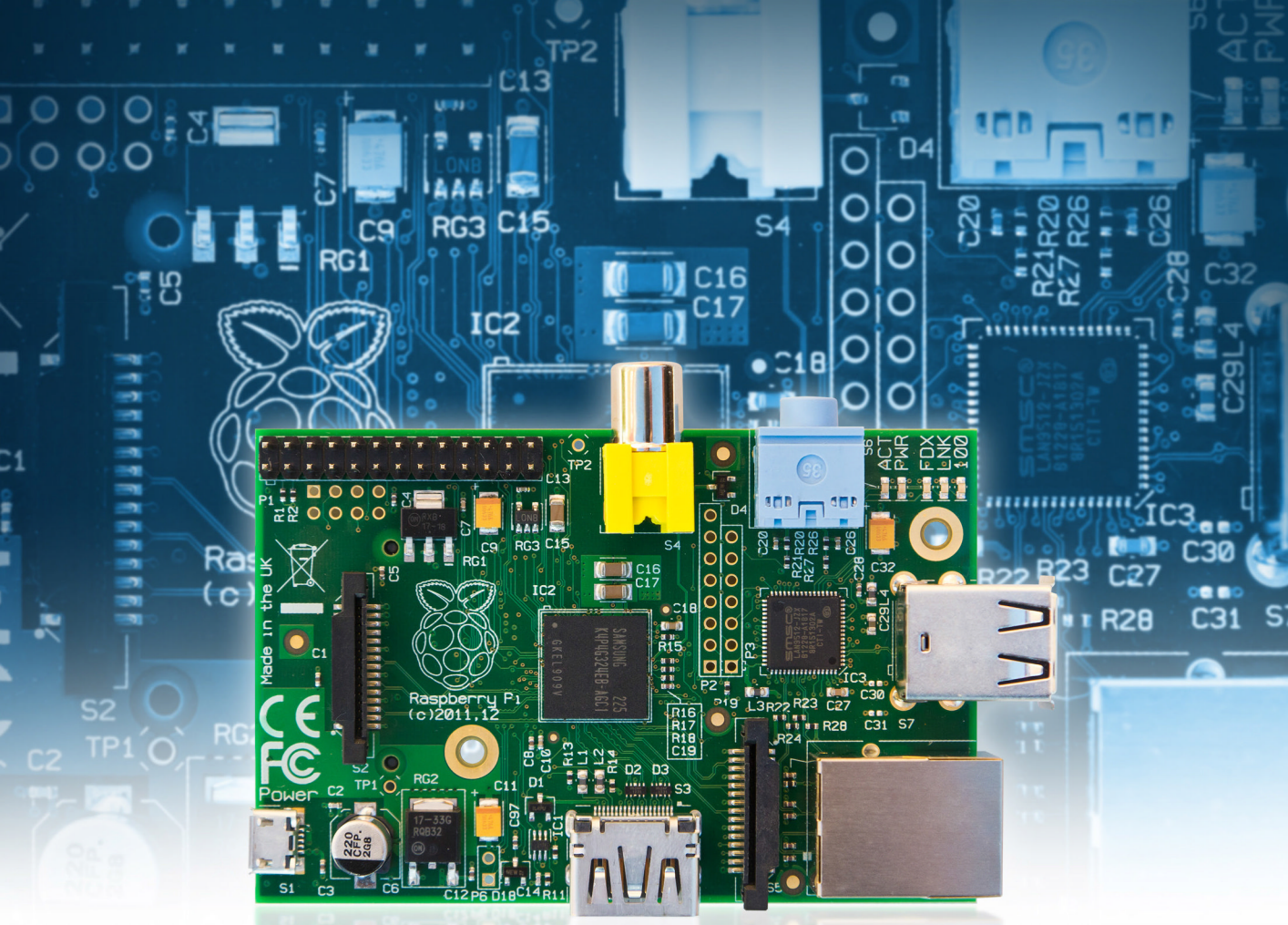


Raspberry Pi®

Hardware Projects

Volume 1

Andrew Robinson
Mike Cook

WILEY

Raspberry Pi® Hardware Projects

Volume 1

Andrew Robinson and Mike Cook
A John Wiley & Sons, Ltd, Publication

WILEY

Raspberry Pi® Hardware Projects, Volume 1

This edition first published 2013

Registered office

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

A catalogue record for this book is available from the British Library.

ISBN 978-1-118-58894-9 (ePub); ISBN 978-1-118-72391-3 (ePDF)

Set in 10 pt Chaparral Pro Light by Indianapolis Composition Services

Contents at a Glance

Some of the people who helped bring this book to market include the following:

Editorial and Production

VP Consumer and Technology Publishing Director
Michelle Leete

Associate Director–Book Content Management
Martin Tribe

Associate Publisher
Chris Webb

Executive Commissioning Editor
Craig Smith

Assistant Editor
Ellie Scott

Project Editors
Dana Lesh, Kathryn Duggan

Copy Editors
Dana Lesh, Kathryn Duggan

Technical Editor
Genevieve Smith-Nunes

Editorial Manager
Jodi Jensen

Senior Project Editor
Sara Shlaer

Editorial Assistant
Leslie Saxman

Marketing

Associate Marketing Director
Louise Breinholt

Marketing Manager
Lorna Mein

Senior Marketing Executive
Kate Parrett

Composition Services

Compositors
Carrie A. Cesavice, Jennifer Goldsmith

Proofreaders
Melissa Cossell, Dwight Ramsey

Introduction

YOU'VE GOT A Raspberry Pi – now what? This book has the answer; it's packed full of fun Raspberry Pi projects to inspire you. From getting your Pi generating comedy insults, testing your reactions and building a talking animatronic toy that tweets to building your own disco light show, prepare to be entertained and amazed by your credit card–sized computer.

One word of warning: After you start you might never stop! Electronics and coding can be addictive; who knows what you might go on to make with the skills you learn from this book.

Appendix A, "Getting Your Raspberry Pi Up and Running", is a beginner's guide to your first steps with the Raspberry Pi. If you've never coded before, Appendix B, "Introductory Software Project: The Insult Generator", will get you started programming in Python. Chapter 1, "Test Your Reactions", will get you wiring up simple computer-controlled circuits. Chapter 2, "The Twittering Toy", will show you how to make your code talk to Twitter and get you hacking household items. Chapter 3, "Disco Lights", shows you how to control LED strips and make them dance in time to the music. Along the way you will pick up the skills you need to develop your own ideas to make projects work exactly how you want them to.

Building and making is incredibly rewarding and satisfying. We want to get more people of the world to become producers of technology rather than consumers. The projects in this book are starting points – step by step, they're easy to follow so you can get results quickly. But then the real satisfaction can come, that which comes from making the project your own. At the end of each chapter there are ideas and suggestions to extend the project, together with background information to point you in the right direction. The real addictive fun begins when you see your own ideas become reality.

Welcome to the world of digital making.

Chapter 1
Test Your Reactions

In This Chapter

- Getting started interfacing hardware with the Raspberry Pi
- Working with basic electronic circuits
- An introduction to electronic components, including transistors and resistors
- How to wire up a switch and an LED

THINK YOU'VE GOT fast fingers? Find out in this chapter as you take your first steps in hardware interfacing to build a reaction timer. You'll program the Raspberry Pi to wait a random time before turning on a light and starting a timer. The timer will stop when you press a button.

Welcome to the Embedded World!

For some people the idea that computers aren't always big black or beige boxes on desks is a surprise, but in reality the majority of computers in the world are embedded in devices. Think about your washing machine – to wash your clothes it needs to coordinate turning the water on, keeping it heated to the right temperature, agitating your clothes by periodically spinning the drum, and emptying the water. It might repeat some of these steps multiple times during a wash, and has different cycles for different types of fabric. You might not have realised it's a computer program. It takes inputs from switches to select the wash and sensors that measure water temperature, and has outputs that heat the water and lock the door shut, and motors to turn the drum and open and close valves to let water in and out.

YOUR TURN! Take a moment to consider the number of appliances and gadgets that need to measure inputs, do some processing to reach a decision and then control an output in response.

A modern kitchen is crammed with computers that watch over and automate our appliances to save us effort. Computers aren't just embedded in practical products either; they're in electronic toys and entertainment devices. After working through this chapter and the other examples in this book you'll be on your way to designing your own embedded systems to make your life easier, or entertain you.

Before you get too carried away connecting things up it's worth considering a couple of warnings that will protect you and your electronic components.

Good Practice

Electricity can be dangerous, so it is important to use it safely. The muscles in your body are controlled by tiny electrical signals, and these can be affected if electricity flows through your body. Your heart is a muscle that can be stopped by an electric shock.

The flow of electricity can cause heating, which will either cause burns to your body (sometimes deep within tissue) or can cause a fire.

WARNING

Electricity can kill! Only experiment with low voltages and currents, and never work with mains. If you are ever in doubt then you should check with someone suitably qualified.

Hardware is less forgiving than software; if you make a mistake with code, you might get an error, the program might crash, or in rare cases you might cause your Raspberry to reset. If you make a mistake in hardware then you can cause permanent damage. As such, hardware engineers tend to check and double check their work before applying the power!

When experimenting you should beware of short-circuiting your projects. Make sure that nothing conductive touches your circuit. Tools, metal watchstraps and jewellery, unused wires, spare components and tin foil have all been known to damage circuits. Keep your working area clear of anything you don't need and make sure that nothing metallic can touch your Raspberry Pi or circuit.

