LED unit is designed to be connected directly to a 12V supply, you don't add your own resistor for RL as that is already included within the LED. So there are only two components needed for each LED: the switch and the resistor that goes between the GPIO port and the gate of the MOSFET.

This completes a single LED, so you just need to choose the GPIO ports to use as outputs. I used the following, which was mainly because I had an LED board that already used those ports so I could use that board to test the code without needing to connect all my disco lights:

- GPIO 4 (pin 7)
- GPIO 17 (pin 11)
- GPIO 23 (pin 16)
- GPIO 24 (pin 18)
- Ground (pin 9)

The completed circuit is shown in Figure 4-26.

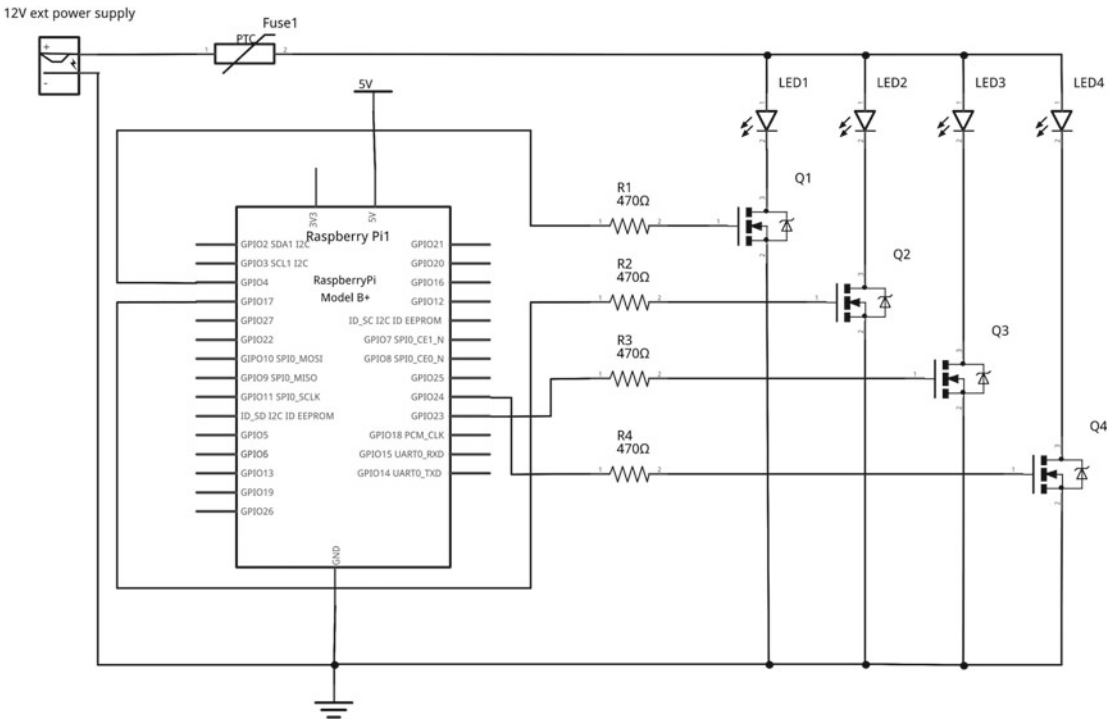


Figure 4-26. *Disco light circuit*

There are a few things to note about this diagram. First, the device shown as Fuse1 is a self-resetting polyfuse used to protect against someone connecting an inappropriate bulb such as a halogen bulb. For simplicity I have only shown the LEDs as a single LED and ignored their internal resistance. This is designed for use with 12V LEDs only and connecting a standard LED in this configuration would most likely result in damage to the LED and possibly other components.

You may also note that the circuit symbol of the MOSFET is slightly different from the one used previously. This circuit symbol is the one used on some of the MOSFETs in Fritzing, which shows that there is an internal reverse biased diode within the MOSFET. This is still the same component. This is another example of how circuit symbols can vary slightly.

Light Sequence Using a Python List (Array)

Let's now turn our attention to the code to make the lights flash in sequence. In this code, you will create a light sequence that goes from left to right. To achieve this, you'll track the GPIO ports and the update the lights using arrays, or in this case a slightly simplified version known as a list. Python does also have support for a specific module for an array, which can provide more efficient code for certain data, but a list is more versatile.

A list is a data structure that can hold the value of multiple variables. So instead of having individual variables called `light1`, `light2`, `light3`, and `light4`, you can have a single list called lights which has four entries, one for each of the lights.

Before you start, though, one thing to consider is that computers start counting from 0 and therefore the first element in the list will be at address 0. Scratch does things a little differently as it is aimed at a younger audience, so if you have used a list in Scratch then you may have started at 1, but in Python and just about all other programming languages, you start at 0.

You'll now look at how to create the first list, which holds the GPIO port numbers for each of the LEDs:

```
LIGHTGPIO = [4, 17, 23, 24]
```

The square brackets are used to denote this as a list (although not strictly required by Python). This creates a list called `LIGHTGPIO` (in uppercase to signify that these won't change later in the code). Inside this list are four values, one for each of the lights. These can be referred to in the code later by the address position. So to access the GPIO number for the first entry which has value for, use the following:

```
LIGHTGPIO[0]
```

To access the second entry (counting from 0), you use `LIGHTGPIO[1]`. Python can store much more than just numbers in a list. In fact, you can use a list to store all the instances of the GPIO Zero LEDs. So this give you:

```
lights = [LED(LIGHTGPIO[0]), LED(LIGHTGPIO[1]), LED(LIGHTGPIO[2]), LED(LIGHTGPIO[3])]
```

That code creates a list of instances of the LED object using the appropriate entry from the `LIGHTGPIO` list for the port number.

So now you can create some code that will flash each light in turn as a light chaser. Copy the following code and save it in a file called `disco-chaser.py`. The file is also included in the source code for the book:

```
from gpiozero import LED
import time

# GPIO port numbers for the light
#9 = gnd, 7 = GPIO 4, 11 = GPIO 17, 16 = GPIO 23, 18 = GPIO 24
LIGHTGPIO = [4, 17, 23, 24]
```

```
# Time between each step in the sequence in seconds
DELAY = 1

lights = [LED(LIGHTGPIO[0]), LED(LIGHTGPIO[1]), LED(LIGHTGPIO[2]), LED(LIGHTGPIO[3])]

# Track our position in the sequence
seq_number = 0

while True :
    if (seq_number > 3):
        seq_number = 0
    for x in range (4):
        lights[x].off()
    lights[seq_number].on()
    seq_number = seq_number + 1
    time.sleep(DELAY)
```

Based on the previous description, you should be able to follow up to the `while` loop. The `while` loop is a `True` loop that will run forever (as long as the program is running). The first step within the loop is to check to see if you've reached the end of the sequence. There are four lights, so 3 is the last entry in the list. If it is, it resets the sequence number to 0.

You then have a `for` loop, which runs the next line four times. This works by using `range(4)`, which expands to four entries 0, 1, 2, and 3. So it runs the loop the first time with x set to 0, and then with x set to 1, until it reaches the end of the fourth loop (number 3), when it exits from the `for` loop.

The line `lights[x].off()` turns the light off for each light. It then turns just the one light on using `lights[x].on()`. So if you were on sequence number 1, this would turn on the second light.

Finally, you increment the sequence number and wait for the set delay (1 second) before turning on the next light.

If you run the code, you should now see each light turn on in turn.

Switching AC Lights Using a Thyristor and a TRIAC

So far all the lights you have been switching have been LEDs using a DC power supply. The bulbs used in the disco lights would normally be powered using an AC power supply—I used a DC power supply as it's easier to get a DC power brick suitable for a portable disco light controller.

The main electricity coming into your home is AC. One of the biggest advantages of AC is that it's fairly easy to step the voltage up and down using a transformer. An example of a transformer designed for running 12V AC halogen lights is shown in Figure 4-27.

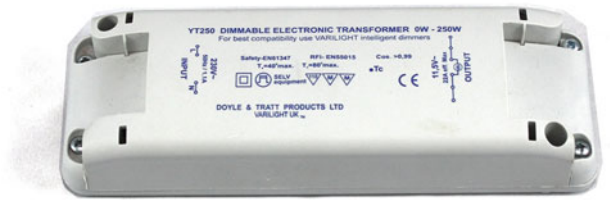


Figure 4-27. *AC main to 12V AC transformer*

The first device you will look at for switching AC power supplies is the thyristor. This device acts as a switch that allows a current to pass in one direction only when it receives a positive signal to its gate terminal. Once triggered, it will normally continue to conduct as long as there is a forward current passing through it, even if the gate is subsequently taken low again. This has limited use in a DC circuit, as it's difficult to turn off, but in an AC circuit, the current will change direction each cycle and so will turn off whenever the current changes direction. By controlling when thyristors switch on, you can use them as dimmers in lighting circuits.

The disadvantage of a regular thyristor is that it works in only one direction, so instead you will look at the TRIAC, which is a bidirectional thyristor. Effectively this is like two thyristors connected back-to-back, but they can be turned on with either a positive or negative voltage to the gate and (assuming AC) turned off with 0V at the gate. The circuit symbol for a thyristor and a TRIAC is shown in Figure 4-28. The symbol on the left is a thyristor and it has an anode and cathode appropriate to the direction of current flow. The one on the right is a TRIAC. It shows the terminals as A1 and A2 as it can be connected either way around.

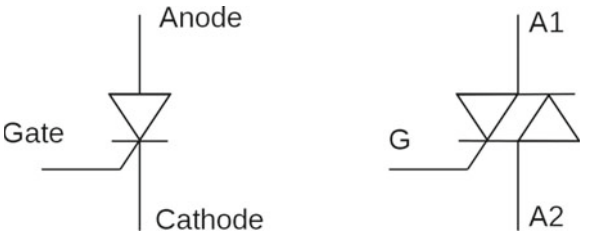


Figure 4-28. *Thyristor and TRIAC circuit symbols*

You could connect the TRIAC directly to the GPIO of the Raspberry Pi (with a suitable resistor), but this would involve connecting the DC circuit to a common connection on the AC circuit. Instead you will use an opto-isolator, which provides a way of controlling one circuit from another without any form of electrical connection between the two. This is achieved by using an infrared or similar light that is controlled at one side of the device, which switches a current at the other side. This is best understood by looking at the circuit symbol in Figure 4-29, which shows an opto-isolator with a phototriac output. The opto-isolator is controlled by providing a DC voltage to the (infrared) LED on the left, which switches on the photo TRIAC on the right.

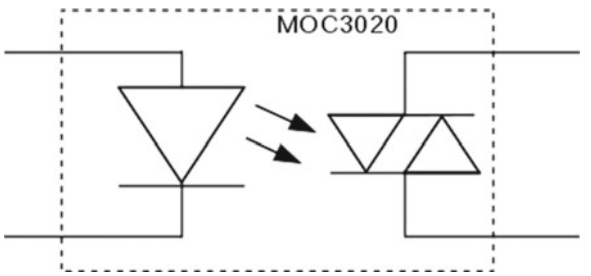


Figure 4-29. *Circuit symbol for a TRIAC opto-isolator*

The complete circuit with the TRIAC and opto-isolator is shown in Figure 4-30.

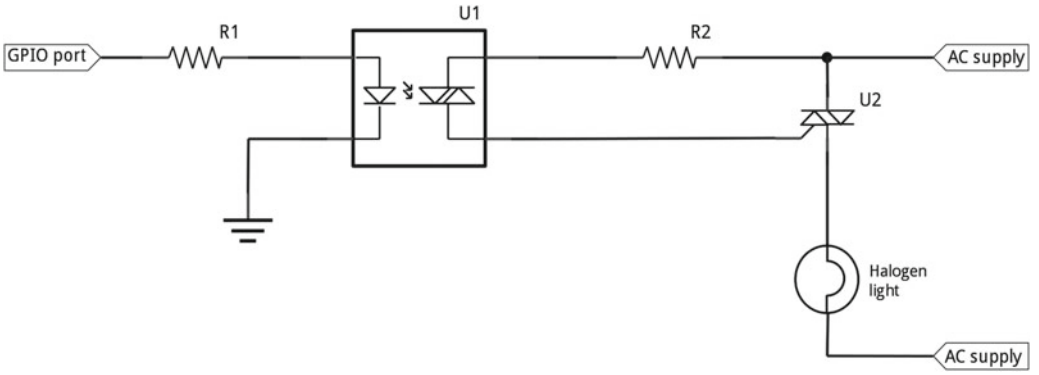


Figure 4-30. *Circuit diagram of an AC TRIAC circuit*

This circuit is intended for use with low-voltage 12V circuits. With appropriate components, this circuit could be used to switch the main electricity, but heed the warning earlier—this can be very dangerous and you should only consider it if you really know what you are doing. In particular the circuit needs to be fully enclosed and the TRIAC itself can have live main electricity connected to its outer body.

More GPIO Zero

This chapter looked at different external power supplies and how you can increase the small signal from the GPIO output to switch a much bigger load. This also provided a first introduction to the Python GPIO Zero module and how lists can be used to track multiple variables. From an electronics perspective, you learned about the transistor, a two-stage transistor known as a Darlington transistor, and a MOSFET, which you used to switch increasingly large loads. You also briefly looked at a TRIAC, which is used to control AC rather than DC circuits.

This chapter introduced the concept of circuit diagrams, which you will be using instead of the breadboard layouts from the previous projects. You should now be able to decide on an appropriate position on the breadboard and wire up the circuit based on the diagram.

If you've followed along with the practical circuits, the first LED should have been easy enough to buy and the second one can use any suitable USB-powered light. The final project does use a more specialist light and power supply, but this can be replaced with the LED from one of the previous two projects (and resistor RL in the case of the first project). Alternatively, you can look at substituting the LEDs for other outputs such as a buzzer. You should avoid motors for the moment, as anything using electrical motors needs some extra protection, which you will learn about in Chapter 8.

You will leave the disco lights for a while, but revisit them in Chapter 6, when you look a little more at the software, and in Chapter 11, when I discuss the disco light printed circuit board that I created.

In the meantime, I suggest that you look at creating code to create different light sequences. Can you make two of the lights come on during each cycle instead of just one? Think of some other sequences and write code that makes them light up the appropriate lights.

CHAPTER 5

More Input and Output: Infrared Sensors and LCD Displays

The last chapter looked at some inputs and outputs using Python GPIO Zero. This chapter will look at working with some more sensors and outputs. Some of these can be used with the GPIO Zero module and others use other Python libraries. These include detecting people entering a room, sending and receiving infrared signals, and displaying output on an LCD display.

PIR Sensor and Pi Camera

In this project, you will learn how to use a PIR (passive infrared) sensor to detect someone entering a room and take a photo of people as they enter. This could be useful for protecting a private possession or for catching a photo of a pet or garden visitor.

The source code for the PIR sensor and the camera is included in the source code for the book, in the `pircamera` directory.

Controlling a Raspberry Pi Camera with picamera

You will start by looking at connecting a Raspberry Pi camera and controlling it using the Python `picamera` module. The Raspberry Pi camera is an official accessory that hooks up to a special connector on the Raspberry Pi. It can be used on any model of the Raspberry Pi except for the early Pi Zero (which doesn't include the camera connector).

The camera is mounted on a small printed circuit board and connects through a ribbon cable. The connector provides direct access between the camera and the processor, which is more efficient than using a webcam, which needs to connect through the USB protocol. Figure 5-1 shows the camera connected to a Raspberry Pi.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_5

Figure 5-1. *Raspberry Pi camera*

There are two types of cameras: the standard Raspberry Pi camera and the Pi NOIR camera. The standard camera is the one used for most purposes. The Pi NOIR camera is the same but without the infrared filter that is useful for night photography using an infrared light source. Using the Pi NOIR camera in normal daylight will result in unusual colors in the photos.

To connect the camera, lift the tab on the connector next to the HDMI connector. Insert the ribbon cable into the connector with the blue side facing away from the HDMI connector. Push the tab back down, which should hold the camera cable firmly into place.

The camera module needs to be enabled through the Raspberry Pi configuration tool. From the menu, choose Preferences and then Raspberry Pi Configuration. The camera is enabled through the Interfaces tab, as shown in Figure 5-2. After enabling the camera, you will need to reboot before you can use it.

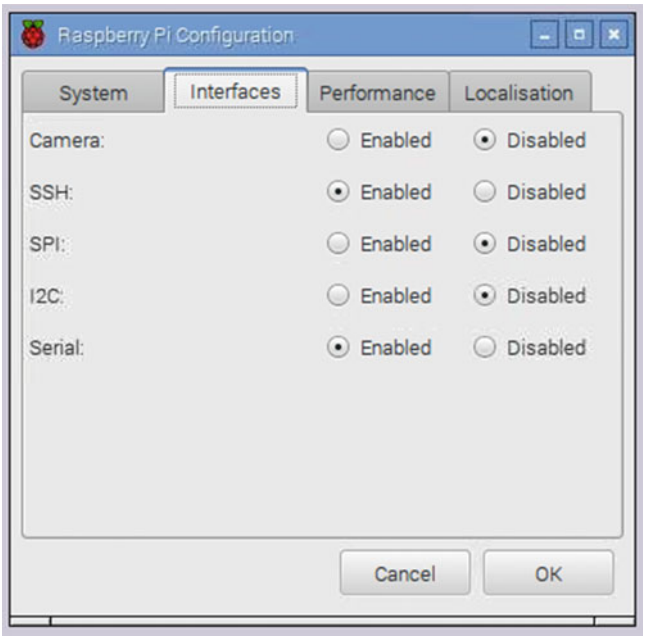


Figure 5-2. *Enable the Raspberry Pi camera*

If you are running the Raspberry Pi headless, this can also be achieved from the command line:

```
sudo raspi-config
```

You can test whether the camera is working correctly using the following code:

```
raspistill -o photo1.jpg
```

This will take a photo and store it as `photo1.jpg`. If you have a screen directly connected to the HDMI port, you will see a preview prior to the photo being taken; otherwise, you will just notice a delay before the photo.

Assuming this command works, you can now proceed to controlling the camera through Python. You'll start with a simple program that takes a photo and saves it to the local computer.

```
import picamera

camera = picamera.PiCamera()

camera.capture('/home/pi/photo1.jpg')
camera.close()
```

This is a simple and straightforward program. The first line imports the `picamera` module using the standard `import` command. This means when you create the object from the `picamera` module, you need to prefix the instruction with `picamera`.

You then create an instance of the `picamera` object called `camera`. You can then call the camera functions. The first is called `capture`. As you can probably guess, this captures a photo and saves it in a file called `photo1.jpg`. Finally, you call `close` to clean up the resources.

You have used the same filename as the previous command, so this will overwrite the file created previously. You need a way of giving each file a new name. There are two simple ways you can do this. The first is to add a unique number that you increment for each new file, which would involve keeping track of the number. The alternative is to add the date and time to each file, which is what this example shows. You will use the `time` module and the `strftime` method, which formats the time into a readable date format.

The date format you'll use is the ISO 8601 date format, which formats the date in order of the most significant part of the date first. This provides the date in the form of *year-month-day*T*hour*:*minutes*:*seconds*. The advantages of this date format are that ordering the files by filename will put them into chronological order and that it's a date format that is recognized anywhere around the world. An issue with using the ISO date format is that the colon character is not allowed on some other operating systems. I used a hyphen to replace the colon to make transferring to another computer easier.

The updated code is shown here:

```
import picamera
import time

camera = picamera.PiCamera()
timestring = time.strftime("%Y-%m-%dT%H-%M-%S", time.gmtime())

camera.capture('/home/pi/photo_'+timestring+'.jpg')

camera.close()
```

`time.gmtime()` gives the current time, which is in seconds since the UNIX epoch (1970). This is converted to a string that's stored in `timestring`. It is then included in the filename.

There is one potential problem with this solution. The Raspberry Pi does not include a real-time clock. If the Raspberry Pi has a network connection (wired or wireless), it will update the time of the Raspberry Pi using network time servers. If the Raspberry Pi does not have a network connection (such as an outdoor sensor to monitor wildlife), then the date and time may not be correct. The files can be renamed when they are transferred to another computer for later viewing. You can test this now by saving the file as `cameratest.py` and running the following:

```
python cameratest.py
```

It will create a single photo each time it is run.

Using a PIR Sensor as a Motion Sensor

To detect someone in the room, you will use a Passive Infrared (PIR) sensor, usually known as a PIR motion sensor. The PIR sensor detects infrared radiation given off in the form of body heat. The sensor detects body heat moving across its view, and that can be used to detect the presence of a person or animal. There are a few different sensors that can be used with the Raspberry Pi. The sensor used here is the HC-SR501. On the sensor board is a small circuit that detects the body heat and triggers a high signal. A photo of the sensor is shown in Figure 5-3.

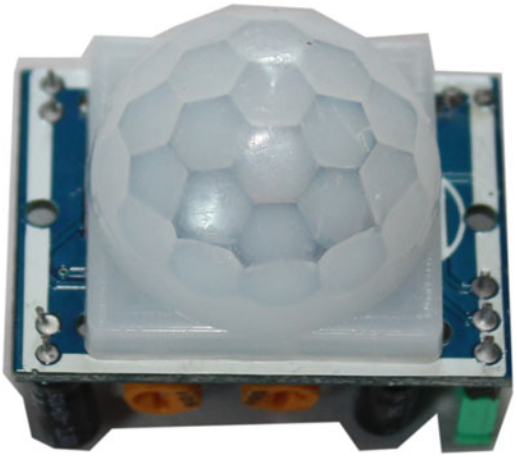


Figure 5-3. *PIR motion sensor*

The motion sensor is powered by 5V, but gives a 3.3V output signal, which is just what you need for connecting to the GPIO port. The sensor needs to be connected to the ground, the 5V power supply, and then the output to any GPIO port. This example uses GPIO port 4 (physical pin 7).

The circuit is very simple for this and does not even need to connect to a breadboard. Assuming that the PIR sensor includes male pins, this can be connected to the Raspberry Pi using female-to-female jumper leads, as shown in Figure 5-4.

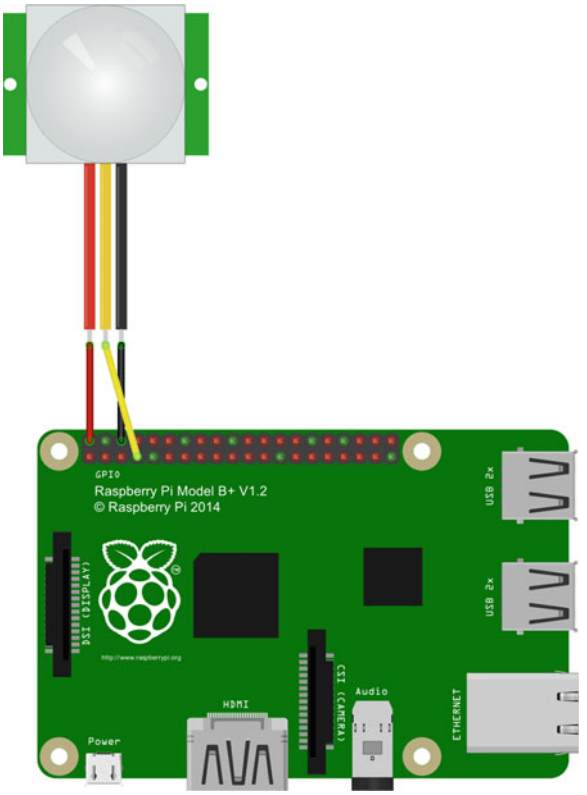


Figure 5-4. *PIR motion sensor connected directly to the Raspberry Pi*

The wire connections to the PIR sensor are based on the terminals being positioned at the top of the sensor, with the sensitivity and delay adjustment variable resistors located at the bottom.

This particular PIR sensor is supported directly by the GPIO Zero module as a `MotionSensor`. The GPIO Zero module also includes two functions specifically for this sensor and they can cause the program to wait until the sensor is triggered, or alternatively cause the program to wait when the sensor is triggered and then continue running after no motion has been detected for a set period of time.

Again, you can make this easier by using the `from import` function:

```
from gpiozero import MotionSensor
```

The `MotionSensor` object can then be created by using `MotionSensor` and the GPIO port as the parameter:

```
pir = MotionSensor(4)
```

The methods are then `wait_for_motion` or `wait_for_no_motion`. An alternative is for your own code to check for the value of `motion_detected`. The method you want is `wait_for_motion`, which will cause the program to wait until the sensor is triggered. By default, this will wait indefinitely if there is no motion, but you can instead add a timeout, which means that the program will continue to execute after that timeout period has been exceeded. I have not included a timeout, but that could be useful if you wanted to periodically check that the Raspberry Pi is still running even if there is no motion detected. I have also included a delay to prevent the sensor from constantly triggering if there is someone present in the room.

Here's a simple test script:

```
pir.wait_for_motion()
```

A simple program that can be used to test the motion sensor is provided here:

```
from gpiozero import MotionSensor
import time

# PIR sensor on GPIO pin 4
PIR_SENSOR_PIN = 4
# Minimum time between captures
DELAY = 5

pir = MotionSensor(PIR_SENSOR_PIN)

while True:
    pir.wait_for_motion()
    print ("Motion detected")
    time.sleep(DELAY)
```

Enter this code and run it to test that the PIR sensor is working. You should try running the code and leaving the room and then re-enter the room to ensure it is working correctly. You may need to adjust the sensitivity variable resistor if the sensor is not triggering properly.

Using the PIR Motion Sensor to Trigger the Camera

You now have the motion sensor and camera working, so it's just a case of combining both parts of the program. The code will wait for motion to be detected and then capture a photo of whoever or whatever triggered the sensor. The files will have the date and time included in the filename.

Here's the complete source code for this:

```
from gpiozero import MotionSensor
import picamera
import time

# PIR sensor on GPIO pin 4
PIR_SENSOR_PIN = 4
# Minimum time between captures
DELAY = 5

# Create pir and camera objects
pir = MotionSensor(PIR_SENSOR_PIN)
camera = picamera.PiCamera()

while True:
    pir.wait_for_motion()
    timestring = time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime())
    print ("Taking photo " +timestring)
    camera.capture('/home/pi/photo_'+timestring+'.jpg')
    time.sleep(DELAY)
```

This code is primarily a merge of the PIR and camera programs listed previously. The main change is for the camera code to be included in the while loop. The `camera.close` entry has also been removed, as you want to continue capturing photos. Ideally this should still be closed when the program is terminated, but as you are relying on using Ctrl+C then the `close` has been removed.

The `print` statement now shows the time that the photo is taken. This is useful during testing, but can be removed once the program is proved to be working correctly.

The code is saved as `pir-camera.py`, which can then be run from the command line as:

```
pir-camera.py
```

One thing that can be useful, particularly if you're installing in a remote location, is to have the program start automatically when the Raspberry Pi is booted. This is explained in Chapter 6, when you look at controlling programs within Linux.

Infrared Transmitter and Receiver

For the next pair of sensors, you will look at an infrared transmitter and receiver pair. This will use a sensor that can send and receive infrared signals that are used by home entertainment remote controls. In this chapter, you will use it to receive signals from a remote control, which can be used as a way to send messages to the Raspberry Pi. You will use the receiver to control a light using a remote control and provide the ability to send remote control signals to an existing infrared device. You will then use this same circuit in Chapter 6 to control a Lego train and in Chapter 7 as a camera trigger for creating stop frame animation.

Infrared Receiver

The infrared receiver used here is the TSOP2438. This is an infrared receiver with built-in pre-amplifier. The sensor has three pins connected to the supply, ground, and a sensor output, which can be connected to a GPIO port on the Raspberry Pi. The sensor can take a supply voltage of between 2.5 and 5.5V. As you need the output from the sensor to work with the 3.3V GPIO ports you need to connect it to the Raspberry Pi 3.3V supply. It is recommended that a 100Ω resistor be used between the 3.3V on the Raspberry Pi and the supply for the sensor and that a 0.1 µF capacitor be used across the supply voltage, although it should still work without these. Details of common capacitor labeling is provided in Appendix C.

This is the first time you have come across the *capacitor*, which is a device that can hold an electronic charge. A *capacitor* is similar to a very small rechargeable battery, but stores the energy as electrical charge rather than converting it to chemical energy. The reason for including the capacitor in this circuit is to smooth the input power supply, because even small spikes can affect the operation of the sensor. This is quite common in electronic circuits.

There are similar receivers that could be used instead. The same series of sensors has a number of different versions, which are similar and different suppliers may only stock a subset of the range. The TSOP2238 is an older version that can be used a direct replacement and the TSOP2436 and TSOP2440 are designed for a slightly different infrared frequency, but should still work for most remote controls. There are, however, some other sensors such as TSOP4438 or TSOP4838 which, while very similar in operation, do have a different pinout so you will need to adjust the circuit accordingly if using one of those. The pinout for the TSOP22xxxx and TSOP24xxxx infrared modules is shown in Figure 5-5.

Figure 5-5. *Pinout for the TSOP2438 and similar infrared receivers*

The output from the sensor is connected to GPIO 18 (physical pin number 12).

Infrared Transmitter

The infrared transmitter you will use is essentially the same as an LED, but designed to give out infrared rather than visible light. Infrared is part of the light spectrum similar to visible light but has a slightly longer wavelength so is outside of the range of human vision. The particular emitter you will use is the TSAL6400 infrared emitting diode, which has a wavelength of 940nm. This particular one is designed for high-power requirements and in practice can give a range of over 5 meters (15 feet) when it has a clear light of sight from the top of the emitter. The circuit to drive the infrared emitting diode is the same one used for the brighter LED in Chapter 4; see Figure 4-6.

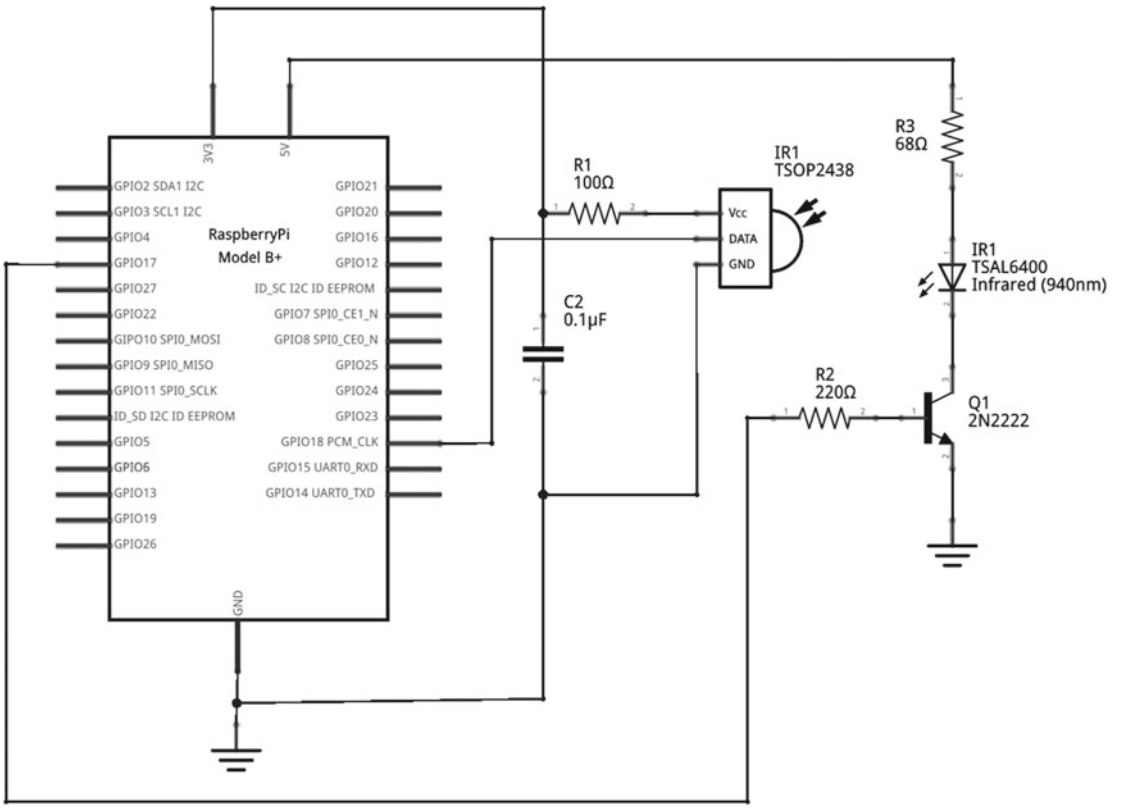


Figure 5-6. *Circuit diagram for the infrared transmitter and receiver circuit*

This will be powered from the 5V supply connection on the GPIO connector.

The infrared emitter diode can operate with up to 100mA of current, but for this circuit I have based the current flow around 60mA, which provides a good compromise between output strength and power requirements. I used a 68Ω resistor for RL and 220Ω for RB.

As with an LED, the infrared emitter diode needs to be connected to the correct pins. The longer lead is the anode and it needs to be connected to the positive side.

The signal for the transmitter circuit is connected to GPIO 17 (physical pin number 11).

Infrared Transmitter and Receiver Circuit

The complete circuit of the infrared transmitter and receiver is shown in Figure 5-6. As mentioned previously, you will see that the sensor is powered from the 3.3V power connection from the Raspberry Pi, but the emitter is powered from the 5V power supply. They both use the same ground connection, although these are shown differently as a way of reducing the number of crossing lines.

In this case, I have shown a single circuit for the transmitter and receiver. This can be very useful as a way of testing the received signal and if required generating the configuration files for the transmitter. If you only need to receive signals to control a Raspberry Pi project then you don't need the transmitter. Likewise, if you only need to send signals to control other devices then you don't need the receiver part.

Configuring the Infrared Transmitter and Receiver Using LIRC

You will use LIRC as a way to communicate with the remote control transmitter and receiver. LIRC stands for Linux Infrared Remote Control and provides a way to send messages through an infrared transmitter and to receive signals from an infrared receiver, which can then be sent to the appropriate application. There are quite a few steps involved in setting up an LIRC, but don't worry, I go through them in this section. You also learn how to generate your own configuration file if you have a remote control for which you don't know or can't find the appropriate codes.

You will need a suitable infrared remote control for this section. You can use most infrared remote controls, but avoid running the project in the same room as the device that the infrared is normally intended for. In my case, I used a Crystallite color-changing LED bulb designed to replace a standard light bulb. This is shown in Figure 5-7.

Figure 5-7. *Infrared color-changing LED bulb and remote control*

The first thing to do is to install the `lirc` package containing the software. To install this, you should first run an `update` and perhaps even an `upgrade` using the following:

```
sudo apt-get update
sudo apt-get upgrade
```

The `apt-get` tool is also used for installing `lirc`:

```
sudo apt-get install lirc
```

There is also another package called `lirc-x` that's used if you want to use an infrared remote control to control the mouse cursor on the desktop. You won't be using it for any of the projects in this book though.

The next stage is to enable the support in the kernel using the device tree overlay. This is a way in Raspbian for enabling only specific hardware drivers to avoid potential conflicts between the hardware drivers. You will need to edit the file `/boot/config.txt` as root.

You can use your preferred editor, which could be leafpad (`gksudo leafpad /boot/config.txt`), nano (`sudo nano /boot/config.txt`), or any other text editor. Note that when using a graphical application, `gksudo` is used to run the application as root. When you're using a command line application, it should be just `sudo`.

Find the entry

```
#dtoverlay=lirc-rpi
```

and remove the #, which is the first character on that line. The hash character is commonly used to indicate a comment and, when used to prevent a command or instruction from being read, it is referred to as being *commented out*. This will cause the Raspberry Pi LIRC driver to be enabled.

Then, as root, create a new file called `/etc/modprobe.d/lirc` and add the two lines:

```
lirc_dev
lirc_rpi gpio_in_pin=18 gpio_out_pin=17
```

The input and output pin numbers can be changed if different ports are being used. These are numbered using the GPIO pin numbering scheme.

Now update the hardware configuration file `/etc/lirc/hardware.conf`. The following options need to be updated to match the wording:

```
LIRCD_ARGS="--uinput"
DRIVER="default"
DEVICE="/dev/lirc0"
MODULES="lirc_rpi"
```

All other lines in the `hardware.conf` file should be left at their default values.

To load the driver, Raspberry Pi needs to be rebooted:

```
sudo reboot
```

The LIRC daemon (`lircd`) will fail during the reboot, which is fine, as you still need to add the details of the remote controls you want to use. The easiest way to add a remote control is to look in the `/usr/share/lirc/remotes` folder or download the appropriate LIRC configuration file from http://lirc-remotes.sourceforge.net. Many hundreds of remote controls are available. If you do find a suitable remote control entry, download it to the `/etc/lirc/lirc.conf.d/` folder with the name of the remote control as the filename. You may need to create that directory first:

```
sudo mkdir /etc/lirc/lirc.conf.d
```

Next add an entry in `/etc/lircd.conf` by importing the appropriate configuration file:

```
include "/etc/lirc/lirc.conf.d/remotename"
```

Replace `remotename` with the name of the remote control file downloaded. You may also need to remove the first line of the `lircd.conf` file if it says `UNCONFIGURED`.

If you cannot find a suitable code for the infrared remote control, then you can discover the codes using `irrecord`. The `irrecord` program provides a way to capture the codes by pressing the remote control buttons. It works with most remote controls although it does not work with all, especially remote controls with only a few buttons.

To use `irrecord`, `lircd` must not be running. If you have not yet added a remote control, it won't have started anyway, but to make sure run the following command:

```
sudo systemctl stop lirc
```

Now run this command:

```
irrecord -d /dev/lirc0 --disable-namespace ~/lightremote
```

The `disable-namespace` option is used to allow you to use any names for the remote control buttons. If you were adding a remote to control a media center then you will want to use the standard controls, but by adding this you can use any appropriate name. I have used the name `lightremote` for the file so that I know which remote control this refers to.

You should then follow the instructions, which involve pressing various remote control buttons so that `irrecord` can determine how the remote control codes are created. It will then ask for you to name each button followed by pressing that particular button.

When the command is complete, you will end up with a file called `lightremote`, which in my case contains the following:

```
begin remote

  name  /home/pi/lightremote
  bits           16
  flags SPACE_ENC|CONST_LENGTH
  eps            30
  aeps          100

  header       9116  4434
  one           637  1614
  zero          637   490
  ptrail        629
  repeat       9116  2193
  pre_data_bits   16
  pre_data       0xFF
  gap          108044
  toggle_bit_mask 0x0

      begin codes
          On                       0xE01F
          Off                      0x609F
          Brighter                 0xA05F
          Dimmer                   0x20DF
          Red                      0x906F
          Green                    0x10EF
          Blue                     0x50AF
          White                    0xD02F
          Yellow                   0x8877
          Aqua                     0x08F7
          Pink                     0x48B7
          Flash                    0xF00F
          Strobe                   0xE817
```

```
        Fade                    0xD827
        Smooth                  0xC837
    end codes

end remote
```

I haven't added codes for all the colors on the remote control, but there are enough for what you need. Feel free to continue adding the other buttons if you would like more color choices.

I made one change to the file manually, which is to replace the entry

```
name  /home/pi/lightremote
```

with

```
name  lightremote
```

The top part of this file has details of the type of remote control, such as the timing and sequence of the codes. The section `begin codes` has the individual codes for each button.

To add this to the LIRC configuration, first create a directory to store it in:

```
sudo mkdir /etc/lirc/lircd.conf.d
```

Copy the file into that directory:

```
sudo cp ~/lightremote /etc/lirc/lircd.conf.d/
```

Then edit the file called `/etc/lirc/lircd.conf`. First remove the line

```
#UNCONFIGURED
```

This is usually the first line of the file. Then add an `include` entry that references the remote control file:

```
include "/etc/lirc/lircd.conf.d/lightremote"
```

You should now be able to start `lircd`:

```
sudo systemctl start lirc
```

You can check that it started correctly:

```
sudo systemctl status lirc
```

You can now send remote control commands using the `irsend` command.

To list all the configured remotes, use:

```
irsend LIST "" ""
```

This should show the `lightremote` added earlier. If it has the path in `/home/pi`, then you should edit the remote control file as the root user and remove the path from the `name` field.

To see more details of the remote control `lightremote`, use the following:

```
irsend LIST "lightremote" ""
```

This lists the buttons that have been configured, as shown here:

```
irsend: 000000000000e01f On
irsend: 000000000000609f Off
irsend: 000000000000a05f Brighter
irsend: 00000000000020df Dimmer
irsend: 000000000000906f Red
irsend: 00000000000010ef Green
irsend: 00000000000050af Blue
irsend: 000000000000d02f White
irsend: 0000000000008877 Yellow
irsend: 00000000000008f7 Aqua
irsend: 00000000000048b7 Pink
irsend: 000000000000f00f Flash
irsend: 000000000000e817 Strobe
irsend: 000000000000d827 Fade
irsend: 000000000000c837 Smooth
```

To send a button press, use `irsend` with the `SEND_ONCE` option followed by the name of the remote control and the button name:

```
irsend SEND_ONCE lightremote On
```

Receiving Infrared Commands Using python-lirc

Now that LIRC is configured you can use the Python module `python-lirc` to get details of the buttons pressed and then perform the appropriate actions. First you need to edit one more file from LIRC so that the messages are sent to your program. Edit (or create) the file `/etc/lirc/lircrc` using a text editor. Add the following entries:

```
begin
    prog = testlirc
    button = On
    config = On
    repeat = 0
end

begin
    prog = testlirc
    button = Off
    config = Off
    repeat = 0
end
```

This shows the first two buttons—On and Off—but you should repeat this for any other buttons you also want to receive. This tells LIRC that when it receives a signal for one of these buttons then it sends the appropriate message to a program listening for `testlirc`. This could be replaced with the name of your own program as long as it matches the Python code you create.

You should now restart `lircd`:

```
sudo systemctl stop lirc
sudo systemctl start lirc
```

The `python3-lirc` library is not installed by default, so install it using

```
sudo apt-get install python3-lirc
```

Now create a Python program, which I have called `testlirc.py`:

```
import lirc

sockid = lirc.init("testlirc")

while True:
    code = lirc.nextcode()
    if (len(code)>0):
        print ("Button pressed is "+code[0])
    else:
        print ("Unknown button")
```

First the program imports the module `lirc` and then creates a socket connection to `lirc` using the same name as you used in the `prog` entry in `lircrc`. The loop then gets the code from `lirc.nextcode()`, which represents the button. First check to make sure that you have the details of the button within the code. For example, if the `config` option has not been entered for a particular button, you will get an empty entry. Assuming you do have a valid button code, then it is stored in `code[0]`, which is printed to the screen. If the On button is pressed then the output of the command will be:

```
Button pressed is On
```

This will continue until Ctrl+C is pressed.

The code can be updated to run a particular Python action when the button is pressed.

If you have an LED connected to GPIO 22, the following example will turn an LED on and off as the relevant button is pressed on the remote control.

```
import lirc
from gpiozero import LED

# GPIO port number for the LED
LED_PIN = 22

sockid = lirc.init("testlirc")
led = LED(LED_PIN)
```

```
while True:
    code = lirc.nextcode()
    if (len(code)>0):
        if (code[0] == "On"):
            led.on()
        if (code[0] == "Off"):
            led.off()
```

The LED can be replaced with any of the light circuits in Chapter 4, depending on how bright a light you want to control.

Sending an Infrared Code Using Python

Previously you used the `irsend` command to send an infrared signal from the command line. You will use the same command in the next example, but call it from within a Python program. This technique can be used for many other command-line programs where a suitable Python module doesn't exist.

The following program is named `sendir.py`, and it will send the `On` command, followed by a delay, and then the `Off` command to turn the device back off.

```
import os
import time

REMOTE = "lightremote"
DELAY = 20

def send_ir_cmd (remote, op):
    os.system("irsend SEND_ONCE "+remote+" "+op);

send_ir_cmd (REMOTE, "On")
time.sleep(DELAY)
send_ir_cmd (REMOTE, "Off")
```

This works by having a function that issues the `irsend` command using `os.system`. You can then just call this command with the name of the remote control and the name of the button for the code you want to send.

More Infrared Devices

You will look at infrared in other projects later in the book, including using it to control a Lego train in the next chapter. It's not too difficult to take the circuit apart and remake it, but if you do have a spare breadboard then this may be a circuit that you want to keep. You will also look at making a more permanent circuit in Chapter 10. An alternative is to buy a pre-made add-on board for the Raspberry Pi; one such device is the Energenie Pi-Mote IR control board (ENER314-IR) and it uses a similar circuit to the one you made previously.

Changing Voltage with a Level Shifter

Before you move on to the next project, you will look at how you can change the voltage of an input or output. Recall that the Raspberry Pi GPIO ports are designed for 3.3V inputs and outputs. In some cases, you can just select sensors and other components that are also designed for 3.3V, but many components are designed to operate at 5V. This includes certain protocols, such as I^2C, which is commonly used for sensors and output devices.

There are three different approaches you can take depending on whether you need to reduce an input voltage that you want to use as an input to the Raspberry Pi, if you need to increase the output of the Raspberry Pi so that it is sufficient for connecting to an appropriate output device, or if you need to be able to communicate in both directions.

Voltage Divider Circuit to Reduce Input Voltage

The first example you will look at is the most simple, but is also very useful. This is used if you have a sensor that had a 5V output that you then want to use as an input to the Raspberry Pi. What you need is a way of dropping the signal voltage from 5V to 3.3V. This can be used by using a voltage divider (also known as a potential divider) circuit consisting of two resistors, as shown in Figure 5-8.

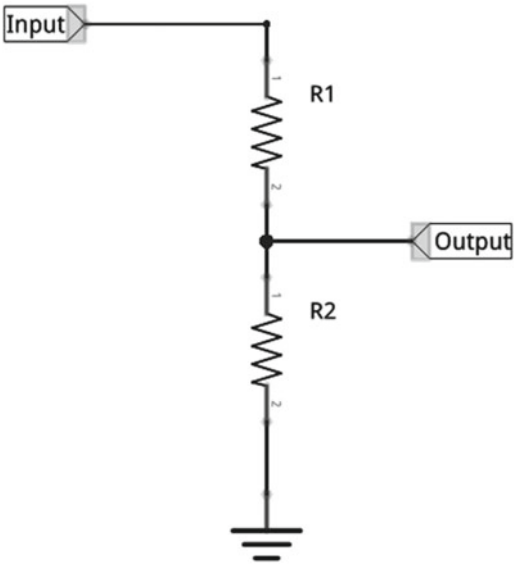


Figure 5-8. *Voltage divider circuit*

The input signal is connected across both the resistors. When a voltage is present from the sensor, a current flows through the two resistors. you take the voltage output from the top of R2, which is the just voltage dropped across R2. This is the same as saying the output voltage is equal to the input voltage less the voltage across R1. As the current is the same through both resistors then the voltage of the outputs is a ratio based on the value of both resistors. If the values of the resistor are equal then the output will be half the input, as 2.5V will be dropped across R1 and 2.5V dropped across R2. You need an output of 3.3V from a 5V input, so you need 3.3V across R2 and 1.7V across R1. This is approximately two thirds. So the value of R2 should be approximately twice as large as R1.

Now you know the ratio but what actual value of resistors do you need? You need to consider the input signal to the potential divider and what will be connected to the output of the voltage divider.

For the sensor, you need to ensure that the current you are going to put through the potential divider is not going to exceed the amount that the sensor can provide. For the output, you need to ensure that you have sufficient current to turn on the device connected and to ensure that the load does not significantly alter the output voltage.

In the previous example, you assumed that all the current flows through the two resistors, but the moment you add a device to the output then some current will flow through that instead of going through R2, which can change the output. If you are using an MOSFET (and in this case you can consider the input to the GPIO to be similar), that is triggered by voltage and very little current will flow into the load. If, however, you connected it directly to a load, or even to a transistor that was switching a large load, then the output current from the voltage divider could change the output. You could reduce this by choosing a low value in the resistors as long as that doesn't exceed the current that the sensor could provide, although that will waste more energy.

As this is going to the GPIO of the Raspberry Pi, which has a high input impedance, you don't need to worry about the amount of current. Therefore, you can just choose resistors that add up to a fairly high value, such as 100kΩ. This results in a current of around 0.05mA (or 50nA). If you have a device that uses more current, choose lower value resistors.

Using the standard sizes, the following are suitable resistors:

R1 = 39kΩ
R2 = 68kΩ

This should give around 3.2V, which is just below the maximum of 3.3V for the input to the GPIO port.

Unidirectional Level-Shift Buffer

Sometimes you may want to increase the 3.3V output from the Raspberry Pi so that it can drive another circuit that expects a 5V input. This is something you will do later using RGB LED strips.

The first thing to check is whether this is actually needed. While a circuit may be designed to operate at a certain voltage, there is usually a range in which it will operate. Also there is a range of voltages that the Raspberry Pi can provide as an output. In the case of the Raspberry Pi output, for a low (off) signal, it will be less than or equal 0.8V. If it's a high (on) signal then it will try to set the output to 3.3V. However, depending on the current of that and other GPIO ports, it may be as low as 1.3V. Obviously 1.3V is very low, but in reality unless there is a high load across the GPIO ports, it is likely to be near to the 3.3V output.

In the case of the WS2812 RGB LEDs, they need a voltage within 0.5V of their supply voltage. So assuming you connect them to a 5V supply for maximum brightness then even at a full 3.3V the output of the GPIO port is going to be well below the 4.5V required. Even then it's possible that the circuit will actually work, but the problem is that when you are operating outside of the designed range for the components, they become unpredictable. Having a circuit that is unreliable can be worse than having one that doesn't work at all; with a system that doesn't work you can track down the problem, but with one that works sometimes can be very difficult to troubleshoot.

There are integrated circuits designed specifically as buffers that are often used in these circumstances. In particular, the 74HCT125 will accept input voltages as low as 1.6V to 2V and provide a high output close to the supply voltage of 5V. Note that this is specifically for the TTL version of the chip denoted by the letter T; the 74HC125 is designed for a higher voltage input and may not work with the Raspberry Pi GPIO under certain conditions.

In a real circuit you may prefer to use a 74HCT125, but as this book is about learning electronics, I think it's useful to take a look at a simpler version that you can create using a MOSFET. The circuit shown in Figure 5-9 shows a simple MOSFET voltage buffer.

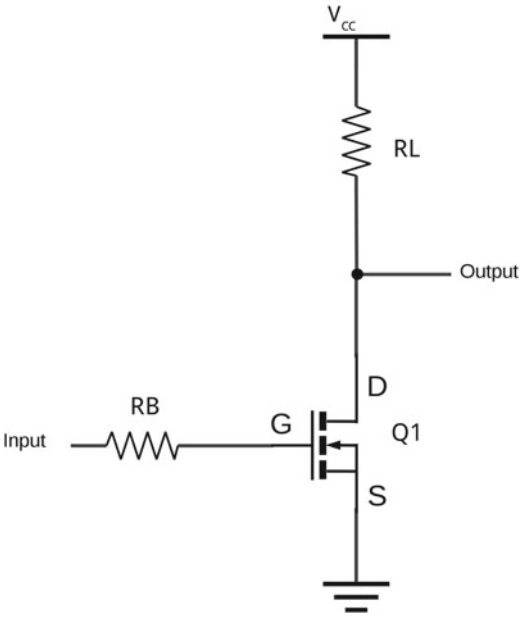


Figure 5-9. *MOSFET-based voltage shift buffer*

The circuit shown in Figure 5-9 is actually an inverting buffer that will produce a high output when the input from the GPIO port is low and vice versa. It's normally possible to change the software to send an inverted signal to counteract this, or a second stage could be added to invert the signal back to its original form. If you need non-inverting, it may be better to swap to a non-inverting IC rather than create your own.

The circuit is essentially the same as the MOSFET switch you used in Chapter 4, but replaces the load (LED) with a pull-up resistor. When the input to the MOSFET is low, the MOSFET is open circuit (switched-off) and the pull-up resistor RL will result in the output being HIGH. When the input to the MOSFET goes high, the MOSFET will switch-on, pulling the output down toward 0V (there will still be the voltage across the MOSFET V_{DS}, which depends on the MOSFET and current, but for this circuit will typically be less than 0.5V).

The value of RB was discussed in the earlier MOSFET switch circuit in Chapter 4 (220Ω), but recall the value of the RL depends on the load and what current that needs to operate. Typically the resistors will be between 1kΩ and 100kΩ, although I have used a lower value when creating an LED RGB circuit to allow for loses in a long cable between the circuit and the LED strip.

Bidirectional Level-Shifter

Sometimes you need a voltage shift in both directions, so that two devices can communicate in both directions across the same wire. In this case, you will look at how you can have an I^2C signal go in both directions from a low voltage (3.3V) Raspberry Pi to a higher voltage I^2C device. This is slightly different from the circuit used for some other communication protocols, as I^2C uses pull-up resistors, which you can take advantage of. The circuit you will look at is based on one used by Adafruit and SparkFun in their bidirectional level shifters. These are based around surface mount MOSFETs on a small breakout board. A photo of the board is shown in Figure 5-10.