Static Discharge

You may have felt a small electric shock due to static sometimes. This occurs when a charge builds up and then discharges to a conductor, which you feel as a small shock. If you are holding a component when this happens, that large voltage will flow through the component and damage it. Other objects such as plastic can become charged too and then discharge through a component. As such, you should take care to avoid this static discharge through components or circuits. In industry, conductive work surfaces and wrist straps are earthed to prevent static buildup. This may be an extreme solution for a hobby; you can discharge yourself by touching something earthed like a water tap, and avoid working on surfaces that are prone to picking up static charge like plastics – for example, avoid working on nylon carpets or plastic bags.

TIP

You may have noticed components are supplied in antistatic bags, or static-dissipative bags or static-barrier bags. These bags are made from special plastic designed to protect the contents from being zapped by static discharges and conduct any charge away. Beware that some of these bags can be slightly conductive and so may interact with your powered-up circuit.

Obtaining Components

Another difference with hardware is that you can't download everything you need from the Internet! However, you can do the next best thing and order parts online. There are a number of online electronics retailers that supply parts, including the two worldwide distributors of the Raspberry Pi, element14/Premier Farnell/Newark and RS Components. Pimoroni, SparkFun, SK Pang, Cool Components, Adafruit and other web stores have a smaller range but cater well to electronic hobbyists.

Maplin Electronics and Radio Shack have shops on the high street with a smaller selection of parts.

An Interface Board

Although the Raspberry Pi has a general purpose input/output (GPIO) connector that you can connect to directly, as a beginner, it is easier to use an add-on board. An interface board can offer some protection to your Pi against burning out if you get your wires crossed!

PiFace Digital

This chapter uses the PiFace Digital interface because it is very easy to use. PiFace Digital has eight LEDs on it so that you can start controlling hardware without any electronics knowledge. Later in this chapter you'll connect your own LEDs and switches to PiFace Digital with the screw terminals. Hopefully you'll go on to use more advanced boards, and eventually you may want to design an interface board of your own!

TIP In computing, *digital* refers to things that can either be on or off – there's no in between. In contrast, *analogue* devices have many points between their maximum and minimum values. A button is digital in that it is either on or off. A temperature is an example of something that is analogue.

Setting up PiFace Digital

PiFace Digital communicates using *Serial Peripheral Interface* (SPI) bus. It's a standard means of connecting peripheral devices to microprocessors. Before you use PiFace Digital with the Raspberry Pi you need to install some software.

SPI

SPI consists of four wires to communicate data from a master (the microprocessor) to a slave device (the peripheral). Data is sent *serially* (that is, the voltage on a wire is switched on and off to communicate a binary number) over time using four wires as shown in Figure 1-1.

- One wire is used for data from the master to the slave (named *master output slave input* [MOSI]).
- Data going to the master from the slave is sent on another wire (named *master input slave output* [MISO]).
- The *serial clock* (SCK) wire is used to synchronise the master and slave so they know when a valid value is being sent (that is, that MISO and MOSI have momentarily stopped changing).
- The *slave select* wire, or sometimes called *chip select* (SS or CS), selects between multiple slave devices connected to the same MOSI, MISO and SCK wires.

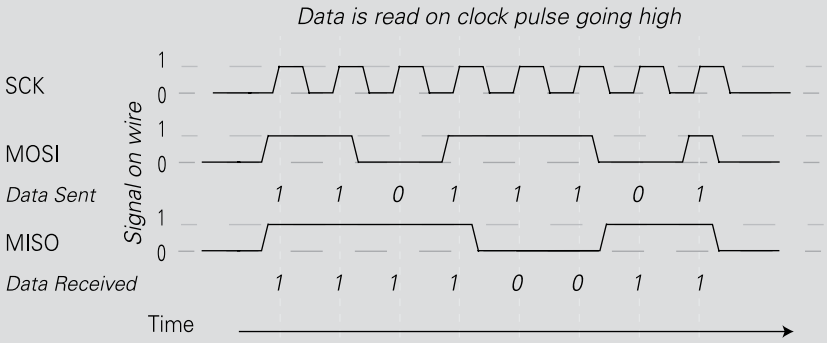


FIGURE 1-1: Example SPI transaction: The microprocessor sends data 11011101, and the device sends 11110011.

Installing PiFace Digital Software

Appendix A, "Getting Your Raspberry Pi Up and Running", mentions drivers that the operating system loads. These make it easy for programmers to write code to interact with hardware devices. Rather than bloat the operating system with drivers for every possible type of hardware, Linux has driver modules. These are loaded separately when needed. As PiFace Digital talks over SPI you need to ensure that the SPI driver is loaded so the Raspberry Pi hardware sends the correct electrical signals on the expansion pins. You will also need to install a Python library that makes the SPI driver easy to use from your Python code.

It is possible to install PiFace Digital software on Raspbian as a Debian package with one command. However, in the future you might need to install software that isn't packaged, or perhaps want to use PiFace Digital on a different distribution that doesn't use Debian packages. As such the following steps show how to manually install software from the source.

Loading the SPI Driver Module

Check to see if the SPI driver is loaded. Type `lsmod` to list all modules. If it is loaded, you will see the following line. Don't worry about the numbers on the right; it is the module name `spi_bcm2708` which is important.

```
spi_bcm2708             4401  0
```

If it is not listed, you need to enable the module. Although the driver module is included, Linux "blacklists" some modules so they are not loaded. To un-blacklist the module, edit the file /etc/modprobe.d/raspi-blacklist.conf. You can insert a # in front of the line `blacklist spi-bcm2708` to comment it out, or delete the line completely. Use your favourite editor to edit the file, such as leafpad, nano or vi. For example, you use nano by typing the following:

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

Enter # at the start of the line `blacklist spi-bcm2708` like so:

```
#blacklist spi-bcm2708
```

Save the file. If using the nano editor, press Ctrl + X and then confirm that you want to save the file before exiting.

TIP

Commenting out is a way of making the computer ignore what is on that line. It works by turning the line into a comment. *Comments* are text intended for the user and not interpreted by the computer.

It is better to comment a line out rather than delete it as it makes it easier to restore later by uncommenting (that is, removing the comment marker). Python also uses # for comments, so you can temporarily remove a line of code by putting a # at the start of the line.

Restart the Raspberry Pi by typing `sudo reboot`.

Git and Source Code Management

Git is a *source code management* (SCM) system that keeps track of different versions of files. SCMs are necessary when multiple people work on the same project. Imagine if two people were working on a project and changed the same file at the same time. Without SCM the first person's changes might get overwritten by the second. SCMs instead help manage the process and can merge both contributions. They also keep previous versions (like wikis do) so it's possible to go back.

Git was initially developed by Linus Torvalds for collaborative development of the Linux operating system but then used by many other projects. There are other SCMs such as SVN, CVS and Mercurial, which provide similar functions.

GitHub (http://github.com) is a website that hosts projects using Git. It's designed for social coding, so everyone can suggest changes and contribute to improving a project. As the Raspberry Pi has such a strong community, there are many projects on GitHub (including the code for Linux itself) for the Raspberry Pi that everyone can contribute to.

GitHub offers free accounts and tutorials so as you become a more proficient coder, you might as well try using source code management for your project. It can be really useful if you start working with a friend or want to go back to a previous version of your code.

Type `lsmod` again and you should see `spi_bcm2708` listed. Programs send a message over the SPI bus by writing characters to a special file. Type `ls -la /dev/spi*` to check whether the special file exists. If you're successful, Linux will show at least one file. If you receive a message such as `No such file or directory`, you need to check that the SPI driver module is functioning correctly.

Installing Python Modules

With the `spi_bcm2708` module loaded, Linux can talk SPI to peripherals. The next step is to install programs that will make the driver easier to use and send the correct messages over the SPI bus:

1. Make sure that the Raspberry Pi is up to date. Because the Raspberry Pi will check the Internet for updates, you need to make sure that your Raspberry Pi is online. Then type `sudo apt-get update`.

   Wait until Linux downloads and updates files, which can typically take a couple of minutes.

2. Install the necessary packages by typing `sudo apt-get install python-dev python-gtk2-dev git`.

   Linux will list any other programs necessary and tell you how much additional disk space is required. Confirm that you want to continue by typing `Y` and pressing enter. After a few minutes the programs will be installed.

3. Get the latest software to control PiFace by typing the following, after typing `cd` to return to your home directory:

   ```
   git clone https://github.com/thomasmacpherson/piface.git
   ```

   This will copy the latest PiFace code and examples into your home directory. Type `ls piface` to list the contents of the directory that have just been downloaded.

4. So that all users of the Raspberry Pi can use the SPI bus, you need to change the permissions on the `/dev/spi*` files. PiFace Digital provides a handy script to set SPI permissions. Run the script by typing

   ```
   sudo piface/scripts/spidev-setup
   ```

5. Restart by typing

   ```
   sudo reboot
   ```

   Lastly, you need to install the Python modules that will make it easy to send the correct messages on the SPI bus.

6. Type `cd` to get back to the home directory.

7. Change the directory to the piface directory and then the python directory by typing `cd piface/python`.

8. Install the Python module that talks to PiFace Digital by typing `sudo python setup.py install`. After a few minutes the necessary files will be installed and you will be ready to start testing with real hardware.

9. Finally, shut down the Pi and remove the power before connecting the PiFace interface by typing `sudo halt`.

TIP It is useful to know how to install software manually, but if in the future you want to install PiFace Digital with one command, instructions are provided at `http://www.piface.org.uk`.

Connecting PiFace Digital

Disconnect the power before connecting or disconnecting anything to or from the Raspberry Pi. This ensures that you don't accidentally short anything and is generally safer. Position PiFace Digital so it lines up with the edges of the Raspberry Pi and check that all 26 pins of the expansion port line up with the holes in the connect. Gently push the PiFace interface down, making sure that you don't push sideways to bend the pins. If it is correctly lined up, it should slide smoothly. Connect the power, log in and start X, as described in Appendix A.

Using the Emulator

This book mentions the importance of regular testing and checking that subsystems work before moving on. This is a great excuse for turning some lights on and off. There's something satisfying about seeing a computer responding to you, lighting a light, obeying your command! Next you will use the PiFace emulator to check that your Raspberry Pi can talk to your PiFace Digital.

Start the emulator by typing `piface/scripts/piface-emulator` in a Terminal window. The emulator window will appear as shown in Figure 1-2.

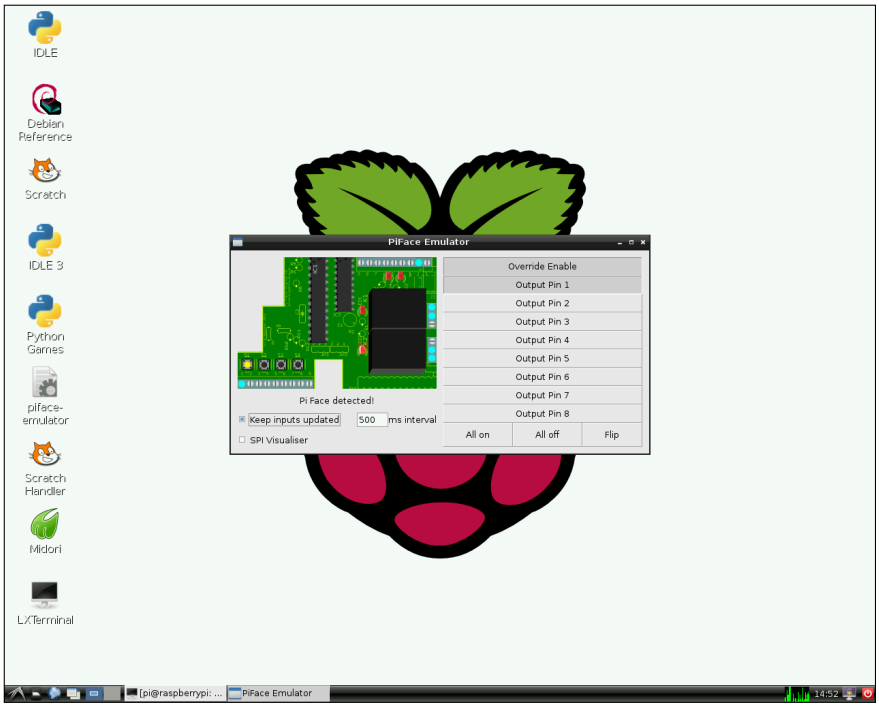


FIGURE 1-2: The PiFace Emulator.

As you want to manually control the outputs, in the PiFace Emulator window click Override Enable.

Toggle Output Pin 1 on by clicking it. The PiFace interface will click as the relay turns on, and the corresponding LED will illuminate. Notice the graphic on-screen updates to show the LED being on, the contacts have changed over on the relay and the first output pin is on.

Try turning multiple output pins on, and notice how the on-screen display updates. Try the All On, All Off and Flip buttons to discover what they do.

When you are finished flashing lights and trying the various options close the emulator. You're now ready to program your Raspberry Pi to take control!

Interfacing with Python

You first meet the Hello World program in Appendix B, "Introductory Software Project: The Insult Generator". The hardware equivalent of Hello World is flashing a light, which similarly, although not exciting in itself, is the first step in getting a computer to start controlling the world. After you've mastered this, there's no limit to what you can start controlling!

Turning a Light On

First, write the code to turn an LED on:

1. Double-click IDLE to begin entering code interactively.
2. Type the following:

```
import piface.pfio as pfio

pfio.init()

pfio.digital_write(0,1)
```

3. The first LED should be lit.

TIP Appendix B mentions that there are two versions of Python – Python 2 and Python 3. Similarly there are two versions of IDLE; `IDLE` that corresponds to the Python 2 command `python` and IDLE3 that corresponds to Python 3.

The `import` statement tells Python to use the `piface.pfio` module so it can talk to the PiFace interface. You must call `pfio.init()` to check the connection and reset the PiFace hardware. The `digital_write(`*`outputpin`*`,`*`value`*`)` function takes *`outputpin`*, which

selects the LED, and *value*, which determines if it is turned on. So `digital_write(0,1)` sets the first LED to have the value 1 (on). This function will be familiar if you have ever programmed the Arduino.

TIP

Computers start counting at 0 as this makes some operations and calculations more efficient. You should see why as you learn more about computers and programming.

Flashing a Light On and Off

Next, you're going to write a program that flashes the LED with a timer:

1. Create a new window in IDLE by going to the File menu and clicking New Window.
2. Type the following:

```
from time import sleep

import piface.pfio as pfio

pfio.init()

while(True):

  pfio.digital_write(0,1)

  sleep(1)

  pfio.digital_write(0,1)

  sleep(1)
```

3. Go to the Run menu and choose Run Module (or press F5).
4. Click OK when Python displays the message Source Must Be Saved.
5. Enter a filename and then click Save.
6. You will see a message saying RESTART, and then Python should run your code. If an error message appears, go back and correct your code and try running it again.

You've now written the hardware equivalent of Hello World – an LED that flashes. The next examples will show how to make the Raspberry Pi respond to the world, as you will learn how to read inputs.

YOUR TURN!

As an extension, you could try flashing a more complex pattern – change the rate the LED flashes at or try flashing multiple LEDs.

Reading a Switch

For a computer to respond to its environment, it needs to be able to read inputs. First you will use the graphical emulator to display the status of the inputs.

Showing Input Status Graphically

In the emulator, click the Keep Inputs Updated check box as shown in Figure 1-3. The interval sets how often the inputs are read; for most cases, it is fine to leave it on 500ms.

FIGURE 1-3: Enable the Keep Inputs Updated check box to show the status of inputs.

Test the input by pressing one of the buttons on the bottom left of PiFace. As shown in Figure 1-4, the on-screen representation changes to indicate the switch that has been pressed.

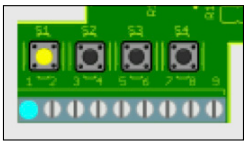


FIGURE 1-4: The status of inputs are shown in the emulator.

You can close the emulator for now by clicking the cross in the top-right corner of the window.

Reading Inputs with Code

As you saw earlier, you can control outputs using the function `digital_write`. Logically, as you might expect, to get the value of inputs, you use the `digital_read` function. Let's write a simple example:

1. Type the following Python code interactively:

   ```
   import piface.pfio as pfio

   pfio.init()

   pfio.digital_read(0)
   ```

   Python prints `0`.

2. Hold down button number S1 and type `pfio.digital_read(0)` again.

   Python prints `1`.

3. Now read the next button along (S2). Type `pfio.digital_read(1)`. Notice how the argument of the function specifies which input to read.

You have seen how easy it is to read an input. Next, you are going to combine turning a light on and reading an input to build a reaction timer.

The Reaction Timer

Now is the time to find out if you've got fast fingers by building a reaction timer game.

Type the following code in a new window and then save it as reactionTimer.py:

```
from time import sleep, time
from random import uniform
import piface.pfio as pfio
#Initialise the interface
pfio.init()
print "press button one when the light comes on"
#Sleep for a random period between 3 and 10 seconds
sleep(uniform(3,10))
#Turn LED on
pfio.digital_write(0,1)
#Record the current time
startTime = time()
#Wait while button is not pressed
while(pfio.digital_read(0)==0):
    continue #continue round the loop
#Button must have been pressed as we've left loop
#Read time now
stopTime = time()
#Time taken is difference between times
elapsedTime = stopTime-startTime
#Display the time taken. str is required to convert
#elaspedTime into a string for printing
print "Your reaction time was: " + str(elapsedTime)
#Turn LED off when finished
pfio.digital_write(1,0)
```

Run the code by pressing F5 from IDLE.

All the lights will go out. Wait until the LED comes on and then press button 0 as fast as you can. The program will print your reaction time.

Next, you'll see if your reactions are any faster with a big red button as you wire up a simple circuit.

Meet PiFace Digital's Connectors

The PiFace Digital interface makes it very easy to wire up simple digital circuits so that you can connect your Raspberry Pi to switches and actuators like lights or motors in the real world. Figure 1-5 labels the connectors on PiFace Digital.