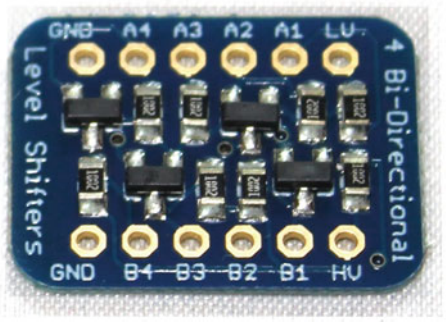


Figure 5-10. *Adafruit bidirectional level-shifter*

PULL-UP RESISTORS

Pull-up resistors are used to create a high signal where the input value is unknown which is the case if there is no signal from a sensor. For example in the case of a simple switch then we can detect when the switch is pressed as that gives a clear signal, but when it is not pressed then electrical noise can trick the circuit into thinking it is pressed when it is not. When the switch is not pressed then the pull-up resistor allows a small current to flow through it which gives a positive voltage (logic high). When the switch is pressed then the bottom of the pull-up resistor is connected through the switch to ground. This overrides the high signal by effectively connecting the input to ground.

This board has four channels, each of which has the same circuit. The circuits use the BSS138 MOSFET. As these use surface mount devices (SMDs), which are fiddly to use and the level-shifters are fairly inexpensive, I recommend just buying a level-shifter. I still provide details of how the level-shifter works. The circuit diagram of one of the channels is shown in Figure 5-11.

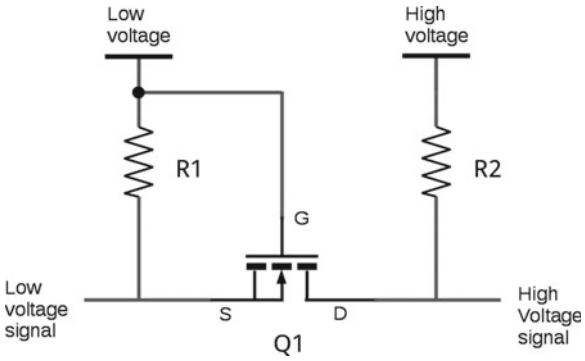


Figure 5-11. *Circuit diagram for one channel of the bidirectional level-shifter*

The circuit uses an MOSFET in an unusual configuration. Don't worry if you can't understand this at first—it should become clear when you have more experience with how electronic components behave.

The drain and source from the MOSFET connect between the input and output. If the input and output are both high, the MOSFET is switched off and the two pull-up resistors ensure that both sides are high using their own supply voltages (the left at 3V, the right at 5V).

If the left side goes low, the MOSFET will be in the forward configuration and thanks to the input to the gate the MOSFET will turn on. This pulls the right hand sound down toward the ground as well.

If the right side goes low, due to an internal characteristic of the MOSFET, it allows a small current to flow in the reverse direction, going from left to right. As this happens, the voltage of the source connection (left side) dips, causing the MOSFET to turn on and hence pull the source down further based on the low signal connected through the MOSFET.

An alternative bidirectional level shifter is to use an IC such as the 74LVC245. The 74LVC245 is not compatible with I²C due to the requirement for pull-up resistors, but can be used with SPI and other sensors that don't required a pull-up resistor.

I²C LCD Display: True or False Game

In the next project you will look at I²C and use it to connect an LCD display to the Raspberry Pi in order to create a True or False game.

LCD Character Display

The display you will be using is an LCD character display. These are commonly used when a simple text display is required. These are perhaps most familiar as the display used for drink and snack vending machines.

For this I have used a display with four rows and 20 characters per row (known as a 4x20 display). Another popular configuration has two rows of 16 characters per row and you could use it instead, but you will need to change some parts of the code and this display only allows short questions.

The LCD character displays can normally operate between 3.3V and 5V, so as long as an appropriate voltage is chosen then it can be connected direct to the GPIO port of the Raspberry Pi. The disadvantage of this is that it needs to use at least six ports from the Raspberry Pi GPIO ports (plus two power supply connections) and quite a bit of wiring. An alternative is to use I²C, which reduces the number of pins to only 2 I²C connections (plus two power supply connections). Even then the I²C connections can be shared with other devices.

I²C

This is another serial communication protocol that is used to communicate with peripherals. It works as a master/slave relationship. Typically the Raspberry Pi will act as the master and communicate with peripherals such as sensors or displays.

I²C is popular for connecting to fairly low-speed devices. It is bidirectional so can be used for input and output devices. You may also see references to SMBus (System Management Bus), particularly in reference to the device drivers. SMBus is a subset of I²C and uses a more strict set of criteria, including a restriction supporting only the lower-speed devices. I will refer to I²C, but if you see SMBus in the software references, that's essentially the same thing.

The main feature of I²C is that it uses only two ports for communication, which can be shared across multiple devices. There are two connections—a data connection referred to as SDA and a clock signal known as SCL. There is a downside to this in that, as multiple devices share the same connections, they also have to share the bandwidth. This is not an issue for devices with low-bandwidth requirements, but some I²C devices (such as cameras) do have high-bandwidth requirements. This is the reason for the Raspberry Pi changing which I²C bus is connected to the GPIO connector. In earlier versions, channel 0 was connected to GPIO pins 3 and 5, but that is also used by the Raspberry Pi camera, which uses a lot of the available bandwidth. The more recent Raspberry Pi versions, including the Raspberry Pi 2, use I²C channel 1 for the GPIO connector. Pins 3 and 5 are connected to SDA1 and SCL1, respectively.

I²C uses a master/slave relationship. The Raspberry Pi will normally be the master and it will communicate with other devices on the bus, telling the slaves when they should send and receive data.

This is shown in Figure 5-12.

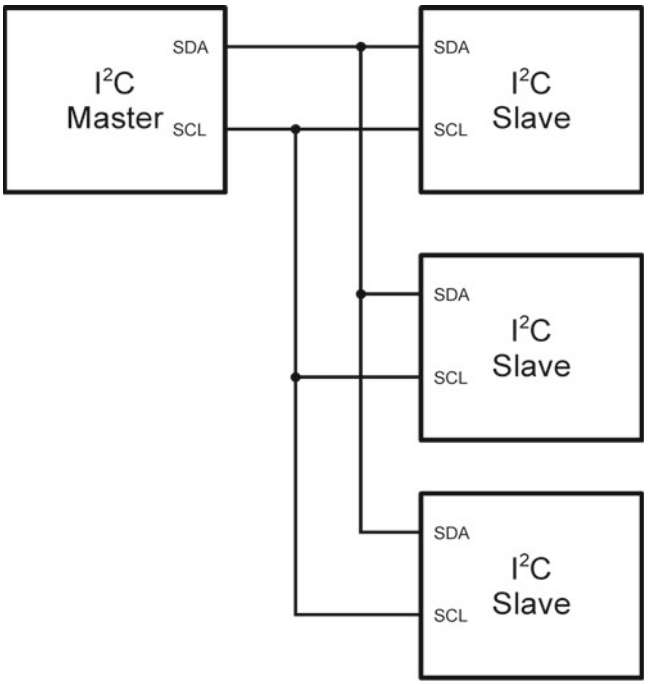


Figure 5-12. *Example of I²C slave devices connected to a single I²C master*

The I²C connections are open drain. This means that an external pull-up resistor is required. There are pull-up resistors on the Raspberry Pi that are enabled through the software, but the level-shifters and some I²C devices also include pull-up resistors onboard.

Although I²C can support a range of different voltages, it is recommended that you use the bidirectional level-shifter when you're connecting a Raspberry Pi to devices that are designed for a 5V.

Each I²C device has an address that needs to be unique to all the devices connected to that bus. To allow multiple devices of the same type to be connected, there are normally three pins on the circuit that can be connected.

I²C Backpack for the LCD Character Display

You will use an I²C backpack as a way of sending the data to an LCD character display using the I²C bus. This is a small circuit board module that can be mounted on to the back of the display, hence the name *backpack*. The use of I²C greatly reduces the number of connections needed, thus leaving more ports free on the GPIO for other components and circuitry. These backpacks are available as standalone modules or pre-soldered onto a suitable LCD display. The backpack that I used is based around a PCF8574 8-bit I/O expander for I²C-bus, which is fairly common. This uses the I²C connection to control up to eight ports, which is perfect for the LCD display. A photo of the LCD display and the I²C backpack is shown in Figure 5-13.

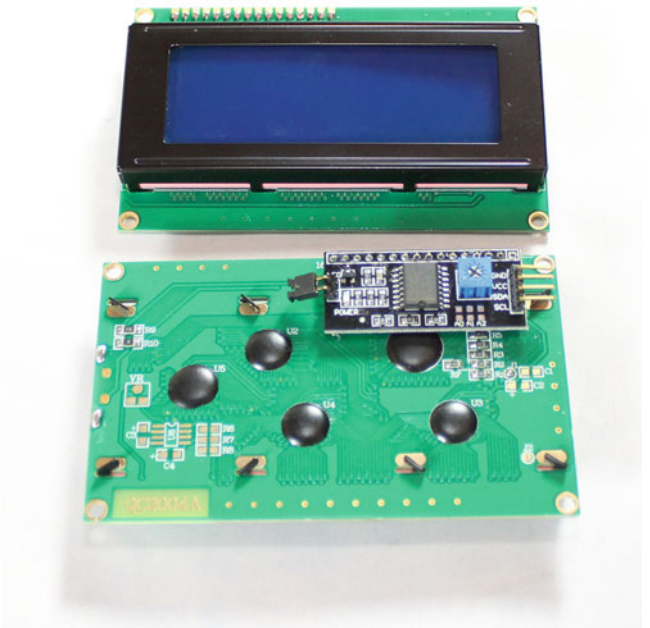


Figure 5-13. *Photo of the LCD character display and backpack mounted on the rear*

There is a trimmer (variable resistor) on the backpack that can be used to adjust the contrast of the screen. If you find that you can't see any characters, you may need to adjust this after wiring up the circuit.

The links that can be used to change the I²C address are visible in Figure 5-13, as pads just below the trimmer. For some devices, the address is selected using jumpers or DIP switches, but in this case they would need to have a link soldered across them to enable that pin. There is no need to add any links as long as there are no further devices that use the same address.

True or False Game Circuit

In addition to the LCD display connected through a level-shifter, I also used three of the other GPIO ports for pushbutton switches. These buttons are for Start, True, and False, which you can use to turn this into a simple quiz game. The ports for the switches are GPIO 23 for the Start button, GPIO 22 for the True button, and GPIO 4 for the False button.

The complete circuit diagram is shown in Figure 5-14.

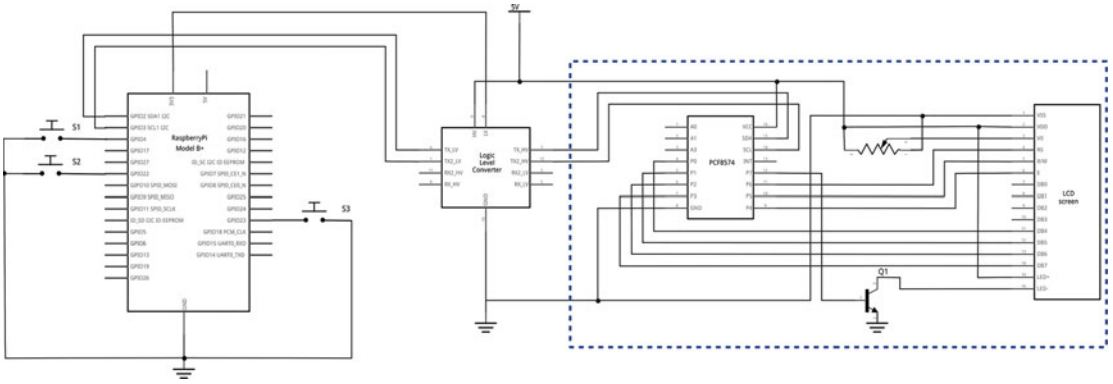


Figure 5-14. *True or false game circuit diagram*

This circuit diagram shows the internal wiring of the backpack (shown in the hashed box), which makes the diagram quite large. As a result, it is difficult to see. I have therefore split the circuit diagram in half. Figure 5-15 shows the backpack and LCD display and Figure 5-16 shows the Raspberry Pi, the level-shifter, and the switches.

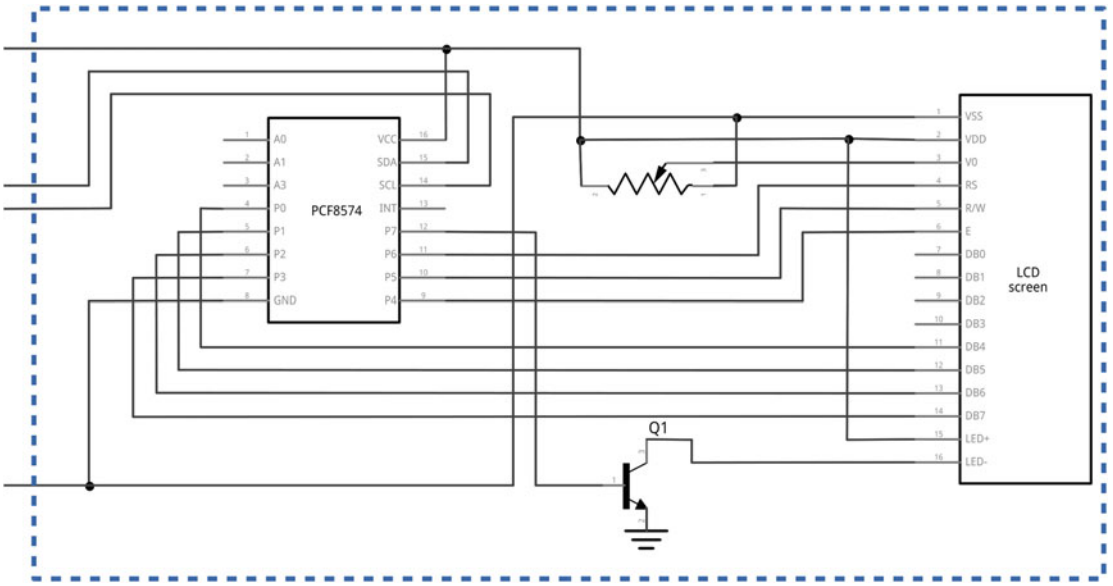


Figure 5-15. *LCD display with I²C backpack*

As you can see, the PCF8574 IC is used to send the appropriate signal to the LCD display. The variable resistor is used to adjust the contrast, which may need to be adjusted to ensure that the characters are clearly visible. The PCF8574, the variable resistor and the transistor are all included on the backpack and so do not need to be added separately.

There are only four connections shown on the left, and two of them are for the I²C channel as well as a connection to the ground and 5V power supply (which can be taken from the Raspberry Pi).

The ports A0 to A3 are the address-selection connections. By changing which of these is set to high and low, that will change the address. You can leave these to their defaults and then perform a search of I²C devices to identify which you need to use.

In Figure 5-16, you can see the level-shifter and the position of the buttons. The switches all connect to the ground, so you will use the pull-up resistors within the Raspberry Pi to provide the high level when not pressed. The part name for the level shifter in Fritzing circuit design software is *Logic Level Converter*. This means the same thing, but refers to changing between different types of logic. The level-shifter is available as a small PCB with headers that need to be soldered on. When they are soldered on, the level-shifter can be inserted into a breadboard.

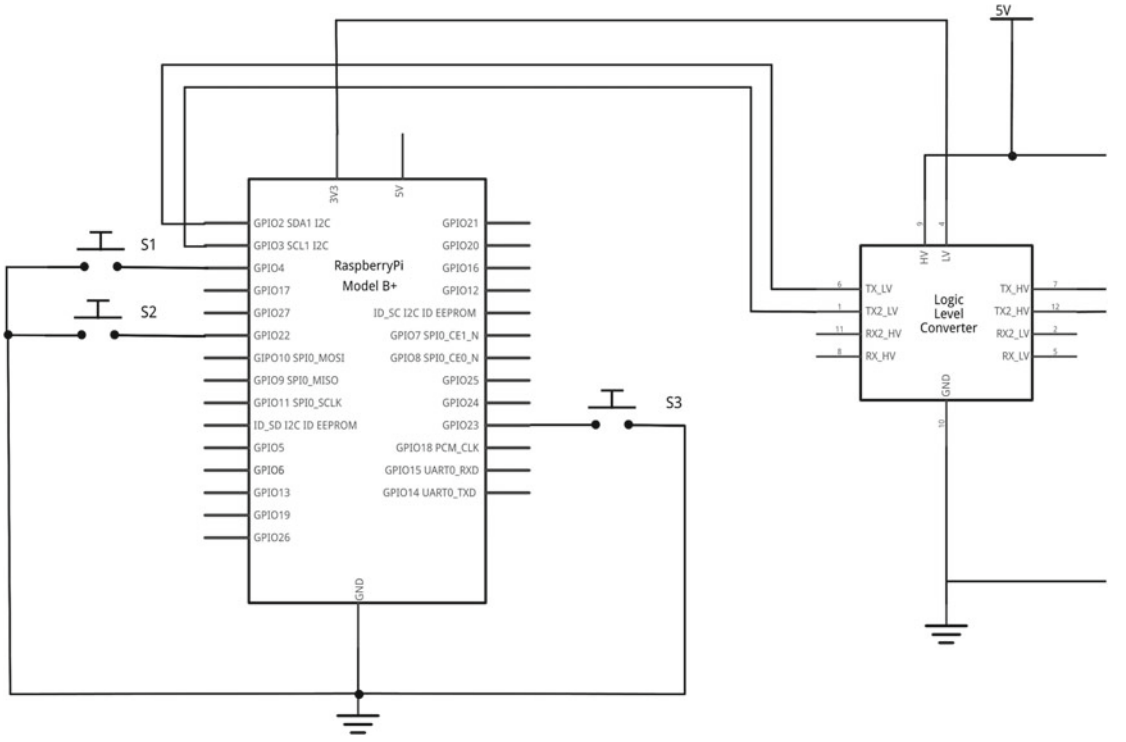


Figure 5-16. *Raspberry Pi with I²C level-shifter and button switches*

Setting Up the I²C Connection and Adding the Code

With the circuit in place, you can now switch to the software to make this work. First you need to enable I_2C on the Raspberry Pi, then identify the port that you need to communicate with, and finally create some code to put it all together.

The first thing that is required is to enable I²C on the Raspberry Pi. This is needed as the drivers are disabled by default. This can be enabled using the Raspberry Pi Configuration tool. Choose the Interfaces tab and then enable the I²C entry. This is shown in Figure 5-17.

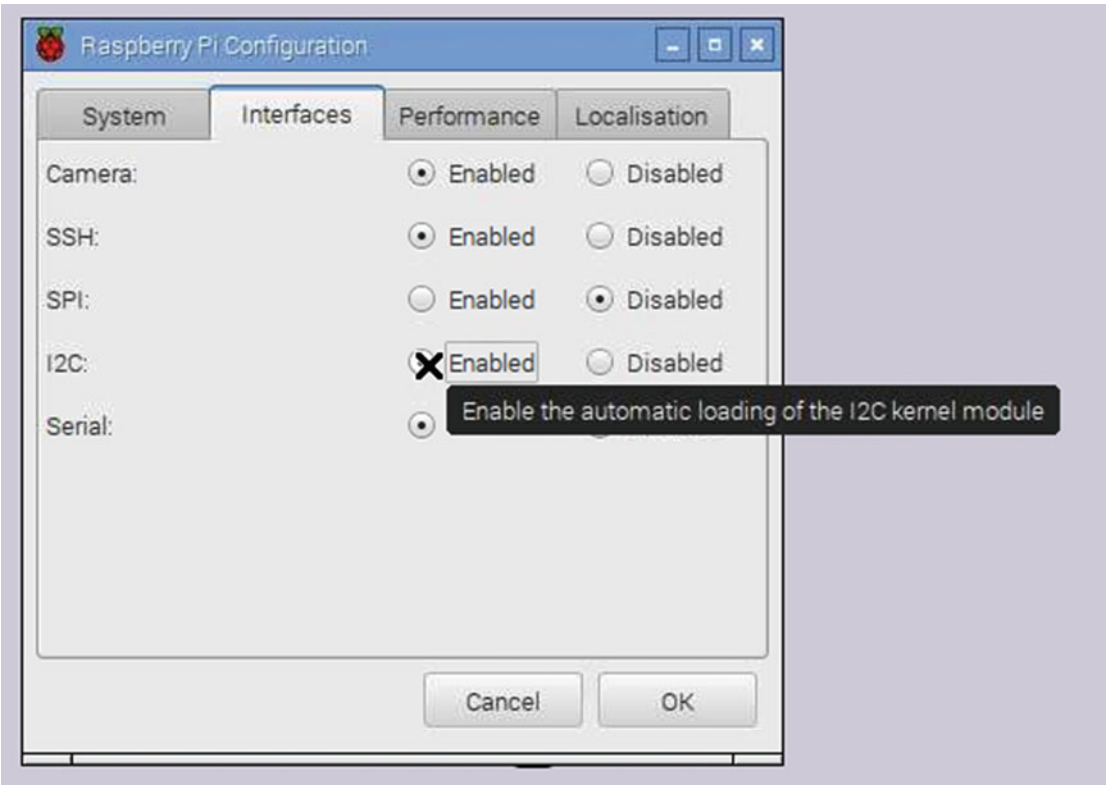


Figure 5-17. *Enabling I²C through the Raspberry Pi Configuration tool*

You'll then need to reboot to ensure that the appropriate kernel module is loaded.
Next, install the Python SMBus module using:

```
sudo apt-get install python3-smbus
```

Identify which address is being used by running `sudo i2cdetect 1`. The option 1 is to scan for bus 1, and it should be replaced with 0 for an original Raspberry Pi. This comes with a scary warning message, but that's fine if you are still trying to get this to work.

```
sudo i2cdetect 1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 3f
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

As you can see from this output, a single I²C device has been found at address `3f`.

There are several libraries around for communicating with I²C devices and for the LCD character display. I didn't find one that met my requirements and was simple enough to use, so I created a slightly modified version based on an existing project (more details are included in the source file).

The code for the Python module is called `I2CDisplay.py`. This is included in the `quiz` folder in the book's source code or can be downloaded directly from GitHub at https://raw.githubusercontent.com/penguintutor/learnelectronics/master/quiz/I2CDisplay.py.

The module has been updated with the address of the backpack using `I2C_ADDR = 0x3f` and the number of characters available on my LCD display using `LCD_WIDTH = 20`. These can be changed to reflect the LCD display if required. This file needs to be downloaded to the same directory as the code you will create.

First you'll create some test code that sends a message to the LCD display. This is called `i2ctest.py` and it's found in the `quiz` folder of the source code:

```
from I2CDisplay import *
import time

# Initialise display
lcd_init()

# Send some test
lcd_string("Learn Electronics",LCD_LINE_1)
lcd_string("with Raspberry Pi",LCD_LINE_2)
lcd_string("by",LCD_LINE_3)
lcd_string("Stewart Watkiss",LCD_LINE_4)

time.sleep(20)
lcd_clear()
```

The first line imports the `I2CDisplay` module. This is a local Python file, so it needs to be stored in the same directory as the file. As all the functions are imported, it's just a case of calling each function directly.

The first function that is called is `lcd_init` and it sets up the LCD display and clears the screen.

The program will then send for lines to the screen, using `LCD_LINE_1` to `LCD_LINE_4` to determine which line to send the output to. It then sleeps for 20 seconds before issuing the `lcd_clear` command to clear the screen.

Load this into IDLE3 and run it. You should see the message displayed on the LCD display. If it doesn't run, ensure that you have the `I2CDisplay.py` file in the same directory as the test file.

You can now add the rest of the code to turn this into a full program:

```
from I2CDisplay import *
from gpiozero import Button
import time

# File holding questions
# 4 lines are the question and then the
# 5th is T for true or F for false
# Repeat for Q 2 etc.
quizfilename = "quiz.txt"
```

```
start_button = Button(23)
true_button = Button(22)
false_button = Button(4)

# Initialise display
lcd_init()

# Send some test
lcd_string("Raspberry Pi",LCD_LINE_1)
lcd_string("True or False quiz",LCD_LINE_2)
lcd_string("",LCD_LINE_3)
lcd_string("Press start",LCD_LINE_4)

start_button.wait_for_press()

# Note that there is no error handling of file not exist
# Consider using a try except block
# Open the file
file = open(quizfilename)

questions = 0
score = 0
answer = ''

while True:
    # print 4 lines as the questions
    thisline = file.readline().rstrip("\n")
    if thisline == "" : break
    lcd_string(thisline,LCD_LINE_1)
    thisline = file.readline().rstrip("\n")
    if thisline == "" : break
    lcd_string(thisline,LCD_LINE_2)
    thisline = file.readline().rstrip("\n")
    if thisline == "" : break
    lcd_string(thisline,LCD_LINE_3)
    thisline = file.readline().rstrip("\n")
    if thisline == "" : break
    lcd_string(thisline,LCD_LINE_4)
    # Next line should be T for answer = True or F for False
    thisline = file.readline().rstrip("\n")
    if thisline == "" : break
    if (thisline == "T"):
        answer = "T"
    elif (thisline == "F"):
        answer = "F"
    # should not reach this unless the question file is invalid
    else : break
```

```
    # wait on True or False pressed
    while (true_button.is_pressed == False and \
        false_button.is_pressed == False):
        time.sleep (0.2)
    # Increment number of questions attempted
    questions = questions+1
    # Once one of the buttons is pressed
    # also check the other is not pressed to avoid cheating
    if (answer == "T" and true_button.is_pressed \
        and false_button.is_pressed == False):
        score = score+1
        lcd_string("Correct!",LCD_LINE_4)
    elif (answer == "F" and true_button.is_pressed == False \
        and false_button.is_pressed):
        score = score+1
        lcd_string("Correct!",LCD_LINE_4)
    else:
        lcd_string("Wrong.",LCD_LINE_4)
    # Wait 2 seconds before next questions
    time.sleep(2)
    # Finished this question return to the start
# Outside of the quiz loop - give the score
lcd_string("End",LCD_LINE_1)
lcd_string("Score",LCD_LINE_2)
lcd_string(str(score)+" out of "+str(questions),LCD_LINE_3)
lcd_string("",LCD_LINE_4)
time.sleep (5)
file.close()
```

Although that's quite long, most of it is a combination of the LCD display test code previously and GPIO Zero used to test for when the buttons are pressed. Something that is new is that you are reading in the contents of a file to get the questions and expected answers. Details of the file format are explained later.

The first thing you do is import the appropriate modules, add a variable with the filename for the quiz file, and then set up the buttons using GPIO Zero. A message is then sent to the LCD display; you use `wait_for_press` to wait until the Start button has been pressed.

The file is opened using `file = open(quizfilename)`. After setting the number of questions and score to 0, you enter a loop to read in each question in turn and get a response from the user. This has been created as a `while True` loop, which is normally used when you don't expect to exit out of the loop. You will instead be exiting at the appropriate point in the loop using a `break` command when you reach the end of the file or when you encounter a problem.

Once in the loop, the first thing you do is to use readline to read the next line from the file. The `readline` command will keep the terminating character, which is the newline character `\n`. You don't want to send that to the LCD display so you use `rstrip` to remove any new lines. The end value, which is the contents of the line without the newline character, is stored in the variable this line. The code then checks if this is an empty string, which will indicate that there are no more questions left. If the line is empty, the `break` command is called and that will cause the execution of the `while` loop to stop and the control to move to the next line outside of the loop.

The first four lines are the question so these are sent directly to the `lcd_string` function. The next line contains a letter T if the answer should be True and the letter F if the answer should be false. This is stored in the variable `answer` so you can compare it against which button is pressed.

The next thing is a `while` loop that waits for the True or False buttons to be pressed, which is as follows:

```
while (true_button.is_pressed == False and \
        false_button.is_pressed == False):
        time.sleep (0.2)
```

You cannot use the `wait_for_press` method as you did previously as this time you need to check for the state of two different buttons. So you use another `while` loop that will continue to run until one of the buttons is pressed. This `while` loop includes a sleep function to sleep for 0.2 of a second. This is added to reduce the load on the Raspberry Pi. Without the sleep, the program will be constantly checking the status. The time is set to be small enough that the user won't notice a delay of 0.2 seconds between pressing the button and the computer responding, but that length of time will allow another program on the system to do some other work.

You then need to check the status of both buttons to check to see if the button pressed matches the one you expect. Even if it matches, you still need to check the other button is not pressed, as the user could cheat by pressing both buttons at the same time.

You then update the score if appropriate and show whether the answer was correct or not. The program will then display the status for two seconds before moving on to the next question.

When you reach the end of the file, the program will exit from the `while` loop and display the score, and then wait for five seconds. It then calls the `close` on the file to ensure that the file is closed correctly. Note that I didn't include an `lcd_clear` function. This means that the display will continue to display the score after it has finished running.

Finally, you need a file containing the questions. Here is the file I used:

```
The Raspberry Pi 2
has 1GB of RAM

True or False?
T
PiZero is the
name of a Python
electronics library
True or False?
F
The Raspberry Pi has
wifi on the main
board.
True or False?
F
The RPi model B+
has 4 x USB2 ports

True or False?
T
The HDMI Connector
on the RPi 2
is a mini-HDMI?
True or False?
F
```

Notice that for each question there are four lines for the questions followed by the letter T or F as appropriate. For blank lines where you don't need all the lines, a space is included. Otherwise, the program will think that you have reached the end of the file.

One problem with the program is that it doesn't include any error checking. So for instance if the `quiz.txt` file does not exist, you will get a rather ugly error message like this one:

```
Traceback (most recent call last):
  File "quiz.py", line 28, in <module>
    file = open(quizfilename)
FileNotFoundError: [Errno 2] No such file or directory: 'quiz.txt'
```

Obviously, this is not a very user-friendly error message, so it is a good idea to add code to check for conditions that have a high probability of happening, such as a missing file.

SPI Analog to Digital Input

So far you have used digital sensors and outputs with the Raspberry Pi. This works well with the digital inputs on the Raspberry Pi, but not all sensors are digital. You will now look at how to connect an analog sensor to the Raspberry Pi. You will use an analog-to-digital convertor (A-to-D convertor, also known as ADC) that converts the analog signal into a digital signal. You will then use an SPI (Serial Peripheral Interface Bus) to read the value from the analog-to-digital convertor.

Creating an Analog Value Using a Potentiometer

The analog sensor you will use is one that you can have control of. You will use a *potentiometer*, which is a resistor whose value you can change. It is also known by other names such as a pot or a variable resistor. These are often rotary devices such as volume controls on music players, although slide potentiometers are also available. They are also available as small devices, and are often not accessible to the end users or they are designed to be rarely changed; these are known as trimmers or trimmer pots.

To use a potentiometer in order to provide an analog signal, you can connect it across the supply voltage and use the variable pin as the analog value. This can then be used as the input to the analog-to-digital convertor. This is shown in Figure 5-18.

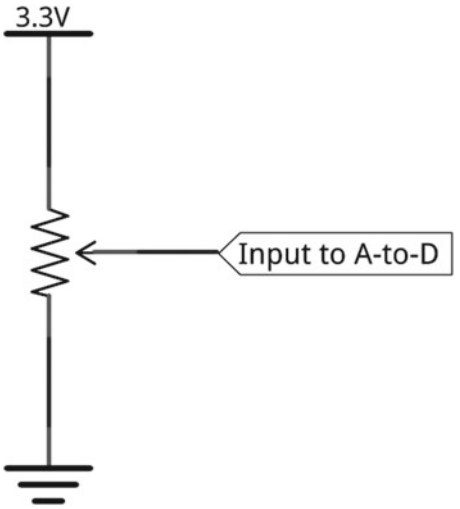


Figure 5-18. *Using a potentiometer to get an analog input*

There are a few different symbols for a potentiometer that show the arrow in a different way, but this particular symbol is good at showing how this works. This uses the same voltage divider circuit you used previously. With the potentiometer set to the center position then there is effectively half the resistance above and below it giving half of the supply voltage as the output. If the potentiometer is moved toward the top of the resistor, the voltage increases, and if it's moved down, the voltage decreases.

Analog-to-Digital Conversion

The particular analog-to-digital convertor integrated circuit that you will use here is the MCP3008. This IC uses an SPI interface to pass the digital value to the Raspberry Pi. Other integrated circuits are available, including ones that you can connect to multiple pins of the Raspberry Pi GPIO connector (although that would use a lot of pins) and I^2C. The main reason for choosing SPI is to show an alternative serial communication protocol that's commonly used for various input and output devices.

The way that this particular IC works is known as *successive approximation* analog-to-digital conversion. It works by storing the input value and then comparing it against a varying signal until a match is reached. This is then the digital value provided as the output.

SPI (Serial Peripheral Interface Bus)

SPI is the main alternative to I^2C as a serial communication protocol for communicating with electronic devices. It is a bidirectional communication protocol that allows full-duplex communication, whereby information can be passed in both directions at the same time. This compares with I^2C which, while still allowing bidirectional communication, allows communication to go in only one direction at a time. The downside is that SPI needs more connections with four wires required for the first device (in addition to ground) and then one more connection for each subsequent device you add. This is shown in Figure 5-19.

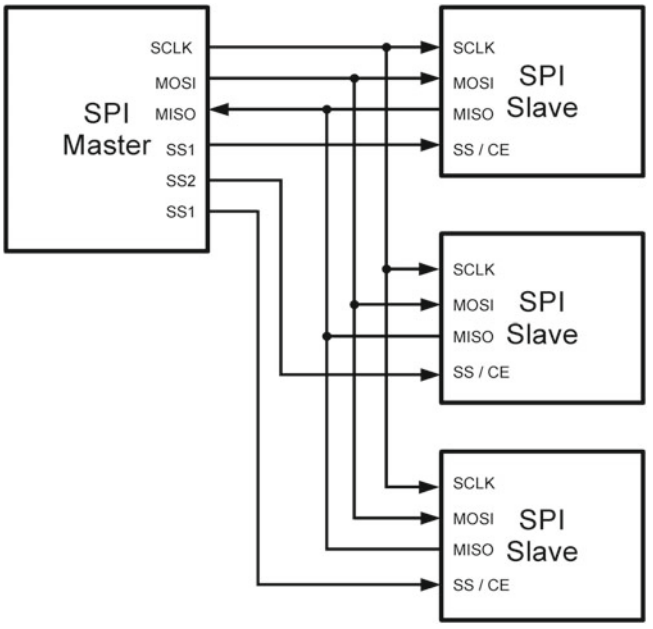


Figure 5-19. *SPI bus connections*

The common connections are:

- SCLK—SPI Clock Signal
- MOSI—Master Out, Slave In
- MISO—Master In, Slave Out

For each slave, a Slave Select (SS) connection is required. On the end device, this could be referred to as Chip Enable (CE) or sometimes Chip Select (CS). The Slave Select is normally low to enable and a high to disable, so it is often shown with a bar (line) above it.

Another advantage of SPI is that it can be connected to a 3.3V power supply, which means no voltage level shifting is required. This assumes that the sensor connected to the ADC can also operate at 3.3V, as the reference voltage should not exceed the supply voltage.

Potentiometer SPI Circuit

The final circuit is shown in Figure 5-20.

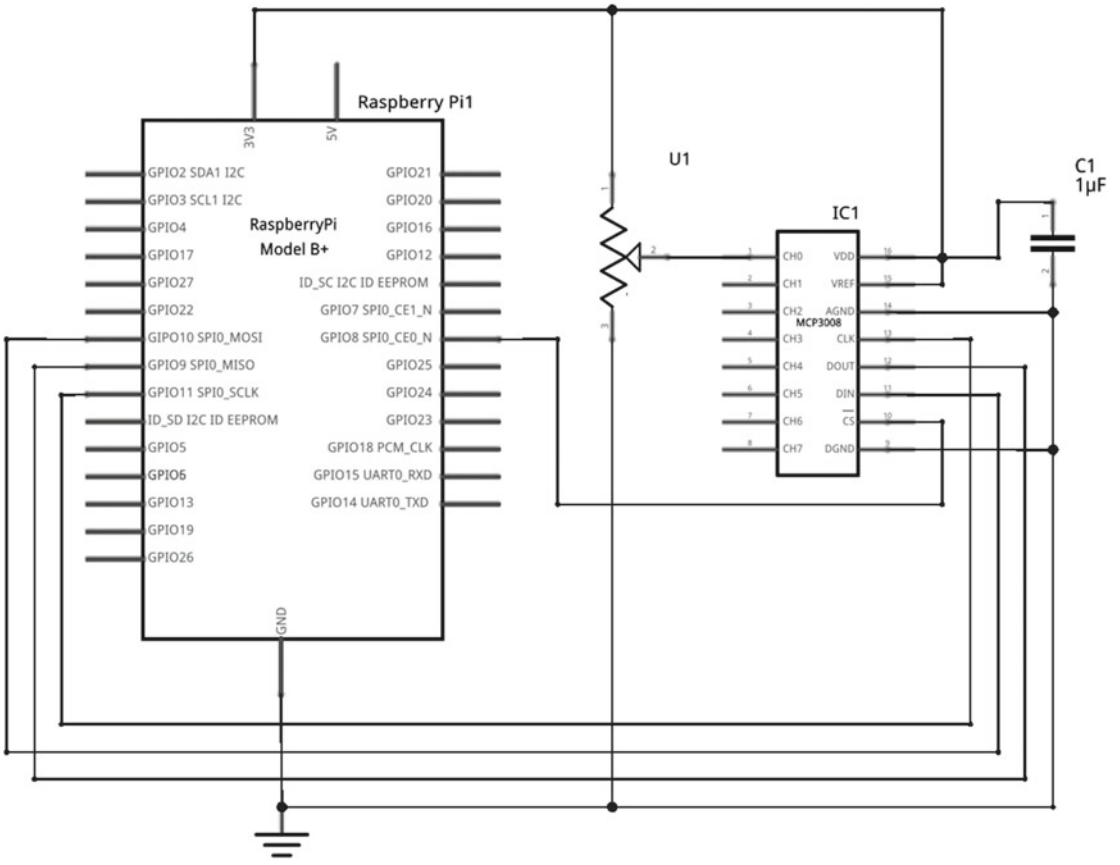


Figure 5-20. *Variable control using a potentiometer and SPI ADC*

This shows the potentiometer connected to channel 0 of the MCP3008 and then the appropriate data lines connected to the Raspberry Pi. Note also that there is a capacitor connected across the supply voltage. This is known as a *bypass capacitor* (to remove noise on the power line) and should be connected as close to the IC as possible. This is sometimes omitted but according to the datasheet should always be included.

Accessing the ADC Using Python

The first thing that is required is to enable the SPI kernel module. This can be achieved through the Raspberry Pi configuration tool, as shown in Figure 5-21. This will require a reboot before the changes take effect.

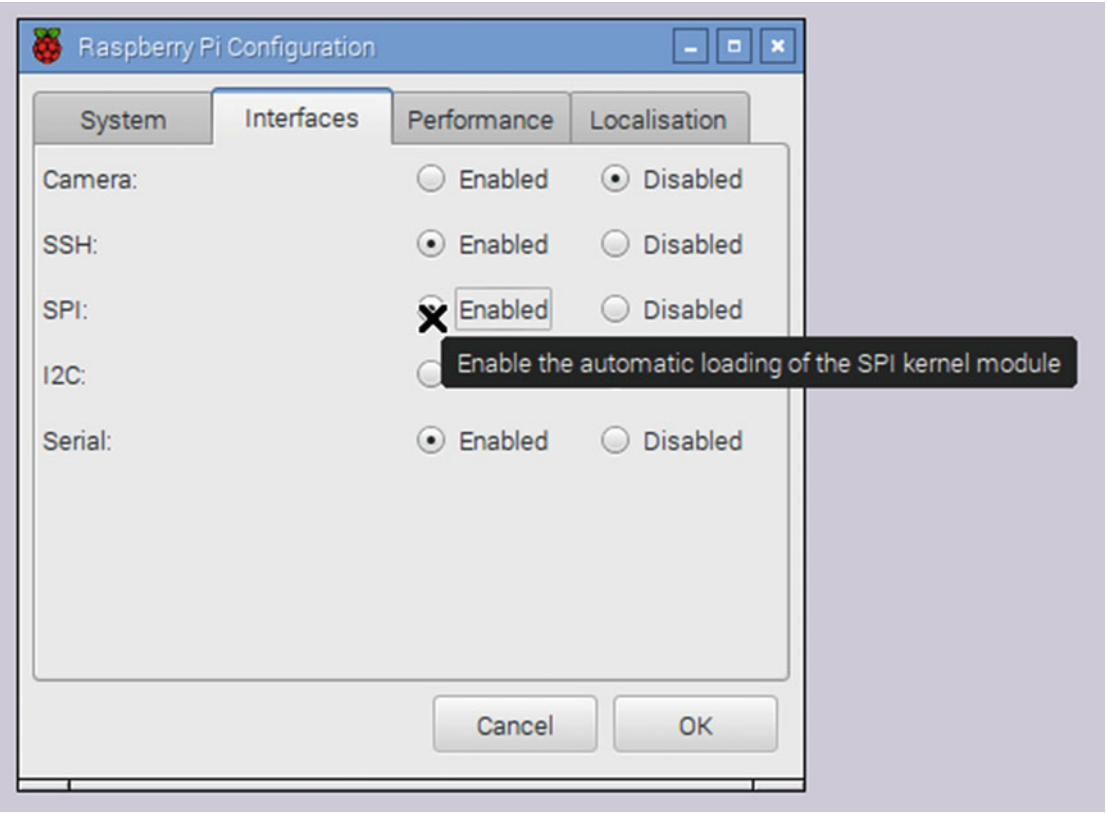


Figure 5-21. *Enable the SPI kernel module*

The Python module also needs to be installed:

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

Sending a request to the ADC using the `spidev` module involves combining several values into a single number, which is done using the shift operator `<<`. A similar process is required for extracting the value from the response; you use the right shift operator `>>`. To make this easier, I've separated this into a function that creates the request in the appropriate format and then returns a single value for the value of the ADC. Functions are covered in the next chapter, but for now you just need to know that you can request the value of an analog port using the `readAnalog` function followed by the channel number on the MCP3008 (in this case, 0).

The test code is provided here and should be saved as `spi-test.py`:

```
import spidev
import time
```

```
# Sensor Port - where analog device is connected to
sensorChannel = 0

spi = spidev.SpiDev()
# Open connection to SPI device: Bus 0, Device 0
spi.open(0,0)

# Read using SPI - A to D convertor
# Has 0 to 7 inputs
# Returns integer of value read 0 to 1023
# or returns -1 on error
def readAnalog(input):
    if (input < 0 or input > 7):
        return -1
    req = 8+input
    # shift to higher bits
    req = req << 4
    # 1 and 0 are start and stop values in tuple
    resp = spi.xfer2([1,req,0])
    # resp is split across two entries [1] and [2]
    # shift and merge these together
    high = resp[1]&3
    low = resp[2]
    return ((high<<8) + low)

while True:
    analogValue = readAnalog(sensorChannel)
    print ("Value : " + str(analogValue))
    time.sleep(1)
```

To communicate with the ADC using SPI, you first need to create an `spidev` object, which is called `spi` here. The `open` command then specifies the bus and device numbers. This example uses SPI bus 0 and the CE0 pin on the Raspberry Pi, so these are both set to 0.

The function handles converting the request into a format that can be used with the `spidev` modules, so you just have a `while` loop that calls `readAnalog`, prints the output (converted from a number to a string), and then sleeps for one second before taking another reading.

The value returned will be between 0 and 1023 (or -1 if an error occurred). A useful way that this could be used is to provide a way of adjusting the time delay on one of your existing programs. Replace the delay with the value returned, divided by 1023 and then multiplied by the maximum delay that you want. One problem with this is that it will see the first divide as being an integer operation, which will work out to the nearest whole number. This will therefore give a value of either 1 or 0 which is not what you want. You can use the `float` operation to convert one of the numbers to a floating point (number with a decimal point) or just use 1023.0 for the division to tell Python to keep the value after the decimal place. So if you wanted to have a delay that goes from 0 to 10 seconds, you would use:

```
delay = readAnalog(sensorChannel) / 1023.0 * 10
time.sleep(delay)
```

or you could use:

```
delay = float(readAnalog(sensorChannel)) / 1023 * 10
time.sleep(delay)
```

If you need the value as an integer (a whole number), you should still have the division performed as a float, but then convert to an integer afterward using `int()`.

■ **Tip** When performing a division, always make it clear to Python where you expect to get a decimal point. This can be done by using `float()` around one of the values or for a fixed number by adding `.0` to the end.

This example used a potentiometer, but other analog sensors can be used such as a light dependent resistor, a temperature sensor, and an analog joystick.

More Input and Output

This chapter looked at different inputs, including a PIR motion sensor camera, a way to communicate with infrared devices using an infrared transmitter and receiver, and how to communicate with devices using two popular serial protocols I^2C and SPI. It also looked at how you can convert between different voltages and how you can read an analog sensor using an analog-to-digital convertor (ADC). You also learned how to add a different output using an LCD character display. With this knowledge, you can now add a variety of different sensors and output devices that use one of the techniques you have used or something similar.

With knowledge of these different sensors, you have now unlocked opportunities for many more circuits and programs. For the PIR motion sensor, you could look at having the Raspberry Pi place a recorded message through its audio output port, or even have it send a tweet whenever a photo is taken.

For the infrared remote control, look at what other devices you have that could be controlled using infrared. The obvious devices are TVs and media centers, but infrared is used in other devices, including toys. You will look at how you can automate turning the light on and off in the next chapter and also see how to use infrared to control a Lego train.

For the LCD character display, you could look at adding scrolling text (particularly if you only have a two-row LCD display) or adding another loop so that the game starts again without needing to restart it.

In the next chapter, you will look at some of the programming techniques that will be useful for the more adventurous projects.

CHAPTER 6

Adding Control in Python and Linux

This chapter covers the software side of the projects. This includes things you can do in the Python programs, but also some of the features of Linux that will allow you to start your programs automatically. This will provide some of the techniques you will need for projects in the later chapters. Some of the techniques listed will help you add functionality to your disco lights, have your PIR camera start automatically, and add a basic graphical interface for controlling electronics projects. The chapter also introduces a reed-switch that will be used to have a Lego train automatically stop at the station and control multi-colored LEDs.

Computers receive information (data), process the data using stored instructions, and then provide an output. The inputs could be a keyboard and mouse or could be some kind of sensor. Likewise, while a computer screen is traditionally an output device, output could be turning on a light or controlling the brakes on a car without any kind of visual screen. There are lots of devices that you have in the home that you may not have considered to be a computer. For example, a programmable microwave oven could be considered a computer, as could an MP3 player or a digital radio. Some modern cars have between 50 and 100 embedded computers controlling various things from the fuel mix and braking system to the entertainment system and navigation system.

The use of stored programs is the part that turns these electronic circuits into a computer. Sometimes the programs are stored in firmware on the computer and sometimes these are programs that can be loaded from disk.

In the case of the Raspberry Pi, the programs are stored on the SD card. The operating system is a Linux-based distribution known as Raspbian. Most of the software created for this book is written in Python, which has lots of modules that making programming the hardware easy. It's a great language for those learning programming.

Taking the Next Steps in Python Programming

The Python programs that you have created so far have been fairly simple. One of the reasons that Python is so simple to use is thanks to the code inside the Python interpreter, which hides a lot of the complexity that other programming languages have. It's also thanks to some of the Python modules that others have been written that make interfacing with the hardware so easy.

If you continued in the same style of programming then the code would become harder to follow and would be a hindrance to what you want to achieve. You will therefore look at some techniques that will make your programs easier to write and understand. This chapter therefore looks at creating functions to make the program easier to follow.

The programming style you will be using is mainly procedural programming, where the program runs in sequence step-by-step. You will stop short of learning object oriented programming (OOP), which is also supported by Python, but needs a different approach. This is not because of any faults of object oriented programming, but more that the space in this book would not be sufficient to give it justice. After some

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_6

experience with OOP, I am now a big fan and think there are many examples where OOP is a much better approach to tackling a programming task. It is however a bit more complicated when starting out.

You'll still use some OOP when you access modules that are written using object orientation, including GPIO Zero. As you get more experienced with programming, it will be useful to learn more about object oriented programming.

You have already used Python to make decisions, which could be moving a character based on the press of a switch or changing the length of a time delay based on the value of an analog input. You've also made decisions in the form of a loop that has repeated a certain number of times or until a certain condition is met. These decisions are key to being able to handle the inputs and outputs from your electronic circuits. Some of these are listed here as a recap, but there perhaps might also be some new ones that can make programming a little easier.

The first is the `if` statement, which performs an action depending on the result of a comparison. It typically determines if a certain variable is equal to, greater than, or less than another value.

```
if myvariable == 10 :
    print ("The value is equal to 10")
```

This checks if the value of `myvariable` is equal to 10. Note that there are two equals characters used, and only a single one will result in the value of 10 being stored into `myvariable`, which is not what you normally want in an `if` statement. To test for being `myvariable` being higher than 10, use the greater-than symbol `>`. Likewise, to test that it is lower than 10, use the less-than symbol `<`. If you want to check for something being not equal to a certain value, use `!=` in place of the double equals sign used previously.

You can also use an `if else` statement, which will perform one action if the condition is met and something else if the condition is not met:

```
if myvariable == 10 :
    print ("The value is equal to 10")
else :
    print ("The value is not equal to 10")
```

In that example, as before, the first `print` statement is called if the value is equal, but if the value is not equal, the `print` statement following the `else` statement will be called.

Another useful method is to chain events together. While we could do this by nesting `if` statements within the `else` statement, this is easier to follow using an `elif` statement (which is a contraction of *else if*).

```
if myvariable == 10 :
    print ("The value is equal to 10")
elif myvariable < 10 :
    print ("The value is less than 10")
else :
    print ("The value is greater than 10")
```

In this case, the `elif` statement checks for the value being less than 10. As you already know that it is not equal to 10, you can deduce that anything that doesn't meet the `if` and `elif` conditions must be greater than 10.

Another useful technique is to be able to put multiple conditions within a single `if` statement. This can be achieved by using the words `and` or `or` between each clause.

```
if ((button_pressed == True) and (myvariable > 10)) :
        print ("Well done score - you won")
```

This example checks that both `button_pressed` is `True` and that `myvariable` is greater than 10. If you wanted this to be run if either condition was met, you would use `or` instead. I added brackets around the various parts of the `if` condition to make it clearer to understand, but these are not required. Brackets can help with understanding what is happening and will avoid any problems due to operator precedence (where certain operators are considered before others).

Loops are another form of decision making. You have mostly used the `while True` loop, which keeps on running forever, or at least as long as the program is running (as the program can be stopped by pressing Ctrl+C or some other external call). Although I say this runs "forever," there is one way you can exit it, which is to include a `break` statement. This is often used in an `if` statement so that if a certain condition is met, it "breaks" from the loop. It's also possible to return to the start of the loop using a `continue` statement. The following code shows both of these in use:

```
myvariable = 0
while True:
    myvariable = myvariable + 1
    print ("In loop")
    if (myvariable < 10) :
        continue
    break
```

This will print the line `"In loop"` ten times. It starts by incrementing the value of `myvariable`, prints `"In loop"`, and then checks to see if it's been printed less than 10 times. If so, it continues back to the top of the loop. Once it has looped 10 times, it will instead reach the `break`, which exits from the loop.

An alternative to the `while True` loop is to use a condition in the `while` statement:

```
myvariable = 0
while myvariable < 10:
    myvariable = myvariable + 1
    print ("In loop")
```

In this example, the program will run as long as `myvariable` is less than 10. This will work the same as the previous example.

Python also provides another way of doing this, which is usually easier to understanding using a `for range` loop:

```
for i in range (0, 10) :
        print ("In loop")
```

This example uses only two lines, which is much less than the previous examples. This is because you no longer have to track your own variable. The `range` function creates a list of values, starting at 0 and going up to 9. The `for` loop then iterates over each of these in turn by copying the value from `range` into the variable `i` and then running that iteration of the loop. You're not limited to using numbers in the `for` loop; you could also iterate over any lists.