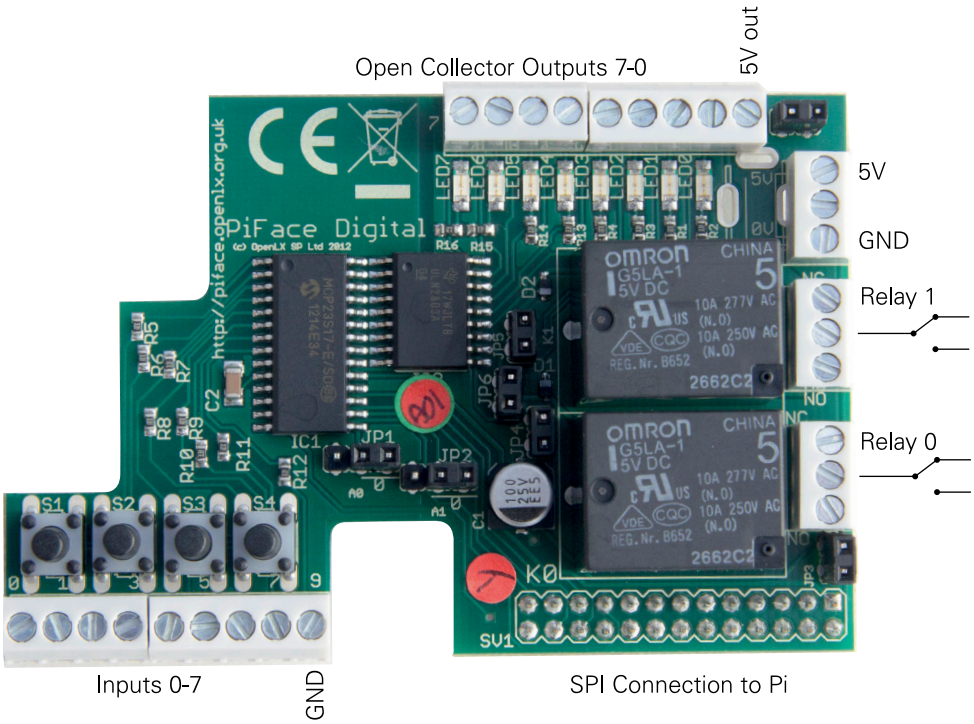


FIGURE 1-5: PiFace Digital's connectors.

Inputs

The screw terminals next to the on-board switches are used to connect external switches. There are eight inputs, numbered 0-7 from the outside of the board to the middle, followed by a connection to ground. PiFace Digital will register an input if there is an electrical connection between the input terminal and ground – that is, there is a path for electrons to flow.

Relays

When you turn either of the first two outputs on you should notice that PiFace clicks. This sound is as the contacts in the relay (the large rectangular components) change over. A *relay* can by thought of as a computer-controlled switch. You'll use relays in Chapter 2, "The Twittering Toy".

Electrical Circuits

Modern computers work with electricity, so to interface with them, you need to understand the basics of how it behaves.

Electricity is the flow of tiny particles, called *electrons,* that carry a charge. Think of electrons as always wanting to get home; for example, with a battery, the electrons want to get back into the other terminal. As the electrons move through components in a circuit on their way home they do work. This work might be to emit light in an LED or move a motor around. If the electrons do not flow, no work is done (and the LED does not shine). Figure 1-6 shows three circuits, but only one has a path for electrons to leave the power source, pass through a component to do work and return home again!

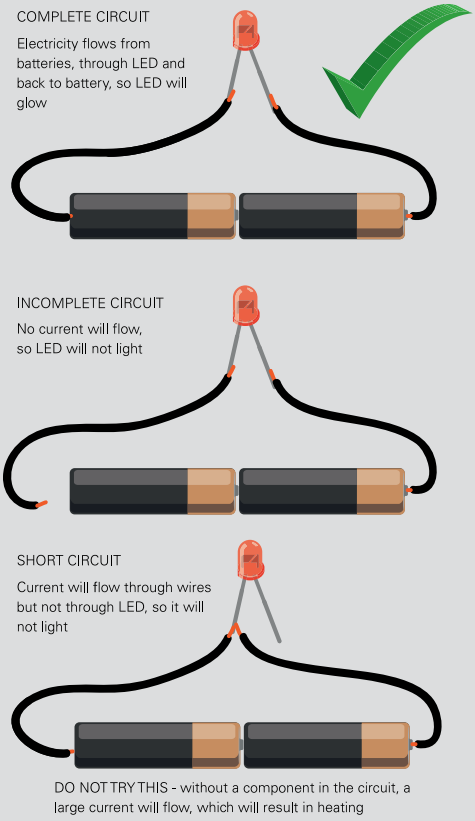


FIGURE 1-6: Electricity needs a complete circuit to flow and do work.

Electronics is all about controlling electrons' journeys! In many cases it is about making or breaking a complete path for electrons to flow.

continued

continued

In describing circuits, there are a few terms that you may come across:

- Voltage – this is, in electrical terms, how "strongly" the electrons are pulled home – that is, how much work they can do while they flow through the circuit. Think of it a bit like a water wheel and a reservoir. The greater the distance the water falls, the more work it can do turning the water wheel as it flows past. *Voltage* describes the work that can be done and is measured in volts – for example, an AA battery has a voltage of 1.5V between its terminals. If another one is connected end to end, then there is greater potential to do work, a voltage of 3V.
- Ground, or 0V (or sometimes referred to as *negative*) – a reference point to measure voltage from. If a point in a circuit is at ground, then it is at 0V, and no work can be done. With the water example, if the water is on the ground, it can't fall any further so can't be harnessed to do any work.
- Resistance – how easily the electrons can flow. Different substances allow electrons to flow with different degrees of ease. Conductors, such as metals, have a low resistance and make it easy for the electrons to flow. Insulators, such as plastics, have high resistances, which make it hard for electrons to flow. Different materials resist the flow of electrons by different amounts. Even water has a fairly low resistance, so it will allow electricity to flow through it, which is why you shouldn't use your Raspberry Pi in the bath!

You can think of these like this: *Voltage* describes the potential to do work (analogous to the height of water), and *current* describes the rate electricity follows (the rate of flow of water passing a point). *Resistance* describes how easily electricity flows through a material – voltage, resistance and current are interrelated – without a voltage existing between two points, no current will flow. The resistance between the two points affects how much current will flow. If you want to know more, look up Ohm's Law online.

Outputs

As well as controlling the on-board LEDs and relays, PiFace Digital has "open-collector" outputs that can be used to control circuits. You can connect to these outputs with the screw terminals next to the LEDs.

Open-collector outputs describe the circuit that drives the output and offers a number of advantages. The term *open collector* describes how the output transistor is connected. *Transistors* are the switches at the heart of computers – there are tens of millions of transistors in the processor at the heart of the Raspberry Pi. Luckily they're only tens of nanometres (a *nanometre* is a thousand millionth of a metre – you could fit 2000 transistors across the width of a human hair) in size. Although transistors behave in a similar way to switches and relays, the direction current flows through them affects how they behave, which needs to be considered when connecting to them.

TIP

There are different types of transistors, which allow current to flow in different ways. For simplicity this chapter just uses examples of the type NPN.

Open-collector outputs can just sink current. That is, they allow current to flow to ground; they are not a source for current. This means that circuits have to be wired up from a power source, through the component being controlled, through the transistor and then to ground. Figure 1-7 shows a typical setup. Remember, current has to flow for electricity to do work, so until the transistor turns on and allows current to flow to ground the LED will not come on. The transistors on PiFace already have the connection to ground wired up.

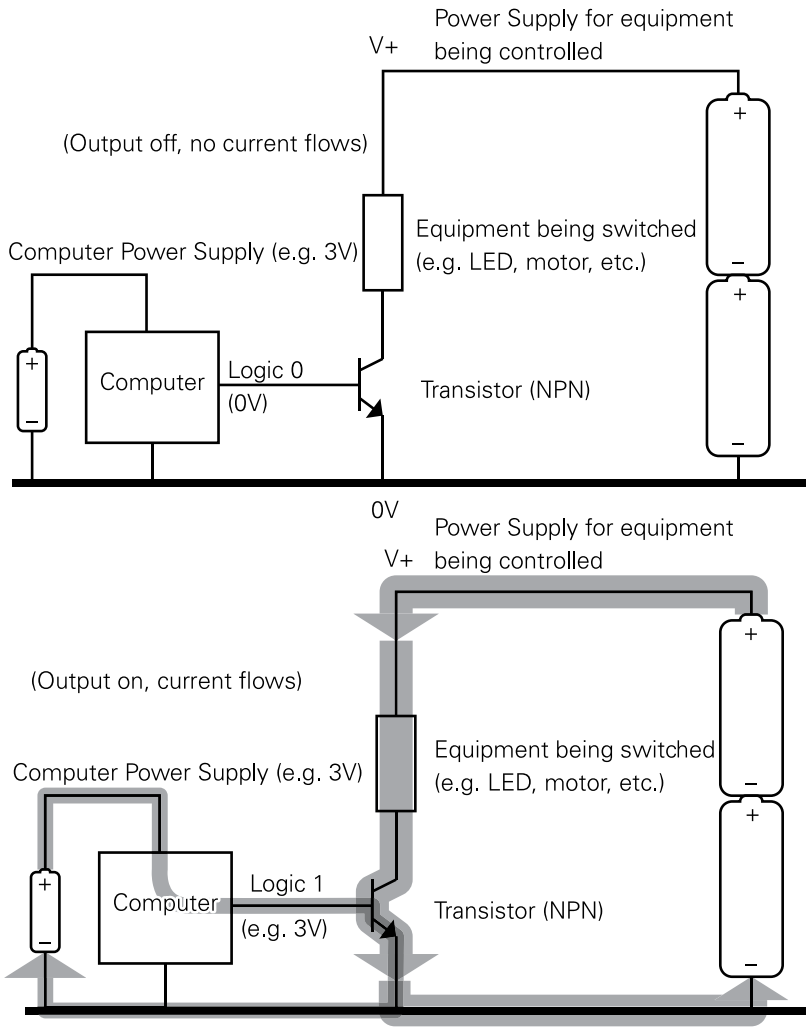


FIGURE 1-7: How to wire up an open-collector output.