```
mylist = ["Loop 0", "Loop 1","Loop 2","Loop 3","Loop 4","Loop 5"]
for thisstring in mylist:
        print ("In "+thisstring)
```

This example creates a list of five entries and then iterates over each one, putting the value into the variable `thisstring`. Each entry is then printed out, one line at a time.

Creating Python Functions

Most of the software in this book so far has been written as a sequential block of code designed to be run from the top to the bottom of the file. I include loops to avoid some repetition, but otherwise if you wanted to do something multiple times, you just add that instruction to the code. This can create duplicate blocks of code, result in very long code, and make it difficult to understand what the code is doing. The solution to this is to move related parts of the code into a function. Then, instead of having to copy that same block of code into multiple places, you can just call the function instead. This also helps with code reuse, where a function can be called multiple times without needing to copy and paste blocks of code within the program.

To create a function, the block of code starts with a `def` statement and includes the title of the function and any parameters, followed by the code inside the function.

For example, when reading the analog input from the SPI ADC, you can create the following function:

```
def readAnalog(input):
    # Code for the function goes here
```

This example defines a function called `readAnalog`, which has a single argument called `input`.

If you want to return information to the calling function, you use `return` followed by the value to return or the name of the variable. The following example shows how you can create a simple function that takes two parameters—`input1` and `input2`—and returns whichever value is the largest (or if they are equal, then the first input):

```
def largestValue (input1, input2):
    if input1 > input2:
        return input1
    elif input2 > input1:
        return input2
    else:
        return input1
```

This can then be called within the main program using:

```
largestValue (3, 2)
```

which will return the value 3.

Adding Disco Light Sequences Using Python Functions

As Python functions are so useful, it's worth including a real example that controls electronics. This example revisits the earlier disco lights and adds some different sequences that can be called from within functions.

If you have created the circuit from Chapter 4 (Figure 4-26), you can use that, but if not, Figure 6-1 shows a simplified circuit that can be used for testing these functions.

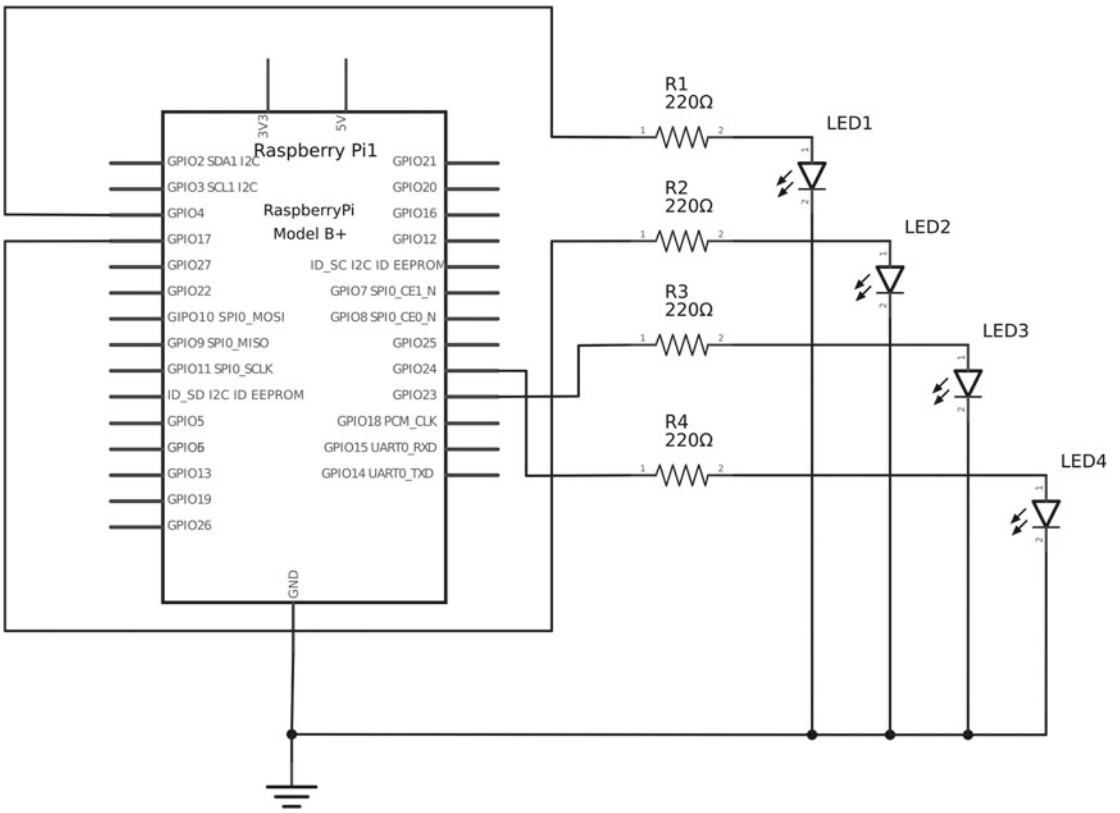


Figure 6-1. *Simplified LED disco light circuit*

This circuit has only four resistors and four standard LEDs. It uses the same layout as the disco light project and so can be used for testing the different sequences.

The following code should be saved as `discolight-sequences.py`:

```
from gpiozero import LED
import time

# GPIO port numbers for the light
#9 = gnd, 7 = GPIO 4, 11 = GPIO 17, 16 = GPIO 23, 18 = GPIO 24
LIGHTGPIO = [4, 17, 23, 24]
# Time between each step in the sequence in seconds
DELAY = 1

lights = [LED(LIGHTGPIO[0]), LED(LIGHTGPIO[1]), LED(LIGHTGPIO[2]), LED(LIGHTGPIO[3])]

def allOn():
    for x in range (4):
        lights[x].on()
```

```
def allOff():
    for x in range (4):
        lights[x].off()

def sequence():
    for x in range (4):
        for y in range (4):
            lights[y].off()
        lights[x].on()
        time.sleep(DELAY)

def repeatSequence(numsequences):
    for x in range (numsequences):
        sequence()

# Main code starts here
allOff()
time.sleep(DELAY)
allOn()
time.sleep(DELAY)
sequence()
time.sleep(DELAY)
repeatSequence(6)
```

In this program, I created four functions. The `allOn()` function turns all the LEDs on, the `allOff()` function turns all the LEDs off, `sequence()` lights the four LEDs in sequence, and `repeatSequence()`repeats the sequence for a number of times as defined in the parameter for the function.

The functions will not run until they are called in the section after the comment `# Main code starts here`. Each function is then called individually with a delay between each one. The function `repeatSequence()` calls another function `sequence()`, which allows even further code reuse. Note that while several of the functions reference the variable `x`, these are local variables that apply only within the function. So even though `repeatSequence` calls `sequence` and both functions refer to the variable `x`, these values are independent of each other. The list that holds the lights is created outside of the function and is visible to all the functions without having to create a new copy for each function; this is known as a *global variable*.

The `allOn()` and `allOff()` functions should be fairly self-explanatory.

The `sequence()` function uses a nested loop. The first loop uses the variable `x` and will be called once for each disco light. The first run within the loop will turn off all the LEDs and then turn the first LED on. The second will turn off all the LEDs and then the second LED on, which repeats up to the fourth, which turns all the LEDs off and then the fourth one back on. This gives the effect of showing a single LED being lit, shifting from left to right. The inside loop turns all the LEDs off. This could have been replaced with a call to the `allOff()` function.

The `repeatSequence()` function takes a parameter called `numsequences`, which should be an integer number. The loop will then call `Sequence()` the number of times that `numsequences` was set to.

Using a Python Main Function

In addition to using functions for small parts of the code, you can create a special function around the main part of the code. This is a requirement in many other programming languages, but is not actually required in Python. You do this by creating a function called `main()` and calling it when the program is started:

```
def main():
    #The main program code goes here

#Run the main function when this program is run
if __name__ == "__main__":
    main()
```

This separates the main function from the rest of the file so that variables in `main` are local to `main` rather than to the global variables. It is generally good practice to avoid using global variables when possible. You can still include global variables in the file, where they are used as global functions in the start of the program and can be created using the `global` keyword. This is a useful thing to know about, particularly when looking at code that others have written.

Another benefit of creating a main function is that it allows your code to be made into a module, but still allows it to be executable. Again, this is considered a more advanced technique, so it won't be discussed further here.

Making Python Programs Executable

The programs you have created so far have been executed from within IDLE 3 or would need to been run on the command line using:

```
python3 myprogram.py
```

It is useful to be able to run the program directly using its name. You can use this by adding a shebang entry as the first line of the program. This takes the form of a hash character #, which is normally used for comments, followed by an exclamation mark. This is a contraction of hash and bang, which is an old nickname for the exclamation mark. To use Python 3, you put the following as the first line in your Python program.

```
#!/usr/bin/python3
```

The program needs the full executable path, which can be found using the `which` command from the command prompt as follows:

```
$ which python3
/usr/bin/python3
```

The one downside of this is that it may not work across different UNIX-like operating systems, including different Linux distributions. This is not going to be a problem for these programs, since they will only work with the Raspberry Pi anyway, but you may see a slightly different entry, which is:

```
#!/usr/bin/env python3
```

This achieves the same thing, but instead of `python3` having to be in a certain directory, the `env` (environment) command will locate it for you.

The next stage in making a program executable is to tell Linux that it has permission to run. By default, Linux gives permission to read and write to the file but not execute it. The `chmod` command can be used to add executable permissions using `+x` as shown:

```
chmod +x myprogram.py
```

After adding a shebang entry and making a program executable, you can run it directly from the command line. The program should be prefixed with `./` (which means run it from the current folder) as shown:

```
./myprogram.py
```

This makes the program a little easier to run and is a bit more professional than having to type `python3` before it. It also makes it easier to run the program with command-line options and to run the program automatically. You will look at these next.

Handling Command-Line Arguments

So far the programs have either run without any user control or have been dependent on interacting with the user directly or through an electronic circuit. It is useful to be able to tell a program what you want it to do without needing to interact with it in real time. This is particularly useful if you want the program to act differently depending on how it runs. You can achieve this by adding command-line arguments. For example, with the time delay light, you may want to change the length of time that the light stays on. You could add an argument to the command that allows you to change the delay.

The arguments to a Python program are provided as a list called `sys.argv`. If you only have a simple requirement, you can just access the list directly, as shown in the following example. Enter the following text and save it as `ledtimer2.py`.

```
#!/usr/bin/python3
from gpiozero import LED
import time
import sys

# Time to keep the light on in seconds
DEFAULTDELAY = 30

# GPIO port numbers for the LED
LED_PIN = 4

if ((len(sys.argv) > 1) and (int(sys.argv[1]) > 0)):
    delay = int(sys.argv[1])
else:
    delay = DEFAULTDELAY

led = LED(LED_PIN)

while True:
        input("Press enter to turn the light on for "+str(delay)+" seconds")
        led.on()
        time.sleep(delay)
        led.off()
```

This is based on the LED timer code used in Chapter 4, but there have been a few changes. First, so that this can be used with the disco light circuit rather than needing the original LED timer circuit, I changed `LED_PIN` to reflect one of the LEDs used in the disco light and removed the need for the button instead of using the keyboard. If you still have the LED timer circuit, you can change this to PIN 22 and reinsert the button code.

Instead of using a constant for the delay, I take the value from the argument and store it into a variable called `delay`. Python doesn't actually differentiate between the constant and the variable (both are in fact just standard variables), but the convention is that variables written all in uppercase should not be changed once set to an initial value.

To run the program, first add permissions to execute it using:

```
chmod +x ledtimer2.py
```

You can then run the program using:

```
./ledtimer2.py
```

Let's look at the code in a bit more detail. First I've added the shebang entry so that this can be called as though it's an executable file.

The module `sys` is imported and you can access the `argv` list containing the command-line arguments. Within the `if` statement are two tests. The first is this:

```
len(sys.argv) > 1
```

It checks to see if you have more than one argument. The reason that this is more than one and not zero is because the first argument is the name of the program you are running. Only if that test passes does it perform the following test to see if it's a number:

```
int(sys.argv[1]) > 0
```

This takes the argument number 1 (remember 0 is the name of the program) and tries to convert it to an integer. If it's a positive number greater than zero, it passes the test. If it's not actually a number (including if someone types four rather than the digit 4), it will show an error and the program will stop. It is possible to add error handling using a `try catch` statement, but that's going beyond the scope of this book. Whenever your code reads in a value that the user provides, you should check that is a valid entry.

Assuming that it is a valid entry, the code copies the value into the `delay` variable using the following:

```
delay = int(sys.argv[1])
```

If you don't use a command-line argument, it uses the default from `DEFAULTDELAY` instead.

You then have a `while` loop that turns the LED on for the amount of delay time. Note that I've replaced the check for the button press with the input function, which asks the user to press the Enter key. When you're using the input function, the return value would normally allow the user to enter data that could be provided into a variable, but in this case you just ignore any returned value and wait for the user to press the Enter key.

If you want to include multiple arguments, you could just read them in using the `sys.argv` list. If you want to do something more complex like have different optional arguments that can be in different positions, you may be better off using the `argparse` module.

Running Python Programs as a Service

One issue with the PIR camera code created in Chapter 5 is that it needs to be started manually whenever the computer restarts. This isn't too bad if the Raspberry Pi is connected to a network because you can connect using ssh, but this could be a real problem if it's used for capturing photos of wildlife. For example, you may want to position the PIR camera in a bird box with no network connectivity and no screen and keyboard to login. In that case, it is much easier to have the program start automatically when you power the Raspberry Pi on. This approach can also be useful for any other server type programs. This section looks at setting the Internet of Things train program to run during startup, but this can equally apply for other services.

On recent versions of Raspbian, the startup and running processes are controlled using `systemd`. It provides you a way to start, monitor, and manage background programs. It has a lot of features, which can make it complex, but the part you are going to use is fairly straightforward.

To register the program as a service, you need a special service file stored in the `/etc/systemd/system/` directory; it must end with the suffix `.service`. I called this file `pir-camera.service`. It needs to be created as the root user using `sudo`.

If you're creating it through a graphical screen, you can use:

```
gksudo leafpad /etc/systemd/system/pir-camera.service
```

Or if you prefer to use the command line, you use this:

```
sudo nano /etc/systemd/system/iot-train.service
```

Add the following details:

```
[Unit]
Description=PIR Camera program

[Service]
Type=simple
ExecStart=/usr/bin/python3 /home/pi/learnelectronics/pircamera/pir-camera.py
User=pi

[Install]
WantedBy=default.target
```

- The `Unit` section provides generic information that applies to the service. In this case, it provides a user friendly description of the service. The `Unit` section may also specify if a different service needs to start first.
- The `Service` section provides the options related to this service. The option `Type=simple` is used for a standard program that has not been specifically designed as a server application.
- The `ExecStart` option specifies the command that runs for this service. In this case, it calls the PIR program. It uses the `Python` command first, but if you updated the `pir-camera.py` file to include a shebang entry, as mentioned previously, you can just use the `pir-camera.py` program. It should still have the full path to the program. In this case it would be:

  ```
  ExecStart=/home/pi/learnelectronics/pircamera/pir-camera.py
  ```

- The `User` option provides the username that the program will run under. In this case, the user `pi`.
- The `Install` section includes the `WantedBy=default.target` entry. This indicates that you want the service to start when the computer is running at the default run level.

You can now start the service using this command:

```
sudo systemctl start pir-camera
```

To set it to start automatically, use this:

```
sudo systemctl enable pir-camera
```

You should now be able to reboot the computer and the service will start up automatically.

Running Programs at Regular Intervals with Cron

Another useful feature is being able to run a program at regular intervals. This enables you to run tasks at set times of each day or to run a command the same day each week. The move to the new Systemd scheduling provides another way to achieve this goal, but this section looks at the older and currently better known method of cron.

Cron is a scheduler that can run commands at regular intervals. It's often referred to as crontab, which is name of its configuration file and the tool used to edit the configuration file.

Each user can have her own list of cron tasks to be run at set times. You can edit these tasks using `crontab -e`, which will load the current crontab entries into the default editor. If you have not yet set a default editor, it will ask you which editor to use and will recommend nano. You should edit this file to list any required tasks. Then save the file (Ctrl+O from within nano) and exit (Ctrl+X). When you exit, the crontab will be loaded and will take effect whenever one of the conditions is met.

Here's an example of the crontab entry:

```
# m h  dom mon dow   command
0 9 * * 6,7 /home/pi/ledtimer.py
30 15 * * 1-5 /home/pi/ledtimer.py
0 11 * * * /home/pi/ledtimer.py
```

This shows the relevant part of an example crontab edit session (there are a number of comments above this entry). This example shows the same command being run on different days and times.

Each line reflects a single entry, which has five time/date fields to specify when the commands are run followed by the command to execute. Using the abbreviations, these are (in order):

```
m - Minutes - 0 to 59
h - Hours - 0 to 23
dom - Days of Month - 1 to 31
mon - Month - 1 to 12 or JAN-DEC
dow - Day of week - 1 to 7 or MON-SUN (or 0 can be used for Sunday)
```

The fields can have a single value, comma-separated values, ranges of values, or an asterisk for any value.

To interpret the top entry in the previous example, it will run on the 0 minutes of the ninth hour (9:00 am) on any day of the month and any month of the year, but only on days 6 and 7 (the weekend). The second entry will run at 15:30 on each day of the month and any month of the year, but only on weekdays (1 to 5). The third entry runs at 11:00 every day.

As well as the time-based entries, there are other special commands that can be used, such as `@weekly` (midnight on Sunday morning), `@hourly` (once an hour), and `@reboot` (which runs when the system is started).

Creating an Automated Lego Train Using Infrared

Let's now create another project to practice some of the techniques covered so far. The first is automating a Lego train. The Lego train uses an infrared receiver on the train with an infrared remote control shaped as a controller. The same type of receiver is used on different Lego models, including remote control cars and trains. The Lego train and infrared controller are shown in Figure 6-2.

Figure 6-2. *Lego train with infrared controller*

For this, you are going to need the infrared circuit that you created in Chapter 5, although you only need the transmitter part. This needs only a few parts and so should be easy enough to put back together on the breadboard.

You are also going to add a reed switch so that you know when the train reaches the station. A magnet controls a *reed switch*. It normally opens when there is no magnetic field nearby and closes when an appropriate magnet comes close to it. If you have seen a burglar alarm sensor mounted on a door frame, these use the same principle. A reed switch is shown in Figure 6-3.

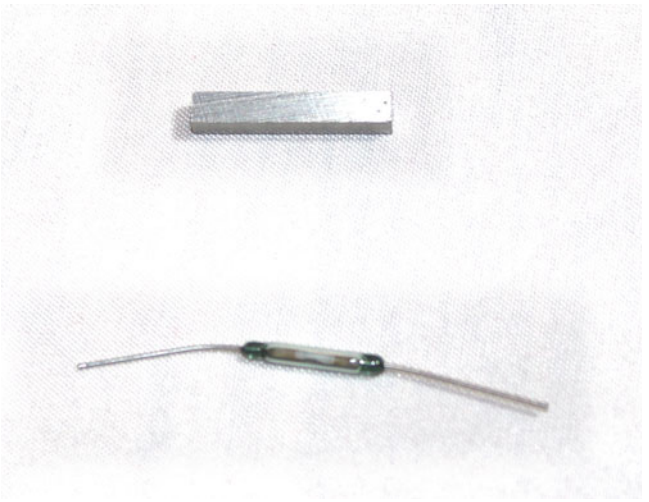


Figure 6-3. *A reed switch with magnet*

You will use the reed switch to pull one of the GPIO input pins down to 0V, the same as you did with the Button GPIO Zero object.

The circuit diagram of the infrared receiver and reed switch is shown in Figure 6-4.

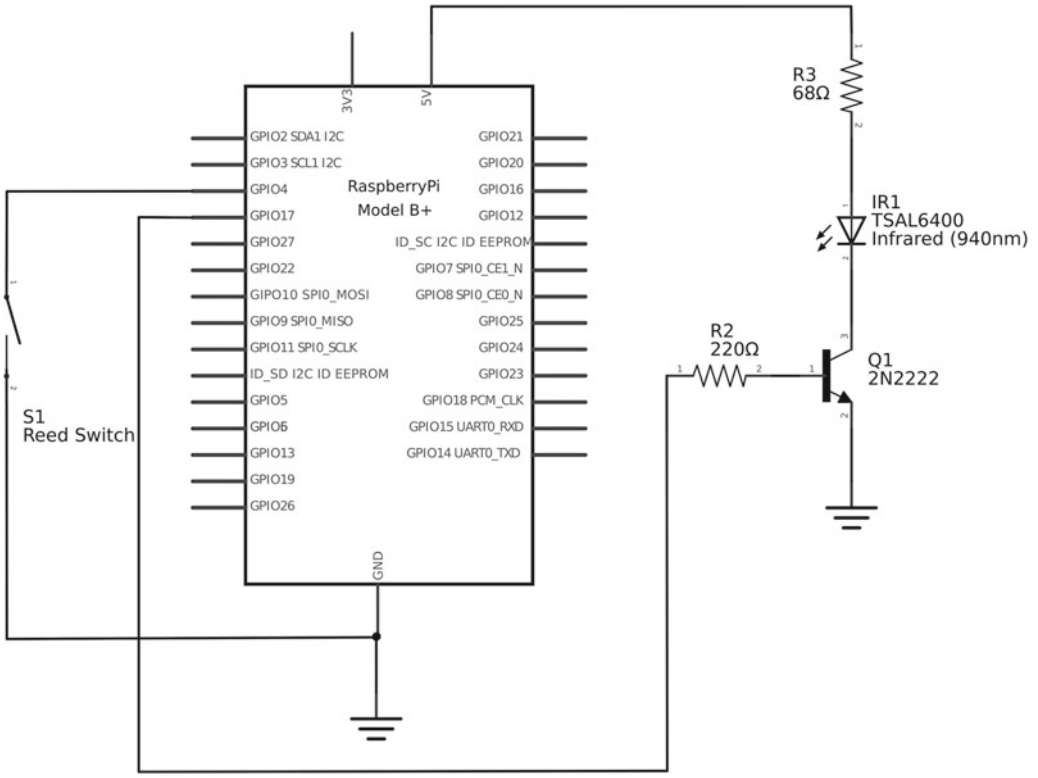


Figure 6-4. *Circuit diagram of infrared transmitter with reed switch*

The infrared transmitter can be made on a breadboard again. I used a long jumper lead to take the infrared emitter diode off the breadboard so that it could be closer to the train. I also found that in certain positions the train could not see the infrared signal. If that is the case, you can add a second infrared emitter diode in series so that two emitters can be pointed at the track from different angles.

The reed switch will need to be positioned inside the train track so that the train goes over it, but so that it doesn't interfere with the train. The reed switch can be soldered to a long wire, as shown in Figure 6-5. (I haven't covered soldering yet, so you could twist the wire against the reed switch lead, which will be sufficient for testing the circuit.) The magnet needs to be stuck to the bottom of the train. In this case I've used Blu-Tack (otherwise known as adhesive putty), but you could use self-adhesive pads instead.

Figure 6-5. *Reed switch mounted under the Lego train track and magnet on the train*

With the circuit in place, you now need to add details about the remote control to LIRC. This assumes that the LIRC has already been configured; if not, refer to the section entitled "Configuring the Infrared Transmitter and Receiver Using LIRC" in Chapter 5.

Although it is possible to teach LIRC the codes using an infrared receiver, these codes are already available on the Internet. To download the codes, enter the following command:

```
wget https://github.com/dspinellis/lego-lirc/archive/master.zip
```

This will download the appropriate files to a file called `master.zip` (it may have a number appended to the end if you already have a file called `master.zip`).

Unzip the contents using:

```
unzip master.zip
```

This will extract into a directory called `lego-lirc-master`.

Now copy the files to the `lircd.conf.d` directory created in the earlier setup:

```
sudo cp ~/lego-lirc-master/Combo* /etc/lirc/lircd.conf.d/
sudo cp ~/lego-lirc-master/Single* /etc/lirc/lircd.conf.d/
```

Add these to the `lircd.conf` file using your preferred text editor. For example:

```
gksudo leafpad /etc/lirc/lircd.conf
```

Add the following lines to the end of the file:

```
include "/etc/lirc/lircd.conf.d/Single_Output"
include "/etc/lirc/lircd.conf.d/Combo_Direct"
include "/etc/lirc/lircd.conf.d/Combo_PWM"
```

Finally, stop and start the `lirc` daemon using:

```
sudo systemctl stop lirc
sudo systemctl start lirc
```

Check that the `lirc` started correctly using:

```
sudo systemctl status lirc
```

Now test that everything is configured correctly using:

```
irsend --count=5 SEND_ONCE LEGO_Single_Output 1R_4
```

This should start the train. Follow it with this command, which should stop it:

```
irsend --count=5 SEND_ONCE LEGO_Single_Output 1R_0
```

If this doesn't work, don't despair. The first thing to do is to make sure that there is line of sight between the infrared transmitter and the Lego train. You may need to take the emitter diode off the breadboard using a pair of male-to-female jumper leads, connect the male end to the breadboard, and push the LED into the female end. Ensure that the diode is connected correctly of course. You may need to have the emitter pointing out toward the track, which can be achieved using a Lego building, such as a train station.

If that still doesn't work, the train may be set for a different channel or using the wrong color channel.

The `irsend` command sends infrared signals, using remote controls that are already configured within `lirc`. It uses several command-line options, which determine which code to send and how to send it. The first option is `--count=5`, which tells the `irsend` command to resend the command five times. This is useful in case the first attempt isn't received correctly. The `SEND_ONCE` argument tells it to send a single set of signals to the device; this is normally required. The alternative to `SEND_ONCE` is to use `SEND_START` to start sending signals and `SEND_STOP` when finished sending signals. This is sometimes used for volume control when you want to keep sending signals for a set period of time. The next option is the name of the remote control, in this case it's `Lego_Single_Output`. These are the codes that can be used to send one action at a time, which is what you will use here. There are also some combination remote control codes that can be used to send multiple commands simultaneously, but the single output is fine this use. The final argument is which particular remote control signal to send (typically the button press on a standard remote control).

This code is made up of three parts. The first part determines which of the Lego channels to use. The Lego receiver and transmitters have a selector that goes from 1 to 4. This is the channel number that you need to use.

The second character is R (red) or B (blue). These are the different positions that the motor can be connected to on the Lego infrared receiver control. On the Lego remote controller, the Red channel is on the left and the Blue channel is on the right.

Finally, there is an underscore character followed by the specific command. The commands used here are 0 to 7, where 0 is stopped and 7 is the fastest speed setting. `BRAKE` effectively applies the brake and brings the train to a stop.

For example, if you have a train on channel 2 connected to the blue side and you wanted it to go full speed, you would use `2B_7`. The complete command would then be:

```
irsend --count=5 SEND_ONCE LEGO_Single_Output 2B_7
```

Dealing with Conflicts

My initial idea was just to use the standard `GPIOZero` module for the reed switch alongside `lirc` for the infrared transmitter. I created the software and set it running. Although the infrared transmitter worked fine, I was unable to see any signals from the reed switch. Perhaps the reed switch was wired up wrong; perhaps the magnet was not coming close enough; or perhaps there was a problem with that part of the code.

I wrote a small test program to find out what was wrong with the reed switch, but when I ran that, it detected the train correctly. The reed switch worked on its own; the infrared circuit and code worked as well, but put them together and there was a problem.

The issue is that different ports are used for the infrared transmitter and the reed switch, yet they use different methods to access the same GPIO port. One conflicts with the other. This is something that you do need to be aware of when creating electronic circuits, particularly when there is some kind of computer involved. This is also one of the reasons that you have to enable certain modules in the Raspberry Pi, as enabling them all by default can cause a problem. Fortunately, I was able to find a workaround for this.

By default, the `GPIOZero` module uses the `RPi.GPIO` module. This is the module that many programmers were using prior to `GPIOZero` and it normally works well. It doesn't appear to work when `lirc` is being used to send infrared codes. I therefore had to change to using `NativePins` for `GPIOZero`, which is a pure Python method of communicating with the GPIO pins. To use `NativePin`, you need to insert the following code d before the `GPIOZero` module is imported (using either `import` or `from`).

■ **Caution** The use of `NativePin` comes with a warning. It is considered experimental and may stop working in the future. Also, while it appears to work with the newer models of the Raspberry Pi, it did not work on all of them. Using a Raspberry Pi 2, it did allow my code to communicate with the reed switch as well as use `lirc` to send infrared signals. With this warning and potential problems, I suggest you avoid using this method for critical applications used in industrial settings, but it's fine for controlling a toy train.

```
from gpiozero.pins.native import NativePin
import gpiozero.devices
# Force the default pin implementation to be NativePin
gpiozero.devices.DefaultPin = NativePin
```

One other down side of the `NativePin` implementation is that it doesn't support PWM (Pulse Width Modulation), although you don't need that here, so it's not an immediate problem.

I did sometimes get a message that the kernel disabled one of the interrupts, but this should not stop the code from controlling the Lego train or from detecting the state of the reed switch. The risk of conflicts is something that you should be aware of, as you could otherwise spend a lot of time looking for errors in the wrong places. A good testing schedule should help catch any errors with conflicts.

Software to Control the Lego Train Using LIRC and GPIO Zero

I created a Python program that starts the train, waits until the train reaches the station, and then slows the train down to a stop. It then pauses before starting again. Rather than just jumping straight to the full speed, I made the train accelerate slowly up to speed and then decelerate slowly back down again. On a short track, the train may trigger the sensor while it's still accelerating, but that is ignored.

The best way to understand this process is to start looking at the code:

```
#!/usr/bin/python3
import os
import time
from gpiozero.pins.native import NativePin
import gpiozero.devices
# Force the default pin implementation to be NativePin
gpiozero.devices.DefaultPin = NativePin
from gpiozero import Button

LEGO_CH = "1"
LEGO_COL = "R"

REED_PIN = 4
ACC_DELAY = 0.5

rswitch = Button(REED_PIN)

MAX_SPEED = 5
STATION_DELAY = 10

def send_lego_cmd (lego_ch, lego_col, op):
        os.system("irsend --count=5 SEND_ONCE LEGO_Single_Output " +lego_ch+lego_
col+"_"+op);

# Go from stop to max speed
def train_speed_up (maxspeed):
    speed = 0
    while speed < maxspeed:
        speed = speed + 1
        send_lego_cmd (LEGO_CH, LEGO_COL, str(speed))
        time.sleep(ACC_DELAY)

def train_slow_down (currentspeed):
    speed = currentspeed
    while speed > 0:
        speed = speed - 1
        send_lego_cmd (LEGO_CH, LEGO_COL, str(speed))
        time.sleep(ACC_DELAY)

while True:
    print ("Leaving the station")
    # Accelerate up to full speed
    train_speed_up(MAX_SPEED)
```

```
    # wait until it triggers reed switch
    print ("Going to station")
    rswitch.wait_for_press()
    print ("Stopping at station")
    train_slow_down(MAX_SPEED)
    time.sleep(STATION_DELAY)
```

I already explained the reason for the extra `import` requirements at the top of the code. First, note that the channel number is entered as a string (with quote marks around it), which is required when you merge it to form the command. The pin number for the reed switch is then included as the acceleration delay, which is how long you wait between each speed increase.

A maximum speed is then set, as while the motor can go up to seven, the train has a tendency to jump off the track at that speed. There is also a station delay, which is how long the train waits at the station before departing again.

The `send_lego_cmd()` function is similar to the remote control function you defined in Chapter 5, but with the code changed so that it is specific to the Lego train. There are two more functions—`train_speed_up()` and `train_slow_down()`—both of which call the `send_lego_cmd()` function.

The main part of the program is contained in the `while` loop, and it calls the functions you already defined and uses the GPIO Zero `Button` object to wait for the reed switch to be triggered. I included some `print` statements, which are once again very useful to understand what the program is doing.

Using the Internet of Things to Control the Model Train

You've no doubt heard about the Internet of Things (IoT), which is all about controlling electronic devices from the Internet. This section looks at using IoT to control the Lego train.

You are going to create your own mini-web server in Python, which is easy thanks to the `Bottle` module. You will also need to use HTML to build the web site and a bit of JavaScript to make it more responsive. Both of these are beyond the scope of this book, so I just concentrate on the Python aspects and how it can send an infrared signal. If you want to take your IoT projects further, it is worth learning more about HTML and JavaScript.

This discussion assumes that you already have the Lego train working with Raspberry Pi, at least as far as being able to use the `irsend` command, as I won't be using the reed switch for this project.

The Python `Bottle` module is effectively an entire web server in a Python module. You simply need to install the module, import it into an application, and then run the appropriate commands.

To get started, you need the module and another file required for the JavaScript. First install the Python module using:

```
sudo apt-get install python3-bottle
```

Now create a directory to hold the program and, inside that, create a public folder that holds the files that you will make available through the web server.

```
mkdir ~/iot-train
mkdir ~/iot-train/public
```

Now change to this directory and download the jQuery file using the following:

```
cd ~/iot-train/public
wget http://code.jquery.com/jquery-2.1.3.min.js
```

Now change to the `iot-train` directory, which is where the program needs to be stored.

```
cd ~/iot-train
```

Rather than rewrite all the code for sending the infrared signals, you will use existing code. To make it so that you can use the code in the file for running as a standalone program and for inclusion in the web application, you use the special Python Main Function code explained earlier. This is shown in the following code, which should be saved as the `legotrain.py` file:

```
#!/usr/bin/python3
import os
import time
from gpiozero.pins.native import NativePin
import gpiozero.devices
# Force the default pin implementation to be NativePin
gpiozero.devices.DefaultPin = NativePin
from gpiozero import Button

LEGO_CH = "1"
LEGO_COL = "R"

REED_PIN = 4
ACC_DELAY = 0.5

rswitch = Button(REED_PIN)

MAX_SPEED = 5
STATION_DELAY = 10

def send_lego_cmd (lego_ch, lego_col, op):

        os.system("irsend --count=5 SEND_ONCE LEGO_Single_Output " +lego_ch+lego_
col+"_"+op);

# Go from stop to max speed
def train_speed_up (maxspeed):
    speed = 0
    while speed < maxspeed:
        speed = speed + 1
        send_lego_cmd (LEGO_CH, LEGO_COL, str(speed))
        time.sleep(ACC_DELAY)

def train_slow_down (currentspeed):
    speed = currentspeed
    while speed > 0:
        speed = speed - 1
        send_lego_cmd (LEGO_CH, LEGO_COL, str(speed))
        time.sleep(ACC_DELAY)

def train_set_speed (speed):
    send_lego_cmd (LEGO_CH, LEGO_COL, str(speed))
```

```
def main() :
    while True:
        print ("Leaving the station")
        # Accelerate up to full speed
        train_speed_up(MAX_SPEED)
        # wait until it trigger reed switch
        print ("Going to station")
        rswitch.wait_for_press()
        print ("Stopping at station")
        train_slow_down(MAX_SPEED)
        time.sleep(STATION_DELAY)

#Run the main function when this program is run
if __name__ == "__main__":
    main()
```

I've also added a new function called `train_set_speed()`, which takes the train straight to that speed without any gradual acceleration or deceleration.

You can give this file executable permission using:

```
chmod +x legotrain.py
```

This will allow you to run it the same as the previous example. The clever thing now is that you can also import it into another program, which enables you to use the functions you want without running the main part of the code. So, to set the speed to three, you can use the following short piece of code.

```
#!/usr/bin/python3
from legotrain import *

train_set_speed(3)
```

As long as this is in the same directory as `legotrain.py`, it will load the existing program as a module and run the `train_set_speed()` function.

The code to run the `Bottle` web server and to handle the appropriate requests should be called `iot-train.py` and be saved in the `/home/pi/iot-train` directory. The code is shown here:

```
#!/usr/bin/python3
from legotrain import *
import sys
from bottle import Bottle, route, request, response, template, static_file

app = Bottle()

# Change IPADDRESS if access is required from another computer
IPADDRESS = 'localhost'
# Where the files are stored
DOCUMENT_ROOT = '/home/pi/iot-train'

# public files
# *** WARNING ANYTHING STORED IN THE PUBLIC FOLDER WILL BE AVAILABLE TO DOWNLOAD
@app.route ('/public/<filename>')
```

```
def server_public (filename):
    return static_file (filename, root=DOCUMENT_ROOT+"/public")

@app.route ('/')
def server_home ():
    return static_file ('index.html', root=DOCUMENT_ROOT+"/public")

@app.route ('/control')
def control_train():
    getvar_dict = request.query.decode()
    speed = int(request.query.speed)
    if (speed >=0 and speed <= 7):
        train_set_speed(speed)
        return 'Speed changed to '+str(speed)
    else:
        return 'Invalid command'

app.run(host=IPADDRESS)
```

First it imports the relevant modules, including the `legotrain.py` file. When importing the bottle module I have included a few functions that we won't actually be using in this program, but they can be useful for other programs so I've left them in.

The app is created as a `Bottle`, which is in effect the web server. Next is the entry `IPADDRESS = 'localhost'`, which sets the IP address that the web server listens on. With this set to localhost, it will be possible to access the server only from a web browser running on the Raspberry Pi. If you want to be able to access this from anywhere on the local network (and depending on your router configuration, potentially the Internet) change this to `0.0.0.0`, which means all addresses.

The `Bottle` module then uses the `@app.route` entries to determine what to do with incoming requests. The first entry says that if you get a request for a file in the `/public` folder, then return a `static_file` (a normal file) from the public directory.

The second `@app.route` entry relates to `/`, which is the web server root directory where you return the default file `index.html`. The final `@app.route` directive is where the real work happens. This says that if an entry begins with `/control`, run the `control_train()` function. That function then gets the value of the `speed` argument from the `get` request and converts it to an integer using the following lines:

```
getvar_dict = request.query.decode()
speed = int(request.query.speed)
```

You then check that it's a valid request by making sure it is a number larger than or equal to 0 and smaller than or equal to 7. This is an important step as you should never accept information provided from a web request unless it has been checked first. If this was not the case, someone could try to insert something inappropriate, which could compromise the security of the web server or the computer.

You can then use the `speed` value to call `train_set_speed()` and send the infrared command to the train.

That function then returns a status message to confirm whether the command was successful. In this case it's just a single line of text, which you can handle in the web browser using JavaScript.

The final step is to start the web server:

```
app.run(host=IPADDRESS)
```

I included the host parameter to change the address; otherwise, it will only listen on the localhost. You can also specify a port parameter, but this example leaves it set at the default port of 8080.

That is the entire web server code. This is in my opinion one of the great things about Python. Thanks to the use of modules that others have created, you can include an entire web server in less than 40 lines of code.

With the web server part complete, you need the HTML code and some JavaScript to send the appropriate requests to the web server. This will all be contained in a single file called `index.html`. The following should be saved in the `/home/pi/iot-train/public` directory as the `index.html` file:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Lego Train Control</title>
<!-- Add Jquery -->
<script type="text/javascript" src="/public/jquery-2.1.3.min.js"></script>
</head>
<body>
<h1>Lego Train Control</h1>

<div id="status">…</div>

<select id="speed">
    <option selected="selected">0</option>
    <option>1</option>
    <option>2</option>
    <option>3</option>
    <option>4</option>
    <option>5</option>
    <option>6</option>
    <option>7</option>
</select>

<script>
// call back function from ajax code
function updateStatus (data) {
    // Update screen with new status
    $('#status').html(data);
}

function changeSpeed (speed) {
    $.get('/control', 'speed='+speed, updateStatus);
}

$( "#speed" ).change(function() {
    changeSpeed($( "#speed" ).val())
});

</script>
</body>
</html>
```

As mentioned, it's beyond the scope of this book to teach HTML and JavaScript as well, but I will still give you a few pointers as to how the code works. You must first load the JavaScript jQuery library using the following:

```
<script type="text/javascript" src="/public/jquery-2.1.3.min.js"></script>
```

This library makes programming in JavaScript much easier and helps you write portable JavaScript that can work across different web browsers.

The next part is very basic HTML, which is not going to look particularly nice (best achieved using CSS) but provides a title, a text position that you will use for status messages, and an option allowing the users to select the speed.

The rest is a block of inline JavaScript. In more advanced programs, this would normally be broken into a separate JavaScript file, but I've just kept it in the HTML file for this example. There are two regular functions—`updatestatus` receives the response from the web server and updates the status text and `changeSpeed` sends a `get` request to the web server.

The last block of JavaScript code overrides the `change` function of the `select` element so that whenever it changes, it calls the `changeSpeed()` function.

To start the server, give it executable permissions using:

```
chmod +x iot-train.py
```

Start the application using:

```
./iot-train.py
```

You will see a status message that informs you which address and port the server is listening on. Whenever a new request is received, it will also show a line of text about what's being requested and the status code.

To access the page, point the web browser at the address of the site followed by `:8080` for the port number. This is shown using the default localhost in Figure 6-6.

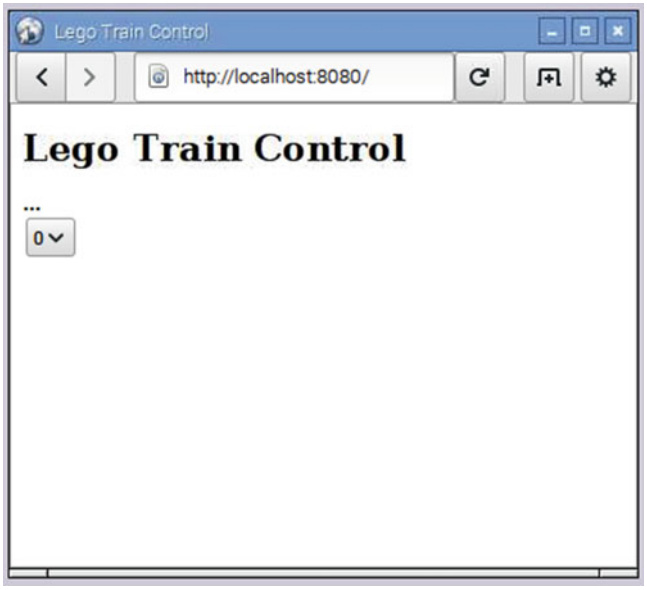


Figure 6-6. *Screenshot of the web page of the IoT train control*

There is no validation or other security checking used in this program. This means that if you allow this on the normal IP address, anyone else on the same network can control your Lego train. For a toy, this might be not a concern, but if you are looking at using this to control something more important, you should add more code to restrict who can use the program.

■ **Caution** If you're using Python Bottle to control something more important than a toy, ensure that you add the appropriate controls so only authorized people have access.

The program works, but it's not particularly great to see or use. You may want to look at improving this program using graphical buttons or Cascading Style Sheets (CSS).

The program can now be set to run automatically by creating a new service file. The service file needs to be stored in the `/etc/systemd/system/` directory and end with the `.service` suffix. I called it `iot-train.service`. The file needs to be created as the root user using `sudo`.

If you're creating it through a graphical screen, you can use:

```
gksudo leafpad /etc/systemd/system/iot-train.service
```

Or if you prefer to use the command line, then you use:

```
sudo nano /etc/systemd/system/iot-train.service
```

Add the following details:

```
[Unit]
Description=IOT Lego Train Control
Wants=network-online.target
After=network-online.target

[Service]
Type=simple
ExecStart=/home/pi/iot-train/iot-train.py
User=pi

[Install]
WantedBy=default.target
```

This is similar to the one you created previously, but also includes the `Wants` and `After` entries. The `Wants` entry ensures that the network is started and the `After` entry means that the code won't start this until after the network is online. It's a good idea to include these with most network applications. If there is a need to wait until another service starts, you need to enter the name of that service in the `After` entry instead.

The service can now be started using this command:

```
sudo systemctl start iot-train
```

To set it to start automatically, use:

```
sudo systemctl enable iot-train
```

You should now be able to reboot the computer and the service will start up automatically. You just need to point your web browser at the page to control the Lego train.

Controlling Color Light Strips Using NeoPixels

Another fun project is using NeoPixel-colored LEDs. NeoPixels is a named used by Adafruit, and these are also known as individually addressable RGB LEDs, or by the codes WS2811, WS2812, or WS28xx.

These LEDs can be connected into a long string. I have a flexible strip, which has 150 of these LEDs mounted, all of which are controlled by a single data connection, plus the power supply. A picture of a slightly shorter strip of these LEDs in action is shown in Figure 6-7.

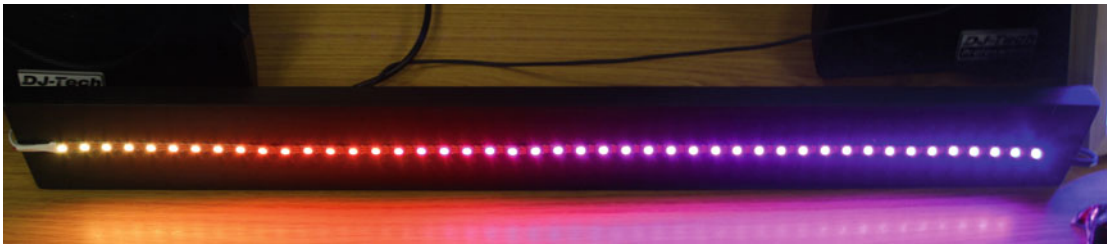


Figure 6-7. *Color light strip using RGB LEDs*

The LEDs are available in a variety of shapes and configurations. These vary from single LEDs designed to be soldered onto a circuit board, to circles, strips, and a matrixes able to create simple pictures.

The circuit that you will create to control these is incredibly simple, but it's the hardware built into the LEDs and the software used to control them that does all the clever work. In fact, you've already come across it in Chapter 5. It's a level-shifter circuit.

The Raspberry Pi GPIO port is 3.3V, but these RGB LEDs work at their brightest when powered from a 5V power supply. You therefore just need to increase the output voltage from the GPIO port to 5V. The basic MOSFET circuit is shown in Figure 6-8.

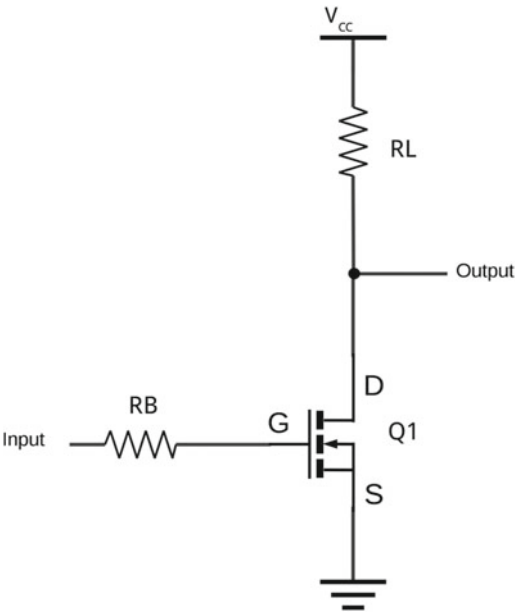


Figure 6-8. *MOSFET level shift circuit for controlling RGB LEDs*

You may recall that this circuit is an inverting buffer, and it gives a low output when the input is high and vice versa. This is something you can resolve by changing a setting in the software. An alternative is to use a 74HCT125 non-inverting buffer IC. This is the approach used by the MyPiFi.net NeoPixel Controller Board, which was a successful Kickstarter project. You could use a level-shifter such as the one used for the I²C LCD display in Chapter 5.

The input to the level-shifter must be connected to GPIO port 18 on the Raspberry Pi, which is the output that supports hardware PWM.

The components I used for this circuit are as follows:

- RL—2.2kΩ
- RB—470Ω
- Q1—2N70000

An example of the circuit using breadboard-mounted NeoPixels in shown in Figure 6-9. This is a good way to get started using NeoPixels. I created different circuits using similar components for controlling the LED strips.

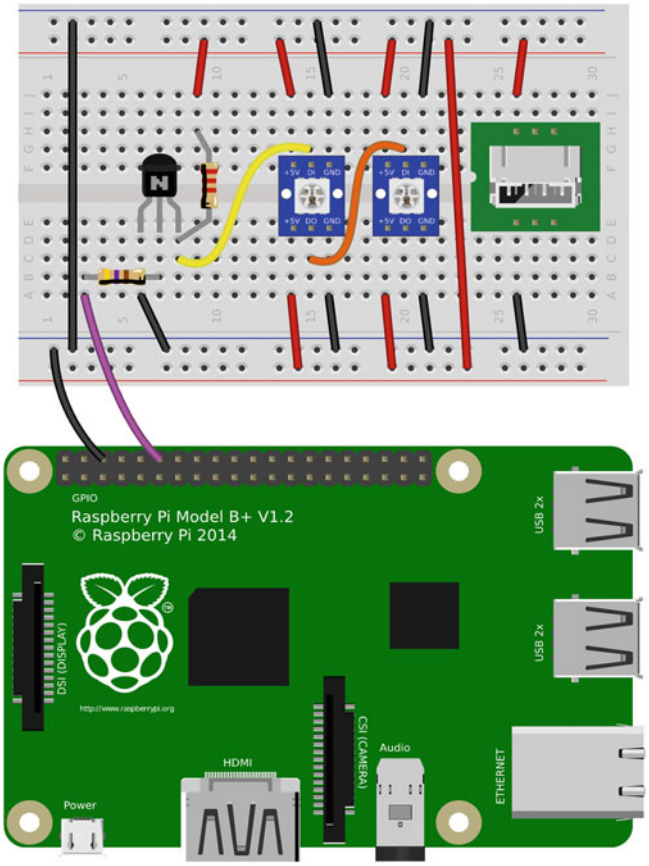


Figure 6-9. *Breadboard version of the NeoPixel circuit*

Note that I used an external power supply in the form of a micro-USB connector, which is the same type of connector used to power the Raspberry Pi. This is not required when using only two LEDs, but a more powerful power supply is required for larger LED strips.

Also note that there are two power supply connections from the same power supply to each of the NeoPixels. This is not actually required either, because there are two +V and two Gnd connections to make daisy-chaining easier. They are included here so that you can see which pins are available for connecting to the power supply. You just need to ensure that one +V and one Gnd connection for each NeoPixel is connected to the 5V power supply.

The data pins are labeled as DI for data in and DO for data out. The signal from the MOSFET connects to the data in of the first NeoPixel. The data out from the first pixel then connects to the data in of the second. Normally, there will be multiple LEDs connected in a series, often on a long LED strip.

■ **Tip** Check that the labels on the pins match those on the diagram. If they don't, the circuit should be adjusted accordingly, as some NeoPixels may have a different pin layout.

Powering RGB LEDs

According to the datasheet for the WS2812, each RGB LED actually consists of three different LEDs in a single package. Each of these LEDs can use up to 20mA when fully lit. So with the color set to white and maximum brightness, you need to be able to provide 60mA per LED. If you have 150 LEDs, this requires a 9A power supply. In reality, I found the LEDs that I used actually ran at around half that, but you do need to ensure that you have a fairly substantial power supply to provide sufficient current to all the LEDs without overheating.

The ground connection of the power supply should be connected to the ground of the Raspberry Pi, but the 5V connection from the power supply should not be directly connected to the Raspberry Pi.

How the RGB LEDs Work

The LEDs are all wired together in a series with the data output from one LED becoming the data input of the next one in the chain.

Each of the LEDs is mounted on top of a built-in controller. This controller IC includes a data input and a data output. The IC receives instructions for all the LEDs on the strip as a long serial string. It strips off the information required from one end to set its own LEDs brightness and then forwards the rest of the information to the next LED in the chain. When the signal reaches the last LED controller, it should only have the information for its own LEDs.

For this to work, the signal needs a very specific timing so that the controller knows when it's the start of a new sequence. To achieve this timing, the Raspberry Pi needs to use the PWM port.

Installing the Python Module

You will be using a version of the Adafruit NeoPixel library that has been modified so that it works on the Raspberry Pi 2 and older models. The Python module is not included in the Raspberry Pi repositories and needs to be compiled from the source code.

The first you need is the developer libraries that allow you to compile the software. These are installed using the normal Raspberry Pi installer:

```
sudo apt-get install build-essential python3-dev git scons swig
```

Next, download the NeoPixel code from GitHub using the `clone` command, which copies all the source code to your local computer.

```
git clone https://github.com/jgarff/rpi_ws281x.git
```

Change to that directory and run `scons` to compile the software.

```
cd rpi_ws281x
scons
```

You then need to change to the Python directory and install the Python module from there:

```
cd python
```

Next, install the Python 3 library file using the following:

```
sudo python3 setup.py install
```

There are some test programs in that directory, but you can also create your own simple test script using Python 3.

Controlling RGB LEDs from Python

You will now use the NeoPixel module to control the RGB LEDs. In this case, I created a simple program to test the circuit and to show how to change the color of the LEDs. As this is designed for the circuit in Figure 6-9, it is written for only two LEDs.

```
#!/usr/bin/python3
from neopixel import *
import time

LEDCOUNT = 2       # Number of LEDs
GPIOPIN = 18
FREQ = 800000
DMA = 5
INVERT = True       # Invert required when using inverting buffer
BRIGHTNESS = 255

strip = Adafruit_NeoPixel(LEDCOUNT, GPIOPIN, FREQ, DMA, INVERT, BRIGHTNESS)
# Initialize the library (must be called once before other functions).
strip.begin()

while True:
    # First LED white
    strip.setPixelColor(0, Color(255,255,255))
    strip.setPixelColor(1, Color(0,0,0))
    strip.show()
    time.sleep(0.5)
    # Second LED white
    strip.setPixelColor(0, Color(0,0,0))
    strip.setPixelColor(1, Color(255,255,255))
```

```
    strip.show()
    time.sleep(1)
    # LEDs Red
    strip.setPixelColor(0, Color(255,0,0))
    strip.setPixelColor(1, Color(255,0,0))
    strip.show()
    time.sleep(0.5)
    # LEDs Green
    strip.setPixelColor(0, Color(0,255,0))
    strip.setPixelColor(1, Color(0,255,0))
    strip.show()
    time.sleep(0.5)
    # LEDs Blue
    strip.setPixelColor(0, Color(0,0,255))
    strip.setPixelColor(1, Color(0,0,255))
    strip.show()
    time.sleep(1)
```

The settings are defined at the top of the file, which among other things, define the number of LEDs and brightness. This code also sets the `invert` option, which counteracts the inverting nature of the MOSFET circuit. These values are passed to `Adafruit_Neopixel` when creating the strip object.