Connecting a Switch and an LED

Enough theory! It's time to wire up the components. For the example, you're using a switch that incorporates an LED, but you could use a separate LED and switch. You will wire them up as shown in Figure 1-8.

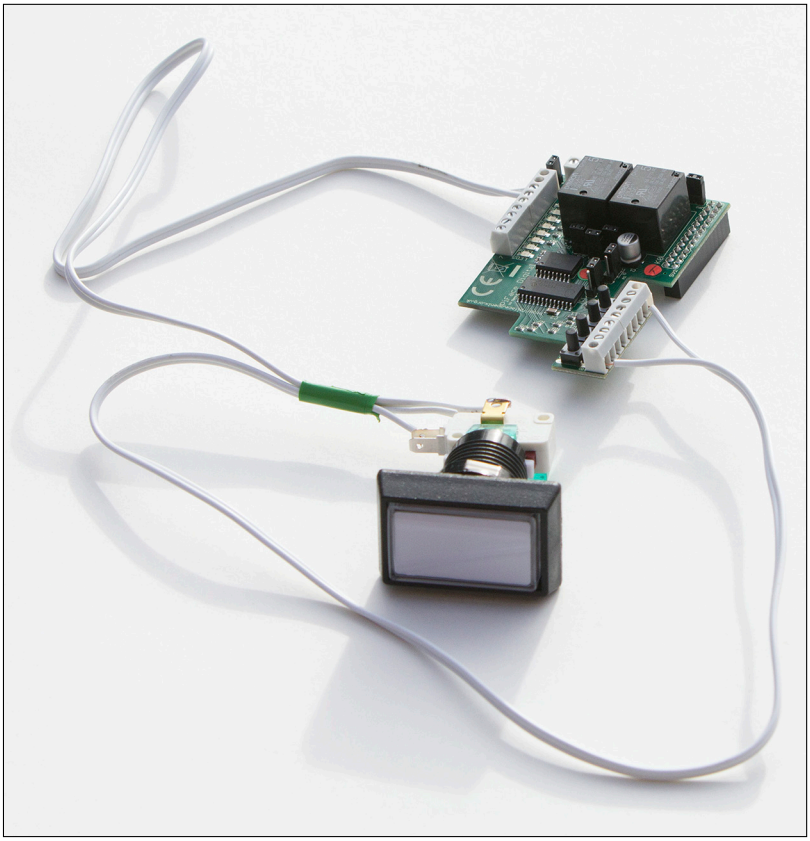


FIGURE 1-8: Wiring up the LED and switch to the Raspberry Pi.

Making Connections

There are a variety of ways to join wires and components together. Figures 1-9 through 1-12 show different ways of making connections. Important considerations are that the joint is secure and that you have a good connection; otherwise, the joint will create a high resistance or come apart.

- Wires can be twisted together (as shown in Figure 1-9) – this is a quick and easy method, but not very secure. Wrapping insulation tape helps to hold things more securely and prevents other connections shorting.

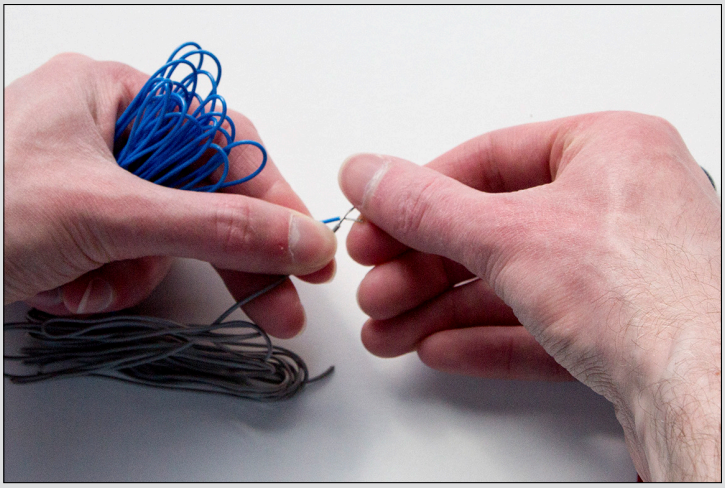


FIGURE 1-9: Twisting wires.

- Screw connectors (in some forms, sometimes called *choc-bloc*) (see Figure 1-10) – these hold wires together under a screw. They're quick, easy and fairly secure, but are quite bulky.

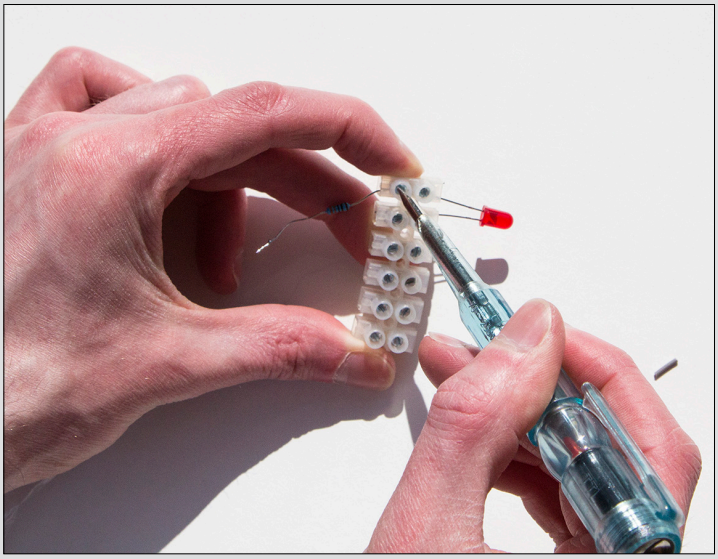


FIGURE 1-10: Screwing terminals.

continued

continued

- Breadboard, sometimes called *binboard* (see Figure 1-11) – is great for experimenting. Breadboard has rows of strips of metal that grip and connect wires. Components can be inserted directly into the breadboard, which makes it good for prototyping circuits. Wires are only loosely gripped so they can be pulled out – good for reuse, but not very secure or permanent. Some cheaper breadboards suffer from poor connectors.

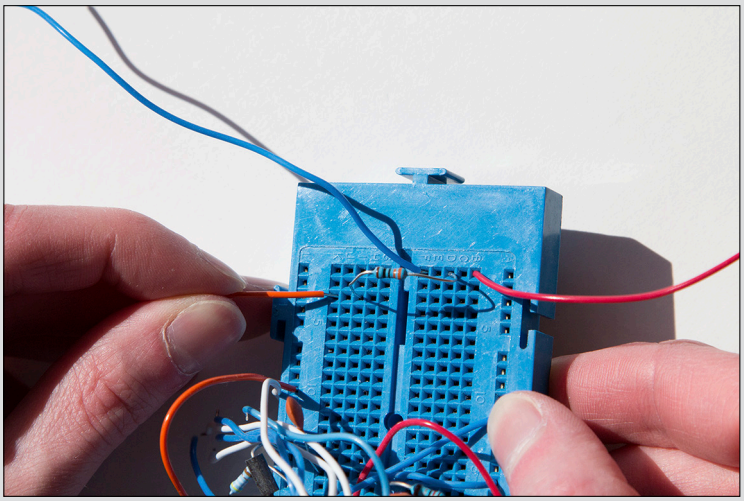


FIGURE 1-11: Using a breadboard.

- Solder (shown in Figure 1-12) – this is the most permanent way of making connections. *Solder* is a mixture of metals and is heated with a soldering iron until it melts and joins the connectors together. It's also possible to re-melt solder to separate connectors, although this is not always easy without damaging the components. Solder will also only join certain types of metals – for example, it won't stick to aluminium. Soldering can be a bit tricky at first; it takes a bit of practise to apply just the right amount of heat in the right place to avoid melting insulation, damaging sensitive components or burning your fingers! It's best to practise to join some scrap components and wire as your first few attempts will be unsuccessful.

It's best to become familiar with all methods of making connections; then you can use whichever method is most appropriate at the time. And, if nothing else, learning how to solder with molten metal can be fun!

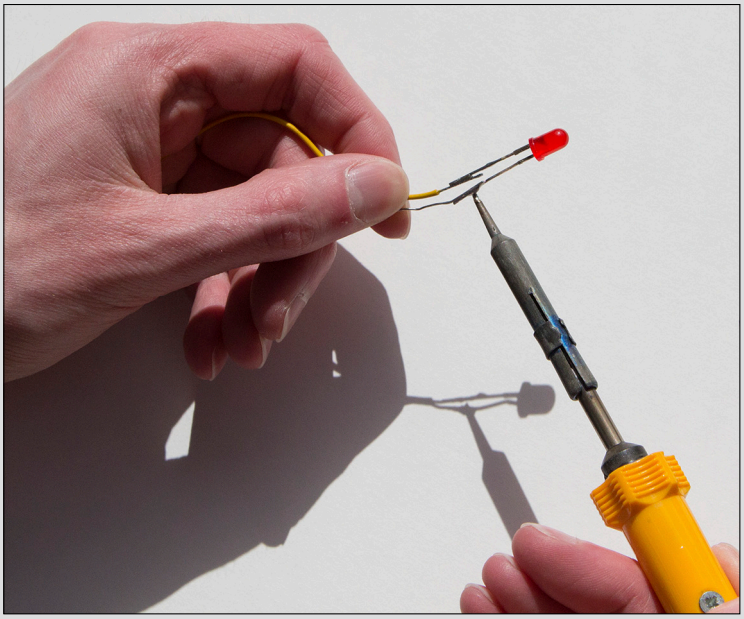


FIGURE 1-12: Soldering.