After creating the strip object, the `begin` method needs to be run before setting the color of each LED. The LEDs are considered to be a pixel, so the method to set the color is `setPixelColor`, which takes the pixel number and a color value.

The color value is created using red, green, and blue, with values between 0 and 255. Turning them all on (255, 255, 255) turns the LED white and setting them all to zero (0, 0, 0) turns it off. You then turn only the red, green, and blue parts in turn. Combining these in different amounts can give different colors.

After setting the color for each pixel, `strip.show()` needs to be called to send the update to the strip.

The NeoPixel library needs to have administrator privileges to access the GPIO ports. The program should therefore be run using `sudo`, which runs it under the root user. I called the file `neopixel1.py` so this would be run as:

```
sudo ./neopixel1.py
```

Creating a Graphical Application Using Pygame Zero

The final stage of the LED project is to create a graphical application, also known as a Graphical User Interface (GUI). Many programmers think creating graphical programs is difficult and requires a lot of code. Unfortunately, this reputation is often well deserved, as the very nature of graphical applications means a lot of code to create the graphical windows, to handle mouse clicks, and to keep the program running in the background. Some programming languages are designed to make this easier for beginners, but they don't have access to the hardware that you need for controlling electronics.

Fortunately, Pygame Zero has come to the rescue. In much the same way that GPIO Zero makes programming the GPIO port easier, Pygame Zero is built on top of the Pygame `games` programming module and makes it easier to create a graphical application. It's still based on Python, so you can use it along with all the GPIO Zero code you've already created. Pygame Zero is still quite new (the first release was in 2015), but is showing a lot of promise. It is installed on the Raspberry Pi.

Pygame Zero is useful for creating games, but in this example, I have created a very basic program using rectangular buttons to select one of the sequences. This is designed for controlling a number of LEDs and, while most of it will work with the two LEDs on the breadboard, it is really designed for connecting to a string of LEDs or perhaps to a NeoPixel ring. The basic graphical interface is shown in Figure 6-10.

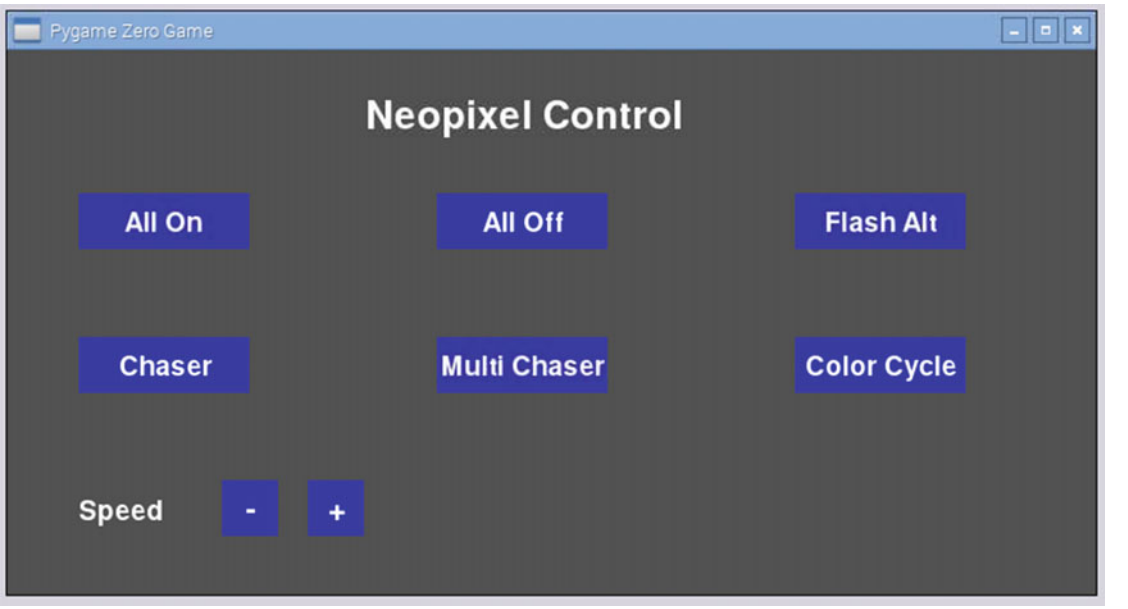


Figure 6-10. *Graphical application for RGB LED control*

A suitable Adafruit NeoPixel ring and strip is shown in Figure 6-11. If you want to use one of these, you need to solder wires on to the appropriate terminals for the 5V power, ground, and data in connections. The soldering is not particularly difficult and you will learn about soldering in Chapter 10.

Figure 6-11. *Adafruit NeoPixel ring and strip*

I saved the code in the neopixel-gui.py file. The code is provided here:

```
#!/usr/bin/pgzrun
from neopixel import *
import time

LEDCOUNT = 10
GPIOPIN = 18
FREQ = 800000
DMA = 5
INVERT = True       # Invert required when using inverting buffer
BRIGHTNESS = 255

WIDTH = 760
HEIGHT = 380

BUTTON_COLOR = 40,40,200
WHITE = 255, 255, 255

buttonText = (
    u"All On",
    u"All Off",
    u"Flash Alt",
    u"Chaser",
    u"Multi Chaser",
    u"Color Cycle"
)
buttonRect = (
    Rect(50, 100, 120, 40),
    Rect(300, 100, 120, 40),
    Rect(550, 100, 120, 40),
    Rect(50, 200, 120, 40),
    Rect(300, 200, 120, 40),
    Rect(550, 200, 120, 40)
)
minusRect = Rect(150, 300, 40, 40)
plusRect = Rect(210, 300, 40, 40)

# Delay counts is number of updates before change in 60th of a second
delay_counts = 30
seq_number = 0
sequence = "All On" # Start with all lights on
timer = 0

# Set up NeoPixel Strip
strip = Adafruit_NeoPixel(LEDCOUNT, GPIOPIN, FREQ, DMA, INVERT, BRIGHTNESS)
# Initialize the library (must be called once before other functions).
strip.begin()

def draw():
```

```
    screen.fill((80,80,80))

    screen.draw.text(
        "Neopixel Control",
        centerx = 360, top = 30,
        fontsize=40,
        color=WHITE
    )

    box = []
    for i in range(len(buttonRect)):
        box.append(buttonRect[i].inflate (-1, -1))
        screen.draw.filled_rect(box[i], BUTTON_COLOR)
        screen.draw.text(
            buttonText[i],
            centerx = box[i][0] + 60, centery = box[i][1] + 20,
            fontsize=28,
            color=WHITE
        )

    screen.draw.text(
        "Speed",
        (50, 310),
        fontsize=28,
        color=WHITE
        )

    boxMinus = minusRect.inflate(-1, -1)
    screen.draw.filled_rect(boxMinus, BUTTON_COLOR)
    screen.draw.text(
        "-",
        centerx = boxMinus[0] + 20, centery = boxMinus[1] + 20,
        fontsize=32,
        color=WHITE
    )

    boxPlus = plusRect.inflate(-1, -1)
    screen.draw.filled_rect(boxPlus, BUTTON_COLOR)
    screen.draw.text(
        "+",
        centerx = boxPlus[0] + 20, centery = boxPlus[1] + 20,
        fontsize=32,
        color=WHITE
    )

def on_mouse_down(button, pos):
    global seq_changed, sequence, delay_counts
    x, y = pos
    # Check position of main buttons
    for i in range(len(buttonRect)):
        if buttonRect[i].collidepoint(x,y) :
```

```
            sequence = buttonText[i]
    # Check position of speed buttons
    if minusRect.collidepoint(x,y) :
        delay_counts = delay_counts + 5
    if plusRect.collidepoint(x,y) :
        delay_counts = delay_counts - 5

def update():
    global timer
    global delay_counts
    global seq_number
    timer = timer +1
    if (timer > delay_counts) :
        seq_number += 1
        updseq ()
        timer = 0

def updseq () :
    global sequence
    if (sequence == "All On"):
        seq_all_on()
    if (sequence == "All Off"):
        seq_all_off()
    if (sequence == "Flash Alt"):
        seq_flash_alt ()
    if (sequence == "Chaser"):
        seq_chaser ()
    if (sequence == "Multi Chaser"):
        seq_multi_chaser ()
    if (sequence == "Color Cycle"):
        seq_color_cycle()

###### Sequences
def seq_all_on():
    for x in range (LEDCOUNT):
        strip.setPixelColor(x, Color(255,255,255))
    strip.show()

def seq_all_off():
    for x in range (LEDCOUNT):
        strip.setPixelColor(x, Color(0,0,0))
    strip.show()

# Uses 2 seq numbers for odd and even
def seq_flash_alt ():
    global seq_number
    if (seq_number > 1):
        seq_number = 0
    colors = [Color(255, 255, 255), Color(0,0,0)]
```

```
    for x in range (LEDCOUNT):
        if (x %2 == 1):
            strip.setPixelColor(x, colors[seq_number])
        else:
            strip.setPixelColor(x, colors[1-seq_number])
    strip.show()

def seq_chaser ():
    global seq_number
    if (seq_number >= LEDCOUNT):
        seq_number = 0
    for x in range (LEDCOUNT):
        strip.setPixelColor(x, Color(0,0,0))
    strip.setPixelColor(seq_number, Color(255,255,255))
    strip.show()

# Needs at least 6 pixels preferably more for this to look correct
def seq_multi_chaser ():
    global seq_number
    if (seq_number >= LEDCOUNT):
        seq_number = 0
    colors = [Color(255, 0, 0), Color(0,255,0), Color(0,0,255)]
    for x in range (LEDCOUNT):
        strip.setPixelColor(x, Color(0,0,0))
    # Set current, one before and one after
    # seq number is always valid
    strip.setPixelColor(seq_number, colors[1])
    # Ensure there is one before - if not put it at the end of the row
    if (seq_number > 0) :
        strip.setPixelColor(seq_number-1, colors[0])
    else:
        strip.setPixelColor(LEDCOUNT-1, colors[0])
    # Ensure there is one after - if not put it at the start of the row
    if (seq_number < LEDCOUNT-1) :
        strip.setPixelColor(seq_number+1, colors[2])
    else:
        strip.setPixelColor(0, colors[2])

    strip.show()

def seq_color_cycle():
    global seq_number
    colors = [Color(248,12,18), Color(255,51,17), Color(255,102,68), \
        Color(254,174,45), Color(208,195,16), Color(105,208,37), \
        Color(18,189,185), Color(68,68,221), Color(59,12,189)]
    if (seq_number >= len(colors)):
        seq_number = 0

    # seq number is used to define the first color then we increment through the colors
    this_color = seq_number
    for x in range(LEDCOUNT):
```

```
        strip.setPixelColor(x, colors[this_color])
        this_color = this_color + 1;
        if (this_color >= len(colors)):
            this_color = 0
    strip.show()
```

Despite this code being much shorter than many other GUI programs, it's still quite long, although some of this is needed for the LED sequences rather than the GUI program itself. I will not go through the code in detail, but will pick out the important things, especially in how they relate to Pygame Zero.

A first thing to note is that the shebang entry does not call the `python3` interpreter, but instead calls `pgzrun`:

```
#!/usr/bin/pgzrun
```

This still uses Python, but running through `pgzrun` will perform the steps required to set up the GUI environment.

You then have the usual NeoPixel configuration details. These are the same as used previously, although the `LEDCOUNT` has been increased to 10. I recommend having an LED strip with at least eight LEDs, such as the strip shown in Figure 6-11, or 16, as shown in the NeoPixel ring.

The next part is the `WIDTH` and `HEIGHT` constants, which determine the size of the graphical window.

```
WIDTH = 760
HEIGHT = 380
```

This is one of the special features of Pygame Zero. These two lines are enough to define the dimensions of the application. In this case, this size was chosen because it fits nicely on the official Raspberry Pi touch screen, as shown in Figure 6-12.

Figure 6-12. *Raspberry Pi touch screen running neopixel-gui application*

Next are two lists: `buttonText` provides the texts for the buttons and `buttonRect` provides a rectangle that will be used for creating the buttons. The `Rect` object is created using `x` and `y` as the first two parameters and then the size along the `x` and `y` axis for the next two parameters. There are then two more rectangles used for buttons to change the speed.

After setting up a few more variables (which will be used as global variables that can be accessed from the other functions), the NeoPixel strip is created and initialized. This is the end of the main part of the code and the rest of the code is all in functions. In a normal Python program, you would need some more code to trigger these functions, but this is handled by Pygame Zero, which calls the `draw()`, `on_mouse_down()`, and `update()` functions as appropriate.

The `draw()` function sets up the layout on the screen. You will see `screen.draw.text()`, which puts some text on the application, and `screen.draw.filled_rect()`, which draws a rectangle for the buttons.

The `on_mouse_down()` function is called whenever the mouse button is pressed. The code checks whether the mouse is over a button at the time (using `collidepoint`) and, if so, it updates the sequence global variable or the speed of the sequence using the `delay_counts` variable.

Pygame Zero calls the `update()` function periodically and it is this function that you'll use to update the sequence. This function is normally called 60 times each second, which is why the `delay_counts` variable uses 60th of a second for each delay count. If the code is within the current cycle, it just increments the timer, but once the timer is larger than `delay_counts`, the code updates the sequence.

The `updseq()` function is not directly related to Pygame Zero; it is instead one I added myself. It is called once each time you update the pixels. It calls the appropriate sequence function, which is in the section after the line:

```
###### Sequences
```

The functions below the sequences update the LED strip based on the sequence number. The `seq_all_on()` and `seq_all_off()` functions are fairly self-explanatory. The `seq_flash_alt()` function uses the modulo operation %. This performs a division operation and provides the remainder as the result of the operation. In this case, by performing a modulo of two, you get a 0 for an even number and a 1 for an odd number.

The chaser functions turn all the LEDs off and then turn the appropriate LEDs on. The final function is a color-cycle function. Rather than calculate the individual colors using sine waves and other mathematical functions, I stored the colors into a list to make this easier to understand.

I saved the file as `neopixel-gui.py`.

Add executable permissions using:

```
chmod +x neopixel-gui.py
```

The program is then launched using:

```
gksudo ./neopixel-gui.py
```

There may be a warning message, as shown in Figure 6-13, which you can dismiss. This is to warn that the program is running as the root user.

Figure 6-13. *Warning message from gksudo*

Assuming there are no problems, the program will run with root permissions. This does, however, mean that if there is a problem, the error messages will not be shown. If you want to see the error messages, first run this:

```
gksudo lxterminal
```

Then, from the new window that opens, run this:

```
./neopixel-gui.py
```

You may notice that most of the sequences are white. This is just to keep the program simple. You need some kind of color selection in order to allow changing colors as well as selecting the sequences.

Adding an Icon to the Raspbian Desktop

You will now add an icon to the Raspbian Desktop to start the program from the Application menu. The latest version of Raspbian includes a menu editor that can be used to add the program to the Start menu.

Launch the Configuration tool from the Preferences menu and Main Menu Editor (shown in Figure 6-14).

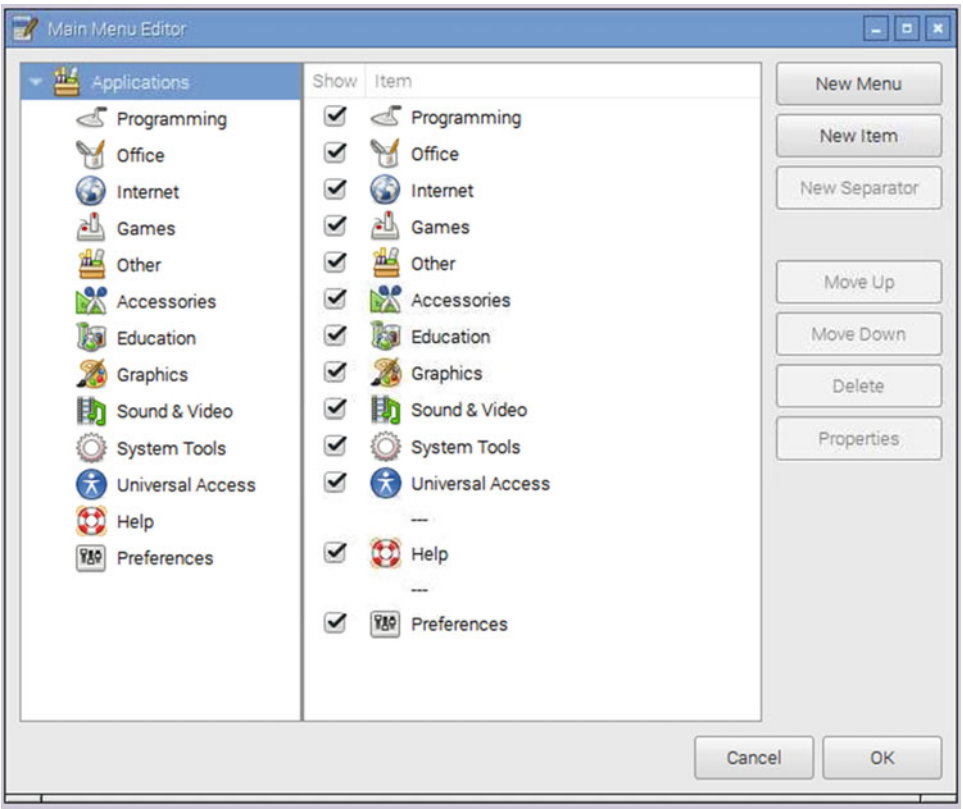


Figure 6-14. *Raspberry Pi Main Menu Editor*

The menu item can be added to one of the existing menus, or you can create a new category using the New Menu button. Some of the menus may not be displayed initially, as they appear only if they have an active item in the menu. After choosing a menu from the left, choose the New Item button on the right. This is shown in Figure 6-15.

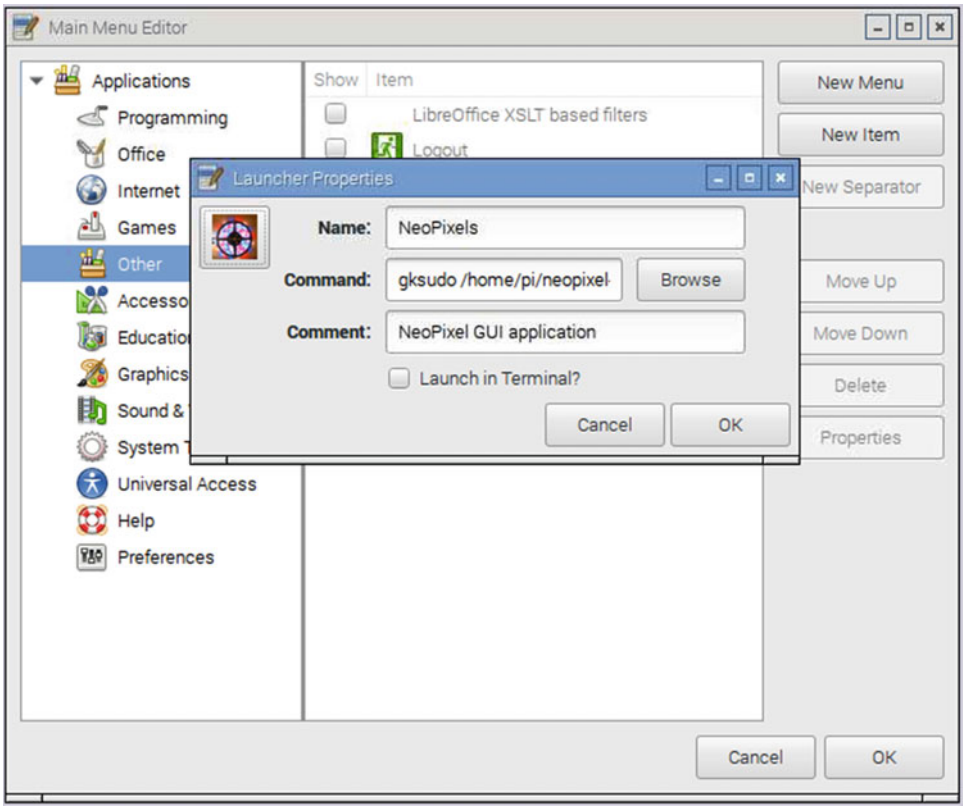


Figure 6-15. *Adding a new application icon in Raspberry Pi*

Give the application a name that is used on the menu. Enter the full command as it would be executed from the terminal, including the `gksudo` command. You can add a comment if you want. To change the icon, click on the box on the left side and then find an appropriate icon image from your computer. A suitably sized `.png` file works well. After you click on the OK, the menu should update to show the new application.

There does appear to be a bug in the Configuration tool. If the new item or menu doesn't show, you can try rebooting. If it still doesn't show up, try removing the menu item using the Delete button and then reading the application using these same steps. Any items added subsequently should then work.

More Linux and Programming

This chapter looked at some aspects of the operating system that you can use with your applications. This includes how to add software, how to register it as a service, having the software start up at boot time, running it at set times, and adding a menu entry for a GUI application.

From the programming side, you learned how to make decisions within your programs and how to create functions to make coding easier and facilitate code reuse. You then went on to create a GUI application using Pygame Zero.

This now gives you the skills to create bigger and more complex programs. You already expanded an existing project by adding more features to the disco lights and used the infrared transmitter to control a Lego train. You also created a new project that incorporated programmable RGB LEDs, which you wrote a simple GUI application for.

This should now give you plenty of opportunity for improving these and other programs. Some suggestions are to use cron to set the infrared light to change colors at different times of the day, add more sequences to the disco and NeoPixel lights, or even write a graphical program to control the Lego train using a touch screen.

In the next chapter, you'll be using the Raspberry Pi camera and taking on the role of director in the next blockbuster movie created using Lego actors.

CHAPTER 7

Creating Video with a Pi Camera

This chapter looks at creating a video using the Raspberry Pi camera. You will control the camera using an infrared transmitter to make it easier to use. This will then be used to create a stop frame animation using Lego characters.

You will start by setting up the Raspberry Pi camera by connecting it to the camera connector directly on the Raspberry Pi. This camera connector has been included on all versions of the Raspberry Pi, except for the Pi Zero.

As a quick recap, the camera needs to be connected by ribbon to the connector located next to the HDMI connector. The camera then needs to be enabled using the Raspberry Pi Configuration tool.

Once it's connected, you can test the camera using the `raspistill` command.

```
raspistill -o photo1.jpg
```

Infrared Shutter Release

You can control the camera from the command line, but when you're creating a stop frame animation, you may not want to or be able to move far from the action. It's easier to use an infrared remote control that can be targeted at the Raspberry Pi infrared receiver whenever a photo is required.

Rather than use a TV remote control (which could end up changing the channel each time I take a photo), I have found an old digital photo frame remote control. If you don't have any spare remote controls, you can use any standard remote control, but preferably in a different room than the appliance it is designed to control. For example, you could use a bedroom TV remote to create a video downstairs or vice versa.

You need an infrared receiver circuit to receive the signal and use it to trigger the camera. If you still have the full transmitter and receiver circuit from Chapter 5, shown in Figure 5-6, you can use that. Alternatively, Figure 7-1 shows a simplified circuit with only the receiver part of the circuit.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_7

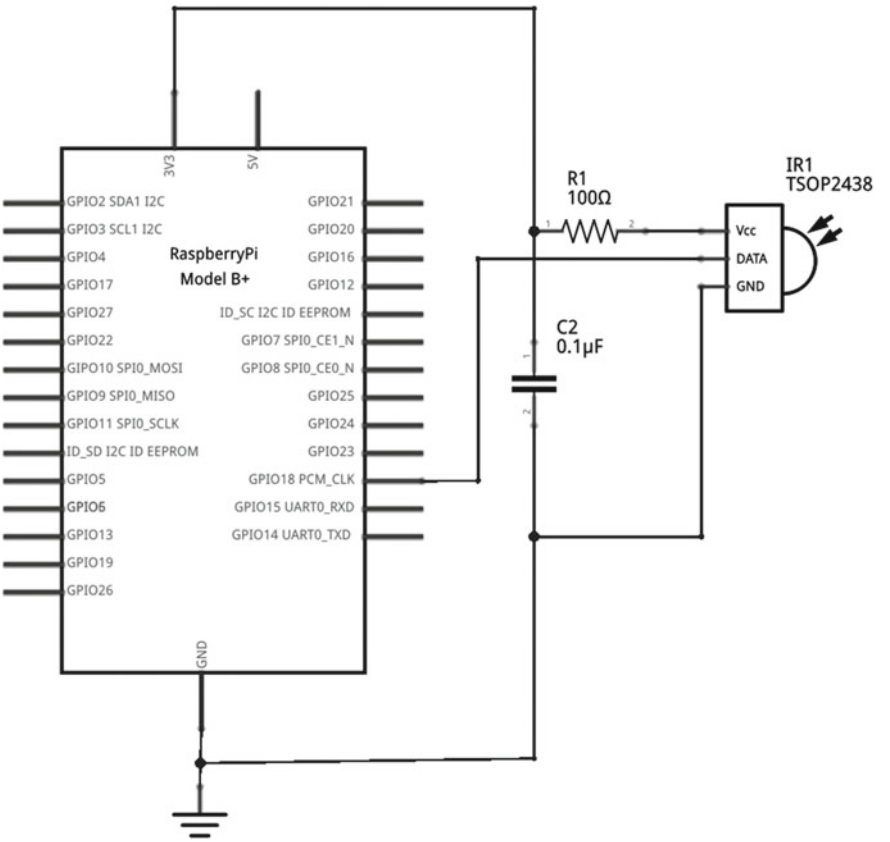


Figure 7-1. *Circuit diagram for the infrared receiver circuit*

This uses the same pin numbers and parts from the previous circuit.

Assuming that `lirc` is still installed as described in Chapter 6, you can now add details of this remote control. The list of remote control codes was created using the same `irrecord` tool used previously, but saving it to a different file.

First stop `lirc` using:

```
sudo systemctl stop lirc
```

Then add the new remote control using:

```
irrecord -d /dev/lirc0 --disable-namespace ~/photoremote
```

The `disable-namespace` option allows you to use any names for the remote control buttons. Follow the prompts to save this to a new file.

I used the name `photoremote` for the file so that I know which remote control this refers to. First edit the name parameter in the `photoremote` file to remove the path information.

The entry should match the following:

```
name  photoremote
```

Then copy the file to the `lircd.conf.d` directory:

```
sudo cp ~/photoremote /etc/lirc/lircd.conf.d/photoremote
```

Edit the `/etc/lirc.conf` file, adding this file alongside the others you already defined. This is done by adding the following line to the end of the file:

```
include "/etc/lirc/lircd.conf.d/photoremote"
```

You should now be able to restart `lircd` using:

```
sudo systemctl stop lirc
sudo systemctl start lirc
```

You can check that it started correctly without any errors using:

```
sudo systemctl status lirc
```

A copy of the `photoremote` file is similar to the one created previously, but with the appropriate buttons for this remote. The following code shows the details, although depending on your remote control, you will most likely have different codes:

```
begin remote

  name  photoremote
  bits           16
  flags SPACE_ENC|CONST_LENGTH
  eps            30
  aeps          100

  header       8942  4524
  one           521  1723
  zero          521   602
  ptrail        520
  repeat       8944  2281
  pre_data_bits  16
  pre_data     0x827D
  gap          107758
  toggle_bit_mask 0x0

      begin codes
          Power                    0x58A7
          Menu                     0xD827
          Up                       0xF20D
          Down                     0xA857
          Left                     0x48B7
          Right                    0xC837
          Enter                    0xCA35
          Random                   0x40BF
          Play                     0x807F
```

```
        Repeat                  0xC03F
        VolumeDown              0x7887
        VolumeUp                0xF807
    end codes

end remote
```

The `lircrc` file will now need to be changed to direct the commands to the video capture program.

Edit (or create) the `/etc/lirc/lircrc` file using a text editor. Remove the previous entries, as they are no longer required, and instead use:

```
begin
    prog = ircamera
    button = Power
    config = Power
    repeat = 0
end

begin
    prog = ircamera
    button = Enter
    config = Enter
    repeat = 0
end
```

I just provided two entries for this. I am using the Enter button for capturing photos and the Power button to exit when I've completed that series of photos.

You will need to restart the `lirc` daemon again using:

```
sudo systemctl stop lirc
sudo systemctl start lirc
```

Assuming the camera is still enabled (if not, see Chapter 5) you need to combine the code from the infrared test code, replacing the relevant part in the camera code. I created this as a new file called `infrared-camera.py`:

```
#!/usr/bin/python3
import picamera, lirc, time, os.path

# Minimum time between captures
DELAY = 0.5

# Directory for photos
photodir = '/home/pi/film';

# Create lirc socket
sockid = lirc.init("ircamera")
```

```
# Create camera object
camera = picamera.PiCamera(resolution=(720,576))
camera.hflip=True
camera.vflip=True

imagenum = 1

while True:
    image_string = u'%04d' % imagenum
    filename = photodir+'/photo-'+image_string+'.jpg'
    # Loop to ensure that filename is unique
    while os.path.isfile(filename):
        imagenum = imagenum + 1
        image_string = u'%04d' % imagenum
        filename = photodir+'/photo-'+image_string+'.jpg'

    camera.start_preview()
    code = lirc.nextcode()
    if (len(code)>0):
        if (code[0] == "Enter"):
            print ("Taking photo " +filename)
            camera.capture(filename)
            time.sleep(DELAY)
            imagenum = imagenum + 1
        elif (code[0] == "Power"):
            break
camera.close()
```

The code uses a sequential number of the files, which makes it easier for combining these into a video. To prevent overwriting an existing file, check that the file doesn't exist before overwriting.

The first part of the code imports the relevant libraries, which have been combined into a single entry. A new one here is `os.path` and it will be used to check if the file already exists.

The `DELAY` constant is now used to create a delay so that holding the remote control button down will not result in lots of photos being taken immediately. Store the files in the `photodir` directory, which you need to create before you run it using:

```
mkdir /home/pi/film
```

It then connects to the infrared socket and creates the camera object using `picamera.PiCamera`. I added a resolution setting to call to the `PiCamera` method, which reduces the resolution of the photos taken by the camera. This will provide smaller files that are easier to work with, but feel free to let it default to a higher resolution if you want a HD quality film.

The mount that I've used holds the camera in the upside down position (with the cable entering from the top), so I've use `hflip` and `vflip` to turn the camera the correct way around. This is required only if the cable to the camera is coming from above.

The `imagenum` variable is used to track the number of the photo. It is initially set to 1, but if there are existing files in the directory, it will be increased.

There there is a loop that will loop through and take a new photo each time the appropriate remote control button is pressed.

The first line in the loop may look a bit unusual at first:

```
image_string = u'%04d' % imagenum
```

The `u` character indicates that you want to format this as a unicode string (a string of text). The `%d` signifies an integer number. Note the `%04d`, which specifies that this is an integer number and will be four digits long. The number 1 will then be rewritten as 0001. The `%` sign indicates that the following are the variables to include in the string. So effectively this will convert the variable number into a four-digit string and store it in `image_string`. You can then use that in your filename, which combines the directory, `photo-`, the formatted number, and the suffix `.jpg` for a JPEG image file.

Next there is another loop that will only run if the filename you have chosen already exists. In that case, the number is incremented and the filename is updated. This will continue to count up until the code finds a filename that doesn't already exist.

Then the `start_preview()` method of the camera shows a preview on the screen. This is so that you can ensure that the camera will take a photo of what you are expecting and give you an opportunity to change the scene. One restriction for the preview is that it will only show on a screen physically attached to the Raspberry Pi (for example through the HDMI port or if using a Raspberry Pi screen connected to the display adapter). It will not be possible to preview the camera through tightvnc or other screen-sharing programs.

It then looks for the appropriate codes from the infrared receiver. Pressing the Enter key will take a photo and increment the number ready for the next photo, or pressing the Power button will close the program. Finally, exiting from the `while` loop by pressing the Power button will result in the `camera.close()` being run, which turns off the camera and cleans up the resources used.

After entering the code, save it as `infrared-camera.py` and make it executable by using:

```
chmod +x infrared-camera.py
```

Then run it using:

```
./infrared-camera.py
```

The program will start by displaying a preview on the screen. Each time the Enter button is pressed a new photo is taken. You can exit using the Power button.

Designing the Film

Now that you have the hardware ready, it's time to focus on the story that it will tell. Professional stop frame animation normally uses expensive flexible models. A good example of this is the *Wallace and Gromit* films, which use models made partly out of clay. If you are good with modeling, then feel free to make your own clay models, but a good way of creating a simple animation on a small budget is to use existing toys such as Lego models. I have used a combination of Lego City and some Lego Friends, and although these are not quite to the same scale they are close enough, especially if you use the Lego Friends model in the background. You can use any other kinds of models or toys, such as action figures, dolls, puppets, or plastic monsters.

I've made some backdrops using photos of places that I've visited and different colored paper and card for the base. You could also look into creating models and landscapes using craft materials.

Before taking the photos, it's a good idea to plan the video. This can be a written story, but it's also a good idea to make a storyboard showing approximately where the characters and props will be. This can be drawn by hand and should provide enough to show the main scenes in your film. A storyboard for my mini-film is shown in Figure 7-2.

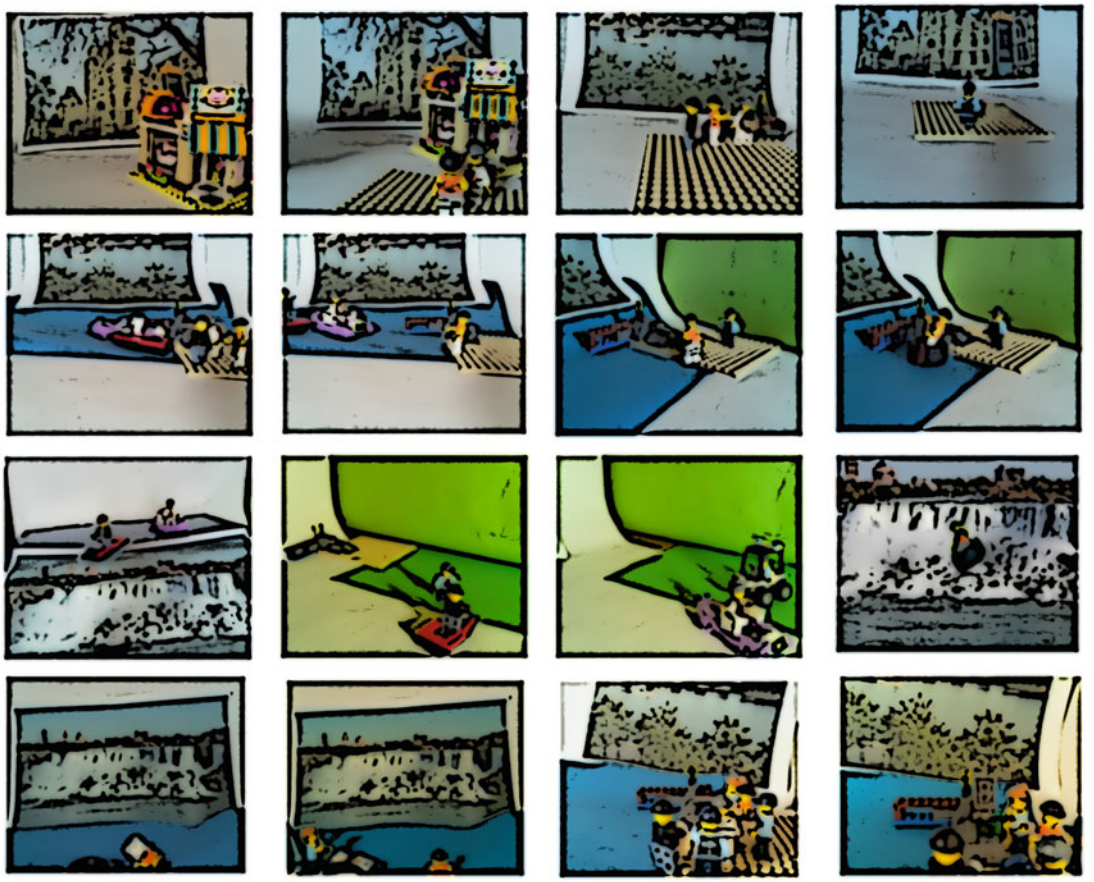


Figure 7-2. *Storyboard for making a film*

The film I am making is about a foiled robbery. In my film, I show the following scenes:

- Starts by showing the cake shop, which is the scene of the crime.
- The thieves run away from the shop with the stolen money.
- They hide the money in a case by the river.
- The police are in pursuit.
- Two of the thieves get into motor boats and speed off.
- There are no boats left for the remaining thief.
- A large tractor tire floats past.
- The thief gets in the tire and paddles away using an oar.
- The thieves come across a waterfall.
- The first thief gives himself up and is arrested.

- The second thief gives himself up and is arrested.
- The third thief goes over the waterfall in the tire.
- He falls out of the tire and is splashing in the water.
- A police officer arrives in a police speed boat and rescues the third thief.
- All the thieves have been arrested. The police officer from the boat finds the box with the money.
- She lifts the lid and takes the money out.
- The end.

Filming the Scenes

To take the photos, you need to create a mini-studio setup. This can be on a table or, if you don't have sufficient space on a table, you can use the floor in the corner of a room. You will need something sturdy to support the backdrops, so a table that is pushed up against a wall is ideal.

You will need the Raspberry Pi connected to a screen so that you can see the preview. The camera should also be mounted on something sturdy. I used a Pimoroni camera stand, which is mounted on top of a mini Gorilla Pod tripod. I also used an extra-long 30cm (12 inch) ribbon cable to attach the camera to the Raspberry Pi, as that provides flexibility. A photo of my setup is shown in Figure 7-3.

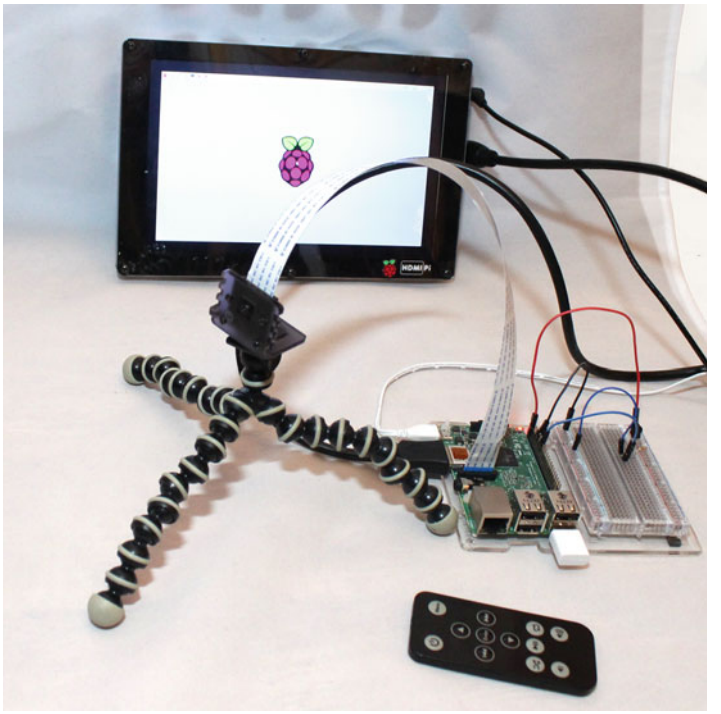


Figure 7-3. *Mini-studio for stop frame animation*

I used a large roll of colored paper for a background and then attached a photo for each scene. My backgrounds were printed on a standard size printer, but if you have access to a larger printer, that would make it easier to fit the entire background across the scene.

Depending on the lighting in the room, you may need to add lighting. Ideally, use a daylight bulb as different bulbs can distort the colors. A standard energy-saving bulb will change the color of the scene, with an LED bulb often making the colors appear bluer than they really are.

With the scene set, you can add your figures and start the program as `infrared-camera.py`. Take your photos by pressing the Enter button on the remote control. There should only be a very small difference in each frame, which is achieved by taking a photo, moving the characters a tiny amount (in my case, moving one leg position at a time), and then taking another photo. The smaller the movement, the more frames can be included per second, and the more realistic the movement. I used a frame rate of around 10 frames per second, which is not particularly smooth but good enough for this video. The standard TV frame rate is normally 24 or 25 frames per second, depending on the country.

Editing the Video

You should now have a series of files starting with `photo-0001.jpg` up to whatever number of photos that you have taken. These still photos can be combined into a video using a script on the Raspberry Pi or by transferring them to another computer first. I will show you how these can be combined into a video on the Raspberry Pi, which is useful if you want to automate the creation of a video, but most likely you will want to transfer this to a PC or laptop, which will provide more flexibility.

This book is about the hardware and software used to capture the photos, and not a guide to video editing, but I will provide some of the basics to get you started and offer some suggestions for special effects.

Creating the Video on a Raspberry Pi

First you will learn how you can combine the photos into a video using the Raspberry Pi.

You will use the `avconv` tool, which can be run from the command line. As its name suggests, it is designed for converting between different video formats, but it also has the ability to take a series of photos and combine them into a video.

First you may want to install some additional codecs. This are not actually required for the basic video creation using `avconv`, but it may be useful to have them installed if you want to do something more advanced. These are installed using:

```
sudo apt-get install libavcodec-extra*
```

Then to install `avconv`, use the following command to install the `libav-tools` package:

```
sudo apt-get install libav-tools
```

You will also need a player to play back your video. I used `vlc`, which can be installed using:

```
sudo apt-get install vlc
```

There are other video players that can be used, such as `mplayer`. If you do install `mplayer`, note that it runs from the command line, whereas `vlc` can be launched from the Applications menu.

Change to the directory containing the photos:

```
cd ~/film
```

Run the following command:

```
avconv -r 10 -i photo-%04d.jpg -qscale 2 video.mp4
```

This command converts the still photos to a video file. It uses a frame rate of 10 and a `qscale` (quality value) of 2, which should give a fairly good quality that can be processed in a reasonable time on a Raspberry Pi. If you have a lot of files, or captured them at a higher resolution, this could take some time to process.

One thing that looks odd at first is the way that the filename is entered `photo-%04d.jpg`. This shows the filename as beginning with `photo-`, followed by four digits (prefixed with zeros), followed by `.jpg`. This ensures that the files are converted using the same numerical order that you used in the Python program.

After this finishes, you can launch `video.mp4` to play the video.

Editing the Video on a PC Using OpenShot

The command-line tools such as `avconv` are okay for automatically combining videos into a sequence, but they don't offer the same flexibility as a non-linear editor. You will therefore take a look at using OpenShot to perform the editing.

OpenShot is primarily written for Linux, but as open source software, it's also available for Apple OS X and Microsoft Windows. You can even run OpenShot on the Raspberry Pi, although, due to the limited memory and performance of the Raspberry Pi, it is very slow and prone to crashing. I recommend trying it only on a Raspberry Pi 2 or 3 and even then only for simple projects.

To install OpenShot on the Raspberry Pi (or Debian/Ubuntu-based Linux), use:

```
sudo apt-get install openshot
```

For OS X or Windows, you can download the program from `www.openshot.org/download/`.

For the following example, I run OpenShot on a laptop running Linux. If you are using a different computer to edit the files, you need to copy the files onto your computer either over the network (using `scp` or similar) or by transferring these using a USB memory stick.

If you have already converted your photos to a video, after launching OpenShot, you can import the video using the Import Files option on the File menu, or by dragging the file into the Project Files area. The video can then be dragged from the Project Files area onto one of the tracks in the timeline area at the bottom of the screen. This is shown in Figure 7-4.

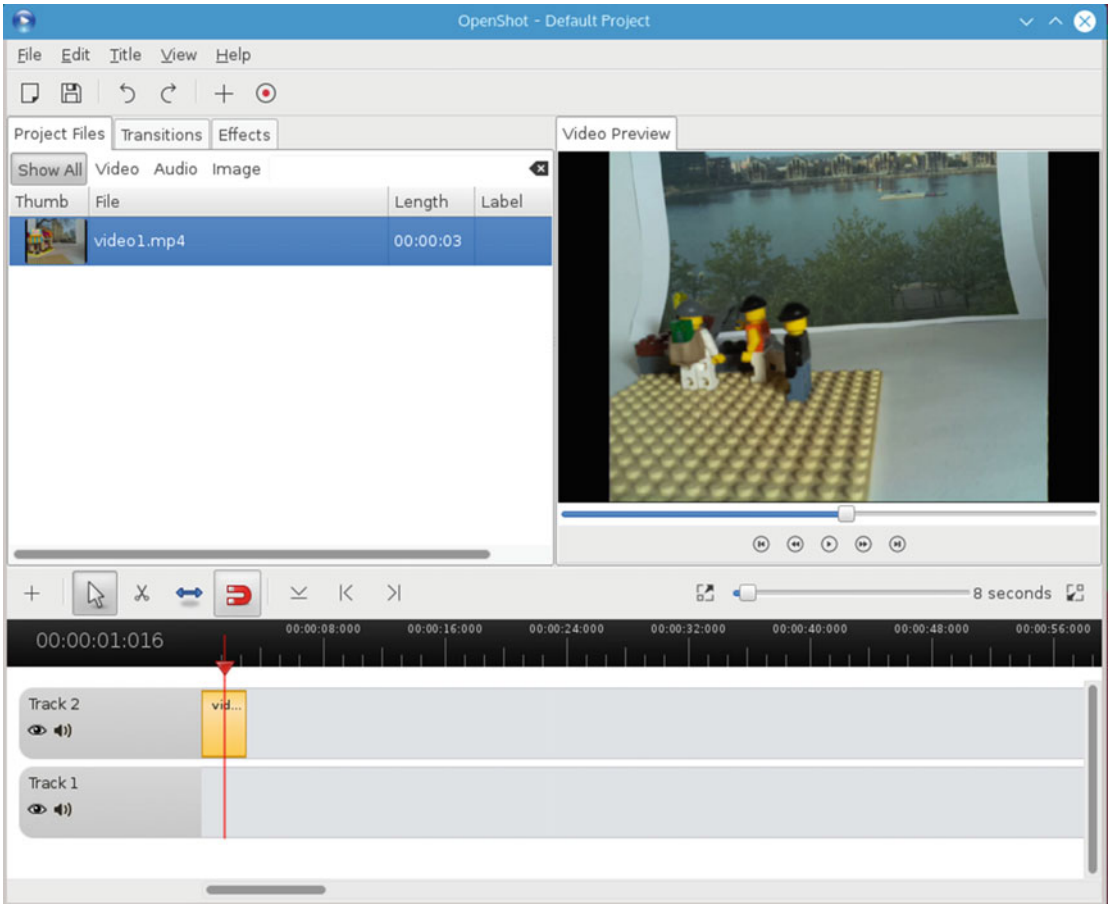


Figure 7-4. *OpenShot with a simple video file*

You can combine this video with other video files or photos and then export it in a suitable format.

If you only have the still photos and haven't yet converted them to a video file, you can import them directly into OpenShot. On the Linux version of OpenShot, this is achieved using the Import Image Sequence option. On other operating systems multiple files can be imported by choosing Import Files ➤ Selecting Multiple Images and then answering Yes from the Import Image Sequence window. On Linux, choosing Import Image Sequence will open the import option shown in Figure 7-5.

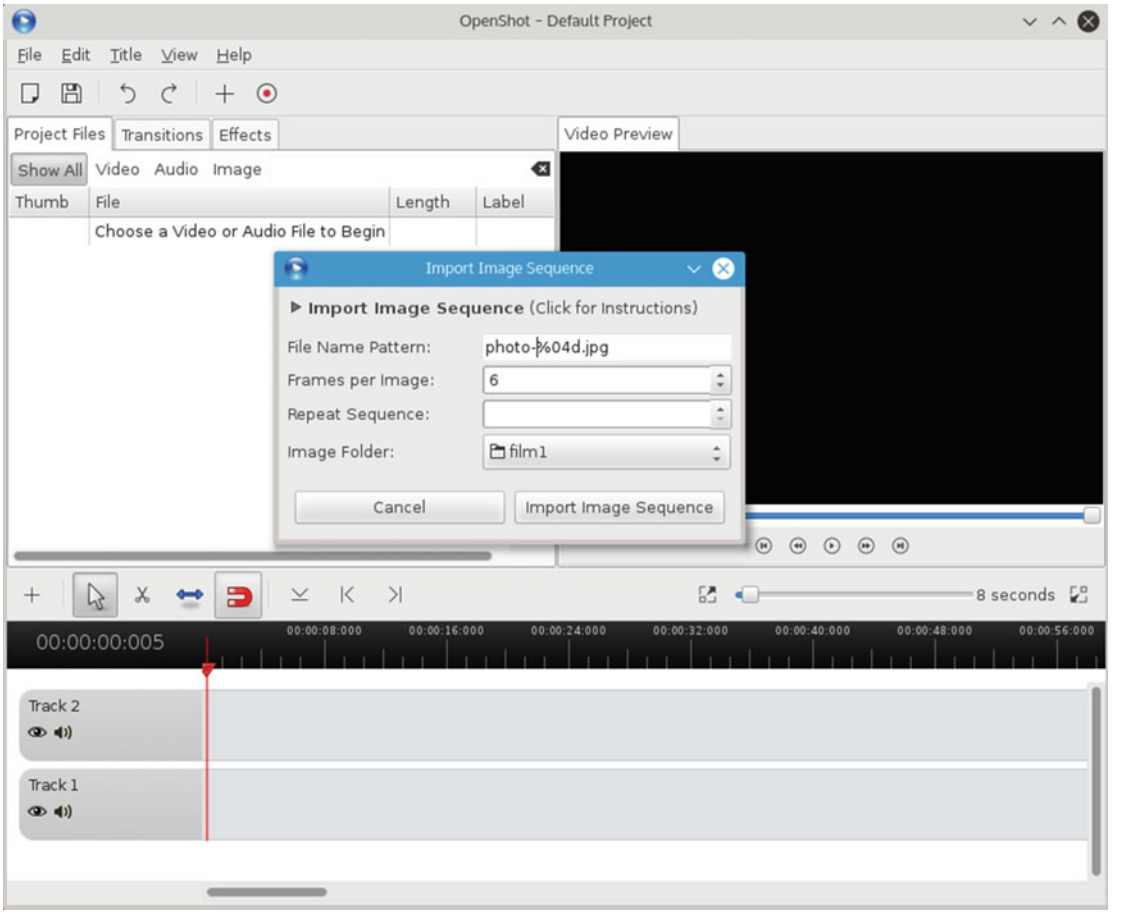


Figure 7-5. *Import a still image sequence into Openshot*

The File Name Pattern field should use a similar file pattern to the one provided with the explanation for `avconv`. This example uses `photo-%04d.jpg`, which looks for files with four digits in the filename. Frames per image is the number of frames that each image will span and this should be set to 2 or 3 to achieve a similar speed to the one you created through `avconv`.

It will then create the video clip from the files and add the clip to the project files in addition to the image files. You will then need to change the frame rate through the video clip's properties.

Adding Effects to the Video

One of the great things about using a computer for video editing is that it is easy to add special effects. You will look at simple special effects using the GIMP video editor and see how you can use green-screen filming to superimpose photos onto a different background.

Adding Special Effects Using GIMP

To add special effects, you need a photo editor. I demonstrate this example using GIMP (GNU Image Manipulation Program), which is available as open source on many computer platforms. On the Raspberry Pi and some Linux distributions, GIMP can be installed using:

```
sudo apt-get install gimp
```

For other platforms, download the program from www.gimp.org/downloads/.

GIMP is a powerful program that is good for editing photos, but as with other powerful programs, it can make it a bit daunting at first. You can of course use a different photo editor if you prefer.

One of the things that can make GIMP difficult for new users is that it launches in multiple windows mode by default. This is shown in Figure 7-6, where there are three windows. The center window is the main editing window where images can be loaded. The left window has the various tools and their options, and the right window provides the Layers and Brushes tools.

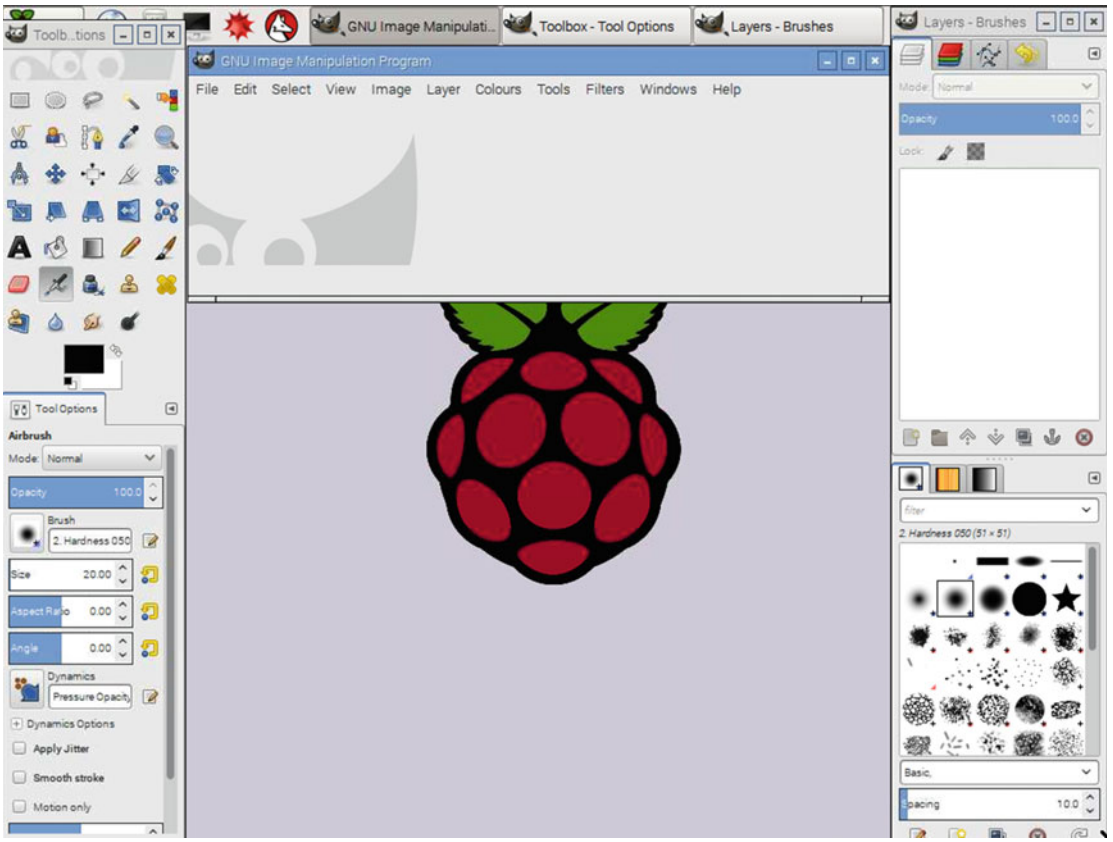


Figure 7-6. *GIMP in multi-window mode*

To change to single-window mode, which for most users will look more familiar, choose Single-Window Mode from the Windows menu. I will be using Single-Window Mode in all the following examples.

Although GIMP is quite usable on the Raspberry Pi, I have done all the editing on my laptop, as that makes it easier to import the edited files directly into OpenShot.

Making Changes to Video Frames

You can now use GIMP to make changes to each of the frames. If you are going for a high frame rate, this can involve making changes to a lot of frames, but it's possible to copy and paste across the images or apply the same image across multiple layers.

For this example, I've made the blue light on the police boat flash. I will just show a small part of the image to make it easier to see.

The picture of the unedited boat is shown in Figure 7-7. This is the frame showing the light when it is off.

Figure 7-7. *Adding a flashing light, unedited image*

I then made the light part of the Lego brighter and added some lines showing the light shining away from the boat, as shown in Figure 7-8.

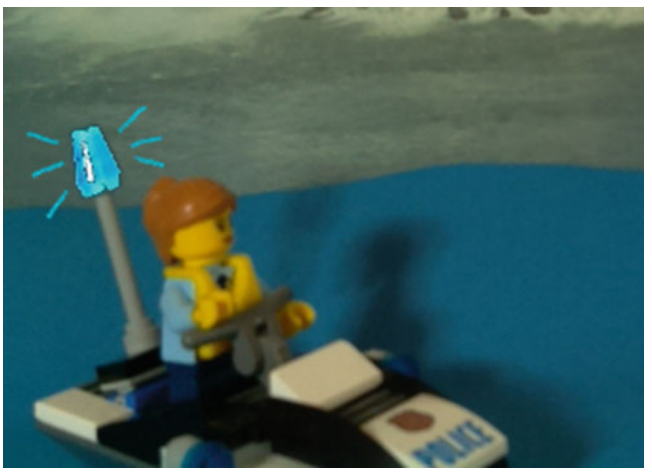


Figure 7-8. *Adding a flashing light, edited image*

The drawing looks basic, but this is in keeping with the Lego animation, which is very blocky. If you were making a more realistic film, you could make it look more natural and might want to add frames with different levels of brightness.

The updated image would be loaded into OpenShot, with alternate frames having the light off in one frame and on in the next. I've also created this as an animated .GIF, which is included in the source code for the book in the `makingvideo` directory.

Using Green Screen Special Effects

A popular technique in professional film production involves using a green screen background, which can be removed and replaced with a different video in post-production. I have referred to this as green screen special effects, although it is officially known as chroma keying or chroma key compositing. Although I've called it green screen, different color backgrounds can be used. The reason for green being used is partly due to technical reasons as green is an easy color for cameras to identify and also because it is a color less likely to conflict with actors features (such as eye color) and clothing.

This technique allows weather forecasters to have a map behind them that can change, for broomsticks to fly across the sky, or for invisibility cloaks to hide the body of a character but leave their head showing. It works by having a single color that is deleted from the film and then having a different video or picture used in the background.

In this case, I used this technique to have a Lego person fall over a waterfall.

First I took pictures of the Lego person against a green background. As he is falling, I used green cotton so that I could lower the figure without the string being visible. One of the photos from this sequence is shown in Figure 7-9.

Figure 7-9. *Using a green screen for the background*

This is then opened in GIMP and the green background is removed. First go to the Colors menu and choose Color to Alpha. Choose the green background for the color. This will identify what color to use as a transparent color channel. This should make the background go partially transparent, but not completely, as there are differences in the color. In film production, they have uniform lighting and better processing, but you can achieve a similar effect using manual selection for the rest of the background. Figure 7-10 shows the Color to Alpha option being used to turn the background invisible.

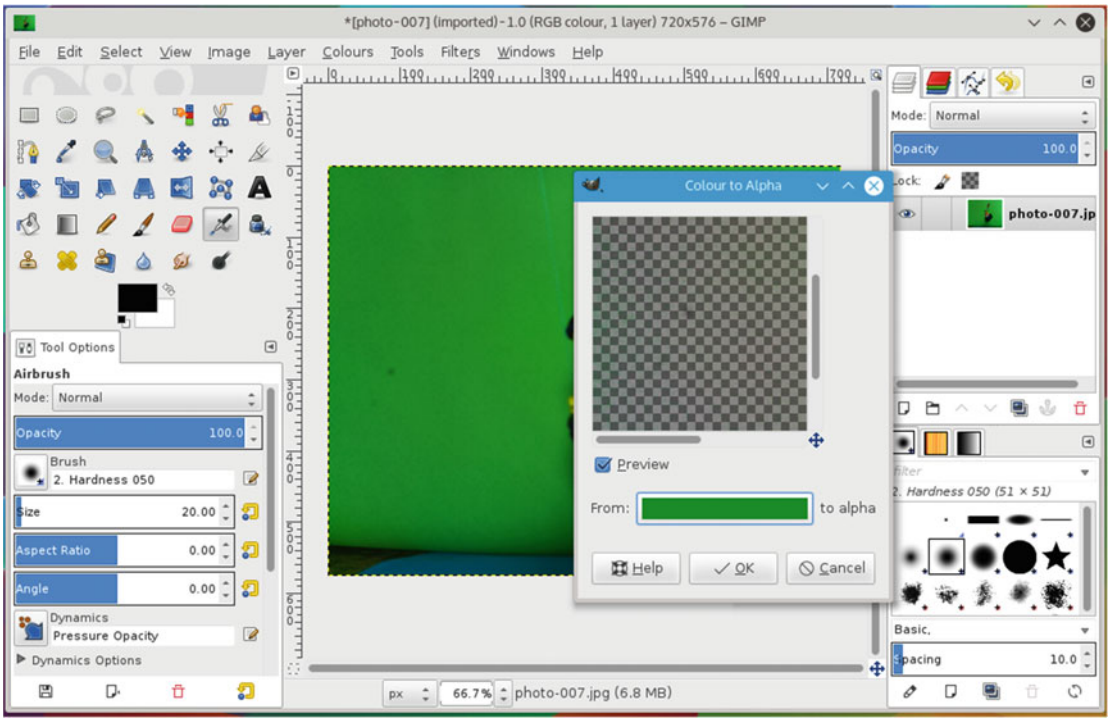


Figure 7-10. *GIMP Color to Alpha tool*

You can remove the rest of the background. Figure 7-11 shows the GIMP toolbox. You'll be using the first four tools on the top row.

Figure 7-11. *GIMP toolbox*

These selection tools are, from left to right, the Rectangle Select tool, the Ellipse Select tool, the Free Select tool, and the Fuzzy Select tool. The Fuzzy Select tool is the most useful for removing a single background color, although the others are useful for selecting specific parts of the picture. Select any remaining green parts of the background and press the Delete key. When it's finished, there should be a checkerboard background indicating the areas that have been removed.

You now need a picture to replace the background. The picture needs to be loaded into GIMP, cropped, and scaled to be the same size as the video footage, which in this case is 720 x 576 pixels. The background image is shown in Figure 7-12, with the Rectangle Select tool set to the correct ratio (see the Rectangle Select box in the bottom-left corner).

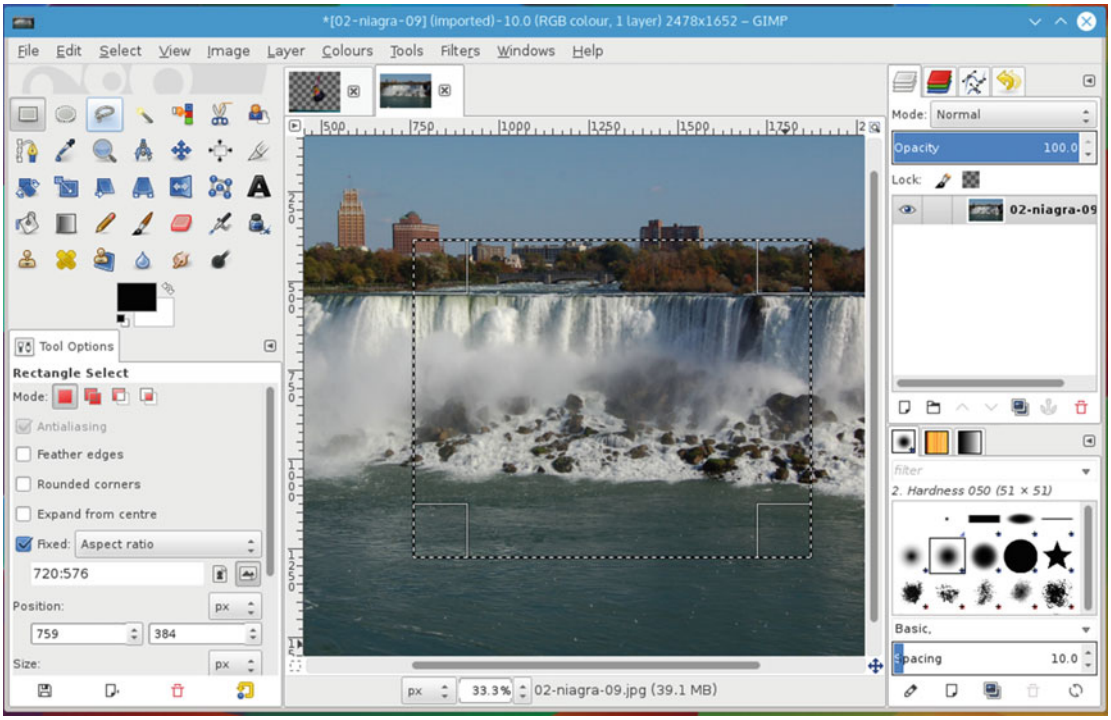


Figure 7-12. *Choosing a background*

Choose Crop to Selection and then Image ➤ Scale Image to resize the background picture.

The first image (with the green background removed) is then copied and pasted as a new layer on top of the background image, as shown in Figure 7-13.

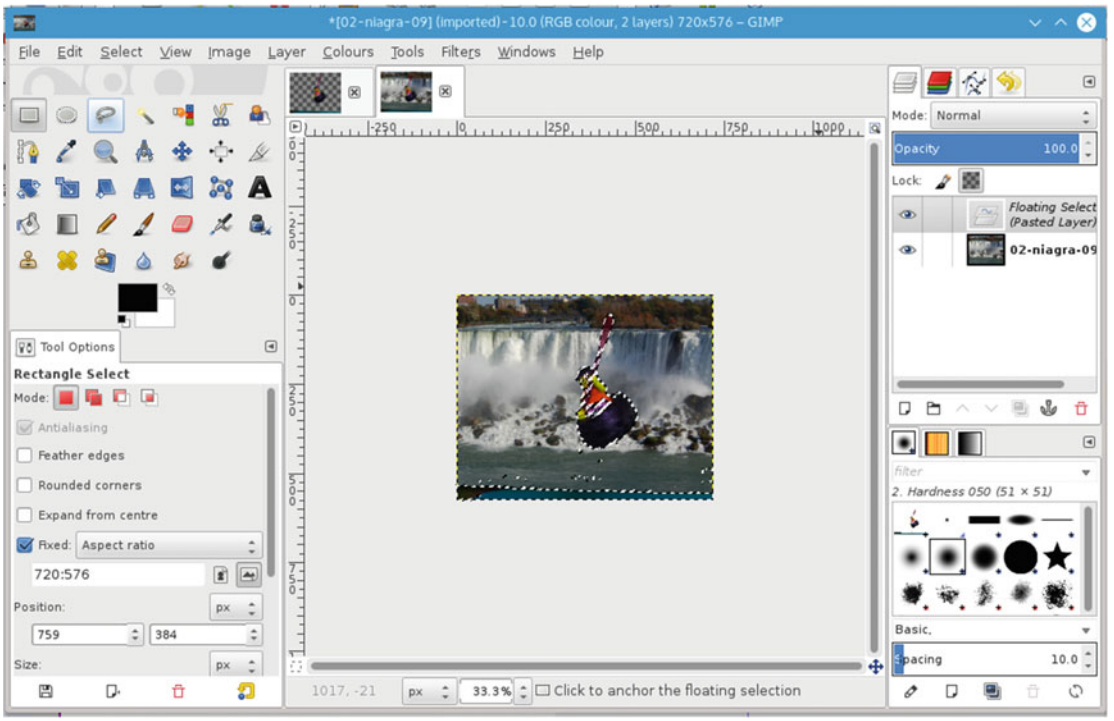


Figure 7-13. *Adding an image over the background*

You can resize and move the new layer if required.

The picture is then exported as a new file, which is the one you will need to include in your video in place of the original frame taken with the Raspberry Pi camera. Repeat these steps for the other images in the sequence.

Adding Sounds to the Video

The video you have created so far does not have any sound, which is fine, but most videos benefit from some kind of sound, whether it's someone talking, sound effects, or background music. Subject to appropriate copyrights, you can find sounds from a variety of different sources and add them to the video using OpenShot. I will briefly mention two techniques for creating your own sound effects and background music.

■ **Tip** If you are interested in finding music that can be included in your videos, consider music licensed under a Creative Commons license. You can find out more about the Creative Commons license and files that are available at `creativecommons.org`.

Recording Sounds with Audacity

I am once again going to use my laptop rather than my Raspberry Pi to record the audio. This is because of the lack of audio input to the Raspberry Pi. It is possible to buy add-on boards or use a USB microphone input if you would rather use the Raspberry Pi.

Although you may already have a sound recording tool on your computer, the one that I normally use is an open source tool called Audacity. It's a powerful tool with lots of features, but is not difficult to use if you stick with the main options.

For Linux, Audacity is normally in the standard repositories. On systems that use the Debian package tools, you can just use:

```
sudo apt-get install audacity
```

For other operating systems, follow the download instructions at www.audacityteam.org/download/.

A screenshot of Audacity in action is shown in Figure 7-14.

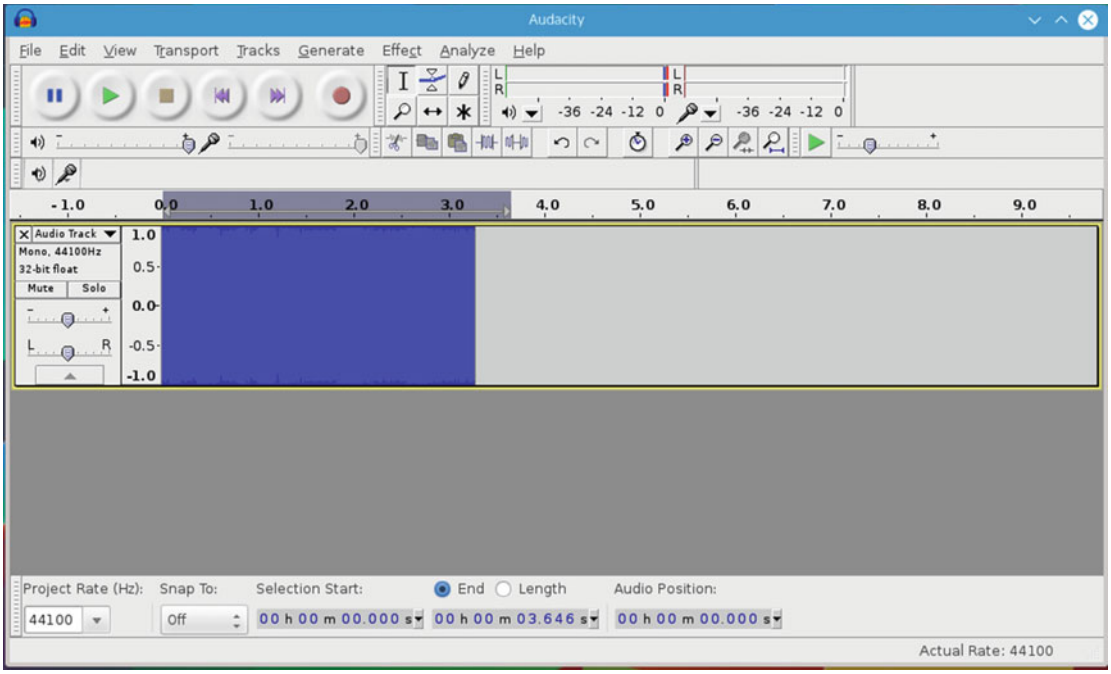


Figure 7-14. *Audacity sound recording and editing tool*

In its simplest form, you click the red circular record button to start recording, the beige square button to stop when you are finished, and then choose File ➤ Export Audio to save the results as a .WAV file.

As well as recording dialogue (talking), you can also create sound effects such as running water, footsteps on gravel, or popping a balloon for the sound of an explosion.

Making Your Own Background Music with Sonic Pi

Sonic Pi is a program for programming sounds. Using a Ruby-based programming language, sound samples can be combined to make great sounds. As well as creating files that can be saved, the music can be changed in real-time, thus providing a live coding experience. If you ever get an opportunity to see live coding in action, it's something I'd recommend.

Sonic Pi is installed by default on the Raspberry Pi Raspbian image. For other operating systems, see `http://sonic-pi.net/`. The Sonic Pi program is shown in Figure 7-15.

Figure 7-15. *Sonic Pi music as a code application*

The program is written in the main part of the screen and the status is shown on the right. To save it as a .WAV file, click the Rec button.

Learning to program for Sonic Pi is beyond the scope of this book, but there are some basic tutorials on the Sonic Pi web site, and a good help system is included in the program.

Adding Sounds to OpenShot

Now that you have the sounds to go with the video, you can import them into OpenShot. This can be added to the Project Files along with the photos and any other video files, and then dragged onto a free track. Figure 7-16 shows the photo sequence on Track 2 with a short audio track called `voiceover.wav` placed on Track 1.

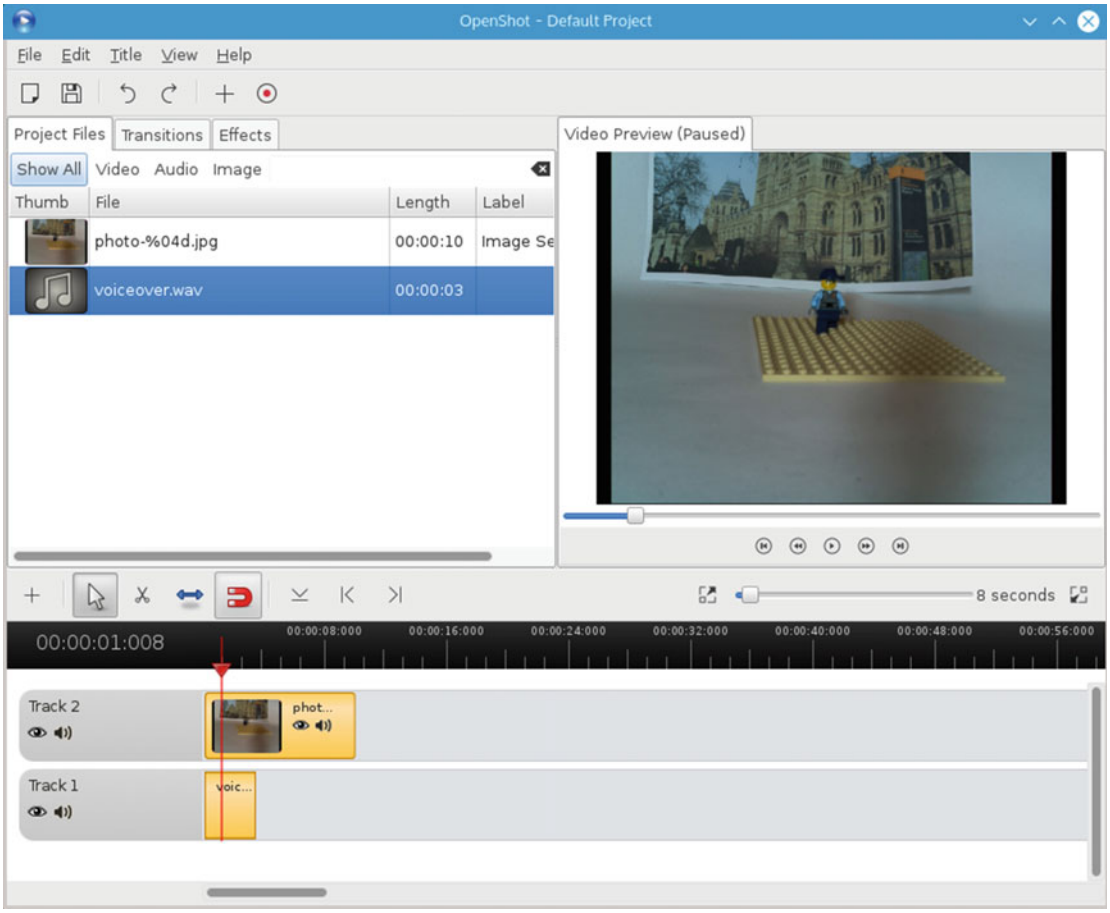


Figure 7-16. *OpenShot with photo sequence on Track 2 and short audio track on Track 1*

Once you've finished adding any photos, music, and sound tracks, you can export it as a video suitable for including on a DVD, or you can upload it onto YouTube or your favorite social media site.

More Video Editing

This chapter provided only a quick overview of some of the things that can be done in terms of creating your own videos. You looked at using infrared to trigger the Raspberry Pi camera and then merging the still photos into a video, using both the command line and OpenShot, which is a visual video editor that can run on a separate PC. You then looked at adding some special effects using GIMP for editing the photos, recording sounds using Audacity, and creating backing tracks with Sonic Pi.

Although these techniques have all been applied to static shots taken with a still camera, these special effects techniques can also be used to apply to live-action footage created using a video camera.

Once you have made your first video, it can be fun to try different techniques and see how they look. You can find that it takes up a lot of time, as hand-editing individual frames can be quite time consuming, but the results can be very rewarding.

In the next chapter, you are going to look at creating your own robot that can be controlled using a web browser.

CHAPTER 8

Rolling Forward: Designing and Building a Robot

This chapter looks at creating a robot vehicle. This process involves selecting and making a chassis, adding motor controls, and writing some software that makes it all work. This is mainly based on one of my first homemade robot vehicles, affectionately named the *Ruby Robot,* although I'll be looking at some alternative chassis and motor controllers. Rather than following it step-by-step, consider this chapter a general guide to how the process works and treat it as an inspiration for creating your own personalized robot to suit your budget, time, and equipment.

Selecting or Making a Robot Chassis

The first thing you are going to need is a chassis on which you can build your robot. There are a variety of robot chassis that can be used, with more coming out all the time. These chassis can vary greatly in cost, from nothing if you recycle junk parts to over $100 for just the base and wheels. This example uses ones at the lower end of the price range.

A robot chassis I have used for some time is the Magician Robot Chassis. It's a two-tier plastic chassis with two powered motors and an omnidirectional third wheel. Although it's been around for a while, it appears to have been recently discontinued. The good news is that there are many other chassis that work in a similar way. The Magician Robot Chassis is shown in Figure 8-1.

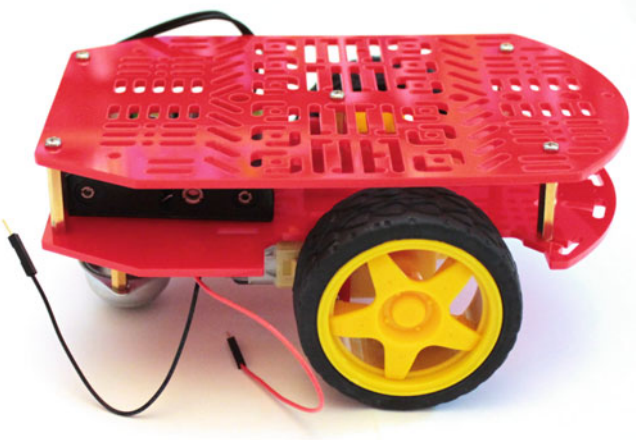


Figure 8-1. *Magician Robot Chassis*

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_8

The main feature that I like about the magician chassis is that it has two tiers, which provides lots of space of adding electronics and batteries in a small space. This is also its bad point, as anything you install on the bottom tier (in my case batteries) is inaccessible except if you remove the top tier. Most of the other similar chassis have a single tier, but tend to be a bit bigger to compensate.

There are different wheel configurations for different robots. I have listed some of these here, along with some advantages and disadvantages of each model.

Two Motorized Wheels and Omnidirectional Wheel

This configuration is used in this chapter. There are typically two motors connected directly to large wheels, with an unpowered omnidirectional wheel (at the front or rear). The omnidirectional wheel often uses a large ball bearing, although this could also be a caster-based wheel.

The main advantages of having two motorized wheels is the cost is typically quite low and the control circuit and software is easier to implement. The main disadvantage is that it's suitable for driving only on flat, smooth surfaces.

Four Motorized Wheels

To overcome the problems with the omnidirectional wheel, it's also possible to have four wheels with one at each corner. This can make negotiating gentle slopes easier, but does require some additional circuitry. It also goes against the principle of car steering, as it means that some of the motors are going against the direction of travel. This makes it more difficult to control.

Caterpillar Tracks

Vehicles with caterpillar tracks can usually negotiate rough terrain more easily. They are often controlled by a motor on each side, so the control circuitry is the same as the two-motor circuit. The main disadvantage is that these tend to be more expensive than the simpler chassis.

Wheels that Steer

Another alternative is to design a robot whose wheels can move similarly to how car wheels move. This is useful if you want to experiment with self-driving cars. This approach requires a complex mechanical mechanism to turn the wheels.

There are other wheel configurations, which are often a variation on the ones discussed here.

Buying a Kit or Making Your Own

There are a variety of robot kits available. I've listed a few examples here, but there are lots of other kits available:

- The Magician Robot provides the chassis and wheels, but no way to control them.
- The CamJam Robotics kit provides the motors and controller circuit, but no chassis (unless you use the supplied box).
- The Ryanteck Robotics kit provides the chassis and motor controllers, and essentially you just need a Raspberry Pi and some batteries.
- PiBorg also provides a variety of comprehensive kits including the chassis, motors, and the controller. These are more robust and as a result are a bit more expensive.

As well as other robot chassis kits that you can buy, you could make your own. If you have access to a laser cutter or 3D printer, you can build a robot to whatever design you can imagine. If you don't have access to that kind of equipment, you could make your own using a sturdy cardboard box or wood. The main criteria is to have somewhere that the motors and wheels can be mounted that won't buckle when used.

Although the CamJam kit doesn't include a chassis, the box that it comes in is quite sturdy and can be used to create a complete robot. The photo in Figure 8-2 shows the CamJam box used as the chassis, with the two motors mounted at one end and an omnidirectional wheel at the other end. This box is big enough to house the motors, a AA battery pack of four, and a Pi Zero with the add-on motor controller board.

Figure 8-2. *CamJam robot kit, with the box used as the robot chassis*

Choosing a Raspberry Pi

At the start of this book, I advised against buying a Raspberry Pi model A/A+. The model A and A+ is a cut-down version of the full Raspberry Pi; it doesn't include wired networking and has fewer USB ports. Normally this is a disadvantage as it likely means that you need to connect to an external USB hub for general computing. It does have some advantages over the bigger model. It is slightly cheaper, uses less power, and is smaller. The power consumption is the main reason that the model A/A+ is probably a more appropriate for use in a robot. The Pi Zero is also good for robots, although you are likely to need an adapter for adding some form of connectivity and the Pi Zero will need a GPIO header soldered on before it can be connected to the motor controller.

■ **Tip** When using a Raspberry Pi with only a single USB connector, you should connect to a USB hub when configuring the wireless connection. Once the wireless connection is established, you can remove the hub and connect through the wireless network.

Controlling the Motors

The main thing you will be looking at is how to control the motors that are used to drive the robot vehicle.

DC Motors and Stepper Motors

Most robots kits use standard DC motors, although if you require more accurate control, you can use stepper motors.

DC motors rely on the magnetic field, which is induced when a current flows through a wire. The wire is wrapped around many times to create a magnetic field that interacts with fixed magnets positioned around the motor. The direction of the electrical current determines the direction of the magnetic field. To reverse the direction of rotation, the motor needs to be connected to the opposite polarity. A DC electric motor will typically spin very fast, much too quickly to drive a wheel directly, so many include an integrated gearbox. The motor in Figure 8-3 shows a DC motor mounted with a configurable gearbox. In this particular motor, it is possible to change the gears to provide a different gear ratio and alter the speed of the drive shaft.

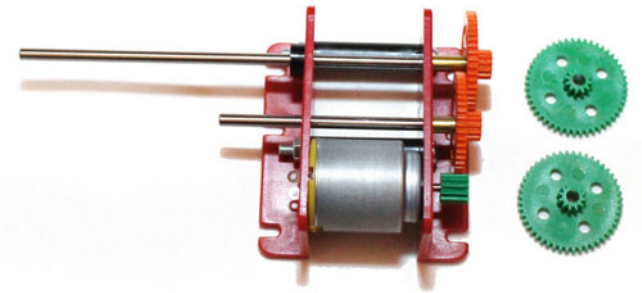


Figure 8-3. *DC motor with adjustable gearbox*

The adjustable gearbox in this motor is useful for understanding how the gears work, but it's a bit too bulky for most robots. In reality, you are more likely to use a motor similar to the one shown in Figure 8-4 (which is from the CamJam robotics kit). This has a fixed-ratio gearbox that is set at the typical speed for a robot vehicle.

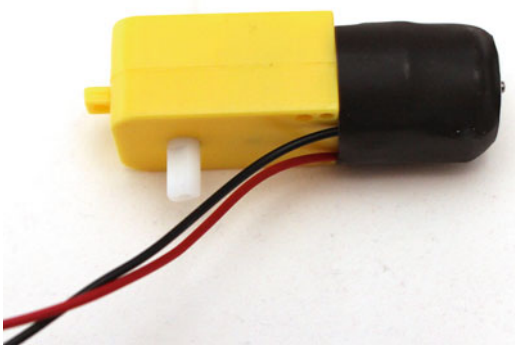


Figure 8-4. *DC motor with fixed ratio gearbox*

One disadvantage of the DC motor is that while it's possible to vary the speed of the motor (which will be explained later), it is difficult to accurately position the vehicle with any degree of accuracy. You could make a guess based on the length of time the motor has been running, or it is possible to add a rotary-encoder, which can approximately measure the rotations of the motor, but it is far from the accuracy that you need to have a robot visit an exact position in a repeatable way.

There is a different motor called a *stepper motor*, which is specifically designed to allow accurate positioning. It uses the same principle of creating a magnetic field using a DC current, but instead of the motor in a spin, the stepper motor just moves along a small amount. The magnetic field then needs to be applied to the next coil, which moves the motor along the next step. If the appropriate magnetic coils can be turned on in the correct order, the motor will turn. The rotation distance is much more accurately controlled than the standard DC motor. Stepper motors are used in a variety of applications, including moving the printhead on a printer and adjusting the focus on a digital camera.

An example stepper motor, known as the Mercury Motor, is shown in Figure 8-5.

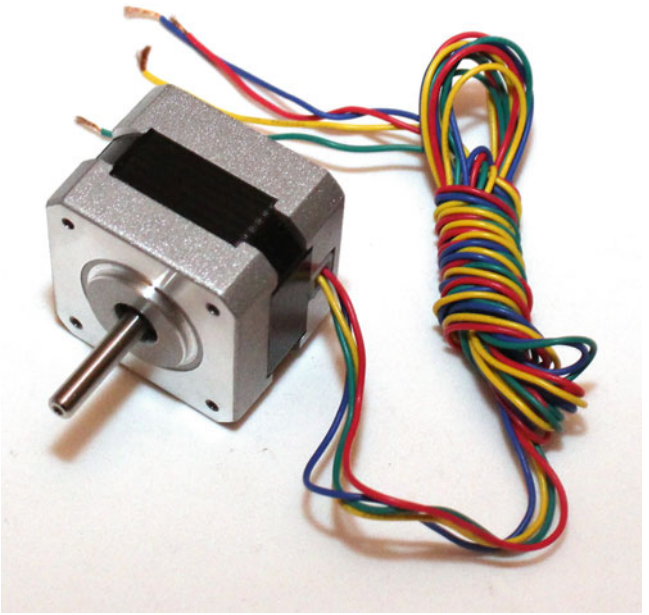


Figure 8-5. *Stepper motor*

This particular stepper is a bipolar stepper motor, which means that it has single pair of windings per phase. In this motor, there are two fields that are connected to four wires: Red (A) and Green (C) for one pair and Yellow (B) and Blue (D) for the other pair. The control circuitry and programming is a little more complicated for a stepper motor, as it needs to apply power to the appropriate coils in order.

Stepper motors are good if you need accurate positioning, although even then, you should be aware that they cannot guarantee against the wheels slipping, which can indicate movement that may not have happened.

For the robot in this chapter, you will use standard DC motors, each with a built-in gearbox.

H-Bridge Motor Control

Before you read about the actual circuit that you will build, it is important to understand how to make the circuit work. As you need to be able to reverse the current across the motor, you need to use an H-bridge motor controller.

The H-bridge configuration is a common way to change the direction of the power supply. The H-bridge is named because it is shaped a little like a letter H and uses two pairs of switches that need to be switched together. It is easiest explained using diagrams.

Figure 8-6 shows the H-bridge controller in the off position. There are four switches, labeled S1 to S4, and all are in the open position. There is no electrical connection to the motor.

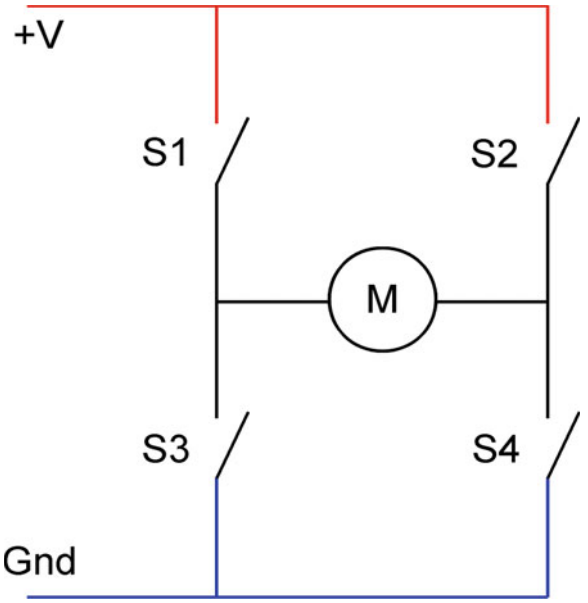


Figure 8-6. *H-bridge motor controller in the off position*

The next setting is shown in Figure 8-7. This shows switches S1 and S4 closed, which allows the current to flow through the motor in one direction.

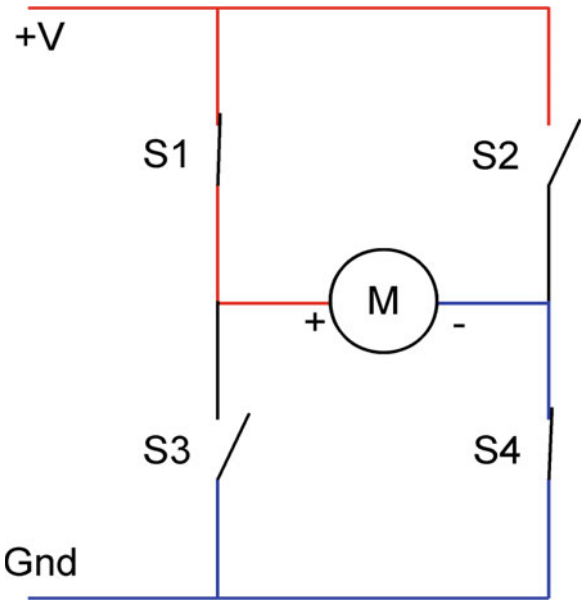


Figure 8-7. *H-bridge motor controller in one direction*

The other configuration is with S1 and S4 open and S2 and S3 closed. This causes the current to travel in the opposite direction across the motor and so the motor moves in the opposite direction. This is shown in Figure 8-8.

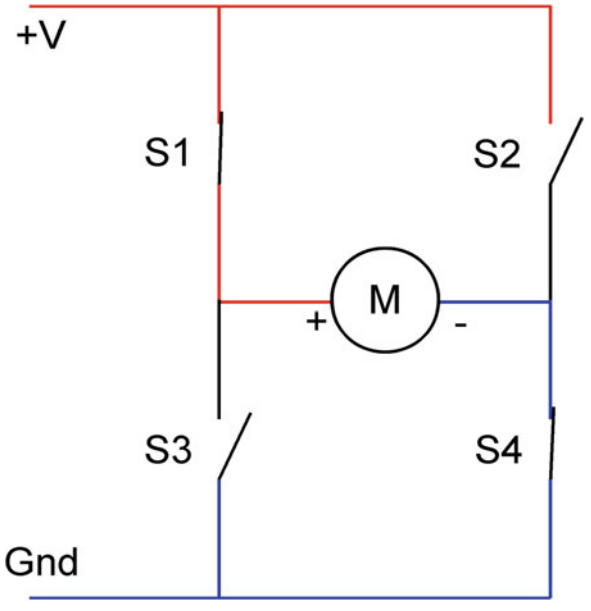


Figure 8-8. *H-bridge motor controller in the opposite direction*

These three are the only valid configuration options. It is important that the two switches on the same side are never turned on together. For example, if S1 and S3 were ever closed together (even for a short time), it would cause a short-circuit, which could cause physical damage to the circuit or power supply.

■ **Caution** Never allow the two switches on the same side to be closed together. If possible, this should be prevented using the physical control circuit.

You can make an H-bridge using MOSFETs and a similar circuit to the one you saw in Chapter 4. The one difference is that you need to use P-channel MOSFETs for those that are on the positive side of the power supply (switches S1 and S2). The P-channel MOSFET is similar to the N-channel MOSFET you have used so far, but works on the voltage at the gate being less than that at the drain to turn it on.

One problem with this approach is the risk of accidentally switching the wrong pairs of MOSFETs, which could create a short-circuit. It is therefore often better to use an integrated circuit, which takes care of all this for you. In this case I am using an SN754410 quad half H-bridge driver IC. As its name suggests, it has four half H-bridge circuits. These are normally grouped into two pairs, which make two full H-bridge drivers, as shown in Figure 8-9.

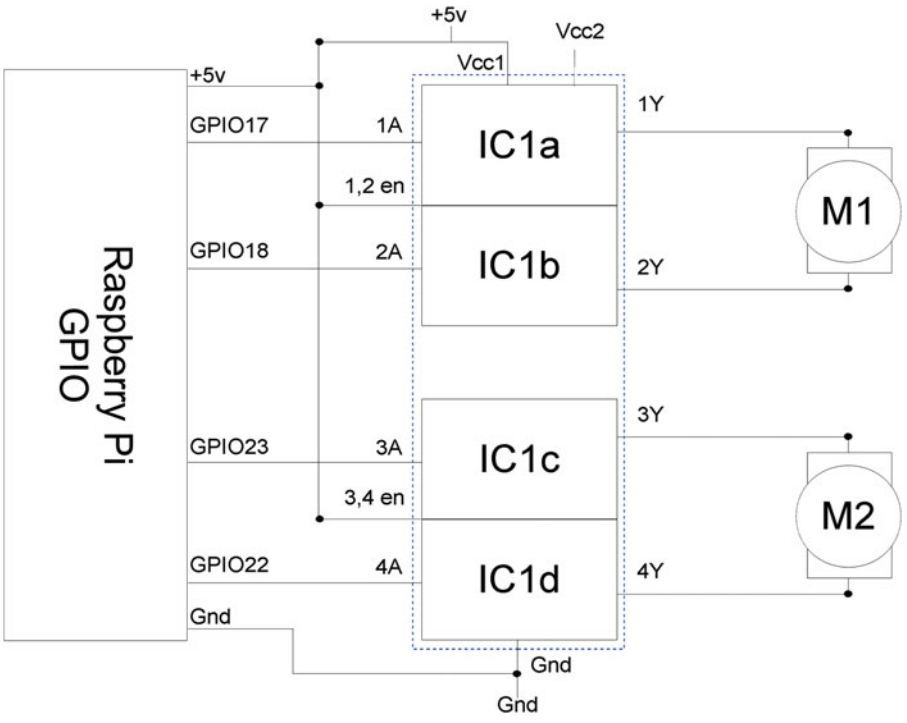


Figure 8-9. *Circuit diagram using SN754410 quad half H-bridge driver IC*

As you can see, there are four half H-bridges in the circuit. These have been grouped such that half H-bridge circuits 1a and 1b form the first full H-bridge and 1c and 1d form the second. In terms of the pin labeling, these are 1A and 2A for the first H-bridge and 3A and 4A for the second. The output is supplied as 1Y and 2Y for the first H-bridge and 3Y and 4Y for the second.

Notice that the diagram in Figure 8-9 is different from the others provided in this book. That is because sometimes it is easier to show a diagram in terms of its functionality rather than the pin layout that Fritzing provides. The Fritzing diagram is shown in Figure 8-10.

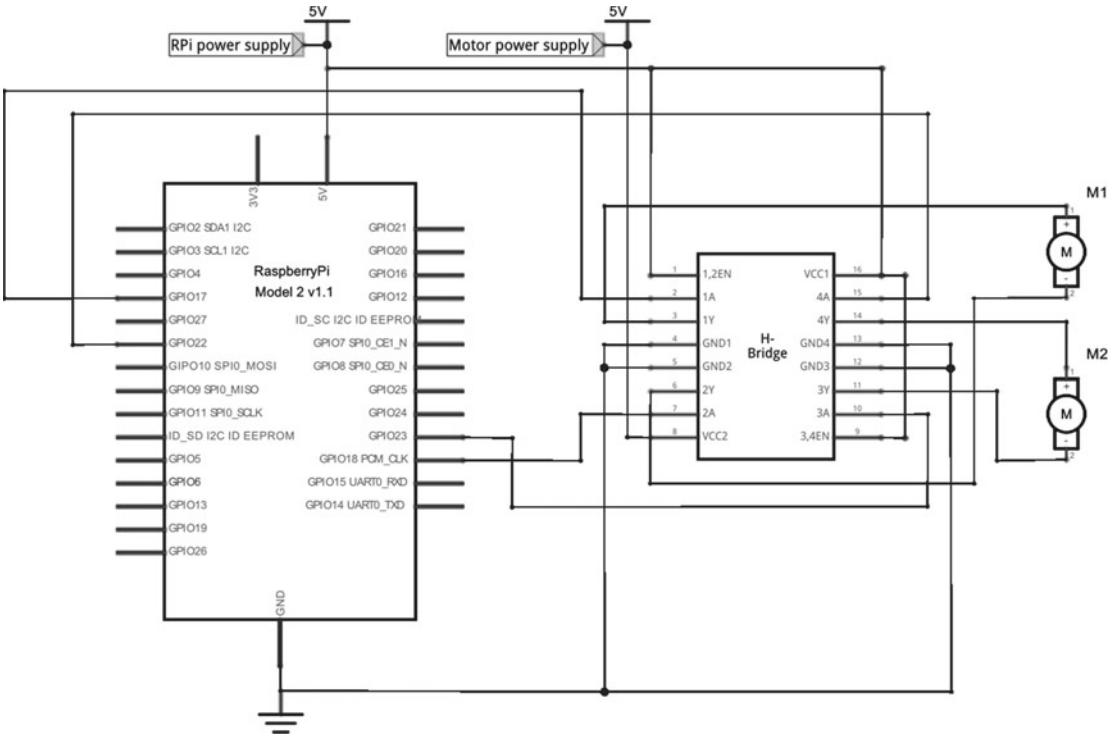


Figure 8-10. *Fritzing circuit diagram using SN754410 quad half H-bridge driver IC*

As you can see, the two diagrams show the same circuit, but the diagram in Figure 8-9 is easier to understand than the more formal style circuit diagram in Figure 8-10.

A few things worth noting on the diagram so far. There are two distinct 5V power supplies shown. This shows that the motors do not have to be connected to the same power supply as is used for the Raspberry Pi and the H-bridge IC. These could be connected to the same supply if you want (which is what I did with my robot), but—due to the noise from driving an inductive load (motors) and the high initial current and potential current draw if the motors stall—there are advantages to keeping these separate.

The enable pins (EN) are connected directly to the 5V power supply. This is one technique and it means that the output of the motor will be enabled whenever the appropriate input is received. With the enable pin set high, the first motor output will be turned on whenever the appropriate inputs are set on 1A and 2A, and likewise for the second motor. An alternative is to connect the enable as another connection to the Raspberry Pi, but as that needs an additional two ports on the GPIO, that has not been used here.

There are no external protection diodes in this circuit. Normally when a magnetic load is used, such as a motor or a relay, as the power is disconnected, there can be a reverse voltage generated from the magnetic device (in effect it becomes an electrical generator). This can be sufficient to damage some electrical components. It is therefore often advisable to include a flyback diode to allow any reverse voltage to be absorbed within the circuit. The SN754410 IC already includes a protection diode in each output, but older versions of the datasheet suggest connecting additional external diodes. The external diode requirement has been dropped on the Texas Instruments 2015 revision of the datasheet and, as this IC only supports small motors, there does not appear to be any problems when they are not included.

Controlling the Speed with Pulse Width Modulation (PWM)

Using an H-bridge motor controller you can change direction, but you also need some way of changing the speed of the motors. For a DC motor this can be achieved by varying the voltage; a higher voltage will allow more current to flow and hence a stronger magnetic field pushes the motor around faster. As with almost all computers, the Raspberry Pi is digital so it can output only in terms of on or off. There are, however, a few techniques that can be used to make it appear to give an analog output with a varying voltage. The one you will use here is known as Pulse Width Modulation (PWM).

PWM works by turning the supply on and off quickly. When connected to an appropriate device, this can be equivalent to providing it with a varying voltage. This doesn't work with all devices, as some devices don't tolerate the constant switching or consider it to be multiple requests. This does work well with motors, as the momentum allows the motor to continue rotating while waiting for the next pulse to push it farther. It also works with LEDs, as they turn on and off so fast that the human eye sees it as a constant, but dimmer, light.

Each of the on states is considered a pulse, and as its name suggests, you vary the width of the pulses to change the equivalent voltage output. This is shown in Figure 8-11, which illustrates the output from the GPIO pin on the top trace and the effective voltage output on the bottom trace.

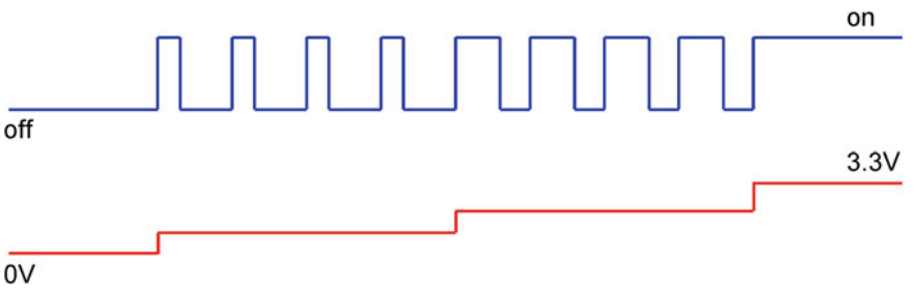


Figure 8-11. *Analog output voltage using PWM*

On this trace, the y-axis represents the output voltage and the x-axis represents time. The time is measured in milliseconds with all this happening very quickly. Starting on the left, it shows the GPIO output completely off which is effectively 0V output. The first four pulses shown have a ratio of 3:7, with the pulse lasting for three-tenths of a cycle, followed by an off period of seven-tenths of a second. Assuming a 3.3V output, this will give you approximately 1V of equivalent output. The next four traces are twice as wide and provide an output for six-tenths of the cycle time, which is an effective output of around 2V. Finally, the last part on the right is fully on, which is 3.3V.

These voltages are based on the output voltage of the Raspberry Pi GPIO port. The voltage supplied to the motor will be relative to the 5V power supply, resulting in 0V, 1.5V, 3V, and 5V, respectively.

The varying output will vary the speed of the motor, but it does not necessarily mean that the relationship between the voltage and the speed of the motor is linear. In particular, there will be a range of low voltages that won't provide enough power to move the motor.

It is possible to provide PWM using both hardware and software. Using hardware, the signal is created using electronics included in the processor, but this can also be achieved using software. In general, hardware PWM is preferred as software PWM makes the CPU work harder. However, earlier versions of the Raspberry Pi had only one PWM port. The newer versions have three hardware PWM ports, although one of those is normally used for the audio driver to generate sound. In practice, unless you need to connect a lot of motors, the impact of performance is acceptable and so I have used software-based PWM.

Powering the Motors and the Raspberry Pi

The circuit diagram in Figure 8-10 shows a separate power supply for the motors and the Raspberry Pi, although it is possible to power these from the same source. There are advantages and disadvantages to both options.

One advantage to having separate power supplies is that it would then be possible to run the motors at a higher voltage. The Raspberry Pi needs a stable 5V power supply. The motors are actually designed for up to 6V and, with the losses in the H-Bridge, you could go up to 7V. Although this works well with the 5V supply, you could potentially get a little more speed out of the motors if you use an extra AA battery or a different power supply. Another advantage to a separate power supply is that motors can introduce electrical noise, which can cause problems with sensitive electronics such as the Raspberry Pi.

The advantages to using a single power supply are the cost, space, and the convenience of not needing to charge/replace two different sets of batteries. I decided to use a single power supply for the Raspberry Pi and the motors. If you do come across power-related problems with the robot, you may want to consider different power supplies.

I used four AA batteries to power the Raspberry Pi and the motors. High-capacity NiMH rechargeable batteries have a nominal voltage of 1.2V, which gives 4.8V. This is just enough to power the Raspberry Pi. It does mean that if the voltage dips a little when the motors are in use, the supply voltage to the Raspberry Pi may dip. Using high-capacity rechargeable batteries appears to work well, but when I tried cheaper batteries, I did find that during a motor stall (when the robot crashed), the power would drop and the wireless network connection would disconnect.

I used a 2.5mm DC socket-to-terminal connector to connect the supplied battery holder to a breadboard used for the electronic circuit. I then connected from the output of the external power supply through a micro-USB cable into the Raspberry Pi. An alternative is that you could supply power from the breadboard into the GPIO port through one of the 5V ports on the GPIO connector.

■ **Caution** Be careful if you're connecting power through the GPIO port, as it bypasses the polyfuse used to protect the power coming from the micro-USB port. Using four AA batteries should work fine with this setup, but consider additional protection if you're using different batteries.

Figure 8-12 shows a micro-USB connector that can be used to take power from a terminal connector to the Raspberry Pi power supply. There are four wires in a USB cable; it is only the power wires that are required and are normally colored red (+5V) and black.

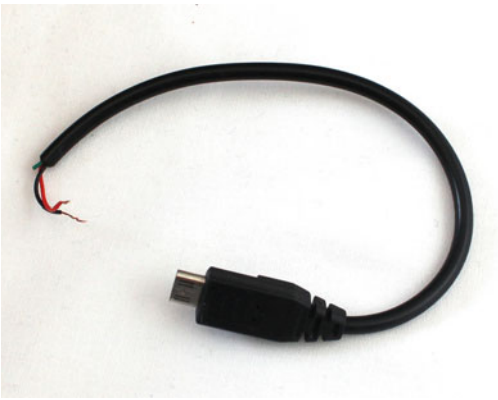


Figure 8-12. *Micro-USB connector with bare wires for connecting to a power supply*

Building the Circuit on a Breadboard

In my first version of the robot, I created the circuit on a breadboard so that I could test that it worked as expected. The breadboard layout is shown in Figure 8-13. The connector shown in the diagram is a 26-way Raspberry Pi cobbler, which makes it easier to connect between the Raspberry Pi and the breadboard using a ribbon cable. This can be replaced with long male-to-female jumpers connecting from the Raspberry Pi to the breadboard. It is also possible to get 36-way cobblers for the newer models, but that would require a bigger breadboard because it would take up the whole of the breadboard.