Cut four lengths of wire 20cm long and strip about 7mm of the insulation off each end. If you are using stranded wire (that is, there isn't a single core inside the insulation, but lots of fine strands), twist the strands together with your fingertips.

TIP

You can "tin" the wires with a soldering iron. This prevents metal whiskers sticking out that can accidentally short, causing undesired connections. To tin wires, twist them together, then run along with the metal from the insulation to the tip with a soldering iron on the top and solder on the bottom. Try and get the speed right so just enough solder flows to bind all the strands together.

Connect a Switch

Connect the switch to an input terminal (such as input 0) as shown in Figure 1-13. Start the emulator up to show the state of the inputs. Press the button and check that the input is registered. If nothing happens, check your wiring. Note that as pins 1-4 are wired in parallel to the switches, the terminal and switch indicate together.

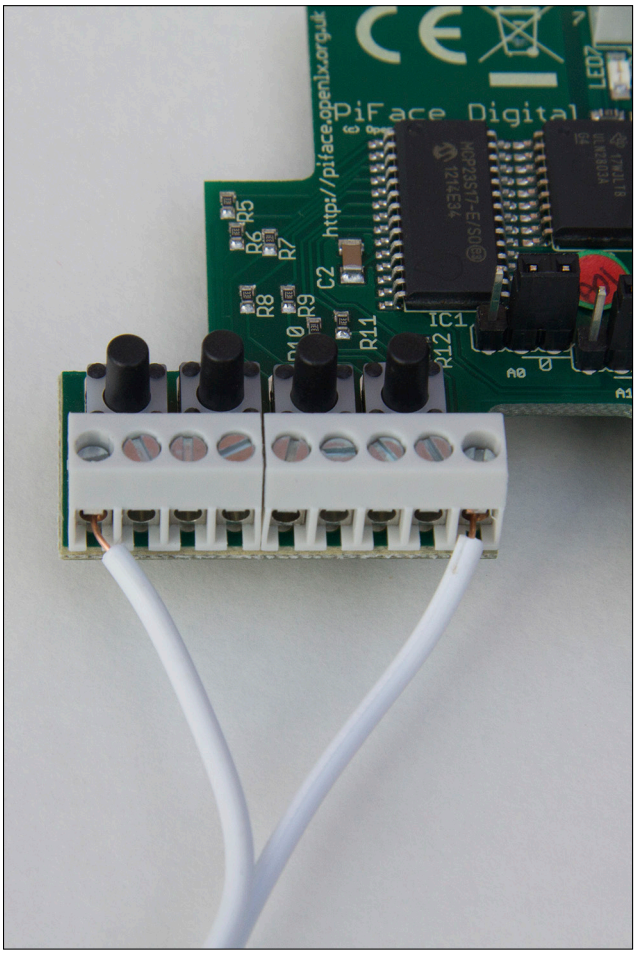


FIGURE 1-13: Wiring up a switch.

Connect an LED

You will connect the LED so current flows from the 5V terminal, through a resistor (to limit the current so the LED is not damaged) and the LED, then through the output terminal on PiFace, through the transistor and to ground. LEDs only work if current flows through them one way, so it matters which lead you connect to positive. For most LEDs the longer leg indicates the *anode*, which should be connected to the resistor and then the 5V terminal as shown in Figure 1-14.

Connect the LED and resistor together and then connect the anode of the LED to the 5V terminal. Now is a good time to check if the LED will work. Briefly touch the other lead of the LED, called the *cathode,* to GND. You should see the LED glow. If not, check your wiring and make sure that you have identified the anode and cathode correctly. (You shouldn't have damaged the LED if you got the polarity wrong.)

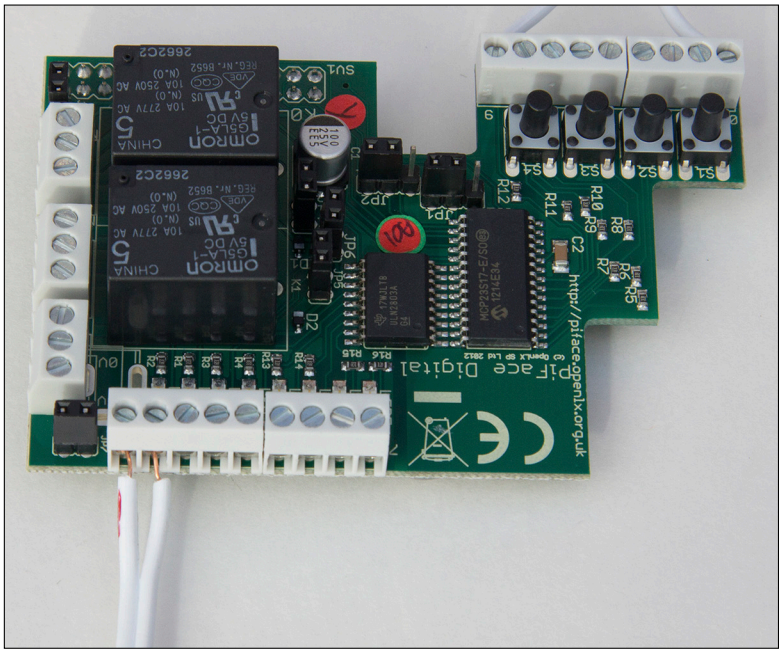


FIGURE 1-14: Wiring up an LED.

When you know the circuit works, disconnect from the ground and then connect to the transistor via one of the output terminals – such as output 0. When the transistor is turned on, it will allow the current to flow to ground and complete the circuit, so the LED will illuminate. Turn the output on from the emulator and check that it lights.

Playing with the Reaction Timer

Now that you have connected an LED and switch, rerun the reaction timer program. Again the LED should light after a random time and the switch should stop the timer. Now that you know your wiring and circuit works, you can really start to have fun – try different output devices, wire up a buzzer instead of the LED and see if your reactions to sound are quicker. You could perhaps mount a feather on a motor and test your reaction to touch. You might find this reaction to be quicker. This is because the inputs (nerves in the skin) can send a message to the outputs (muscles) without going through a complicated processing system (your brain) – the principles of computing apply to lots of other systems too! Experiment with different input switches too – you could attach a switch to different parts of your body and see if they respond as fast as your hand.

Have a go at making your own switch! Instead of a pre-made switch, you could wire up pieces of aluminium foil as the contacts and detect when they are connected. Maybe you could make a pressure pad for your foot, or have pieces of foil taped to your knees and complete the circuit by bringing them together.

YOUR TURN!

Make your own switch – such as a pressure pad – and see if you have the same reaction speed with your feet.

Connect up multiple switches and LEDs and modify the code to make a game. You could have a different score for each button, or deduct points if you press the wrong button at the wrong time. You could build your own game as shown in Figure 1-15, which is similar to games in amusement arcades like whack-a-mole.

When you become more experienced at coding, if you are really adventurous you could create a network version and play against a friend over the Internet.

FIGURE 1-15: My giant raspberry game.

What Will You Interface?

Computing becomes more interesting when it is connected to the world. Interfacing allows computers to sense and manipulate the world through inputs and outputs. Most computers sense the world by detecting a voltage on an input pin, and affect the world by allowing a current to flow through an actuator such as an LED or motor. Relays can be considered as an equivalent to switches, whereas open-collector outputs have to be wired up in a particular way.

With a basic understanding of electricity and the right sensors and actuators you can make the world a smarter place. As computers become more sophisticated we can build even smarter solutions. In the future, you might build a robot that would listen for your commands. Or one day it might even watch what you do and then learn to do the task for you. Although that may seem complicated to program, as you have seen, computing is about breaking a challenge down into lots of simple parts, and then each little part becomes solvable.

Chapter 2
The Twittering Toy

In This Chapter

- Make a soft toy read out tweets and moves
- Learn about text-to-speech
- Discover object-orientated programming
- Gain experience building Python modules
- Access Twitter from your program

IN THIS CHAPTER, you'll make use of the Twitter library to bring a soft toy to life. At the end of this chapter, you can have your very own animatronic chicken that waddles and moves its beak as it reads aloud tweets from a particular user or containing a certain hash tag.

This project illustrates that one of the joys of the Raspberry Pi is the ease of reusing existing code. Programmers strive for efficiency (some people call it laziness) and aim to never type the same thing twice. What's even better is never having to write the code in the first place and using someone else's!

This chapter will cover how to hack a toy to connect it to the Raspberry Pi and how to install a Python module to talk to Twitter and interact with an external program (in this case a text-to-speech engine) from Python.