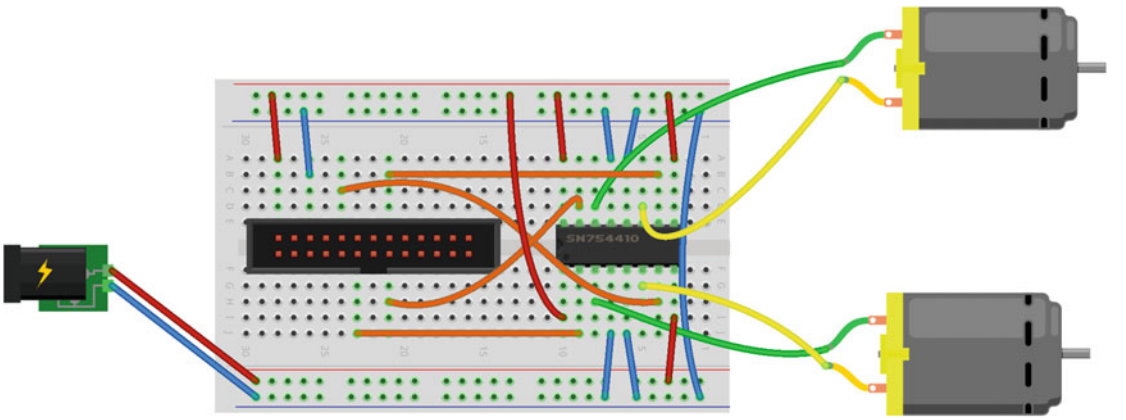


Figure 8-13. *Robot motor control circuit on a breadboard*

One thing to be aware of is that the position of the motor wires may need to change to ensure that they are going in the right direction. This will depend on the motors and the direction that they are facing when mounted inside the robot. It is also possible to change this around in the code, although that depends on the way the motors are defined.

Add-On Motor Controller Boards

Creating the circuit on a breadboard is a good way to learn how the electronics work, but it's more convenient to have it on a printed circuit board that can be mounted directly onto the Raspberry Pi. There are a number of these available, but this section looks only at two: the Ryanteck Motor Control Board and the one supplied in the CamJam Robotics Kit, made by 4tronix.

The Ryanteck board is available as part of a complete robotics kit or as just the motor controller. The controller board is normally supplied as a PCB and components that need to be soldered together, although you may also be able to buy one pre-soldered. The main appeal of this controller board is that it is identical to the breadboard circuit used previously. So the same code can be used for the robot whether it is connected to the breadboard circuit or the Ryanteck board.

I originally created the breadboard circuit using the same circuit but different port numbers. When the Ryanteck board came out, coincidentally using the same circuit, I simply changed the port numbers on my breadboard to match the Ryanteck board so that the same code could be used. This board includes pins for the I2C port so that can be used or a tall connector can be soldered onto the 26-way connector, thereby allowing the other GPIO ports to be used. Figure 8-14 shows the Ryanteck Motor Control Board.

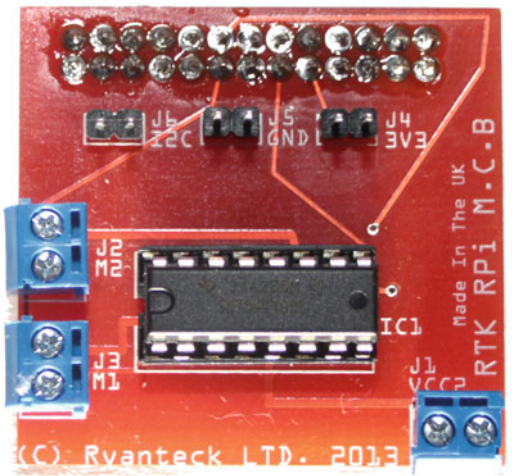


Figure 8-14. *Ryanteck Motor Control Board*

The CamJam robotics kit includes a small add-on printed circuit board, which is pre-soldered. The board uses a different integrated circuit than the one used previously, although essentially it works in the same way. The control board uses GPIO pins 7, 8, 9, and 10 for controlling the motors, with some of the other GPIO ports provided on a female header on top of the circuit. The pins are some of the ports used for SPI but there is still I2C and several other GPIO pins available. Figure 8-15 shows the board mounted on a Pi Zero.

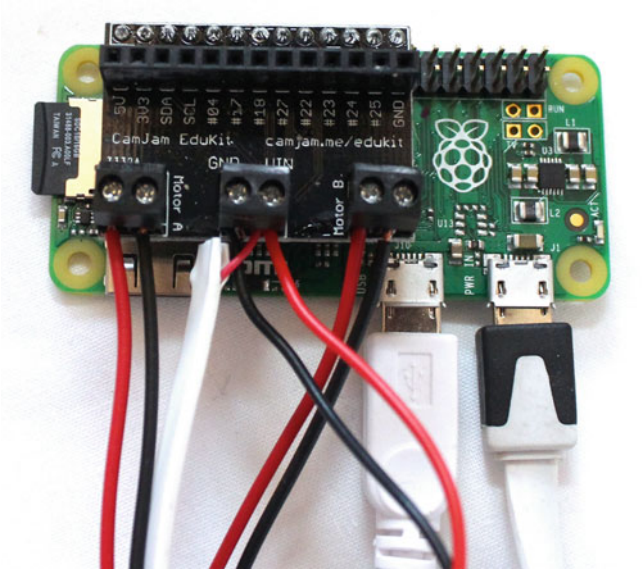


Figure 8-15. *CamJam Robotics motor control board*

Both of these motor controllers are designed for the original 26-way Raspberry Pi, so if you're using one of the more recent Raspberry Pi models that has a 40-way GPIO connector, there is scope for adding sensors to the remaining ports. You will learn about those ports in some of the modifications later in this chapter.

There are other motor controller boards, such as the Gertduino and boards from PiBorg. These are a little more expensive than the two boards discussed here, but they do have more functionality if you want to go beyond the basic motors used in this example.

Controlling the Robot Using Python

You will look at a few different ways to control the robot. The first way is using a simple program that runs on the command line. The program will look for certain keys being pressed and act accordingly. For the direction control, I used the numbers from the numeric keypad on a keyboard as they are arranged in a convenient grid. If you are using a compact keyboard that doesn't include a numeric keypad, you may want to change these to use appropriate letters instead. The numeric keypad and the directions are shown in Figure 8-16.

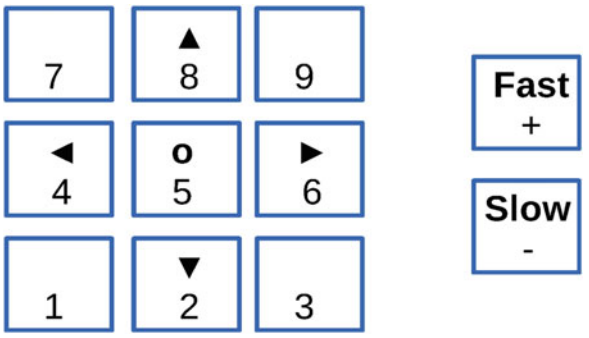


Figure 8-16. *Direction keys for the robot*

The number 0 is also used to instruct the camera take a photo, and Q is used to exit (quit) the program.

You will be using GPIO Zero to control the robot. The code will set the speed and direction and then wait for a key to be pressed, which will be used to update the speed or direction as appropriate. The code for this is included in the source files in the `robot` directory in a file called `robotcontrol.py` and is included here:

```
#!/usr/bin/python3
import sys, tty, termios
import picamera, time
from gpiozero import Robot

robot = Robot(left=(17, 18), right=(22, 23))
camera_enable = False
try:
    camera = picamera.PiCamera()
    camera.hflip=True
    camera.vflip=True
    camera_enable=True
except:
    print ("Camera not found - disabled");

photo_dir = "/home/pi/photos"

# get a character from the command line
def getch() :
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    try:
        tty.setraw(sys.stdin.fileno())
        ch = sys.stdin.read(1)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    return ch

# list to convert key presses into motor on/off values to correspond with the direction
# direction based on number keypad
# 8 = fwd, 4 = left, 5 = stop, 6 = right, 2 = rev
# the key for the list is the character
```

```
direction = {
    # number keys
    '2' : "backward",
    '4' : "left",
    '5' : "stop",
    '6' : "right",
    '8' : "forward"
}

current_direction = "stop"
# speed is as a percentage (ie. 100 = top speed)
# start speed is 50% which is fairly slow on a flat surface
speed = 50

print ("Robot control - use number keys to control direction")

print ("Speed " + str(speed) +"% - use +/- to change speed")

while True:
    # Convert speed from percentage to float (0 to 1)
    float_speed = speed / 100
    if (current_direction == "forward") :
        robot.forward(float_speed)
    # rev
    elif (current_direction == "backward") :
        robot.backward(float_speed)
    elif (current_direction == "left") :
        robot.left(float_speed)
    elif (current_direction == "right") :
        robot.right(float_speed)
    # stop
    else :
        robot.stop()

    # Get next key pressed

    ch = getch()

    # q = quit
    if (ch == 'q') :
        break
    elif (ch == '+') :
        speed += 10
        if speed > 100 :
            speed = 100
        print ("Speed : "+str(speed))
    elif (ch == '-' ) :
        speed -= 10
        if speed < 0 :
            speed = 0
        print ("Speed : "+str(speed))
```

```
    elif (ch in direction.keys()) :
        current_direction = direction[ch]
        print ("Direction "+current_direction)
    elif (ch == 'O' and camera_enable == True) :
        timestring = time.strftime("%Y-%m-%dT%H.%M,%S", time.gmtime())
        print ("Taking photo " +timestring)
        camera.capture(photo_dir+'/photo_'+timestring+'.jpg')
robot.close()
```

The code uses a `Robot` object from the `gpiozero` library. This takes the GPIO pins for the left and right motors. These are the pins that go to the appropriate inputs on the H-bridge integrated circuit.

You can also use the Raspberry Pi camera so that it can take still photos when requested. A potential problem is that you may want to run this on a robot that doesn't have a camera. To handle this, the camera is added using a `try` statement to gracefully handle any errors. If this was not in the `try` statement—if the camera was not found—the program will give an error message and stop. The camera is mounted upside down on my robot (with the cable coming from the top), so it is flipped horizontally and vertically after creating the camera object.

The name of the `getch()` function stands for the *get character,* which is taken from the name of a similar function in C. The reason this function is required is that Python normally reads keypresses from the standard input device. This is a buffered device that will hold all the characters until Enter is pressed and then send them all the program as a line. That would mean you would have to press Enter after each command, which wouldn't be so easy. This function sets the input to raw mode so that each keypress can be read individually. After the key is read, it resets the terminal back to standard mode; otherwise, this may mess up the terminal when you exit the program.

The direction keys are then stored as a list. If you want to add more keys to this list, you can do so as long as they have the same value as the existing ones. The `current_direction` is set to stop and the speed to 50%. This speed was chosen as it's the minimum speed for my particular robot to be able to move forward. You may need to change this up or down depending on your particular robot. Although this is stored as a percentage in the speed variable, GPIO Zero expects a number between 0 and 1, so this is stored as a floating-point number in the `float_speed` variable.

Setting the robot in motion is as simple as calling `robot.forward()`, `robot.backward()`, `robot.left()` and `robot.right()`. The optional parameter is the speed that sets the PWM ratio. There is also `robot.stop()`, which stops both motors. Upon exiting, `robot.close()` is called to clean up.

The code should be saved onto the Raspberry Pi on the robot. It could be controlled directly using a wireless keyboard on the robot, but that would need the program to be started whenever the robot was powered on, and without a screen performing a clean shutdown or restarting the program, that would be difficult. It is perhaps more convenient to run this over a wireless network connection using ssh. If you find that the wrong motor turns when the appropriate key is pressed, or that the robot moves in the wrong direction, you can fix it by swapping the wires going to the motors, or by changing the pin settings when the robot object is created toward the start of the code.

The previous code uses the generic robot within the GPIO Zero library. It also includes two other robot types for the Ryanteck and CamJam robots. These are used in the same way, but as the pin numbers are fixed for these controllers, they are pre-set when the object is created. To use one of these boards, replace the `import` and robot creation entries with the following:

```
from gpiozero import RyanteckRobot
robot = RyanteckRobot()
```

or

```
from gpiozero import CamJamKitRobot
robot = CamJamKitRobot()
```

The only thing about using the specific robot controllers rather than the generic "robot" one is that the pins cannot be changed in the code. So if the motors don't go in the right direction, the only way to change that is to change the physical wiring. You can always use the generic robot for either of these robot controller boards, which gives you the ability to change the pin numbers as required.

Measuring the Distance Using an Ultrasonic Range Sensor

Once you have the robot working, you can add other sensors to make the robot a bit more autonomous. The example you will look at is an ultrasonic range sensor, which can be used to measure the distance from an object. In this case, you will use it to determine when the robot is getting close to a wall or other obstacle. The ultrasonic range sensor is shown in Figure 8-17. It works in a similar way to the parking sensors on cars. It sends out an ultrasonic signal and then times how long it takes to get a signal reflected back.

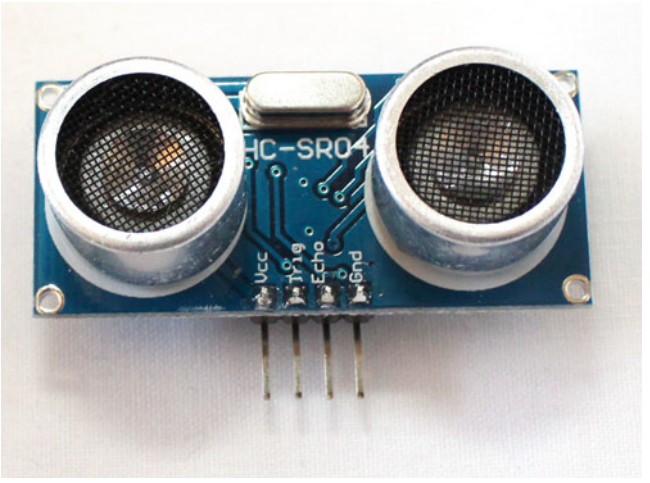


Figure 8-17. *Ultrasonic range sensor*

There are four connections to the sensor. Two of these are for a 5V power supply and ground. The other two connections are for the Trigger signal and the Echo response. The sensor is designed for a 5V signal. In the case of the input to the ultrasonic sensor, the 3.3V from the Raspberry Pi is sufficient to trigger the outgoing signal, but connecting the sensor output direct to a GPIO pin could damage the Raspberry Pi. You learned about the voltage divider circuit in Chapter 5, which reduces the voltage to a safe level for the Raspberry Pi. This is shown in the circuit diagram in Figure 8-18.

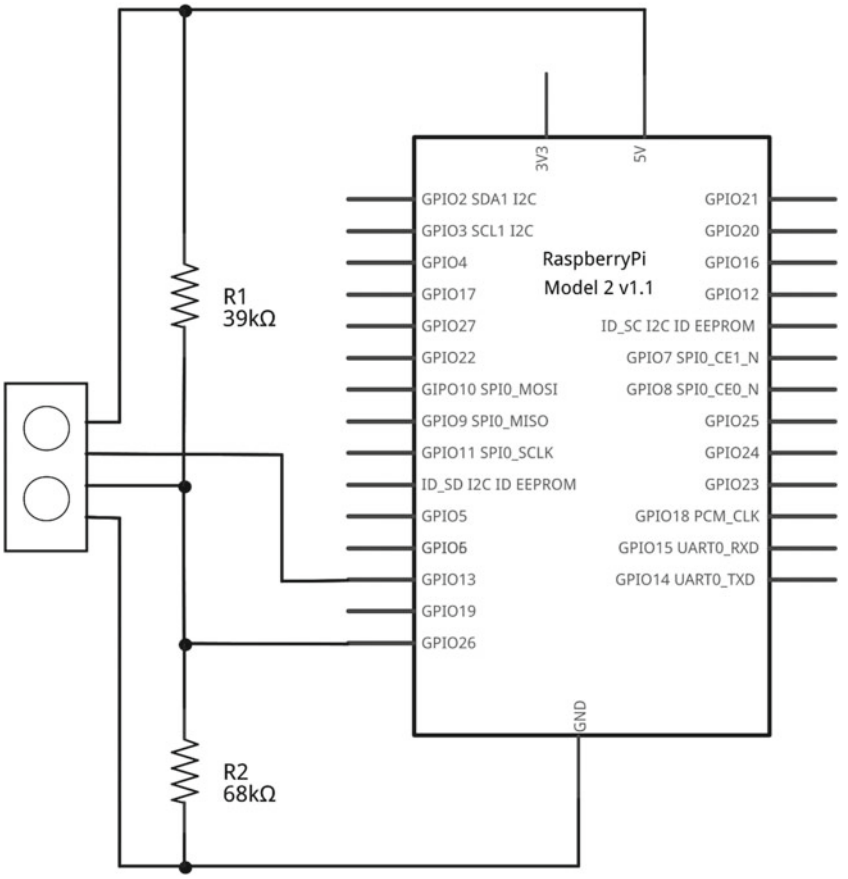


Figure 8-18. *Circuit diagram for ultrasonic range sensor*

The circuit diagram shows the ultrasonic range sensor part, assuming that the motor control circuit is still in place. The trigger pin is connected to GPIO 13 and the echo response to GPIO 26. These have been used because they are on the extra pins used in the later models of the Raspberry Pi and are not covered by the 26-pin motor control circuit board if you're using the PCB. If you're using a version 1 Raspberry Pi, you will need to change the pin number for the wiring and in the code.

Before you create the full program to control the robot, it's useful to test that the ultrasonic circuit is working correctly. First I created this code in the `distance.py` file:

```
#!/usr/bin/python3
import time
import RPi.GPIO as GPIO

# Ultrasonic range sensor (distance)
pin_trig = 13
pin_echo = 26
```

```
# Set up GPIO input/output and disable warnings
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings (False)
GPIO.setup(pin_trig, GPIO.OUT)
GPIO.setup(pin_echo, GPIO.IN)

# Set ultrasonic sensor off and wait to settle
GPIO.output(pin_trig, 0)
time.sleep(0.5)

while True:
    # sleep to ensure we don't try to recalc distance before it's settled
    time.sleep(0.5)

    # Check for distance in front
    # Send 10micro sec pulse
    GPIO.output(pin_trig, 1)
    time.sleep(0.000001)
    GPIO.output(pin_trig, 0)

    # Wait until input goes high - then wait for it to do low
    while (GPIO.input(pin_echo)==0):
        pass
    # Start timer
    start_time = time.time()

    # wait until it goes low again
    while (GPIO.input (pin_echo)==1):
        current_time = time.time()
        # If no response in reasonable time then not working (perhaps too close to an object
        - or too far)
        if ((current_time - start_time)>0.05):
            # print dot so we know it's happening
            print (".")
            break

    # Calculate response time
    response_time = current_time - start_time;

    #distance is time * speed sound (34029cm/s) - divide by 2 for return journey
    distance = response_time * 34029 / 2

    print ("Distance is "+str(distance));
```

The ultrasonic sensor is not included in the GPIO Zero library, so this is written using the classic `RPi.GPIO` library. This uses a bit more code to set the mode and configure each of the ports, which can then be turned on and off using `GPIO.output` and read using `GPIO.input`.

The code works by first turning the trigger off and waiting for the sensor to settle. A 10µs pulse sends an ultrasonic signal. It waits for the echo to go high before starting a timer and then waits in a loop until the response is received. The ultrasonic signal travels at the speed of sound, the time taken is for the outgoing signal and the response. The calculated value is then printed in centimeters (there are approximately 2.5cm in 1 inch).

The values shown on the screen were crucial in establishing how to use this to control the robot. The first observation is that having an object either too close or too far away gives the same value of around 850cm (which is down to the timeout of the `wait` loop). This is the case for obstacles up to 3mm away from the sensor, which is pretty close to touching. Then there is an area up to about 5cm where the reading is not particularly reliable, giving a fairly random reading of between 3.5 and 5cm. After 5cm, the sensor is fairly accurate up to about 3 meters (10 feet); then it jumps to the 850cm mentioned previously.

As long as you are looking to measure a distance of between 5cm and 300cm, this method should work. I wrote a program (called `robot-distance.py`) that drives the robot forward until gets within 13cm of an obstacle and then turns right. If the robot then has space to move, it will move forward again—otherwise, it will move right some more. The value of 13cm is based on experimentation and allows for the motor continuing to move forward for a short while after reading the distance. This is enough to prevent the robot from getting stuck into a corner. The distance and the speed of the robot may need to be adjusted for different robots and motor configurations. The full code to control the robot using the distance sensor is here:

```
#!/usr/bin/python3
import time
from gpiozero import Robot
import RPi.GPIO as GPIO

robot = Robot(left=(17, 18), right=(22, 23))

# Ultrasonic range sensor (pins)
pin_trig = 13
pin_echo = 26

## These may need to be adjusted depending upon the size of the robot and the speed of the
motors
# minimum distance before we turn in cm
min_distance = 13
# speed when moving forward (between 0 and 1)
speed_forward = 0.8
# speed when turning (between 0 and 1)
speed_turn = 0.6

# Set up GPIO input/output and disable warnings
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings (False)
GPIO.setup(pin_trig, GPIO.OUT)
GPIO.setup(pin_echo, GPIO.IN)

# Set ultrasonic sensor off and wait to settle
GPIO.output(pin_trig, 0)
time.sleep(0.5)

while True:
    # sleep to ensure we don't try to recalc distance before it's settled
    time.sleep(0.5)

    # Check for distance in front
    # Send 10micro sec pulse
    GPIO.output(pin_trig, 1)
```

```
    time.sleep(0.000001)
    GPIO.output(pin_trig, 0)

    # Wait until input goes high - then wait for it to do low
    while (GPIO.input(pin_echo)==0):
        pass
    # Start timer
    start_time = time.time()

    # wait until it goes low again
    while (GPIO.input (pin_echo)==1):
        current_time = time.time()
        # If no response in reasonable time then not received a response (either too close
        to an object - or too far)
        if ((current_time - start_time)>0.05):
            break

    # Calculate response time
    response_time = current_time - start_time;

    #distance is time * speed sound (34029cm/s) - divide by 2 for return journey
    distance = response_time * 34029 / 2

    # if we are close to a wall then turn right
    if (distance < min_distance) :
        print ("Too close " + str(distance) + " turning")
        robot.right(speed_turn)
    else :
        robot.forward(speed_forward)

robot.close()
restore_terminal()
GPIO.cleanup()
```

As you can see, this code uses the GPIO Zero robot library to control the motors and the `RPi.GPIO` library to send and receive the appropriate details from the ultrasonic range finder. In practice, the range finder works well on walls and large objects, but can't see certain obstacles that are too high or low compared to the position of the sensor.

Controlling the Robot Using a Wii Remote

Previously you used Python to control the robot using keyboard presses from another computer (using ssh) or a wireless keyboard. This works if you have a laptop, but it is often more convenient to control the robot using a wireless controller. This section looks at how to control the robot using a Wii Remote controller. The Wii Remote is a popular way to control a robot and it uses Bluetooth, which is now included on the board for the Raspberry Pi 3 or available as a low-cost USB dongle for the other models. For this example, I am assuming a Raspberry Pi A+ is being used, which does not include the Bluetooth and has only a single USB port for the Bluetooth dongle.

You will be using existing software libraries for this. Unfortunately, this has not yet been updated to run with Python 3, so you will instead be using Python 2.7. The existing robot code does work with Python 2.7, so it's just a case of changing the code to call `python` instead of `python3`. If you're using IDLE, you should run the version titled Python 2 (IDLE).

Assuming you have only a single USB port on the Raspberry Pi, you will find it easier to use a USB hub, as you will need to download some new software from over the network and they program will need the USB Bluetooth dongle. After you have finished, you can set this up so that the network connection is no longer required.

First update the operating system on the Raspberry Pi to include the latest updates. This is a good idea any time, but it's worth checking as Bluetooth hardware is a recent addition and there may have been some useful updates since. Assuming the Raspberry Pi is connected to the network, it can be updated using:

```
sudo apt-get update
sudo apt-get dist-upgrade
```

Now insert the USB Bluetooth dongle.

Check the status of the Bluetooth service:

```
sudo service bluetooth status
```

Be sure it shows as being active. If it doesn't, you may need to check that the Bluetooth dongle is supported on the Raspberry Pi. Next check that it's possible to see the Wii Remote:

```
hcitool scan
```

Once you've entered the command, press the remote control buttons 1 and 2 simultaneously so that it is visible. You should now see an entry identifying itself as a Nintendo controller. I used a Wii Remote with MotionPlus, so I got an entry that read `Nintendo RVL-CNT-01`.

Assuming that you do see this (don't worry if you see some other devices such as a tablet or phone as well), you can move on to adding some Python code to listen for instructions. You will use the `cwiid` library, which you need to install:

```
sudo apt-get install python-cwiid
```

As before when you were using the ultrasonic distance sensor, I provided some test code prior to including this into the program to control the robot. This program is called `wii-remote.py` in the book's source code:

```
#!/usr/bin/python
import cwiid
import time

delay = 0.2

print ("Press 1 + 2 on the Wii Remote")
time.sleep(1)

wii = cwiid.Wiimote()
print ("Connected\n")
```

```
# Testing button mode
wii.rpt_mode = cwiid.RPT_BTN

while True :
    buttons = wii.state["buttons"]
    button_string = ""

    # The value of each button is added together so use a
    # 'bitwise and' to test each button
    if (buttons & cwiid.BTN_1):
        button_string += " 1"
    if (buttons & cwiid.BTN_2):
        button_string += " 2"
    if (buttons & cwiid.BTN_A):
        button_string += " A"
    if (buttons & cwiid.BTN_B):
        button_string += " B"
    if (buttons & cwiid.BTN_UP):
        button_string += " up"
    if (buttons & cwiid.BTN_RIGHT):
        button_string += " right"
    if (buttons & cwiid.BTN_DOWN):
        button_string += " down"
    if (buttons & cwiid.BTN_LEFT):
        button_string += " left"
    if (buttons & cwiid.BTN_PLUS):
        button_string += " plus"
    if (buttons & cwiid.BTN_MINUS):
        button_string += " minus"
    if (buttons & cwiid.BTN_HOME):
        button_string += " home"

    if (button_string != ""):
        print ("Buttons pressed :" + button_string)
    time.sleep(delay)
```

Most of this code checks for each button to see which is pressed. Start by loading the Python library and attempting to connect to the remote control. During this time, buttons 1 and 2 on the remote need to be pressed simultaneously. If it fails to connect, the program will stop, but assuming that works, you can enable the button mode and then enter a `while` loop, where you can constantly check and print out the status of the buttons.

The value received from the `wii.state` command is a sum of the different buttons. To see if a button is pressed, perform a bitwise `and` against the appropriate mask value, which is defined as a constant in the `cwiid` library. This way, you can check for multiple buttons pressed simultaneously. Each of the buttons that is pressed is added to a string, which is then printed to the screen. Pressing one or more buttons prints a list of the buttons that are pressed.

Assuming this test program works, you can update the previous program by replacing the code to monitor for keypresses from the keyboard to watch for button presses from the Wii Remote. This is saved as wii-robotcontrol.py:

```
#!/usr/bin/python
import sys, tty, termios
import picamera, time
import cwiid
from gpiozero import Robot

robot = Robot(left=(17, 18), right=(22, 23))
camera_enable = False
try:
    camera = picamera.PiCamera()
    camera.hflip=True
    camera.vflip=True
    camera_enable=True
except:
    print ("Camera not found - disabled");

photo_dir = "/home/pi/photos"

# delay between button presses
delay = 0.2

current_direction = "stop"
# speed is as a percentage (ie. 100 = top speed)
# start speed is 50% which is fairly slow on a flat surface
speed = 100

print ("Press 1 + 2 on the Wii Remote")
time.sleep(1)

# Keep trying to connect to the remote
while True:
    try:
        wii=cwiid.Wiimote()
        break
    except RuntimeError:
        print ("Unable to connect to remote - trying again")
        print ("Press 1 + 2 on the Wii Remote")

print ("Robot control - use arrow buttons to control direction")
print ("Speed " + str(speed) +"% - use +/- to change speed")

wii.rumble = 1
time.sleep(0.5)
wii.rumble = 0

wii.rpt_mode = cwiid.RPT_BTN
```

```
while True:
    last_direction = current_direction
    # Convert speed from percentage to float (0 to 1)
    float_speed = speed / 100
    if (current_direction == "forward") :
        robot.forward(float_speed)
    # rev
    elif (current_direction == "backward") :
        robot.backward(float_speed)
    elif (current_direction == "left") :
        robot.left(float_speed)
    elif (current_direction == "right") :
        robot.right(float_speed)
    # stop
    else :
        robot.stop()

    time.sleep(delay)

    # Get next key pressed
    buttons = wii.state["buttons"]

    # set button to stop so that if no buttons pressed we stop
    current_direction = "stop"

    # + and - = quit
    if ((buttons & cwiid.BTN_PLUS) and (buttons & cwiid.BTN_MINUS)) :
        break
    if (buttons & cwiid.BTN_PLUS):
        speed += 10
        if speed > 100 :
            speed = 100
        print ("Speed : "+str(speed))
    if (buttons & cwiid.BTN_MINUS):
        speed -= 10
        if speed < 0 :
            speed = 0
        print ("Speed : "+str(speed))
    if (buttons & cwiid.BTN_UP):
        current_direction = "forward"
    if (buttons & cwiid.BTN_DOWN):
        current_direction = "backward"
    if (buttons & cwiid.BTN_LEFT):
        current_direction = "left"
    if (buttons & cwiid.BTN_RIGHT):
        current_direction = "right"
    if (buttons & cwiid.BTN_A and camera_enable == True):
        timestring = time.strftime("%Y-%m-%dT%H.%M.%S", time.gmtime())
        print ("Taking photo " +timestring)
        camera.capture(photo_dir+'/photo_'+timestring+'.jpg')
```

```
    # Only print direction if it's changed from previous
    if (current_direction != last_direction) :
        print ("Direction "+current_direction)

robot.close()
```

I added a few more changes to this so that you can have this running continuously. First, when you try to connect to the Wii Remote, this is now done in a loop using a `try` statement, which means you can keep trying if the first attempt fails. It also tells the Wii Remote to vibrate when it connects so that the user knows that the pairing was successful.

I've also set the speed to 100%. Previously, you used a slower speed for the robot as you didn't want it to get out of control when you pressed a key. The remote control is far more responsive than trying to control the robot using a keyboard, so running at full speed isn't an issue.

You now just need to be able to remote the USB hub so that the robot can move freely. As you can only have the Bluetooth or the WiFi dongle connected, you will no longer be able to connect through ssh to start the program. You can instead add the program to the startup, which is something you did for the Internet-controlled train in Chapter 6.

To make the program start automatically, first create a service file called `/etc/systemd/system/wii-robot.service`. Create the file as the root user (using `sudo`); it should contain the following:

```
[Unit]
Description=Wii Remote Robot Control
Wants=bluetooth.target
After=bluetooth.target

[Service]
Type=simple
ExecStart=/home/pi/robot/wii-robotcontrol.py
User=pi

[Install]
WantedBy=default.target
```

You already read about the features of the service file. The main new thing here is that this code instructs it to start after the Bluetooth service using the `Wants` and `After` entries.

You can test the service by starting it and verifying that it starts correctly:

```
sudo service wii-robot start
sudo service wii-robot status
```

Assuming these work correctly, issue this command to have it start automatically:

```
sudo systemctl enable wii-robot
```

It should now be possible to reboot the server without the WiFi, but with the Bluetooth dongle connected. The service will run in the background, waiting until it detects a Wii Remote, which it will then connect to and respond to the appropriate button presses. For me, this is a much more natural way to control the robot compared to using the keyboard.

More Robotics

In this chapter, you chose a suitable robot chassis and added motor control. You first used a breadboard and then swapped for a PCB-based motor controller. Two controllers were used in the example. These motor control boards are both supported by GPIO Zero, although other boards can be used by adding the appropriate pin numbers to the generic GPIO Zero robot library. You also looked at how the speed of the motor can be changed by using Pulse Width Modulation (PWM). You then looked at how to automate the robot using a distance sensor and how the robot can be controlled using a Wii Remote controller.

This is just a small taste of what you can achieve with a Raspberry Pi-powered robot. An alternative interface would be to create a web interface using the `Bottle` framework covered in Chapter 6. For a different type of automation, the CamJam robot includes a line-follower sensor that could be used to follow a black line.

In the next chapter, you look at how the physical world and the virtual world can interact using a joystick to navigate around Minecraft.

CHAPTER 9

Customize Your Gameplay: Minecraft Hardware Programming

Minecraft is a block-based construction game in which you can make buildings and other objects. It is a wildly popular game available for the PC, gaming consoles, and as the Minecraft Pocket edition for tablets and phones. It's also available as a special Minecraft Pi edition, which is provided for free with the Raspberry Pi Raspbian image. Using the Raspberry Pi edition, you can communicate with Minecraft using Python, which means that you can interface with the electronics sensors and outputs that you have used.

This chapter will use the joystick controller from Chapter 3. The joystick can be used to move the Minecraft character, Steve, around the world. The buttons will provide a specific function in the Minecraft game and you will code some LEDs to light up to show some useful information about the game.

The standard version of Minecraft includes a survival mode that has creatures that you have to defend against. The Pi edition is a more basic version and only has creative mode.

Tip While Minecraft will run on any of the Raspberry Pi models, it is processor intensive so a Raspberry Pi 2 or higher is recommended for performance reasons. You will also be using GPIO ports on the more recent models (B+ and later).

Connecting to Minecraft with Python

You will start by creating a new Minecraft world and connecting to it with Python. There is an Application Program Interface (API) provided that makes it easy to interact with Minecraft.

Minecraft needs to be running first and can be found under Games on the Start menu. Choose Start Game and then Create New to build your first world. You will notice that the game starts above a terminal window. Use the terminal window when you want to move the screen around or maximize it. You can pause the game using the Esc key, which will allow you to move the screen or switch to your Python program. Clicking on Back to Game will return you into the game mode. To switch from Minecraft to another program without pausing, use the Tab key.

Tip Minecraft Pi edition draws directly to the screen memory. It is not possible to run the game through most remote desktop clients such as TightVNC.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_9

You will be in the game controlling the Steve character, who is carrying a sword. In the game you can move around using the WASD keys (W for forward, A for left, S for backward, and D for right). The space key is used to jump, and a double press of the space key will allow Steve to fly. The E key will bring up the inventory allowing you to change the tool/weapon or use another item from the inventory. The mouse allows you to look around and the mouse button uses the current tool.

Once you familiarize yourself with moving around Minecraft, you can create your first program. This can be written using IDLE for Python 3. Add the following code and then choose Run Module from the IDLE Run menu:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.postToChat("Hello Minecraft World")
```

After you import the Minecraft library, a Minecraft object is created named `mc`. The `postToChat` displays a message on the screen, as shown in Figure 9-1.

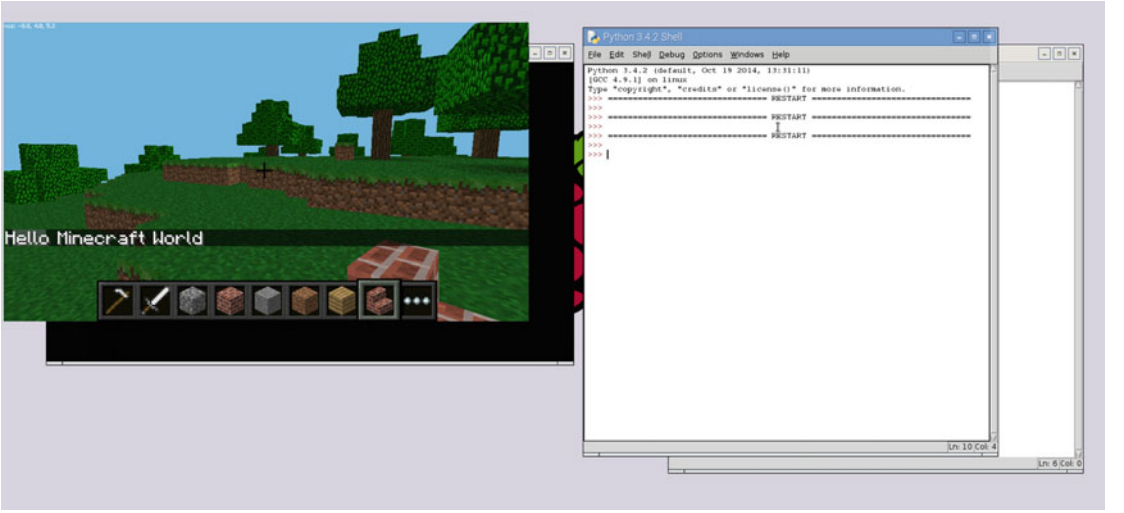


Figure 9-1. *Hello Minecraft World message*

As well as sending information to Minecraft, you can get the status of your position in the Minecraft world. The following script will get information about Steve and his environment:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
position = mc.player.getTilePos()
print ("X position :"+str(position.x)+", Y position :"+str(position.y)+", Z position:"+str(position.z))
```

When this is run, it displays your position within the Minecraft world. Minecraft is a 3D environment and so there are three values needed to show the position.

- The X position is the longitude (east-west position). As the character moves farther east, the x position increases and, as the character moves west, the x position decreases. The position is relative to the starting position so goes negative when you're moving west compared to the starting position.
- The Y position is the altitude (height relative to sea level). As the character moves upward (up a hill or into the sky), y increases and as the character moves down, it decreases. As the character goes underground, the y position goes negative.
- The Z position is the latitude (north-south position). As the character moves north, z decreases. Moving south causes the value of z to increase. This is relative to the starting position of the character. The output to the print statement is shown in the IDLE run window, as shown in Figure 9-2.

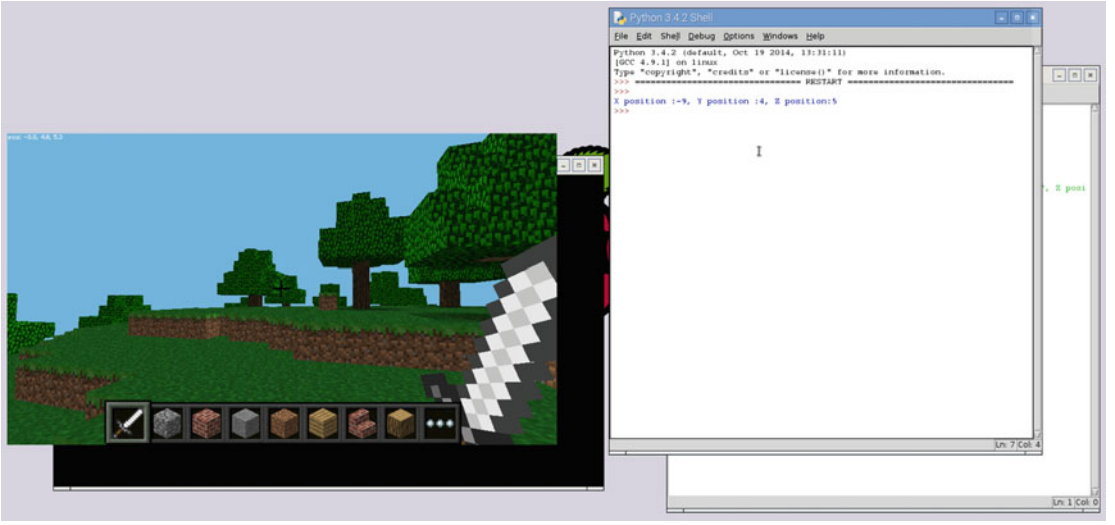


Figure 9-2. *Finding your location in Minecraft*

As well as reading the position, you can change the position of the character using `setPos`. The following program will return to the start position:

```
from mcpi.minecraft import Minecraft
mc = Minecraft.create()
mc.player.setPos(0, 0, 0)
```

If you run this, you may find a problem. The character always starts with X = 0 and Z = 0, but it starts at the normal ground level for that position. Unless the start position happened to be at sea level (Y = 0) or in a valley, Steve will most likely be trapped underground and won't be able to move. So you need a way to check what kind of block is at that position.

The next bit of code will check to see what kind of block is at that position. If it's an air block, it decreases Y to find the first solid block to stand on. If it's a solid block, it increases Y to find the nearest air block. This uses the `getBlock` from the `mcpi.block` module to find the type of block at a particular position. This has been put into a function as you will need this later.

```
from mcpi.minecraft import Minecraft
import mcpi.block as block

# Find nearest empty block based on an x, z position
def getSafePos(x_pos, y_pos, z_pos):
    block_id = mc.getBlock(x_pos, y_pos, z_pos)
    # If y position is in the air then move down to find the first non-air block
    if (block_id == block.AIR.id):
        while (block_id == block.AIR.id):
            y_pos = y_pos - 1
            block_id = mc.getBlock(x_pos, y_pos, z_pos)
        # we have found the first solid block - we want to go one above that on the first air block
        y_pos = y_pos + 1
    # If y position is underground then count up to find the first air block
    else :
        while (block_id != block.AIR.id):
            y_pos = y_pos + 1
            block_id = mc.getBlock(x_pos, y_pos, z_pos)
    # Return full address
    return (x_pos, y_pos, z_pos)

mc = Minecraft.create()
mc.player.setPos(getSafePos(0, 0, 0))
```

This tests to see if the block is of type `block.AIR.id`, which indicates that it is free space. There is a constant of each type of block, such as `block.WOOD.id`, `block.COBBLESTONE.id`, and `block.GLASS.id`.

Moving Around Using a Joystick

You should now have an idea of how you can use Python to talk to Minecraft. You can add some electronics to interact with Minecraft. You already created most of what you need when you made the arcade game in Chapter 3. You will add some LEDs to show certain information later in this chapter, but for now, you'll focus on the buttons and on controlling Steve with the joystick.

You need to think of what you'll do if there is a block in the way of where the character will move. You therefore need a button to act as a jump button. I have also provided two other buttons. Auto-jump will automatically move Steve up one block and long-jump will cause Steve to climb or safely fall any number of blocks in a single move. The long-jump button depends on a suitable position, and the joystick needs to be held down at the same time as it's pressed.

```
# Move around in Minecraft using a joystick
from mcpi.minecraft import Minecraft
import mcpi.block as block
from gpiozero import Button
import time
```

```
# Set up various buttons and connect to minecraft
mc = Minecraft.create()

JOY_NORTH = 23
JOY_EAST = 17
JOY_SOUTH = 4
JOY_WEST = 25

BTN_JMP = 22
# use to enable automatic jump by a single block
BTN_AUTOJMP = 9
# Jump, or safely fall any number of blocks
BTN_LGEJMP = 11

# Time to wait before moves
DELAY = 0.2

# Main loop to monitor buttons
def main():

    joy_north = Button(JOY_NORTH)
    joy_east = Button(JOY_EAST)
    joy_south = Button(JOY_SOUTH)
    joy_west = Button(JOY_WEST)

    btn_jmp = Button(BTN_JMP)
    btn_autojmp = Button(BTN_AUTOJMP)
    btn_lgejmp = Button(BTN_LGEJMP)

    # Autojump setting
    auto_jump = False

    while True:
        # jump status can be 0 = no jump, 1 = jump now, 2 = autojump, 3 = long_jump
        jump = 0
        # check for each button and set appropriate status
        if (btn_jmp.is_pressed) :
            jump = 1
        # toggle auto jump status
        if (btn_autojmp.is_pressed) :
            if (auto_jump == True):
                auto_jump = False
                mc.postToChat("Auto jump disabled")
            else :
                auto_jump = True
                mc.postToChat("Auto jump enabled")
        if (auto_jump == True):
            jump = 2
        if (btn_lgejmp.is_pressed):
            jump = 3
```

```
        # Don't check it's a safe position to move to - we test that later
        # Get current position and apply joystick movement
        position = mc.player.getTilePos()
        if (joy_north.is_pressed):
            position.z = position.z - 1
        if (joy_south.is_pressed):
            position.z = position.z + 1
        if (joy_east.is_pressed):
            position.x = position.x + 1
        if (joy_west.is_pressed):
            position.x = position.x - 1

        if (jump == 2):
            # Jump only if next position is solid
            block_id = mc.getBlock(position)
            if (block_id != block.AIR.id):
                position.y = position.y + 1
        # Now apply appropriate jump to new position
        if (jump == 1):
            # Jump regardless
            position.y = position.y + 1
        # auto jump uses getSafePos
        if (jump == 3):
            position = getSafePos(position.x, position.y, position.z)

        # Now we have the position to move to, check it's not a solid block
        block_id = mc.getBlock(position)
        if (block_id == block.AIR.id):
            mc.player.setTilePos(position)
        # Otherwise not a valid move so ignore

        # Delay before next instruction
        time.sleep(DELAY)

# Find nearest empty block based on an x, z position
def getSafePos(x_pos, y_pos, z_pos):
    block_id = mc.getBlock(x_pos, y_pos, z_pos)
    # If y position is in the air then move down to find the first non-air block
    if (block_id == block.AIR.id):
        while (block_id == block.AIR.id):
            y_pos = y_pos - 1
            block_id = mc.getBlock(x_pos, y_pos, z_pos)
        # we have found the first solid block - we want to go one above that on the first air block
        y_pos = y_pos + 1
    # If y position is underground then count up to find the first air block
    else :
        while (block_id != block.AIR.id):
            y_pos = y_pos + 1
            block_id = mc.getBlock(x_pos, y_pos, z_pos)
    # Return full address
    return (x_pos, y_pos, z_pos)
```

```
#Run the main function when this program is run
if __name__ == "__main__":
    main()
```

This code uses a combination of GPIO Zero and the Minecraft API.

One issue with the joystick is that it always moves according to the compass directions. If Steve is facing north, this works well, but if he's not, it may not feel natural. This is a limitation of Minecraft Pi edition as unfortunately there is no way to determine which direction Steve is facing.

There are three more buttons on the console. You'll use one to show the current location, the second to create a new house, and the third (the big button) to transport you to the most recent house. These are all useful when you are in survival mode and want to escape from a danger.

The complete console will then look as shown in Figure 9-3.

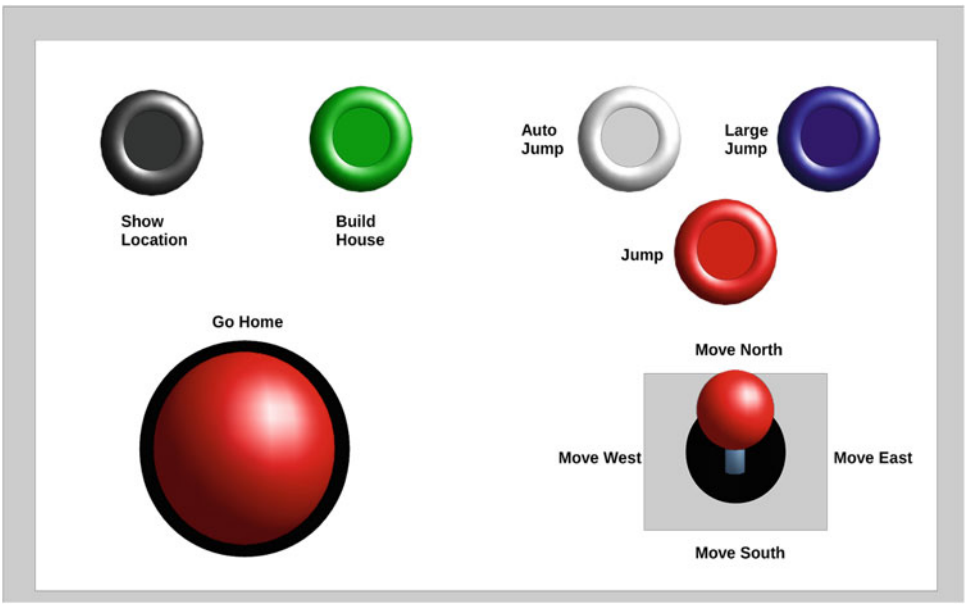


Figure 9-3. *Joystick and button positions for use in Minecraft*

Building a House in Minecraft

One of the good things about writing code is that you can use it to automate things that would otherwise take much longer to do manually. Building a house is a good example of this, as you can create some simple for loops to place the blocks for the walls. Using code, this can be done much quicker than if you had to position them all manually.

Before creating the walls, you need to create the space for the house; otherwise, you could end up with the ground coming up inside of the house. It would take extra code to do this a layer at a time, but fortunately there is a quicker way. You'll use `setBlocks`, which allows you to select an area of blocks to change to a particular block type.

You then create four walls and a roof. Initially the walls will be plain, but you will add a door and windows afterward. Here's the code to build the house:

```
# Build a house in Minecraft
import mcpi.minecraft as minecraft
import mcpi.block as block
from mcpi.minecraft import Minecraft
from gpiozero import Button
import time

# Set up various buttons and connect to minecraft
mc = Minecraft.create()

BTN_HOUSE = 18

# House size
house_size_x = 16
house_size_y = 6
house_size_z = 10

# Save position of house here
# allows us to go to it later
# if not defined then leave at 0,0,0
# vec3 allows us to create a position vector
house_position = minecraft.Vec3(0,0,0)

# Time to wait before moves
DELAY = 1

# Main loop to monitor buttons
def main():
    btn_house = Button(BTN_HOUSE)

    while True:
        if (btn_house.is_pressed) :
            # Set position to current location
            # This will be the center of the house
            house_position = mc.player.getTilePos()
            build_house(house_position, house_size_x, house_size_y, house_size_z)
            # Delay before next instruction
            time.sleep(DELAY)
    time.sleep (0.2)

def build_house (house_position, house_size_x, house_size_y, house_size_z):
    # clear blocks where the house is to be built
    mc.setBlocks (                                  \
        house_position.x - (house_size_x / 2),      \
        house_position.y,                           \
        house_position.z - (house_size_z / 2),      \
        house_position.x + (house_size_x / 2),      \
        house_position.y + house_size_y,            \
```

```
    house_position.z + (house_size_z / 2),     \
    block.AIR.id)
# Build floor
mc.setBlocks (                                 \
    house_position.x - (house_size_x / 2),     \
    house_position.y - 1,                      \
    house_position.z - (house_size_z / 2),     \
    house_position.x + (house_size_x / 2),     \
    house_position.y - 1,                      \
    house_position.z + (house_size_z / 2),     \
    block.COBBLESTONE.id)
# build front wall - north wall
mc.setBlocks (                                 \
    house_position.x - (house_size_x / 2),     \
    house_position.y,                          \
    house_position.z - (house_size_z / 2),     \
    house_position.x + (house_size_x / 2),     \
    house_position.y + house_size_y,           \
    house_position.z - (house_size_z / 2),     \
    block.BRICK_BLOCK.id)
# build rear wall - south wall
mc.setBlocks (                                 \
    house_position.x - (house_size_x / 2),     \
    house_position.y,                          \
    house_position.z + (house_size_z / 2),     \
    house_position.x + (house_size_x / 2),     \
    house_position.y + house_size_y,           \
    house_position.z + (house_size_z / 2),     \
    block.BRICK_BLOCK.id)
# build side wall - east wall
mc.setBlocks (                                 \
    house_position.x + (house_size_x / 2),     \
    house_position.y,                          \
    house_position.z - (house_size_z / 2),     \
    house_position.x + (house_size_x / 2),     \
    house_position.y + house_size_y,           \
    house_position.z + (house_size_z / 2),     \
    block.BRICK_BLOCK.id)
# build side wall - west wall
mc.setBlocks (                                 \
    house_position.x - (house_size_x / 2),     \
    house_position.y,                          \
    house_position.z - (house_size_z / 2),     \
    house_position.x - (house_size_x / 2),     \
    house_position.y + house_size_y,           \
    house_position.z + (house_size_z / 2),     \
    block.BRICK_BLOCK.id)
# Build roof
mc.setBlocks (                                 \
    house_position.x - (house_size_x / 2),     \
    house_position.y + house_size_y + 1,       \
```

```
        house_position.z - (house_size_z / 2),     \
        house_position.x + (house_size_x / 2),     \
        house_position.y + house_size_y + 1,       \
        house_position.z + (house_size_z / 2),     \
        block.WOOD.id)

    # Frame of house now built - add doors and windows
    # Make doorway out of air block
    mc.setBlocks (                                 \
        house_position.x,                          \
        house_position.y,                          \
        house_position.z - (house_size_z / 2),     \
        house_position.x + 1,                      \
        house_position.y + 2,                      \
        house_position.z - (house_size_z / 2),     \
        block.AIR.id)

    # Add 2 windows
    mc.setBlocks (                                 \
        house_position.x + (house_size_x / 4),     \
        house_position.y + (house_size_y / 2),     \
        house_position.z - (house_size_z / 2),     \
        house_position.x + (house_size_x / 4) + 2, \
        house_position.y + (house_size_y / 2) + 2, \
        house_position.z - (house_size_z / 2),     \
        block.GLASS.id)

    mc.setBlocks (                                 \
        house_position.x - (house_size_x / 4),     \
        house_position.y + (house_size_y / 2),     \
        house_position.z - (house_size_z / 2),     \
        house_position.x - (house_size_x / 4) - 2, \
        house_position.y + (house_size_y / 2) + 2, \
        house_position.z - (house_size_z / 2),     \
        block.GLASS.id)

#Run the main function when this program is run
if __name__ == "__main__":
    main()
```

This looks a lot of code. That is partly because each `setBlocks` command uses eight lines, as it's split across multiple lines using the \ continuation character. This is not required but makes it easier to see how each of the X, Y, and Z positions are calculated.

You need to be careful when deciding where to position the house in the Minecraft world. While the code will carve out the space needed for the house, it would look odd if much of the house was in the air or if the door is inaccessible. Ideally the house should be on a fairly level area. Also if the wall with the door is positioned on the side of a hill, you may need to dig yourself out.

If you want to teleport yourself to the house in future (I saved the big button for this), the house's position is saved in `house_position` and can be used with `setTilePos()`.

The finished house is shown in Figure 9-4.

Figure 9-4. *Minecraft house built using code*

Adding Status LEDs

So far we've been using buttons and a joystick to interact with Minecraft, but you can also use Minecraft as an input to your electronic circuits. One thing that you'll find yourself doing a lot in Minecraft is mining for resources buried underground. What would be easier is if there were some way that you could detect if there were useful resources underneath. It's also useful to know if there could be lava underneath to stop you from accidentally falling into it. You'll build code that can look at the blocks underneath you and use that information to show the status using LEDs. You'll use tri-color LEDs for some of these, which as the name suggests, can show three different colors.

The tri-color LEDs consists of two LEDs in a single package. One of the LEDs is green and the other is red. Turning both on together provides the third color, which is orange. The LEDs share the same negative connection, known as a common cathode. There are other multi-colored LEDs available, but the advantage of these is that they needs only two GPIO outputs and can provide three status colors.

■ **Tip** If you are unable to find tri-color LEDs (also known as duo-LEDS), you can use an RGB LED instead. The RGB LED has an extra terminal for blue which is not required. The RGB LEDs are available as common cathode or common anode. The common cathode type can be used as a direct replacement, leaving the terminal for the blue LED disconnected.

You'll use these LEDs to indicate the amount of a material available—red for none, orange for a small amount, and green for a larger quantity. One standard red LED will warn that lava present.

I mounted the LEDs on some cardboard Minecraft blocks, but these could be mounted into a cardboard box with labels indicating what each LED refers to. The blocks you are looking for are:

- `COAL_ORE` (lit furnace)
- `IRON_ORE` (unlit furnace)
- `DIAMOND_ORE` (glass)
- `LAVA` (flashing LED on wood)

The label in brackets is the cardboard block where you mount the LEDs. You can see the layout in Figure 9-5. The LED mounted on the wooden block will flash.

Figure 9-5. *Minecraft blocks with tri-color status LEDs*

These are useful things to know in the standard Minecraft, although `LAVA` and `DIAMOND_ORE` blocks don't appear to occur naturally in the Pi edition. Instead, the `LAVA` LED will come on when you dig too deep. In the standard edition, you normally hit bedrock first, but digging too deep in the Pi edition will result in Steve falling to his death. To make up for the lack of diamonds, you will write another program to add some later.

Each of the LEDs is wired back to the breadboard, where it is connected through a resistor to the Raspberry Pi GPIO port. This has been done using male-to-female jumper leads, although some of these needed to be joined together as a single wire was too short.

The circuit diagram showing how these connect to the GPIO ports is shown in Figure 9-6.

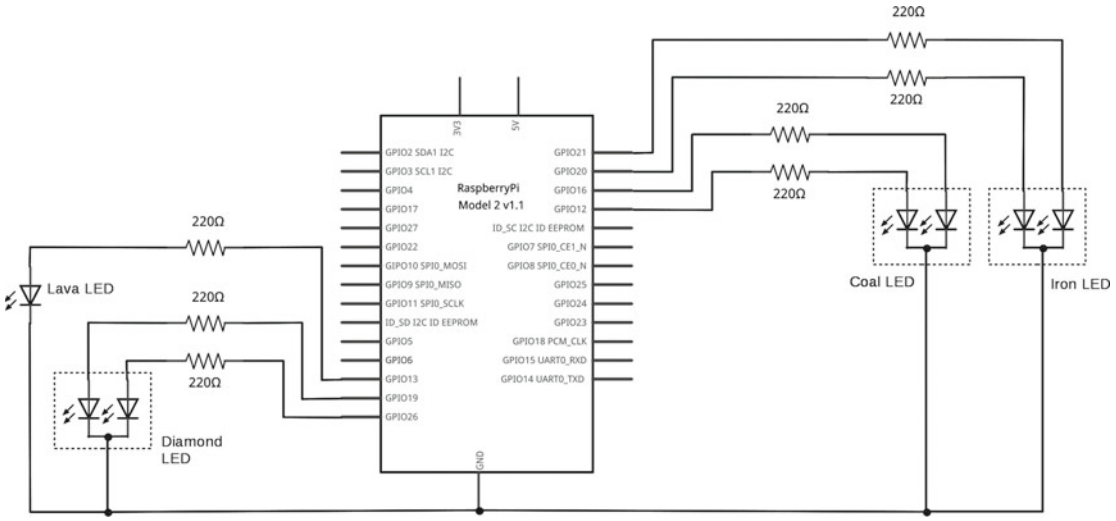


Figure 9-6. *Circuit diagram showing tri-color status LEDs*

This circuit diagram shows just the connections to the LEDs, which have been added to some of the GPIO ports included in the Raspberry Pi B+ and later. The buttons and joystick can therefore remain connected. You won't actually be using the joystick and the LEDs at the same time, but it's something you may want to look at adding yourself later. Details of the ports used for the buttons and joystick are in Chapter 3, Figure 3-24.

Here's the code to find the relevant blocks and light the LEDs:

```
# Scan underground blocks and light LEDs to show status
from mcpi.minecraft import Minecraft
import mcpi.block as block
from gpiozero import Button, LED
import time

# Connect to minecraft
mc = Minecraft.create()

DELAY = 0.2

# Number of blocks to scan for
DEPTH = 50

COAL_RED = 12
COAL_GREEN = 16
IRON_RED = 20
IRON_GREEN = 21
DIAMOND_RED = 26
DIAMOND_GREEN = 19
LAVA_RED = 13
```

```
coal_red = LED(COAL_RED)
coal_green = LED(COAL_GREEN)
iron_red = LED(IRON_RED)
iron_green = LED(IRON_GREEN)
diamond_red = LED(DIAMOND_RED)
diamond_green = LED(DIAMOND_GREEN)
lava_red = LED(LAVA_RED)

# track current status of lava - so we can make it flash
lava_flash = 0

while True:
    # variables to identify the number of blocks we have found
    coal_found = 0
    iron_found = 0
    diamond_found = 0
    lava_found = 0

    # Get position
    position = mc.player.getTilePos()

    for i in range (0, DEPTH):
        # Get ID of next block
        block_id = mc.getBlock(position.x, position.y - i, position.z)
        if (block_id == block.COAL_ORE.id):
            coal_found = coal_found + 1
        elif (block_id == block.IRON_ORE.id):
            iron_found = iron_found + 1
        elif (block_id == block.DIAMOND_ORE.id):
            diamond_found = diamond_found + 1
        elif (block_id == block.LAVA.id):
            lava_found = lava_found + 1

    # Update LEDs based on what found - 0 = red, 1 = orange, s+ = green
    if (coal_found < 1):
        coal_red.on()
        coal_green.off()
    elif (coal_found < 2):
        coal_red.on()
        coal_green.on()
    else: # 2 or more
        coal_red.off()
        coal_green.on()

    if (iron_found < 1):
        iron_red.on()
        iron_green.off()
    elif (iron_found < 2):
        iron_red.on()
        iron_green.on()
```

```
    else: # 2 or more
        iron_red.off()
        iron_green.on()

    if (diamond_found < 1):
        diamond_red.on()
        diamond_green.off()
    elif (diamond_found < 2):
        diamond_red.on()
        diamond_green.on()
    else: # 2 or more
        diamond_red.off()
        diamond_green.on()

    # different with lava - instead flash LED
    # toggle status on each iteration around the loop
    if (lava_found > 0 or position.y < -50) :
        lava_flash = 1 - lava_flash
        if (lava_flash == 1) :
            lava_red.on()
        else :
            lava_red.off()
    else:
        lava_red.off()
```

This code looks at the blocks that match the position of the player for the X and z positions, searching from y=0 to y=-50. It counts any of the matching blocks and then uses these to determine whether the red, green, or both LEDs should be on. The LAVA LED is different, as it is just a standard LED. The LAVA LED toggles by checking the status of `lava_flash`. Subtracting the current status from 1 will result in the status toggling from 1 to 0 on each turn around the loop, and hence flashing on and off.

Here's the code to place diamond blocks at random locations:

```
# Scatter some diamond ore in random positions underground
from mcpi.minecraft import Minecraft
import mcpi.block as block
import random

mc = Minecraft.create()

# We may not end up with quite this many if we get any
# duplicate positions
num_diamonds = 50

# min depth and max depth should both be positive (relative to 0)
max_depth = 30
min_depth = 5
max_distance = 100

for i in range (0, num_diamonds):
    x_pos = random.randrange(0, max_distance * 2)
    x_pos = x_pos - max_distance
    z_pos = random.randrange(0, max_distance * 2)
    z_pos = z_pos - max_distance
```

```
    y_pos = random.randrange(min_depth, max_depth,1)
    y_pos = y_pos * -1

    # set the appropriate block to a diamond
    mc.setBlock (x_pos, y_pos, z_pos, block.DIAMOND_ORE.id)
```

This uses `random.randrange` to get a number in a certain range. It only provides positive numbers so for the X and Z positions, you get a range between 0 and 200 and then subtract 100. This will give a number between -100 and 100 (actually it goes up to 99 as the top number in the range is never returned). For the Y position, a positive random number is provided and it's multiplied by -1 to change it into a negative number. The `setBlock` function is then used to set that block to a diamond.

You now have some LEDs showing the status of the various blocks. It makes it a little easier to find the blocks you require, although it still requires some searching. You will need to move fairly slowly to avoid missing any blocks. You could make the loop run faster by reducing the delay, but that may reduce the performance of Minecraft and slow the computer down when you're trying to dig.

Find the Glowstone Game

You are going to finish this chapter with a game inside Minecraft—a game within a game.

When the game runs, a glowstone block is hidden somewhere, and it's your job to find it as quickly as possible. The stone will be buried underground and the only help is a single LED that indicates whether you are getting closer (green), are the same distance away (orange), or are moving farther away (red). A catch is that only a single LED is used for all three directions (X, Y, and Z). If you move a step in the right horizontal direction, but in doing so move higher and hence farther away, this will count as neither closer nor farther away. A glowstone is used as it is normally only found in the Nether, so there is no risk of coming across one naturally. If you come across it underground, it should light up the area so you can see it.

The best way to understand this is to play the game. Here's the source code:

```
# Find the Glowstone
from mcpi.minecraft import Minecraft
import mcpi.block as block
from gpiozero import LED
import time, random

mc = Minecraft.create()

RED_LED = 12
GREEN_LED = 16

red_led = LED(RED_LED)
green_led = LED(GREEN_LED)

# min depth and max depth should both be positive (relative to 0)
max_depth = 30
min_depth = 5
max_distance = 100
```

```
# Hide the Glowstone
x_pos = random.randrange(0, max_distance * 2)
x_pos = x_pos - max_distance
z_pos = random.randrange(0, max_distance * 2)
z_pos = z_pos - max_distance
y_pos = random.randrange(min_depth, max_depth,1)
y_pos = y_pos * -1
mc.setBlock (x_pos, y_pos, z_pos, block.GLOWSTONE_BLOCK.id)

start_time = time.time()

last_position = mc.player.getTilePos()

while True:
    position_difference = 0
    current_position = mc.player.getTilePos()
    # Diff in x direction
    old_diff = abs(x_pos - last_position.x)
    new_diff = abs(x_pos - current_position.x)
    position_difference = position_difference + new_diff - old_diff
    print_string = print_string + " X diff:"+str(new_diff-old_diff)
    # Diff in y direction
    old_diff = abs(y_pos - last_position.y)
    new_diff = abs(y_pos - current_position.y)
    position_difference = position_difference + new_diff - old_diff
    print_string = print_string + " Y diff:"+str(new_diff-old_diff)
    # Diff in z direction
    old_diff = abs(z_pos - last_position.z)
    new_diff = abs(z_pos - current_position.z)
    position_difference = position_difference + new_diff - old_diff
    print_string = print_string + " Z diff:"+str(new_diff-old_diff)

    if (last_position.x != current_position.x or \
        last_position.y !=  current_position.y or \
        last_position.z != current_position.z):
        print (print_string+" Total:"+str(position_difference))

        if (position_difference > 0) :
            red_led.on()
            green_led.off()
        elif (position_difference < 0) :
            red_led.off()
            green_led.on()
        else :
            red_led.on()
            green_led.on()

        last_position = current_position
```

```
    # check to see if the block has been destroyed
    block_id = mc.getBlock(x_pos, y_pos, z_pos)
    if (block_id != block.GLOWSTONE_BLOCK.id):
        break

    time.sleep (0.1)

end_time = time.time()
timetaken = int(end_time - start_time)
mc.postToChat("Well done - it took you "+str(timetaken)+" seconds")
```

Save this as `minecraft-game.py`. Start a new game in Minecraft then run `minecraft-game.py` from IDLE or by running the following:

```
python3 minecraft-game.py
```

Once you've started the program, the clock is ticking. Move around watching the LED and try to find the glowstone. Whenever the LED is green you are getting closer and red means you are getting farther way. Orange means that you are neither farther away nor closer, but that will show only if you move in two different directions at the same time.

More Minecraft Hardware Programming

This chapter showed you how software and hardware can come together using the popular game Minecraft. You used a joystick and switches to send commands to Minecraft, which even included building a house at the press of a single button. You then turned the tables and had status LEDs light up in response to finding certain blocks in Minecraft. You finished by creating a game that used an LED to give additional feedback on your position relative to the glowstone.

The electronics are simple in this chapter so as to focus on the interaction with Minecraft, but that doesn't need to be the case. You can use any of the other sensors from earlier chapters or use your movement in Minecraft to send instructions to the robot.

I left the large button for teleporting back home. I have however left this as an activity for the reader. You will need to save the position when the house is created and then use `setTilePos()`. If you are interested in doing more in Minecraft Pi, there is more information on the API at http://www.stuffaboutcode.com/p/minecraft-api-reference.html.

In the next chapter, you will be looking at moving from the breadboard to create more permanent circuits using a soldering iron and you'll learn how to perform testing on electronic circuits.

CHAPTER 10

Making Your Circuits Permanent

Most of the circuits you have made so far were built using a breadboard. Creating circuits on a breadboard is particularly useful when you're designing a new circuit, as it allows for easy modifications to the design and reuse of the components. It does not work so well for permanent circuits, as components and wires can get accidentally pulled out of the breadboard. In this chapter you will look at how you can make your circuits permanent so that they can survive a reasonable amount of use without the wires falling out. This configuration will also allow you to use components that cannot be connected directly to a breadboard or components that need header pins to be soldered on before they can be connected to a breadboard. This includes many of the NeoPixels that you looked at in Chapter 6.

This will involve soldering, which can sound daunting if you haven't done any before, but it's a useful skill and not as difficult as you may think.

You'll also take a brief look at how to create a more professional-looking product by placing the circuit into an enclosure and you'll look at some simple diagnostic tools for when the circuit doesn't behave as it should.

Soldering Basics

Soldering is a technique that joins two metal objects by adding a metal filler. In the case of electronic circuits, this is a process commonly used to join an electronic component to a layer of copper on a printed circuit board or to join a wire to the lead of a component.

The metal used to form the join is known as *solder*. It has a lower melting point than the components you want to join, which allows you to melt the solder, which then flows around the joint. The solder then sets hard when it's allowed to cool. A good way to practice soldering is with a simple electronic kit. These don't need to be Raspberry Pi based, although there are some good ones that can be used with the Raspberry Pi, including the Ryanteck motor control board and a GPIO cobbler, which makes it easier to connect the Raspberry Pi to a breadboard. An electronics kit will often include a printed circuit board (PCB) along with some components that need to be soldered into place.

In addition to the kit, you are going to need a few tools to get started. The minimum is a soldering iron with a stand, a pair of wire cutters (preferably side-cutters), and a reel of solder, although there are other tools that can make soldering a little easier.

Gathering the Essential Tools

While it is possible to buy a basic soldering iron that is connected directly to a main electrical plug, it is useful to buy one with some kind of temperature adjustment. The one that I use includes a digital temperature control, as shown in Figure 10-1. The digital display is a nice luxury, but when starting out, a less expensive one with a simple temperature control knob is adequate.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_10

Figure 10-1. *Temperature controlled soldering iron*

Soldering irons also have a power rating. This does not necessarily mean that a soldering iron with a higher power rating will be at a higher temperature than one with a lower power rating. A soldering iron with a higher power rating (such as 50W or higher) will heat up quicker and maintain the temperature better. This is most important when soldering to a thick gauge wire, which can absorb a lot of heat from the soldering iron. For soldering on a PCB, the power rating is usually less important.

Most soldering irons have interchangeable tips with different shaped tips for different types of soldering. I normally use a chisel shaped tip because that makes it easier to get a good contact with the component and PCB. See Figure 10-4 later in this chapter for an example of the tip that I normally use. An alternative is a pointed tip, which can be useful for soldering small components.

A soldering iron stand is essential to store the soldering iron when it's not in use. These usually include a sponge that should be kept moist for cleaning the tip between each use. An alternative to using a sponge is to use a tip cleaner made of brass shavings, which also avoids the need to regularly moisten the sponge.

A pair of wire cutters is essential for cutting wires and removing the excess from component leads. They can also be used for stripping the insulation from wires if you don't have dedicated wire strippers. A sharp pair of wire cutters makes it much easier to strip the insulation from the wire.

■ **Tip** Don't be tempted to use your teeth as wire strippers. A pair of wire strippers is much cheaper to replace than the potential damage you could do to your teeth.

Side-cutters such as the ones shown in Figure 10-2 are most useful, as they have a flat side allowing you to cut close to the joint.

Figure 10-2. *Pair of side-cutters*

Choosing Solder

A good quality solder intended for hobby electronic circuits is normally an alloy made up of a mix of silver, copper, and tin. It normally includes a non-corrosive flux often made from rosin. The flux prevents oxidization during soldering and helps form a clean joint.

In the past, solder often contained lead, but due to concerns about health and the environment, these have become less popular. Some countries have imposed restrictions on the use of lead in commercial products. While some people prefer to use a lead-based solder, a good quality lead-free solder is easy enough to use for most purposes.

Although there is no lead in lead-free solder there are still many other toxins can cause serious health problems. Those involved in soldering on a regular basis should use a fume extraction system, which may be mandatory in some work environments. A proper fume extraction system may be prohibitively expensive for hobby use. I have a simple fume disperser (sometimes referred to as a fume absorber), shown in Figure 10-3. These fume dispersers will not remove the toxins in a rosin-based flux. They instead draw the fumes away from the person soldering and disperse them around the room. These should only be used occasionally and in a well-ventilated area where the fumes will be able to disperse outside and where other people will not breathe the air coming out the rear.

Figure 10-3. *Bench-top fume disperser*

Safety Tips When Soldering

While the idea of holding a hot soldering iron may sound dangerous, as long as you follow the basic safety precautions it is a relatively safe activity. The following are some basic precautions you should take when you're soldering:

- Hot **soldering iron—**It goes without saying that an iron that is hot enough to melt solder could easily burn skin. Accidentally touching your finger to a soldering iron can leave a small burn with no long lasting effects, but grabbing the end of the soldering iron could cause a more substantial burn. For this reason, it's important to place the soldering iron into a holder and ensure that the wire is not going to get pulled by mistake. Also remember that a soldering iron will remain hot for a long time after it's unplugged. If you do burn yourself, flood the area with water and leave it under running cold water for at least 10 minutes. If you have any concerns seek professional medical advice.
- **Soldering iron stand—**Always place the soldering iron in a suitable stand when not in use.
- Solder **fumes—**Breathing in solder fumes regularly can cause serious health problems. Always solder in a well-ventilated area or use an appropriate fume extraction.
- Protect **your eyes—**When cutting leads from components, beware of any ends that may fly off and go into your eyes. This can be avoided by holding the loose end leads in one hand when snipping them off, or by wearing the appropriate eye protection.
- Never **solder a live circuit—**Always remove power from a circuit board prior to any soldering. This is important to protect the power supply (or battery), components, and from a safety point of view.
- Adult **supervision—**Soldering is not a difficult activity and can be undertaken by children of an appropriate age as long as it is under appropriate adult supervision.

Soldering to a Printed Circuit Board

One of the easiest ways to start soldering is using a printed circuit board (PCB). These often have the component positions marked on the top so that the component can be easily inserted and then soldered on the bottom of the board. When soldering, you usually want the component to be as close to the PCB as possible. Perhaps the biggest challenge is keeping the circuit board still and ensuring that the component stays in the right place when soldering. This can be tricky, particularly with large components. To make this easier, start with the smallest components first and support the rest of the PCB with padding. I will sometimes use a piece of Lego or something similar to support the PCB.

Once you have the component in place, follow these steps:

1. Place the tip of the hot soldering iron so it is in contact with the pad on the PCB and the lead of the component.
2. Bring the solder in and allow sufficient solder to melt to form the joint.
3. Remove the solder, but leave the soldering iron in place for a few seconds to allow the solder to flow around the pad and the component's lead.
4. Remove the soldering iron and allow the joint to cool.
5. Ensure that the solder makes a good joint (if not, repeat this process).
6. Check that the solder does not expand beyond the pad, thus causing a bridge to any other components or pads.
7. Snip off the excess wire from the component.

Figure 10-4 shows the soldering iron touching both the PCB pad and the component lead.

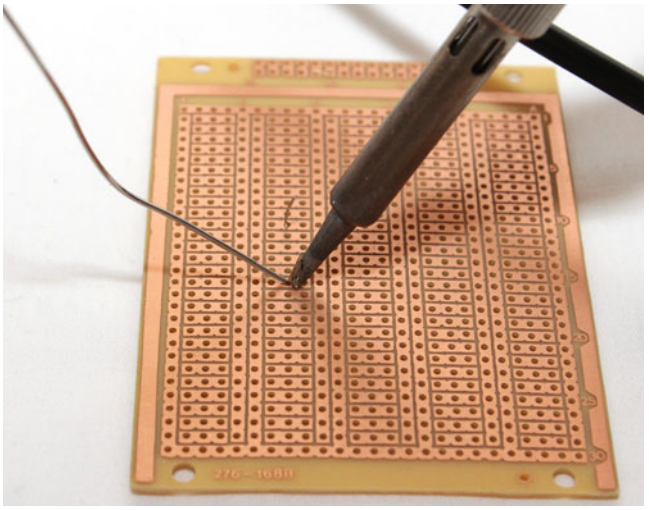


Figure 10-4. *Soldering iron just before making a solder joint*

Soldering is a skill best learned through practice. Don't get too disheartened if your first solder joint looks a bit untidy. Try again and your skills will improve over time.

■ **Tip** Always check for any stray bits of solder and short-circuits before connecting the circuit to your Raspberry Pi or power supply.

Here are a few additional hints and tips when soldering:

- Regularly clean the tip by wiping it on a damp sponge or using a brass tip cleaner.
- Tin the tip of the soldering iron using a small amount of solder on the tip.
- Avoid using too much solder.

Soldering Direct to Leads

Soldering leads and wires directly together can be a little more difficult, but only because the wires don't stay where you need them to. This is where a set of helping hands can be very useful. A traditional way to do this is with two crocodile clips mounted on a stand, often with a magnifying glass attached. I personally prefer the SparkFun Third Hand, which comes with two crocodile clips on flexible stands. I have added two additional crocodile clips to mine, two of which are padded with heat-shrink tubing and the other two are bare crocodile clips. Figure 10-5 shows my setup with a wire ready to be soldered to an LED.

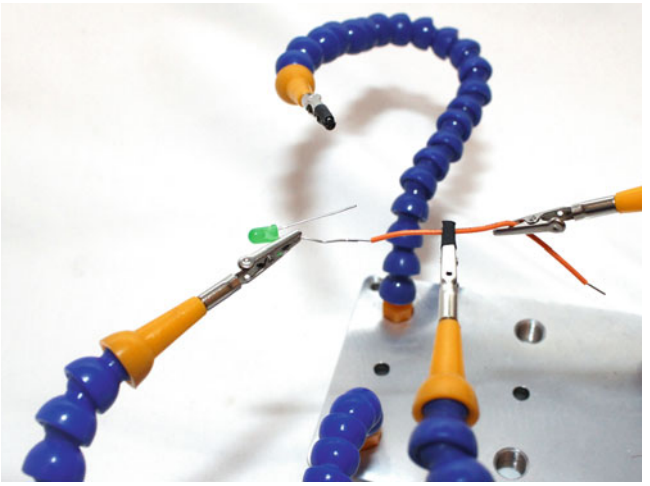


Figure 10-5. *SparkFun Third Hand used to solder a wire to an LED*

With careful positioning, it should be possible to touch the soldering iron to both the LED and wire and then apply the soldering along the two to form a joint. If you need to insulate the leads, some heat-shrink tubing can be placed over the wire before soldering and then moved into position.

Heat-shrink tubing is hollow tubing that can be placed over a wire and will shrink when heated. A hot air gun (such as the ones used as paint strippers) should be used to shrink the tubing.

Stripboard

You will look at making a custom PCB in the next chapter, but when you're getting started, stripboard is often cheaper and easier. Stripboard (also known by the brand name Veroboard) is a circuit board with copper strips or pads on one side. These are a permanent equivalent of the breadboard. In fact, stripboard is available that follows a similar layout to the breadboard you have been using. Figure 10-6 shows some stripboard in a variety of shapes and sizes.

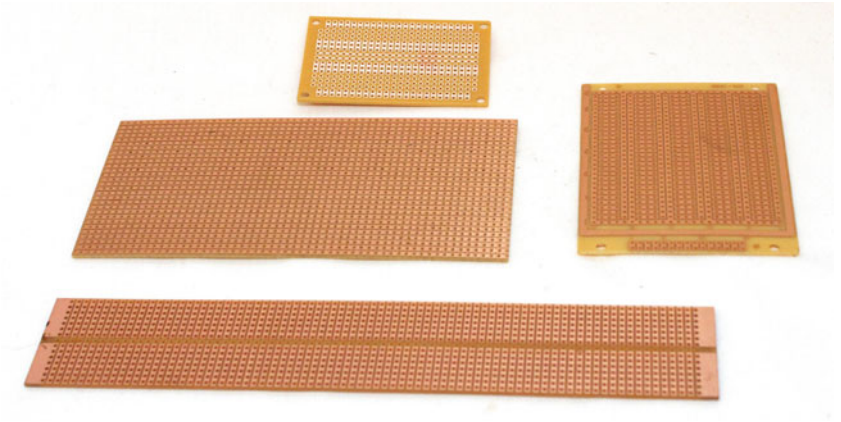


Figure 10-6. *Stripboard*

The components are positioned across the rows much like you would create your breadboard circuit, but with soldering required to join each component to the copper strip. Some designs of stripboard have strips that are broken in a central row, which is good for mounting integrated circuits. Otherwise, you can use a stripboard cutter. A stripboard cutter is a hand tool that sometimes looks like a drill with a screwdriver handle. These can be twisted against the track to remove the copper strip over one position at a time.

Perfboard

Perfboard is similar to stripboard, but without any of the copper pads joined together. The copper pads can be used to hold the components in position, but then wires need to be added to join the components. I personally recommend stripboard over perfboard, as I find it can be difficult to get tidy joints when using perfboard.

Raspberry Pi Prototyping Boards

One drawback of stripboard and perfboard is that they don't lend themselves to mounting on top of the Raspberry Pi. One problem is that the connector to fit on the GPIO connecting needs to be mounted on the underside, but it's also difficult to get it positioned just right. They are useful for circuits that are mounted away from the Raspberry Pi and can still be connected to the Raspberry Pi using wires or a cobbler.

A better solution is a prototyping board designed to connect on top of the Raspberry Pi. I have taken two of the circuits you used previously and mounted them on top of a prototyping board. The first uses a Slice of Pi, which is a perfboard-like add-on. I used this to create a version of the NeoPixel MOSFET-based circuit from Chapter 6. This is shown in Figure 10-7 and the underside shows why I am not such a fan of perfboards.

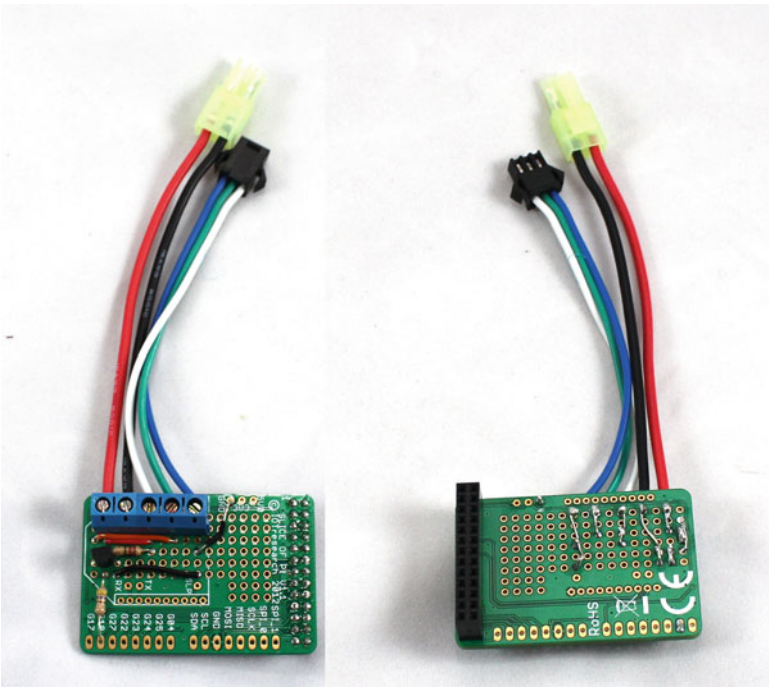


Figure 10-7. *NeoPixel MOSFET circuit on a Slice of Pi add-on board*

My preferred prototyping board is the Adafruit Perma-Proto Pi HAT. This is designed specifically for the Raspberry Pi B+ or later and includes an EEPROM, which the Raspberry Pi can use to set the appropriate input and outputs on the GPIO ports. Figure 10-8 shows the infrared receiver and transmitter circuit from Chapter 5. You will see that I have squeezed the components up to one end as I plan to add some sensors to the rest of the board in the future, but if you want to build a similar circuit, I suggest spacing the components out more similar to the breadboard layout.

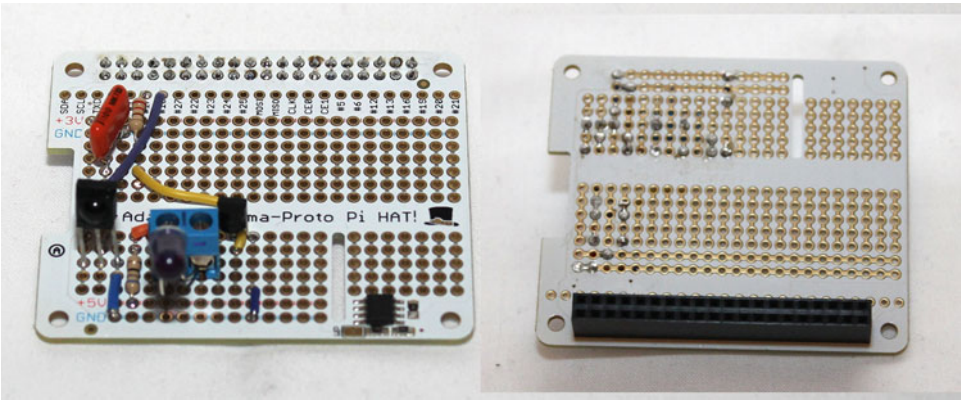


Figure 10-8. *Perma-Proto Pi HAT with infrared transmitter and receiver circuit*

Cases and Enclosures

Once you have your permanent circuit, it is often a good idea to put it in a box or enclosure to protect the circuit and make it look a little more professional. If you have access to a 3D printer, you can print your own case, but many people need to make use of what is readily available. Usually that involves a simple box shape, although I've also used a metal rack-mounted enclosure for my own version of the disco lights project from Chapter 4.

The easiest material to use for an enclosure is plastic, as it can be easily cut and shaped to fit the required components. I created an enclosure for the True or False game from Chapter 5, which is shown in Figure 10-9.

Figure 10-9. *Finished True or False game*

This is based on a cuboid box with a Pi Zero inside with the level shifter wired directly to the GPIO connector. The Pi Zero is attached inside the case using double-sided sticky pads and holes made into the box for the power lead as well as the HDMI connector (if required). I replaced the mini buttons from the breadboard circuit with panel-mounted pushbuttons, which fit into holes I drilled into the front panel. The hole for the LCD display was cut using an electric rotary tool. It has been finished off with labels from a Brother labeling machine.

Testing Tools

So far we've been assuming that everything works first time. While that is the aim, sometimes things go wrong and you'll need to run some tests. This is the electronics equivalent of debugging software. There are two tools that you will look at. The most important tool is the multimeter, which is an essential piece of equipment for an electronics toolkit. The second is a bit more advanced, which is a Raspberry Pi based digital oscilloscope.

Multimeter

A multimeter is a test tool that can take a number of different measurements on an electronic circuit. These can vary in price considerably depending on the required functionality and accuracy. My primary multimeter is a reasonable quality meter shown in Figure 10-10. A less expensive meter should be sufficient for most purposes, although I recommend looking for one with an audible connectivity tester (sometimes missing from the cheaper ones). It's not essential but can make it a little easier when you don't have to look at the screen when testing for a broken connection.

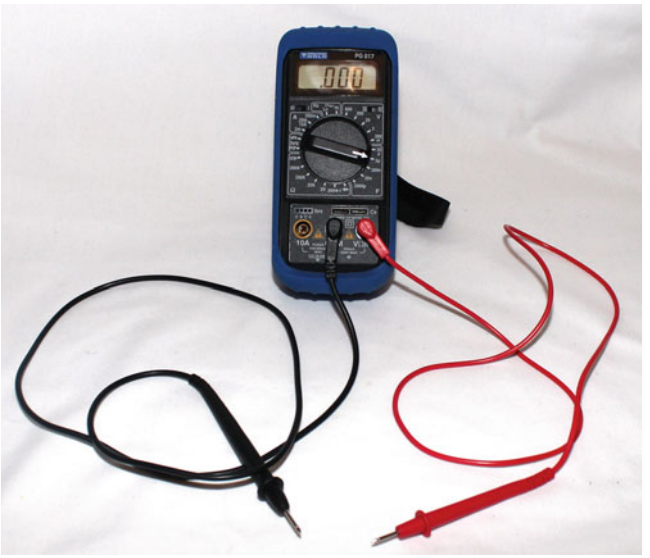


Figure 10-10. *Multimeter*

Across the bottom of the meter are three connectors for connecting the test leads. Two of these have test leads connected, which is the most common setting. The black lead is connected to the central common connector and the red lead is connected to the VΩmA (voltage resistance and milliamps). Most of the testing can be done with the probes in the positions, with the red wire being moved to the left connector when you're measuring large currents.

The rotary selector is then used to choose the appropriate measurement. The most common settings are:

- Voltage—Connect the black lead to the ground of a circuit and then use the red lead to measure the voltage at a certain point. The circuit needs to be connected to a power supply when testing the voltage.
- Current—Measuring the current involves making a break in the circuit and connecting the red and black leads across the break. The current will then flow through the ammeter where it is measured. For small currents, the leads are used in the normal positions, but for larger currents (above 200mA for this model), the other lead position should be used. There is often a fuse that may blow if you exceed the current rating.

- Resistance—To measure the resistance of a particular component or wire, first disconnect the power and place the probes across the component. If the component is in a circuit then other components may affect the measured value. Do not use the resistance setting on a live circuit.
- Continuity—On many multimeters, the lowest resistance range (typically 200Ω) often includes an audible connectivity tester. This can be useful for testing for short-circuits or breaks in a connection, as you don't need to watch the display.
- Other measurements—Some multimeters also include additional features. The one shown in Figure 10-10 includes a transistor tester, capacitor measurement, and frequency setting.

Some of the measurements have several different positions with the maximum rating on them. When measuring voltage and current, you should not exceed the value selected. If you are not sure, start with a higher value and then move the selector to a lower setting until you are in the correct range.

On this particular multimeter, there is a separate switch for use on AC and DC circuits, although for other multimeters this may be part of the rotary selector.

Oscilloscope

A limitation of a multimeter is that it can only show a single value for the voltage and cannot show how that changes over time. An *oscilloscope* (*scope* for short) is a measurement tool that can be used to show how the voltage at a point in the circuit changes over time. It used to be that these were too expensive for electronic hobbyists, but computer-based scopes have been introduced that are more affordable. In particular, the BitScope Micro is a low-cost scope that can be connected directly to a Raspberry Pi. The BitScope connects to a USB port, thus providing a way to connect two standard oscilloscope channels or up to eight connections when used as a logic analyzer. The BitScope is shown in Figure 10-11.

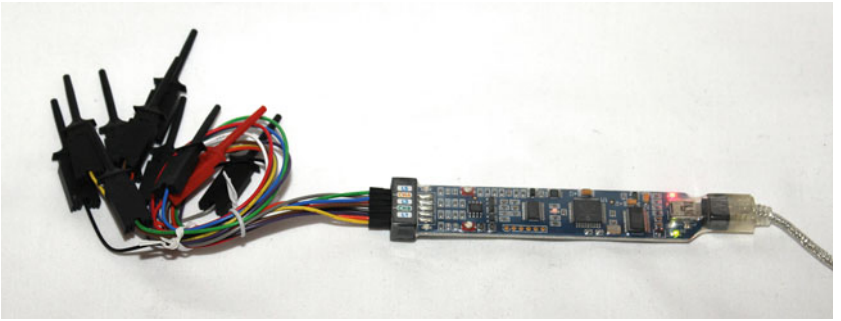


Figure 10-11. *BitScope Micro with test leads*

The software that runs on the Raspberry Pi is shown in Figure 10-12. In this case, it shows a square wave signal.

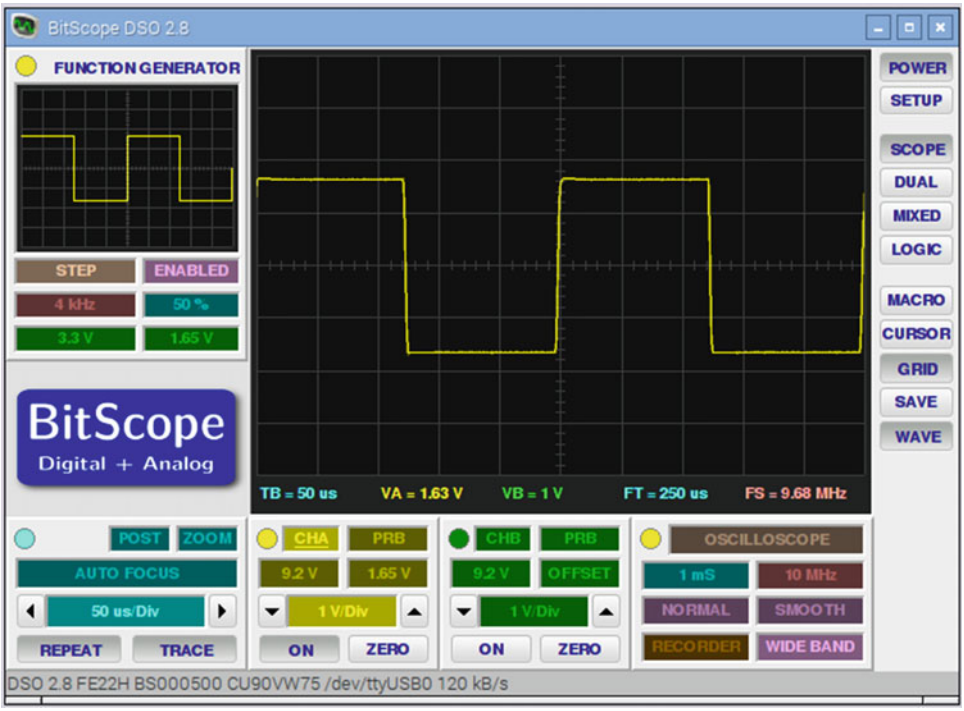


Figure 10-12. *BitScope DSO running on a Raspberry Pi*

More Project Making

This chapter looked at how you can use soldering to turn your temporary breadboard circuits into a permanent circuit. When getting started, stripboard is a good way to build circuits without the expense of a custom printed circuit board. In particular, the Perma-Proto Pi HAT is useful for creating circuits that can be mounted on top of the Raspberry Pi. You can find more information about soldering, including a video guide, at `www.penguintutor.com/electronics/soldering`.

You then looked at how to make the circuit look more professional by putting it in an enclosure. Finally, you looked at some test tools that are useful when the circuit doesn't work as expected.

In the next chapter, you will look at how you can design and make your own circuits, including how to design a circuit in Fritzing that could be made into a customer printed circuit board.

CHAPTER 11

Let the Innovation Begin: Designing Your Own Circuits

In this chapter, you will look at how you would go about designing your own circuits. Designing your first circuit may sound like a big jump compared to following existing designs, but you have already learned many of the basic elements used in many electronic circuits. Many electronic circuits are built from the basic building blocks you have read about, and many are modular with larger circuits designed by combining these small circuits into a larger one. This is also how integrated circuits are designed, and in fact a computer processor with millions of transistors starts with a switch circuit similar to the MOSFET switch you created in Chapter 4.

There is a good chance that there is already an integrated circuit that includes some of the functionality that you are looking to implement. Searching a component supplier's web site or a general Internet search may lead you to an integrated circuit that you can use in your own design.

This chapter will show you where to find information about components and circuits, will explain some tools for designing circuits, and will describe how they can be used to create circuits, including custom printed circuit boards (PCBs). Finally, you will see how to create a power supply suitable for powering a Raspberry Pi.

The Design Process in a Nutshell

Designing a circuit is usually a multi-step process. First you start with the idea, then you research the available components, and then, having decided on the components, you design it into a circuit showing how they will be connected. You can then prototype the circuit by making a temporary circuit before creating the final finished one. The final circuit could be built on an off-the-shelf board such as stripboard or made into a complete printed circuit board, depending on your budget and the complexity of the circuit.

Each of these stages can be repeated as necessary until you come to the final design. As you move through the stages, the potential cost increases both in terms of money and the time, so the earlier you identify any potential problems the less it will cost. Don't be afraid to go back to the start rather than trying to continue with a design that isn't working.

The prototyping stage is useful, as this is when you get to see if the circuit you designed works. This is often done using a breadboard. You don't necessarily need to test every single component at this level, but you should test all the basic blocks. For example, when designing the disco lights in Chapter 4, I tested only one of the MOSFET switches. The other three followed the same design, so I did not need to test them separately. The more comprehensive your testing, the better your chance of identifying any potential problems.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3_11

There is one additional step that is used in professional circuit design and that's to perform a circuit simulation. Circuit simulation involves computer modeling of the behavior between components to see how they will work together. It can be a complicated process and should not be required for the type of circuits you will be making at this stage.

The idea is something you can come up with yourself, although hopefully some of the ideas in this book may help inspire you. This chapter therefore looks at some of the sources of information and the tools you can use to design a circuit.

Looking at Manufacturer Datasheets

When designing circuits, you need to understand the particular characteristics of the components you use. Components that look similar to each behave in different ways when used in practice. To get this information, manufacturers provide a datasheet that explains the characteristics of their components. Interpreting the information from a datasheet is one of the key skills you need when designing your own circuit.

Datasheets for different types of components may be completely different, but there are a number of sections that are often included. Rather than covering just one type, I have created a few examples of the sort of thing that you may see on a datasheet. I suggest you download some real datasheets so that you can get familiar with their content. The best place to download datasheets is from the component supplier. Some companies, such as Farnell/Element 14, provide links to the datasheets in the component listings, others include practical examples and tutorials, and some don't provide any information. If you don't find what you are looking for, search the Internet for datasheet followed by the component name.

The datasheet includes a descriptive title, the component model number, and the name of the manufacturer. This is normally followed by a paragraph explaining what the component is and where it is useful. In some cases, a single datasheet may include multiple components (for example, the TSOP2438 infrared receiver shares a single datasheet with 23 other similar receivers).

There is often a diagram showing the pin layout. Obviously this will look different for a discrete component such as a transistor than for an integrated circuit. For a transistor or something similar, it may be a 3D picture with labels on the leads. In the case of an integrated circuit, it may look like the example shown in Figure 11-1.

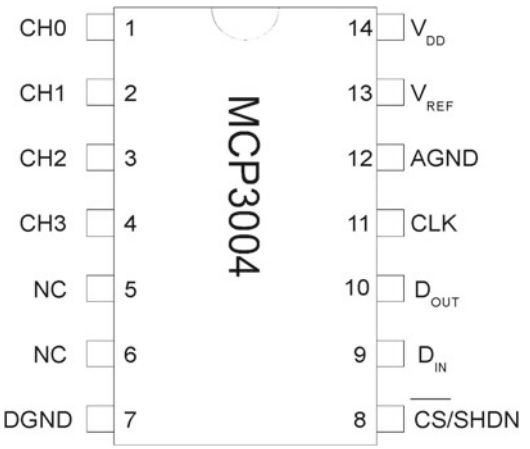


Figure 11-1. *Pin layout for the MCP3004*

This is the MCP3004 analog-to-digital integrated circuit that you used in Chapter 5. This is a good example of the different types of pin labels that you may come across.

The numbers on the inside of the IC are the pin numbers. Pin 1 is indicated at the top right and it's normally identified on the chip by a semicircle or small dot on the side with pin 1. CH0 to CH3 are the input channels (analog inputs). NC means not connected. This normally means that it is not used at the moment and so the pin should be left without physically connecting to the rest of the circuit. Normally GND is used to signify the pin that needs to be connected to the 0V ground part of the circuit. In this case, it has a separate ground connection for the digital and analog parts of the circuit indicated by DGND and AGND, respectively. An alternative is V_{SS}, which is used by some manufacturers to refer to the ground instead.

The pin marked CS/SHDN is used for chip select or to shut down the input. An alternative that you may see on some components is EN, which stands for enable and normally means the same as chip select. You will see that there is a line above CS. This indicates that it is inverted, so a high input has the opposite meaning as the wording. In this case, "chip select" happens when it receives a low signal rather than a high signal.

The pins labeled D_{IN} and D_{OUT} are the data in and out pins that are used to communicate with the Raspberry Pi using I2C. CLK stands for clock, where a timing signal is required. The V_{REF} pin is used for the analog part of the circuit as a reference voltage that the input is compared against. Finally, the V_{DD} pin is connected to the positive power supply, and on some circuits this may be V_{CC} instead.

The pin layout may also show different variants of the component. This may include an option for a surface mounted device (SMD) or through the hole device. You will normally want to avoid the SMD components when making the circuit by hand, as they can be difficult to solder.

The next part of the datasheet is normally a table of absolute maximum ratings, which means you should not exceed these values. Going outside of these ranges can cause the component to operate incorrectly or may cause permanent damage to the component.

Figure 11-2 shows part of the information for the IRL520 MOSFET.

ABSOLUTE MAXIMIM RATINGS (T_C = 25°C unless otherwise noted)					
PARAMETER			**SYMBOL**	**LIMIT**	**UNIT**
Drain-Source Voltage			V_{DS}	100	V
Gate-Source Voltage			V_{GS}	± 10	
Continous Drain Current	V_{GS} at 5V	T_C = 25°C	I_P	9.2	A
		T_C = 100°C		6.5	
Pulsed Drain Current			I_{DM}	36	

Figure 11-2. *Absolute maximum ratings for the IRL520 MOSFET*

This shows the maximum voltages between different pairs of pins as well as the continuous and pulsed currents. Exceeding these values is likely to damage the MOSFET. Also note that some values may depend on certain conditions and in this case the maximum drain current is less as the temperature increases.

There are then often several tables that provide more information about the behavior of the component under certain conditions. For logic circuits, this will include the valid voltage ranges for a signal to be considered as a true or false; for a transistor, this includes the typical gain; and for an LED, it may include the luminous intensity. There may be other information that is useful under certain circumstances, such as timing information or effective capacitance.

Some information may vary depending on the input, temperature, or other characteristics. In this case, the information may be presented as graphs or timing diagrams.

Another common section is a circuit diagram that shows the internal characteristics of the inputs or outputs. Use these when you're deciding what additional components may be needed to connect the circuit to another component. For instance, if a circuit has an open collector output, as shown in Figure 11-3, if you're connecting to a logic circuit, it would need a pull-up resistor to ensure that the input to the next stage was not floating when the transistor was switched off.

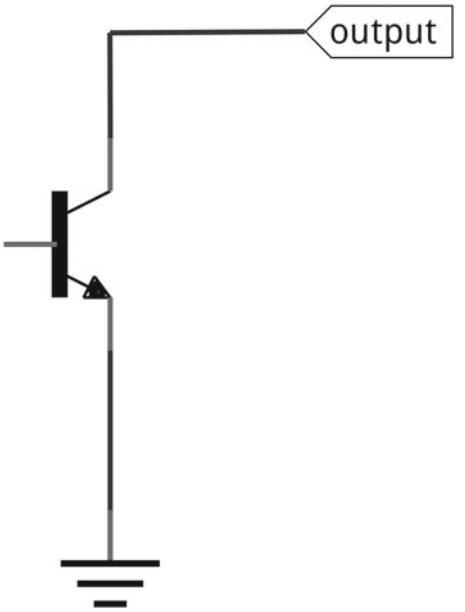


Figure 11-3. *Open collector used on certain ICs*

Figure 11-4 shows the SN754410 H-Bridge IC, which includes the effective output circuit from the chip and the protection diodes.

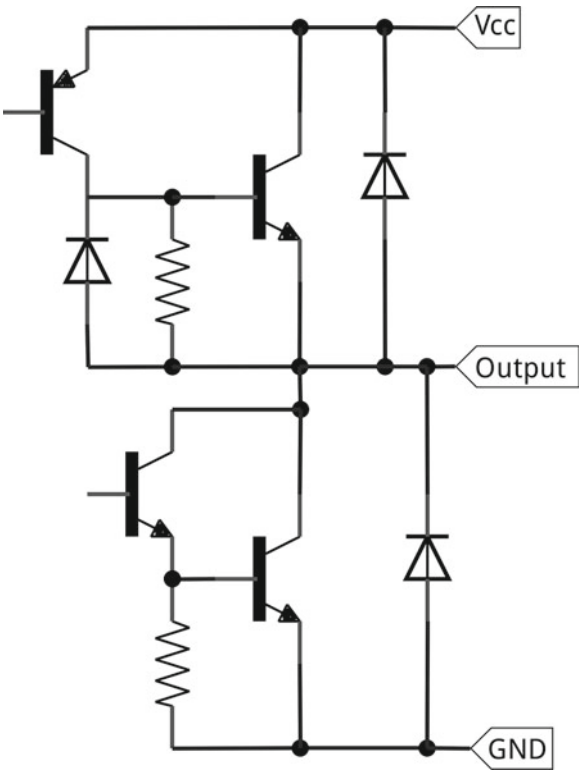


Figure 11-4. *Datasheet representation of output from SN754410 H-Bridge IC*

The datasheet may also include examples of how the component can be used in a circuit and often includes the dimensions of the component and how it connects to a printed circuit board.

Some datasheets will be more useful than others, but as you can see, it can be an invaluable source of information when designing a circuit.

Designing with Fritzing

Having read this far, you may be wondering how the circuit diagrams and breadboard images were created. The majority of these were created using a program called Fritzing. Fritzing is open source software available for Linux, Mac OS X, and Windows and can help you create circuit diagrams (schematics), breadboard layouts, and printed circuit board designs. You can download Fritzing from `http://fritzing.org/download/`. The site requests a donation to support the further development of the software, but you can download without giving a donation. It should be possible to download the source code and compile the program to run on a Raspberry Pi 2 or Pi 3, although it is easier if you choose one of the pre-compiled programs instead.

Designing a Circuit Diagram/Schematic

Although the breadboard view is the first tab in Fritzing, I recommend starting with a circuit diagram, which is called a schematic in Fritzing. This allows you to design the circuit as you would like it to connect logically and it will then guide you to creating the physical layout for a breadboard or PCB. The initial schematic view is shown in Figure 11-5.