Hacking the Toy

You are going to take an animated toy and "hack" it so the Raspberry Pi can control its movement. You'll do this by wiring one of the relays on the PiFace interface to replace the toy's on-off switch.

NOTE The word *hack* in regard to computing has become associated with illegal activity. In this sense, hacking is bad and not something to engage in. However, to programmers, a *hack* is a way to make something work, particularly when reusing something or repurposing it in a clever way.

There are many animatronic and robotic toys available online and in novelty shops. It's easy to modify the simple toys that have just a basic on-off switch. Before hacking your toy, it's worth considering what happens if something goes wrong – it's best not to hack an expensive toy or one you're particularly fond of, just in case you struggle to put it back together again. You may wish to build your own toy from components instead.

Building the Chicken

I chose to build a twittering chicken around the Head and Mouth Mechanism shown in Figure 2-1 from Rapid Electronics (`www.rapidonline.com/Education/Head-and-mouth-mechanism-60675`). The mechanism contains a battery case, a motor, gears and a switch. On this web page, you'll find a free data sheet with a fabric pattern for making the chicken cover (as well as for a bird and a whale with other mechanisms). These writing instructions make it easy to follow this chapter step by step. But you can substitute other mechanisms.

FIGURE 2-1: A naked chicken – the mechanism that makes the toy move.

Yellow fur for the body and red felt for the wattle and comb can be purchased from a local fabric shop, and a local craft shop or a market is an ideal hunting ground for suitable materials like stick-on eyes. What's great about building your own toy is the opportunity for customisation – feel free to experiment. It shouldn't be too hard to modify the chicken pattern into a "tux the penguin". You could share your pattern online for other people, too.

Wiring

You will connect the relay in parallel with the switch. This allows the relay to override the switch to turn the toy on.

Open the case by removing the four screws as shown in Figure 2-2, taking care not to lose any bits or move parts too far out of alignment.

TIP

An important skill of hacking is remembering how you take something apart so you can put it all back together. You could try filming it with a camera phone or similar, so you can play it back to see which part goes where.

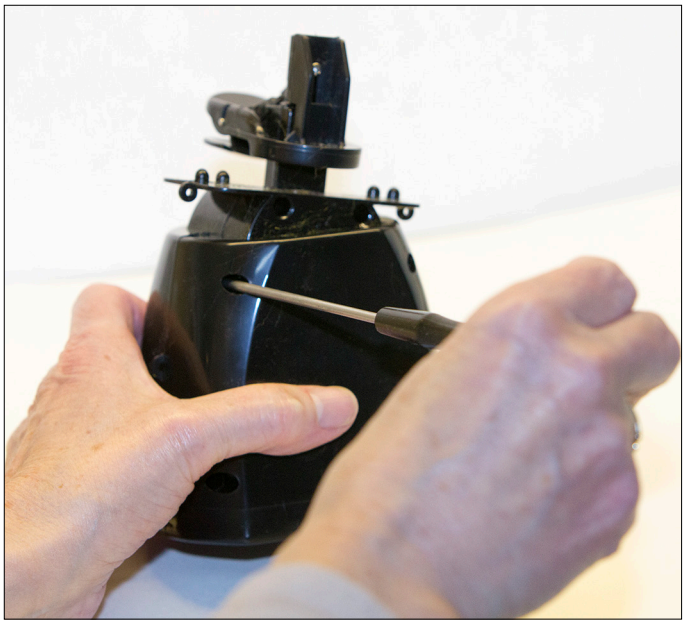


FIGURE 2-2: Removing the case of the movement mechanism.

Try and identify how the wires are connected to form the circuit. Find the terminals on the power switch. These may be covered with glue, but this can be carefully peeled or scraped away. In many cases, there will be only two wires going to the switch; if there are more, then the wiring is more complex, and it may be better to hack a different model if you cannot easily identify how the circuit works.

Series and Parallel Circuits

Series and *parallel* are two names to classify how components in a circuit are connected. In series, as the name suggests, the electric current flows through all the components in a series, one after another. As such, if you have switches wired in series, then all of them have to be closed for the current to flow. Breaking one of them will break the circuit.

In parallel, the flow of electricity splits and so closing any switch wired in parallel will allow the current to flow. This is because electricity tends to take the path of least resistance, and in this case, when the relay contacts close it has a much lower resistance than the almost infinite resistance of the open switch.

Attach another wire to each of the wires already connected to the terminals as shown in Figure 2-3. A soldering iron is the most secure way to do this, but you could also twist the wire onto the terminal and secure with tape. Briefly, touch the free ends of the wires you just joined together to check that your model moves. Find an appropriate hole to pass them through to the outside of the model, taking care that they don't catch on any moving parts. Adding a blob of glue or tying a loose knot on the inside will relieve some of the strain on the connections.

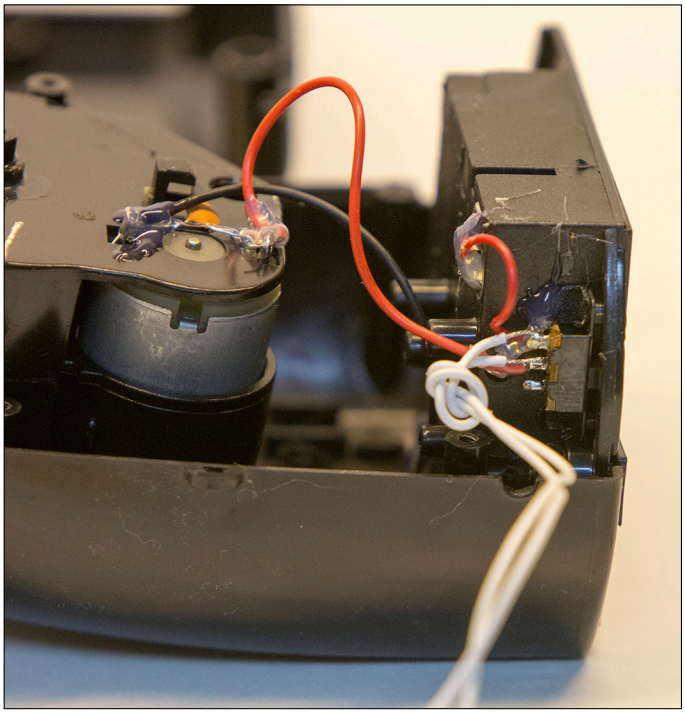


FIGURE 2-3: Add wires in parallel to the switch and add a knot to stop it from pulling through.

It is now time to reassemble your toy and the time when you discover how much attention you were paying when you took it apart! All things being well, your toy should go back together looking no worse after its surgery, as shown in Figure 2-4, and you shouldn't have any parts left over. Briefly touch the wires together again to make sure that your toy still comes to life.

FIGURE 2-4: Toy post-surgery showing no ill effects. Touch the wires together to make the toy move.

Making It Talk

You'll use the espeak text-to-speech (TTS) program to read aloud the tweets.

How TTS Works

A TTS engine typically works by splitting the words into syllables and then synthesising the word by concatenating (combining) corresponding sounds. In most TTS engines, the computer has no understanding of the words, so the output is monotone. However, TTS is an exciting area of research in computer science, attempting to make it sound more human. Latest research efforts include modelling the human voice box to generate more realistic sounds on virtual vocal chords, and another project to try and understand what is being said to vary pitch, delivery speed and rhythm.

After you've followed the instructions below to install espeak, you can see more about what espeak is doing if you run it with `-X`. For example, from a terminal run `espeak -X "hello world"`.

Uses of TTS

TTS is used when a visual display isn't appropriate. This includes assistive technology for people with visual impairments or when it is not possible to look at a screen, a satnav or automatic telephone exchanges.

These applications use TTS because the range of words that can be said is large. If the computer is required to say just a few hundred different words, then for better quality, an actor will record the separate words, with the computer forming sentences and playing back a sequence of clips separated by short pauses. In some applications, there are too many different words to have a recording of each one, so the computer will generate them from syllables. In the case of a satnav, it's a lot of work recording the name of every road and every place. Some words will be frequently used so they may be recorded, and a mixture of synthesized TTS and recorded may be used. You might like to try putting your own voice in the soft toy by recording a set of words and writing a program that plays them back. See the "More Ideas" section at the end of this chapter.

Install espeak by typing the following in a terminal:

```
sudo apt-get install espeak
```

Plug in some speakers and then test that TTS works by typing this in a terminal:

```
espeak "hello world from espeak"
```

TIP

If you cannot hear sound, you may need to change how audio is routed. The Raspberry Pi can output sound via the HDMI cable or the audio jack plug. It will try to automatically select the correct one, but that doesn't always work with some displays. To switch it by hand, type `sudo amixer cset numid=3 n`, where *n* is `0` for automatic, `1` for audio jack and `2` for hdmi.

Using Espeak in Python

The next step is to call the espeak program from Python. Because you might want other programs that you write in the future to use espeak, you'll create a module for it. It's worth studying how the module works because it shows how you can run another program from Python, which can be very useful. Enter the following code into `espeak.py`:

```
import subprocess

def say(words):
    #Speaks words via espeak text to speech engine
    devnull = open("/dev/null", "w")
    subprocess.call([
            "espeak",
            "-v", "en-rp", # english received pronunciation       ↵
   words],
            stderr=devnull)
```

Let's take a brief tour of the code. The first line imports the subprocess module that allows other programs to be called. Put simply, a program running on a computer is called a *process,* so if it wants to run another program, then (similar to calling a function) it creates another subprocess with a call.