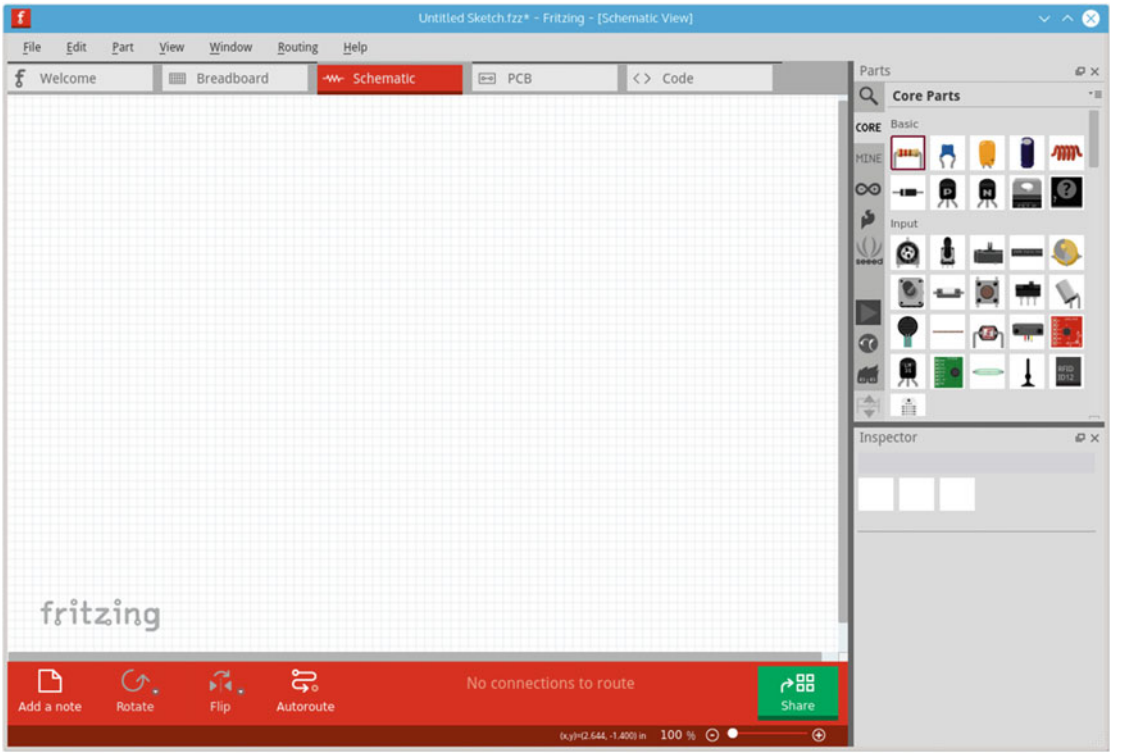


Figure 11-5. *Fritzing schematic view*

The main part of the screen contains an editor area, which is a white background with a grid for laying out the circuit design. The Parts area in the top right contains the available components and circuit boards and the Inspector area at the bottom right allows you to change any of the parameters relating to the current component.

In the Parts selector, most of the standard components are included under the CORE tab. This includes common components such as a resistor and PCB, but also labels, wires, and even an entire computer in the form of the Raspberry Pi. The components can be dragged across to the Edit area. If the component has some options available, you can select these in the Inspector area. This can be used to select values such as resistor values and different component types such as THT (through-hole technology) or SMD (surface mount devices). When presented with the choice of THT and SMD, you will usually want to choose the THT components. They have leads that can pass through a printed circuit board or mounted into a breadboard. SMD are very small components that are mounted to only one side of a PCB and are very difficult to solder by hand without additional tools. The inspector panel for a resistor is shown in Figure 11-6.

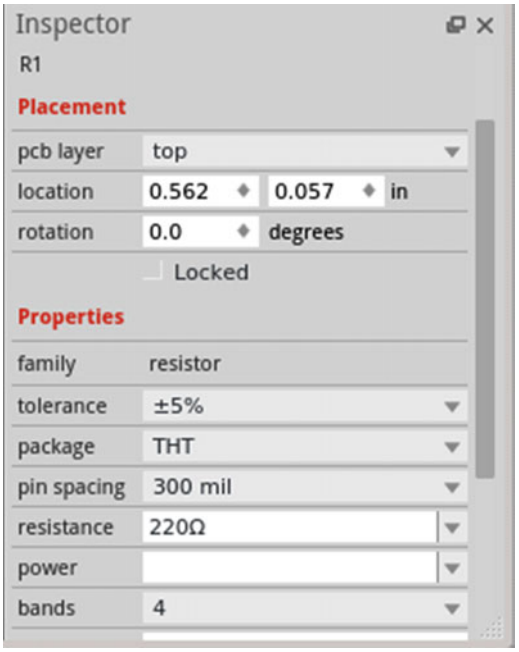


Figure 11-6. *Fritzing inspector panel for a resistor*

You then connect the components by dragging the mouse from the lead of one component to the other. Each time you create a connection between the components, it is called a net. You can connect more components to the same net by right-clicking on a connection and choosing Add Bendpoint. You can also connect to a net label (under the core components), at which point any other net labels with the same name are also connected. A complete circuit for the basic transistor LED circuit is included in Figure 11-7.

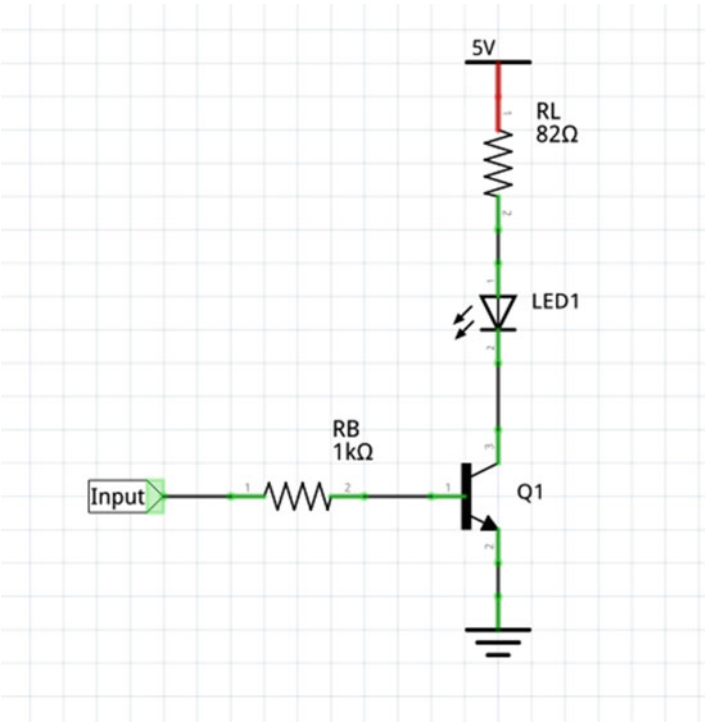


Figure 11-7. *Transistor LED circuit schematic view*

I haven't included the Raspberry Pi on this diagram, although you can include one by dragging it from the core components. In this case, I used a net label at the point where the circuit connects to the GPIO, instead of showing the Raspberry Pi on the diagram.

Design Conventions

When creating a circuit diagram (in Fritzing or otherwise), there are a few standard conventions that are followed when possible. These are not fixed rules, but should be followed when possible. Some common rules are:

- The positive supply is usually placed at the top of the diagram.
- The ground (0V) supply is normally placed at the bottom of the diagram.
- Inputs are normally positioned toward the left and outputs toward the right. As a result, the data signals normally flow from left to right.
- If a battery is being used, this may be displayed using the battery symbol, but often the supply is shown by marking the appropriate lines instead.
- Lines are normally drawn horizontally or vertically straight. A change in direction is normally through a 90-degree turn. In Fritzing, the connections will normally take the shortest path, but can be dragged out to form 90-degree turns.

- Avoid crossing lines where possible. When lines join, a dot (filled circle) is used to indicate a join. When lines cross but don't join, they normally pass in a straight line.
- Each component is normally given a reference. This is prefixed with a letter or few letters to denote the type of component. Examples include:
  - R for resistor
  - D for diode
  - Q for transistor
  - BT for battery
  - C for capacitor

Creating a Breadboard Layout

Now that you have a circuit diagram you can convert it to a breadboard layout by switching to the Breadboard tab. If you do so, it will likely look a bit of a mess, such as the diagram in Figure 11-8.

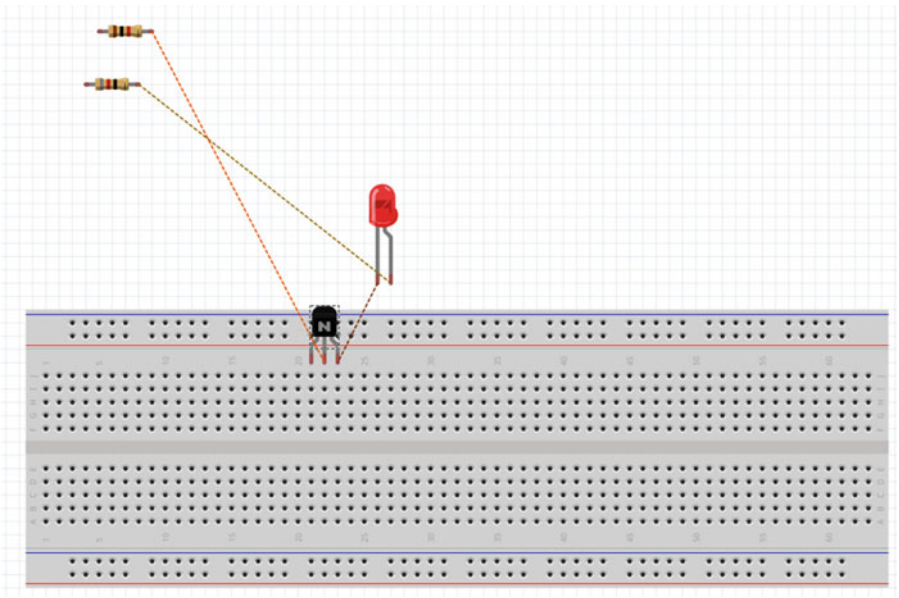


Figure 11-8. *Fritzing initial breadboard view*

Not only does this look a mess, but for some circuits, components will be on top of each other, thereby hiding those below them. First you should click on the breadboard and use the inspector to set it to the appropriate type of breadboard. The most common type is the half+ breadboard.

You can now drag the components to an appropriate position, rotating them as necessary. When you place a component onto the breadboard, the row of pins that it will connect to will change to green. The dashed lines represent the nets that you created in the circuit diagram. Clicking on one of these will turn it into a wire that you can place as appropriate. You also need to add connections for the positive and ground power supplies, which are normally connected to one of the appropriate red or blue strips along the breadboard (not all breadboards include the colored lines). The final layout is shown in Figure 11-9.

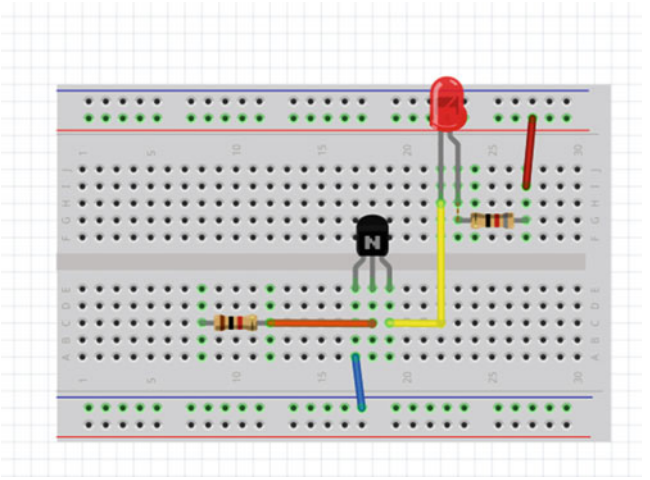


Figure 11-9. *Fritzing updated breadboard view*

The resistor that goes to the base of the transistor normally connects to the Raspberry Pi, but I have not shown the Raspberry Pi in this view.

Creating a Stripboard Layout

If you want to create a layout for a stripboard (see Chapter 10), you do that using the same breadboard view. You will lose the view of the breadboard so you may want to save it with a different name first. Click on the breadboard and choose Delete and then drag the stripboard part from the core parts. There are only a couple of different types of stripboard included, but you can adjust the number of columns and rows. You can also create breaks in the rows by clicking on the stripboard between a pair of holes.

Designing a Printed Circuit Board

The final part you will look at is creating a professional printed circuit board (PCB). This is not something you should consider lightly as, creating custom PCBs can unfortunately be expensive, but it can provide a professional look for your project. The price for PCBs varies depending on manufacturer, lead-time, and quantity. The quantity is an important factor, as producing only one or two copies of a PCB is very expensive, but if you want to create hundreds, the cost per board is significantly less.

While I don't recommend you have a PCB made for this simple circuit, I will still use it to illustrate the process of designing a PCB. Clicking on the PCB tab will show a grey area for the PCB board. You will find all your components stacked up on the top corner and they will need to be moved around.

First set the size of the PCB and move the components to appropriate positions on the board. Before you go about working out how the copper will be laid out you need to add some way of connecting to the power supply and to the Raspberry Pi. For the power supply, you can use one of the included power connectors and for the Raspberry Pi you may want to include a 30-way connector. However, these would be overkill for such a simple design, so I am instead just going to add three terminals—two for the power supply and one for the GPIO connection. I will use the part for a terminal connector, but you could just solder wires directly to the pins rather than using a physical connector.

Under the core parts, look for connections and for Generic Female Header—2 pins. The number of pins can then be changed in the Inspector tab. The PCB so far is shown in Figure 11-10.

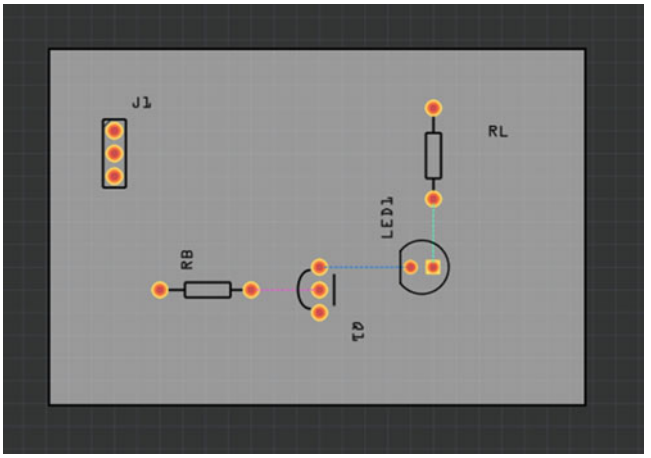


Figure 11-10. *Initial PCB layout without routing*

The next stage is to add the copper tracks that go between each component. There is an Autoroute option, but you can just do this manually as you did in the breadboard layout using the dotted lines as a guide. There will be no connections shown to the connector terminal so you will need to add these as well.

Before you start dragging out the connections, you may want to specify which layer to run the connections on. While it technically doesn't matter whether these are on the top or the bottom, I prefer to put as many on the bottom as possible. This is more due to my experience creating single layer boards in the past. To do this, look for the icon on the bottom that says Both Layers and choose Set Bottom Layer Clickable. Any tracks you draw will be orange, which indicates that they are on the bottom. If you change it to the top layer, they will be colored yellow.

While you could have the tracks going diagonally directly to the components, I prefer to lay the tracks in straight lines and at 90-degree bends when possible.

Once the tracks are connected, you should have something looking similar to Figure 11-11.

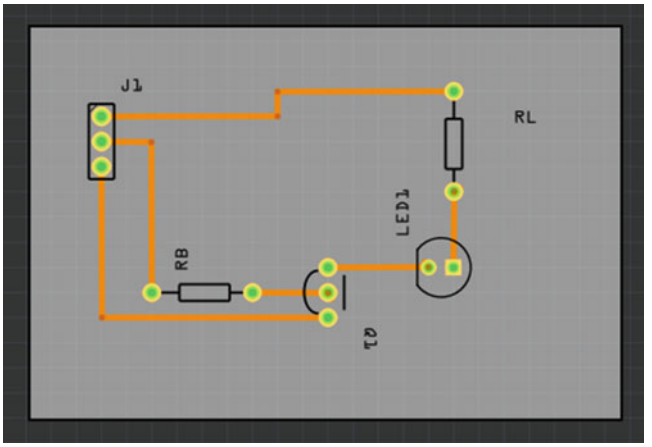


Figure 11-11. *PCB layout with copper tracks between the components*

If you now switch to the schematic view, you will see that the connector component has been added and that it has dashed connections to the power supply and resistor RB. This is one of the features of Fritzing, as you can add a component in any of the different views and it is added to all the others. You may want to tidy the other diagrams up by repositioning the connector and wires. Alternatively, if you didn't want to show the wires, you can use the Delete Ratsnest Line option to hide the dashed line. Take care when using this option, as it is a useful way to check that you have connected your circuits correctly.

It's a good idea to rearrange the text labels for the components at this point and then label the connections for J1. You can add text using the logo part, which can be used for adding any kind of text, not just logos. The updated PCB is shown in Figure 11-12.

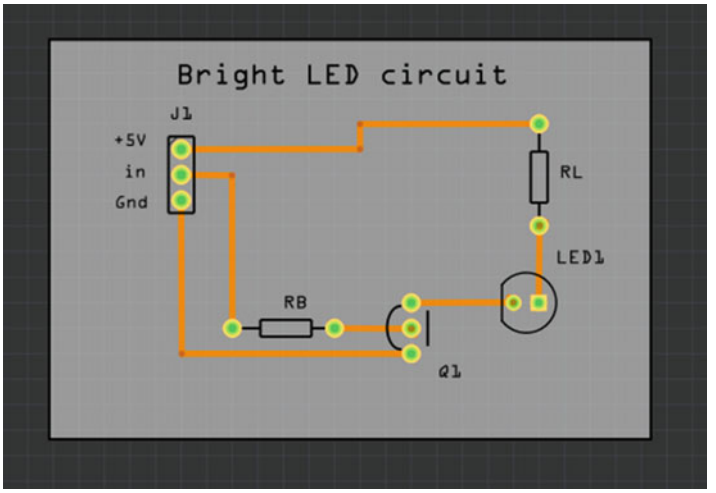


Figure 11-12. *PCB layout with added text information*

This is almost ready to be made up as a printed circuit board. The final thing you should do to the design is add a ground or copper fill. This will fill most of the PCB with copper. In the case of a ground fill, that will be connected to the ground which helps avoid electromagnetic interference. Some manufacturers, including Fritzing, automatically apply a copper fill, as it saves on the amount of copper that needs to be etched, but it's a good idea to do so yourself.

To add a ground fill, right-click on the Gnd Connection and choose Set Ground Fill Seed. Then from the Routing menu, choose to apply a ground fill. The entire board will then be filled in with copper except for gaps around each of the connections. If you look at the bottom layer, you should see that the ground connection from the connector to Q1 is merged into the ground fill.

The PCB is now ready to be sent for manufacturing. Before you send it off, it is a good idea to perform some checks on the finished layout. First run the Design Rules Check (DRC) from the Routing menu, which will check for any common problems. Then export the PCB layout using File ä Export for Production and choosing PDF. Print the component layout and check that the components fit. Double-check that all the components are orientated correctly on the PCB layout.

Once you've checked everything twice, check the PCB once more before you send the files to the manufacturer. Remember the extra cost and time if there is a mistake. If you want to use the Fritzing lab (which funds the development of Fritzing software), you can use the Fabricate button to load a web page for submission. There are a couple of advantages to using Fritzing. The first is that you can just send the Fritzing file as it is and the second is that they will perform some additional checks.

If you would rather send the PCB to a different manufacturer, you can export it for production as Extended Gerber files. In this case, you may need to fine-tune some of these files. Check the submission rules, especially for what .TXT files they expect to receive.

As I have said, it is not worth the cost of sending this particular circuit to manufacturing, but I created a more complex PCB that I used in one of my own projects. Figure 11-13 is a circuit that combines the disco lights and NeoPixel circuits onto a single board. I used this circuit at a real disco.

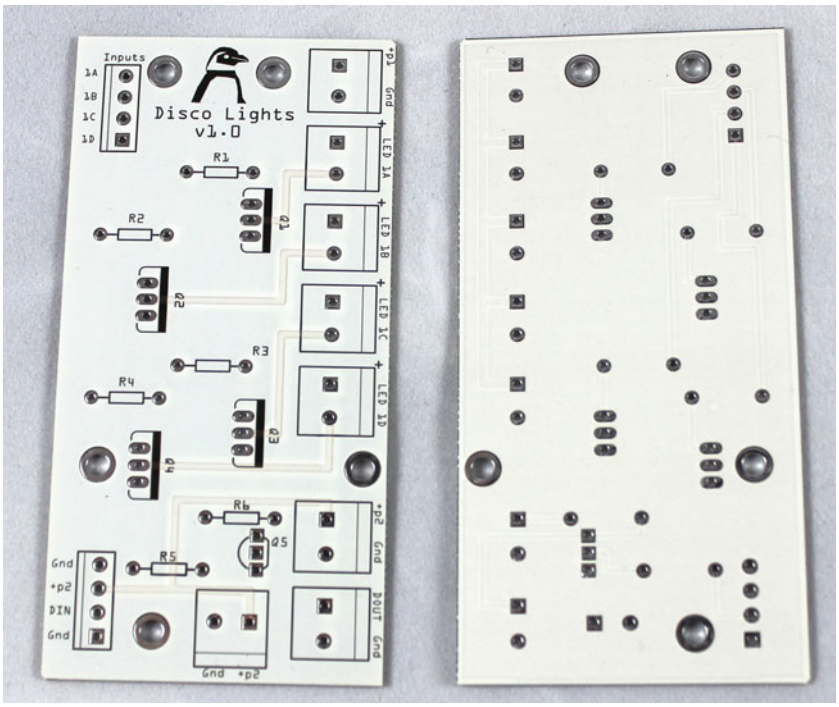


Figure 11-13. *Manufactured PCB for disco lights and NeoPixels*

Powering the Raspberry Pi

The Raspberry Pi is normally powered from a dedicated power supply through the micro-USB connector. This is fine for normal use, but what if you want to include the Raspberry Pi in a complete project that already has a power supply? In particular, when I created the disco light project which runs on 12V, I didn't want to have to add a second power supply for the 5V needed by the Raspberry Pi. First, I will show you a circuit that you can make yourself so that you can understand how to add a power supply directly into your own circuit. Then I will show you a more efficient solution where you can buy a converter and connect it to the circuit.

78xx Linear Voltage Regulator

The 78xx series of voltage regulars are small components that can be easily included directly onto a PCB or stripboard circuit. The regulators take a higher voltage on their input and provide a constant lower voltage for their output. In this case, 12V for the input and 5V for the output. The only extra components are capacitors at the input and output, which are there to smooth out any noise. The circuit diagram is shown in Figure 11-14.

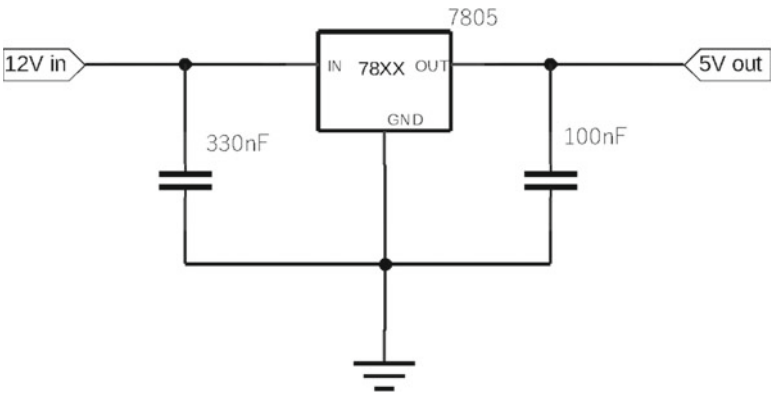


Figure 11-14. *7805 voltage regulator circuit*

The voltage regulator comes with a variety of output voltages, which are denoted by the last two digits. For example, the 5V regulator suitable for the Raspberry Pi is the 7805. The input voltage needs to be higher than the output voltage, so in the case of the 7805, it needs to be at least 7.3V.

One disadvantage of a linear regulator is that the voltage difference between the input and output is turned into heat, wasting power. If this is supplying 600mA (a typical current draw for a Raspberry Pi) using a 12V power supply then the amount of power lost would be 4.2 watts (7V dropped x 0.6A). This is quite a significant amount of heat and a heatsink would be required to prevent the regulator from overheating.

Buck Converter

A more efficient way of powering the Raspberry Pi from a 12V power supply is by using a buck converter, also known as a DC-DC converter or a DC switching regulator. I used one of these to power the Raspberry Pi in the disco light project. The buck convertor is normally bought pre-built on a printed circuit board. The output voltage can usually be changed with a trimmer built onto the PCB. You need to use a multimeter to check the output voltage. A buck converter is shown in Figure 11-15.

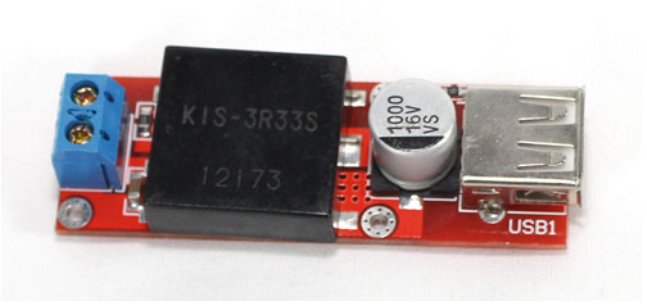


Figure 11-15. *Buck converter*

Buck convertors are available through various suppliers such as Amazon and eBay. These are more expensive than the linear voltage regulator and they take up more space, but they are better when it comes to power efficiency.

Designing More Circuits

This chapter discussed the information needed to design your own circuits. You saw how datasheets provide the information needed when designing a circuit and how to extract the relevant information. You then looked at Fritzing and how it can be used to design circuits, including creating a schematic layout, designing a breadboard and stripboard layout, and finally creating a professional printed circuit board.

This chapter provides the basics for getting started with Fritzing, but only scratches the surface of what Fritzing is. I encourage you to spend more time learning about Fritzing. There are also other circuit design software programs, including KiCAD (which is open source software) and Eagle PCB design software (which is a commercial product but free to use for small circuits). Neither of these provide the ease of use and breadboard layouts that Fritzing does.

Through this book you have learned about electronic circuits, about how to connect them to the Raspberry Pi, and the steps involved in designing your own circuit. With this information, you can now design your own electronic circuits to have the Raspberry Pi interact with the real world.

Look around at what others have made, come up with your own ideas, and design your own electronic circuits.

APPENDIX A

Required Tools and Components

This appendix includes the most common components and tools you will need for each of the projects in the book. There are suggested design alternatives for some of the projects so you may want to explore other components. I also list suggested sizes or types for some of the tools, which should be considered only as a guide. It's often a good idea to start with a small set of basic tools and then expand that collection or add better-quality tools as you gain more experience.

I've listed the tools in a few groups, which will allow you to add to your tools as you progress. I've listed the components for each of the projects, some of which may have already been used by other projects.

Tools Required

When getting started then only a few basic tools are required. This includes a breadboard to build the circuits on and a pair of side cutters to cut and strip wires.

When making permanent circuits then the number of tools required does increase, although some of these are normal DIY tools that you may already have or that you may find useful around the home.

Basic Breadboard Circuits

These are the recommended tools in Chapters 1 to 7 (and later):

- Raspberry Pi (preferably Raspberry Pi 2)
- Breadboard (half size)
- Side cutters (small wire cutters)
- Crocodile clips with wires
- Jumper wires and/or solid core wire
- Small screwdriver (cross-head and straight, or multi-head)
- Cobbler (optional)
- Board to mount the Raspberry Pi and breadboard (optional)

Crimping and Soldering Tools

A crimping tool is useful in Chapter 3; the rest of the tools are used in Chapters 10 and 11.

- Combined crimp tool and wire strippers
- Soldering iron with suitable tip (usually a fairly small chisel tip)

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3

- Soldering iron stand and cleaner
- Solder (normally lead-free)
- Stranded wire (not essential, but more flexible than the solid core used for breadboards)
- Heatshrink tubing
- Stripboard
- Spot face/stripboard cutter

Manufacturing Tools for Enclosures

Most of these circuits are designed to be made up and then taken apart. The exception is the arcade game in Chapter 3. You may also find that you do want to make a permanent version of some of the projects, such as the True or False game in Chapter 5. Ultimately, these will be useful when you go on to create your own projects in future.

- Safety glasses or goggles
- Electric drill
- Small hacksaw
- Small files (needle files)
- Permanent marker pen
- Rotary tool (optional)

Meters and Test Equipment

While not essential, an inexpensive multimeter can be very useful for all types of circuits. A BitScope would be a luxury at this stage, but may be useful in future. Chapter 10 includes the use of these tools.

- Multimeter
- BitScope (optional)

Components for Each Project

As with the tools the number of components can start small and then increase as you create the more ambitious projects. The components are listed against the project names they are used in, but you may also want to consider building up a stock of components for future projects. A set of resistors is a good start (either the E6 or E12 series listed in Appendix C) along with a few spare transistors and LEDs. You can then expand the collection by buying a few extra spares when you order components for your next project.

Chapter 1: Simple LED Circuit

- 9V PP3 battery
- 9V battery clip (with wires)
- LED (5mm any color)

- Miniature pushbutton switch (SPST)
- 470Ω resistor

Chapter 3: LED Circuit

- 5mm red LED
- 220Ω resistor

Chapter 3: Switch Input Circuit

These are required in addition to the parts in the LED circuit.

- 12mm push-to-make pushbutton switch.
- Optional switch cap

Chapter 3: Robot Soccer

These are a list of all the components needed, excluding those listed in the breadboard list, at the start of this appendix.

- 5mm red LED
- 5mm green LED
- 220Ω resistors (2)
- 12mm push-to-make pushbutton switches (2)
- Optional switch caps (2)

Chapter 3: Mars Lander

These are the parts required, excluding the Raspberry Pi.

- Box to mount the switches to (Really Useful Box 4 Litres)
- Joystick (micro-switch)
- Large button switch
- Arcade button switches (5 are used in example)
- Breadboard (optional)
- 26 or 40 way Raspberry Pi Cobbler (optional)
- Crimp terminals (red, female).

Chapter 4: Brighter LED

- 10mm white LED
- Transistor, 2n2222 or BC548
- 82Ω resistor
- 1kΩ resistor

Chapter 4: Brighter LEDs with Darlington Transistors

- USB LED light
- 5V USB power supply with suitable connector
- BD681 Darlington transistor
- 220Ω resistor
- 12mm push-to-make pushbutton switch

Chapter 4: Disco Lights

- PAR 16 theatre lights (4)
- MR 16 LED bulbs (4)
- 12V power brick with suitable connector
- IRL520 MOSFET transistors (4)
- 470Ω resistors (4)
- 5A fuse with holder or polyfuse

Chapter 5: PIR Sensor and Pi Camera

- Raspberry Pi camera
- PIR motion sensor (HC-SR501)

Chapter 5: Infrared Transmitter and Receiver

- TSOP2438 infrared receiver
- TSAL6400 infrared emitter
- 2N2222 transistor
- 68Ω resistor
- 100Ω resistor
- 220Ω resistor
- 0.1µF capacitor
- Infrared color-changing LED and remote control

Chapter 5: I²C LCD Display True or False Game

- Bidirectional level shifter (Adafruit/Sparkfun)
- LCD display
- I²C LCD display backpack
- Push-to-make switches (3)

Chapter 5: SPI Analog to Digital Input

- MCP3008 SPI ADC
- 1µF capacitor
- Potentiometer (variable resistor), 10kΩ or similar

Chapter 6: Automated Lego Train

The parts for the infrared transmitter are the same as those used in the infrared transmitter circuit in Chapter 5.

- Lego train or similar infrared device
- 68Ω resistor
- 220Ω resistor
- 2N2222 transistor
- TSAL6400 infrared emitter
- Reed switch and magnet

Chapter 6: NeoPixels

- 5V power supply and connector
- 470Ω resistor
- 2.2kΩ resistor
- 2N70000 MOSFET
- NeoPixels (2 breadboard NeoPixels or strip of NeoPixels)

Chapter 7: Video Capture

The infrared receiver components are the same as those used in Chapter 5.

- TSOP238 infrared receiver
- 100Ω resistor
- 0.1µF capacitor
- Selection of Lego figures or similar characters
- Background pictures

Chapter 8: Breadboard Based Robot

- Magician robot chassis with motors
- SN754410 H-Bridge IC
- Cobbler (optional)
- Battery power supply (four AA batteries or alternative 5V power supply)
- WiFi dongle for the Raspberry Pi

Chapter 8: Robot Using Ryanteck Motor Control Board

As with the breadboard-based robot, but with a Ryanteck Motor Control Board instead of the SN754410 and no cobbler.

Chapter 8: CamJam Robot

- CamJam Education Kit 3 (Robotics)
- WiFi dongle for the Raspberry Pi

Chapter 8: Robot with Ultrasonic Range Sensor

- Any of the robot kits listed previously
- Ultrasonic Range Sensor
- 39kΩ resistor
- 68kΩ resistor

Chapter 9: Minecraft Hardware

The Minecraft projects use the same enclosure, joystick controller, and switches as used in the 3 Mars Lander project in Chapter 3.

- 220Ω resistors (7)
- Tri-color common cathode LEDs; also known as duo-LEDs (3)
- Red LED
- Minecraft blocks (optional)

If you are unable to buy tri-color LEDs, you can use common cathode RGB LEDs instead.

Chapter 10: Permanent Circuits

The additional tools required are listed at the top of this appendix.

- Perma-Proto Pi HAT or Slice of Pi prototyping board
- PCB terminal connectors (optional)

To create a permanent circuit also requires the components listed in any appropriate project earlier in the book. Good examples are the NeoPixel controller and the infrared transmitter and receiver circuits.

Chapter 10: True or False Game

The following tools are in addition to the components from the True or False game in Chapter 5.

- Enclosure/box
- PCB mounting screws and spacers

Chapter 11: Powering the Raspberry Pi

- 7805 voltage regulator
- 330nF capacitor (also known as an 0.33µF)
- 100nF capacitor (also known as an 0.1µF)
- Heat sink (depending on the power requirements)

or

- Buck converter

APPENDIX B

Electronic Components Quick Reference

These are some of the common electronic components provided as a quick reference. Some technical details have been included for the common components, but these depend on the particular model. Check the datasheets for more details.

Resistors

A *resistor* is normally used to reduce the amount of current that can flow through a circuit. This protects a component from being damaged due to too much current. Resistors can also be used for dropping the voltage at a point in the circuit by creating a voltage divider.

The size of the resistor is measure in ohms, Ω, which is marked on the side of the resistor using a color code (see Appendix C). The resistors are available in certain values (depending on the resistor series) and so you should select the nearest common value.

Variable Resistors

As its name suggests, a *variable resistor* is a resistor whose value can change. They normally have three terminals, two providing the specified value of the resistance and a third that can be adjusted to provide a resistance value between the other two.

The variable resistors can be accessible to the user, such as the volume control buttons on a speaker, or can be small components that are hidden inside an enclosure away from the user. The ones that are accessible to the user often have control knobs and are often called *potentiometers*. The smaller variable resistors that are not accessible to the user are often used to calibrate a particular circuit are commonly called *trimmers*.

Switches

Switches create a break in a circuit or join two parts of a circuit together. This can be used to turn a circuit off by creating a break in the circuit or to direct the current through a different part of the circuit.

These are most commonly available as pushbutton switches, toggle or rocker switches, rotary switches (such as a key switch), or micro-switches.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3

The switches are known by the number of number of poles (individual switches within the package) and the number of throws (number of positions switched by each pole). Common examples are:

- SPST single pole single throw: On/off
- SPDT single pole double throw: Switch between A and B outputs
- DPST double pole double throw: Two on/off switches

Pushbutton switches are also known by whether the switch makes or breaks the circuit. The most common form is the momentary push-to-make, where the switch closes (completing the circuit) when it is pressed and then opens when the button is released. This is how a door bell switch works. They are also available as push-to-break, which is the opposite or locking/latching, where one press closes the switch and a second opens the switch.

Diode

A *diode* is a component that allows current to flow in one direction, but not in the other. It effectively acts like a one-way valve. There is normally a white line around the body of the diode positioned nearest to the negative end (cathode), and the other end (anode) should be connected to the positive end of the circuit.

Light Emitting Diode (LED)

An *LED* is a specific type of diode that gives out a light when an electric current passes through it. An important thing about an LED is that like any other diode it needs to be inserted the correct way in the circuit. The diode has an anode, which should be connected to the more positive end of the connection and a cathode, which goes to the other side.

You can tell which end is the anode (the positive terminal), as it normally has a longer lead. There is often a flat area on the plastic casing that indicates the cathode (negative terminal).

An LED does not limit the current flowing through, so a resistor is normally required to protect the LED.

Multi-Colored LEDs

Most LEDs emit only a single color of light. To allow them to provide different colors, two or three LEDs may be combined into a single package. They can be made to show a range of different colors depending on how much current is allowed to flow through each of the LEDs. The multi-colored LEDs usually have a common connection where all the LEDs connect to a single lead. These are known as common anode, where the positive end of the LEDs are connected and a common cathode, where the negative end of the LEDs are connected.

For a common anode-type LED, the LEDs are normally connected to a common positive power supply and by switching the other end down to ground the LEDs light. For a common cathode, the cathode is normally connected to ground and a positive voltage applied to the appropriate color anode connection.

Sometimes the LEDs are connected to a controller circuit such as the WS2812 used in Chapter 6. The built-in controller circuit turns on the appropriate LED based on a digital signal sent to it.

Bipolar Transistor

A *transistor* is an electronic component that is used to switch a larger current compared to its input. The bipolar transistor has three connections, known as the collector, base, and emitter (represented by the letters C, B, and E on diagrams). When a small current flows between the base and emitter, it allows a much larger

current to flow between the collector and the emitter. The transistor is an analog component, so varying the base current changes the corresponding collector current.

The bipolar transistor comes in two types: one is called NPN and the other is PNP. The name is based on the way that the transistor is made. The NPN needs an input voltage that is higher than the emitter voltage to allow it to conduct, whereas the PNP needs an input voltage that is lower than the base voltage for it to conduct.

Darlington Transistor

A bipolar transistor increases the amount of current that can flow, but sometimes a second stage is required to increase the current even more. This could be achieved by connecting two transistors with the output of the first stage used to drive the input of the second stage. This is known as a *Darlington pair*.

It is also possible to have both of these inside a single component and that's usually known as a Darlington transistor. This can be used in place of a standard transistor, but with a much higher gain.

MOSFET Transistor

A *MOSFET transistor* is different type of transistor that increases the signal based on the input voltage rather than the input current. They have three connections known as the drain, gate, and source (represented by the letters d, g, and s). MOSFETs are available as N-channels and as a P-channel component. For the N-channel MOSFET, a small voltage at the gate will allow a current to flow from the drain to the source. A P-channel MOSFET works in an opposite manner. When a negative voltage is applied to the gate, a current can flow from the source to the drain.

Capacitor

A *capacitor* is a device for storing electrical charge. It's a bit like a small rechargeable battery that can charge and discharge while connected in a circuit. The capacitor has a variety of uses in analog circuits, including acting as a filter to remove signals outside a specific frequency.

In digital circuits, capacitors are often used to smooth a power supply or signal, thereby removing any stray electrical noise.

Capacitance is measured in farads (F), but a farad is a really large value, so they are often denoted in micro-farads (μF, which is 0.000001 of a farad), nano-farads (0.000000001 of a farad), or pico-farads (0.000000000001 of a farad).

Thyristor

A *thyristor* is a component that allows current to flow in only one direction and only then when it receives a signal on the gate terminal. It is rather like a diode that needs to be turned on first. Once it has been turned on, it stays on until the power supply is removed or goes negative. They are suitable for use in AC circuits. Whereas a large reverse voltage would damage a MOSFET or transistor, a thyristor will work with a high reverse voltage. It does only allow for one half of the AC cycle to pass through, which you can overcome using a triac.

Triac

A *triac* is effectively two thyristors connected in the opposite polarity, but sharing the same three terminals. If a signal is applied to the gate, it will switch on and allow current to flow. If the signal is removed, it will stop conducting once the supply voltage reverses (on the opposite phase of the cycle).

APPENDIX C

Component Labeling

While many components include a part number on them, some components instead use specific colors or special codes to denote their values. The use of a code is often used on small components such as resistors in cases where it would be very difficult to read the value if it was printed in words. Some common component labels are explained in this appendix.

Resistor Color Codes

Resistors are normally labeled using four colored bands across the body of the resistor. This consists of three bands together indicating the resistance value of the resistor (ohms) and the fourth band, is used to indicate the tolerance (accuracy) of the resistor. The tolerance band is usually silver or gold which makes it easy to identify, although on some resistors the fourth band may be the same as the standard colors. In the case that the fourth color code is one of the standard colors then there will normally be a larger gap between the third and fourth bands. This is shown in Figure C-1.

Figure C-1. *Resistor showing the color codes*

The bands relate to the resistance as follows:

- First band: First significant figure of resistor value
- Second band: Second significant figure of resistor value
- Third band: Decimal multiplier (applied against the first two bands)
- Fourth band: Tolerance (if not present, tolerance is 20%)

The first three bands can be one of 10 colors representing digits from 0 to 9. The tolerance is normally gold or silver, although other colors are sometimes used. The different colors and their meanings are listed in Table C-1.

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3

Table C-1. *Resistor Color-Code Meanings*

Color	Significant Figures	Multiplier	Tolerance
Black	0	$x10^0$	-
Brown	1	$x10^1$	±1%
Red	2	$x10^2$	±2%
Orange	3	$x10^3$	-
Yellow	4	$x10^4$	-
Green	5	$x10^5$	±0.5%
Blue	6	$x10^6$	±0.25%
Violet	7	$x10^7$	±0.1%
Grey	8	$x10^8$	±0.05%
White	9	$x10^9$	-
Gold	-	$x10^{-1}$	±5%
Silver	-	$x10^{-2}$	±10%

The resistor in Figure C-1 has the colors green, blue, black, and gold.

This translates to 5 (green), 6 (blue), $x10^0$ (black), with a tolerance of 5%. This works out at 56Ω.

The tolerance is not normally a major consideration in most digital circuits, but if the exact value of the resistor is required, that band should be considered as well.

Resistors are made in certain sizes. Normally you can select the nearest size, but in some circumstances you may need to choose the nearest higher or lower as appropriate. For example, if you are calculating a resistor for the maximum current that can flow, you should choose the next size up, rather than a lower value.

The commonly-used E12 series uses 12 equally spaced values across every multiple of 10. A common alternative is the E6 series (Table C-2), which has six values per multiple of 10. If you intend to have some resistors in stock, it is useful to have the E6 series as a minimum and then just buy other specific sizes as required.

Table C-2. *E6 Resistor Series*

10Ω	15Ω	22Ω	33Ω	47Ω	68Ω
100Ω	150Ω	220Ω	330Ω	470Ω	680Ω
1kΩ	1.5kΩ	2.2kΩ	3.3kΩ	4.7kΩ	6.8kΩ
10kΩ	15kΩ	22kΩ	33kΩ	47kΩ	68kΩ
100kΩ	150kΩ	220kΩ	330kΩ	470kΩ	680kΩ
1MΩ					

The E12 resistors series has more resistor values, as shown in Table C-3.

Table C-3. *E12 Resistor Series*

10Ω	12Ω	15Ω	18Ω	22Ω	27Ω	33Ω	39Ω	47Ω	56Ω	68Ω	82Ω
100Ω	120Ω	150Ω	180Ω	220Ω	270Ω	330Ω	390Ω	470Ω	560Ω	680Ω	820Ω
1kΩ	1.2kΩ	1.5kΩ	1.8kΩ	2.2kΩ	2.7kΩ	3.3kΩ	3.9kΩ	4.7kΩ	5.6kΩ	6.8kΩ	8.2kΩ
10kΩ	12kΩ	15kΩ	18kΩ	22kΩ	27kΩ	33kΩ	39kΩ	47kΩ	56kΩ	68kΩ	82kΩ
100kΩ	120kΩ	150kΩ	180kΩ	220kΩ	270kΩ	330kΩ	390kΩ	470kΩ	560kΩ	680kΩ	820kΩ
1MΩ											

SMD Resistors

While color codes are used for standard through-hole-technology resistors, surface-mount resistors normally have a numeric code instead. The most common code is a three-digit number code. The first two digits indicate the first and second significant figures and the third digit indicates the multiplier.

A resistor marked as 180 will be 18Ω and a resistor marked 221 will be 220Ω.

Electrolytic Capacitors

Electrolytic capacitors are usually quite large, both in physical size and in their capacitance. They normally have the value of the capacitor written on them in micro-farads (μF). An example of an electrolytic capacitor is shown in Figure C-2.

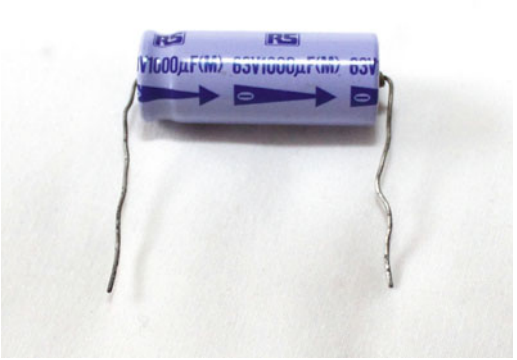


Figure C-2. *Electrolytic capacitor*

As well as showing the capacitance, the capacitor will also have the maximum voltage written on the body, which sometimes merges with the capacitance value. For example, a capacitor labeled:

```
63V1000µF
```

can be used up to 63V and has a capacitance of 1000µF. Electrolytic capacitors are polarized and need to be connected the right way around. The negative terminals are normally denoted by an arrow marked with a 0 pointing toward it.

Polyester Capacitors

Polyester capacitors, such as the one in Figure C-3, normally have the value written directly on them. These may be missing the units, in which case they are normally in µF.

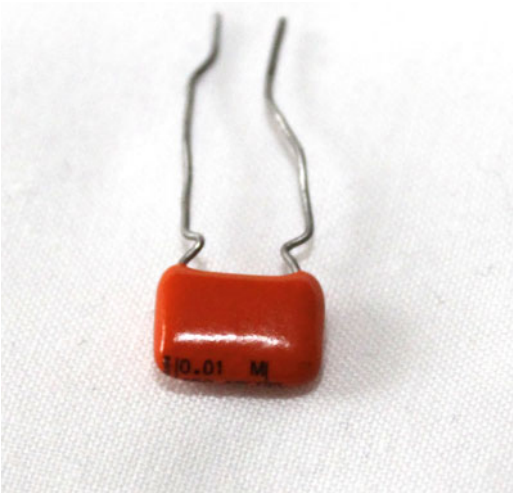


Figure C-3. *Polyester capacitor*

For example, a capacitor labeled as

```
0.01
```

will be 0.01µF, which is 10nF.

Ceramic Capacitors

Ceramic capacitors are normally the smallest in physical size, so often have a code instead of showing the actual value, as seen in Figure C-4.

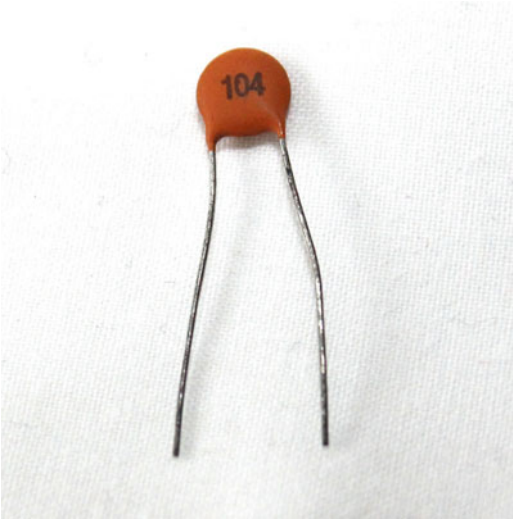


Figure C-4. *Ceramic capacitor*

The most common code is usually three digits. These give the value in pF (pico-farads). The first two digits are the significant digits and the third digit is a factor of 10 multiplier.

The 104 capacitor is 10 x 10^4 = 100,000pF, which is 100nF.

Some other capacitors are labeled with the letter n or p to denote nano-farads or pico-farads, respectively. A capacitor labeled:

```
1n0
```

has value of 1nF.

APPENDIX D

GPIO Quick Reference

This is a summary of the information about the GPIO pin layouts, provided for easy reference. For more details, see the explanation in Chapter 2.

GPIO Pin Layouts

The GPIO pin layouts have changed between different versions of the Raspberry Pi. The first Raspberry Pi had 26 pins with some changes in the port allocations between revision 1 and revision 2. The B+ and later models were then expanded to 40 pins. A diagram showing the pin allocations is shown in Figure D-1.

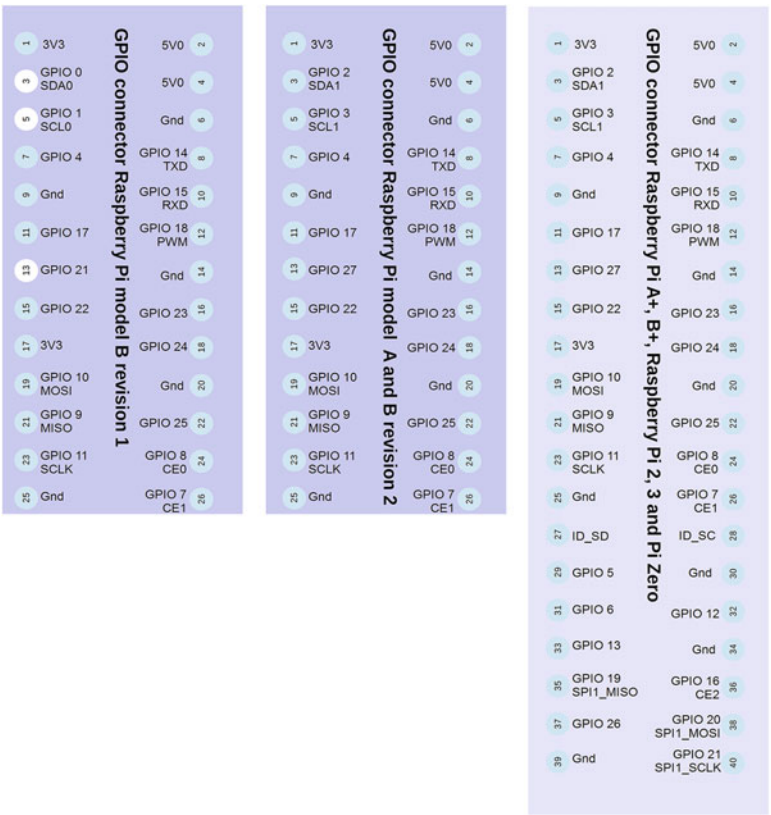


Figure D-1. *GPIO port numbers*

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3

GPIO Ports with Alternative Functions

A summary of the ports and common alternative functions is provided in Table D-1. This is based on the B+ and later.

Table D-1. *GPIO Pin Reference*

Pin Number	GPIO Port	Alternative Function	Comments
1		3.3V power supply	
2		5V power supply	
3	GPIO 2	SDA1 (I2C data)	Changed from revision 1 boards
4		5V power supply	
5	GPIO 3	SCL1 (I2C clock)	Changed from revision 1 boards
6		Ground	
7	GPIO 4		
8	GPIO 14	Serial/console TXD	Serial transmit
9		Ground	
10	GPIO 15	Serial/console RXD	Serial receive
11	GPIO 17		
12	GPIO 18	PWM	Pulse Width Modulation
13	GPIO 27		Changed from revision 1 boards
14		Ground	
15	GPIO 22		
16	GPIO 23		
17		3.3V	
18	GPIO 24		
19	GPIO 10	MOSI (SPI)	SPI Master Output, Slave Input
20		Ground	
21	GPIO 9	MISO (SPI)	SPI Master Input, Slave Output
22	GPIO 25		
23	GPIO 11	SCLK (SPI)	SPI clock
24	GPIO 8	CE0 (SPI)	Chip Enable (slave select)
25		Ground	
26	GPIO 7	CE1 (SPI)	Chip Enable (slave select)
27		ID_SD	For HAT EEPROM
28		ID_SC	For HAT EEPROM

(*continued*)

Table D-1. (*continued*)

Pin Number	GPIO Port	Alternative Function	Comments
29	GPIO 5		
30		Ground	
31	GPIO 6		
32	GPIO 12		
33	GPIO 13		
34		Ground	
35	GPIO 19	SPI1_MISO	
36	GPIO 16	CE2 (SPI)	Chip enable (slave select)
37	PIO 26		
38	GPIO 20	SPI1_MOSI	
39		Ground	
40	GPIO 21	SPI1_CLK	

For more details, see the schematics of the Raspberry Pi and related documentation at `www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/README.md`.

Index

S. Watkiss, *Learn Electronics with Raspberry Pi*, DOI 10.1007/978-1-4842-1898-3

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y, Z