`def` defines the `say` function that is passed a string of words to say. Next, the file `/dev/null` is opened for writing. On Linux, `/dev/null` is a special file that throws away anything written to it. It's a really good way of ignoring any output produced. In this case, it's used to hide any error messages produced by espeak. If you replaced `/dev/null` with a real filename, then any error message would be saved there instead.

Finally, the `subprocess.call` function calls the program `"espeak"` with an array of arguments to pass to it. In this case, if the `words` argument contained `hello`, it would be the same as typing the following:

```
espeak -v en-rp hello
```

This shows that multiple arguments can be passed to a command. The `-v en-rp` is used to specify how espeak sounds. It can be fun to play around with different pronunciations, or even languages.

Testing the Espeak module

Now is a good time to check the Python module you've just created. Enter the following into the file `try_espeak.py`:

```
#!/usr/bin/env python
# try_espeak.py
# Show use of espeak.py
import espeak as talker
def main():
    talker.say("Hello World")
```

```
if __name__ == '__main__':
    main()
```

Run `try_espeak.py` and check that you hear "Hello World" from the speakers.

YOUR TURN!

Try espeak with different pronunciation options, such as American or caricatures of British accents such as Northern, West Midlands or Scottish. The codes needed are described in the espeak documentation at http://espeak.sourceforge.net/languages.html. You could add another argument to the `say` function and pass information to set the accent. If you are really adventurous, you could try passing pitch, speed and voice arguments, and you'll get some very silly sounding voices.

Making It Move

The next step is to turn on the motors of the toy to make the mouth move while it is speaking. To control the motors, you'll use PiFace Digital. You should have set up PiFace Digital as specified in Chapter 1, "Test Your Reactions". Because there are LEDs on the board, these will indicate that the outputs are active so you can test the code before connecting the chicken. Create a new file named `chicken.py` and enter the following:

```
import piface.pfio as pfio
import espeak as talker

VERBOSE_MODE = True

class Chicken():
    #The wobbling/talking chicken
    def __init__(self, pfio, pin):
        pfio.init()
        self.pfio = pfio
        self.relay_pin = pin

    def start_wobble(self):
        #Starts wobbling the chicken
        self.pfio.digital_write(self.relay_pin,1)
        if VERBOSE_MODE:
            print "Chicken has started wobbling."

    def stop_wobble(self):
        #Stops wobbling the chicken
        self.pfio.digital_write(self.relay_pin,0)
```

continued

```
            if VERBOSE_MODE:
                print "Chicken has stopped wobbling."

        def say(self, text_to_say):
            #Makes the chicken say something
            if VERBOSE_MODE:
                print "Chicken says: %s" % text_to_say
            talker.say(text_to_say)


def test_chicken():
    #create a sample Chicken object called tweetyPi
    tweetyPi = Chicken(pfio,1)
    tweetyPi.start_wobble()
    tweetyPi.say("hello world")
    tweetyPi.stop_wobble()

if __name__ == '__main__':
    test_chicken()
```

With a speaker connected, run the preceding script and check that it works. You should hear the relay click (and see the corresponding LED illuminate) on the PiFace interface, the sound "hello world" come from the speaker and then the relay click off again.

Looking back at the code, there are a couple of points to notice. The line `VERBOSE_MODE = True` is an example of a constant. It is set by the programmer and then never changes – in other words, it remains constant. In this case, it provides an easy way to turn on and off debugging information in one place. With it turned on, you'll receive messages saying when the chicken should start moving and stop. Without these messages, if the chicken wasn't moving, you wouldn't know if the problem was with this module, or whether the problem was elsewhere.

TIP Being able to see what is going on in a program is very important when debugging. Computer scientists describe it has having *good visibility*. One way to achieve this is to have `print` statements at key points in a program so as it executes you can follow its progress and the state of data. It's very common for a program to not work perfectly the first time, so designing it to make it easy to debug can save time in the long run, particularly when you might add a new feature later.

Creating Classes

The file `chicken.py` makes uses a particular style of programming called *object-orientated programming (OOP)*. You don't need to worry too much if you don't understand the code at the moment.

OOP is a handy way to group code and data together. Think about any object in the real world: It has characteristics, also called *attributes,* and it has things that can be done to it. For example, a balloon's attributes include its colour and if it is inflated. Actions that can be done to it include blowing it up or popping it. You may have a room full of different balloons, but they can all be considered to share the same set of attributes and actions (even though the value of the attribute, like the colour, might be different). As such, they are all the same class of object.

In OOP, the programmer designs his or her own classes of objects, which have a set of attributes and functions. In `chicken.py`, the `Chicken` class of the object is defined by `class Chicken():`. The next indented block defines the methods (another name for functions) of the class. There are methods to start and stop the chicken from moving and one to make it speak. The `__init__` method is a special method that is called when an object is created. In this case, it calls the `pfio` initialisation method (to initialise the PiFace interface) and sets up the object's attributes. This can be handy, as you can use your object without worrying about what you need to do to initialise things first. So far, the program has only created a class for the chicken. It hasn't created an actual object, merely a description of what a `Chicken` object will be like. It's a bit like creating a cookie cutter – you've defined what shape the cookies will be, but you've not actually created any yet.

Creating Objects

A `Chicken` object is created in the `test_chicken` function. It is outside the class definition (and therefore not part of the `Chicken` class) because it's not within the indented block. The statement `tweetyPi = Chicken(pfio,0)` creates a new `Chicken` object called `tweetyPi`. The arguments are used to pass in the PiFace digital interface and identify which pin the motor is connected to.

Breaking Up Your Code

Appendix B, "Introductory Software Project: The Insult Generator", talks about splitting programs up into functions and how important it is to structure your code in computer science.

In this example, separate files are used for modules to help structure the program. Classes also help by grouping related data and functions together.

As you have seen, what may have sounded like a daunting task of making a Twitter-enabled soft toy move and talk becomes manageable when tackled in smaller chunks. You also tested each chunk so it's easier to see where a problem is. Similarly, as you become more experienced, you'll be able to take almost any big and seemingly hard problem, and split it up into little steps. Why not try it? Think of a project you want to do, and then think how you can split it up into smaller parts. Hopefully, as you complete the projects in this book, you'll learn the skills necessary to implement each of these little parts, and you'll be able to build almost anything!

Imagine if you had multiple chickens, one connected to each of the relays on PiFace. You could create two chickens by typing the following:

```
tweetyPi1 = Chicken(pfio,0)
tweetyPi2 = Chicken(pfio,1)
```

Now, if you wanted to start them both wobbling you could type this:

```
tweetyPi1.start_wobble()
tweetyPi2.start_wobble()
```

By using objects, you can use the same function for each chicken. You don't have to worry about which pin they are connected to after you've created the object because the object stores it for you. This is an example of why using OOP can be advantageous.

OOP can be tricky to understand at first, but it should become clearer as you see more examples. For now, you can ignore the details, accept that it works and hide it in the chicken module, in the same way you use other modules without knowing what is inside them. Believe it or not, professional programmers do this too – as long as the interface to a module is clear (that is, the functions it provides are clearly documented), it doesn't matter if they don't fully understand how it works!

Testing As You Go

If you wrote a huge program and tried to run it, chances are it wouldn't work the first time. You'd then have to go through all of it trying to find where the problem was. Instead, it is better to test as you go, checking each component before moving on to the next. Python provides a good means to do this. Toward the end of the file are the following lines:

```
if __name__ == '__main__':
    test_chicken()
```

This code calls the `test_chicken()` function if the file is being run by itself, but doesn't call the function if it is imported as a module. As such, it's a good way of writing code that will test the behaviour of a module. As you learn more about programming, you will understand the importance of testing and which tools and techniques can help.

TIP Surrounding or starting a word with __ in Python (and some other languages) indicates a special meaning. As such, it's better not to start and end your own variables and functions this way unless you really know what you're doing!

Connecting to Twitter

The `python-twitter` module makes it very easy to read from Twitter. Unfortunately the module isn't prepackaged for Debian Linux. Luckily it's not too difficult to build it from source and doing so will give you good experience that will come in handy if you need to install another module in the future. You'll also see what it's like to use someone else's module, which will be an advantage if you write modules that you want other people to reuse. You'll discover that it is just as important to write good documentation as it is to write good code.

Building and Installing Python Modules

The module's home page http://code.google.com/p/python-twitter contains a summary of how to build the module. If you've never built a module before then you're better off following the more detailed steps in this chapter.

The website lists dependencies; these are other modules that must be built first. python-twitter requires simplejson, httplib2 and python-oauth2 to be installed. Step-by-step installation instructions are provided in this chapter.

TIP It's possible to download files from the command line in Linux without using a web browser. There are two programs to choose from: either `curl` or `wget`. Both provide similar functionality, so deciding which one to use comes down to personal preference and/or availability. `wget` is used for the examples in this chapter.

simplejson

Clicking the http://cheeseshop.python.org/pypi/simplejson link redirects you to http://pypi.python.org/pypi/simplejson.

TIP Note that the version numbers may be different as the library is updated, in which case, you should replace `simplejson-3.3.0.tar.gz` with whatever filename you have downloaded.

Using tar

You can create your own zipped archives by typing the following:

```
tar czvf <archivename.tar.gz> <list of files and directories>
```

Tar has many different options, but in most cases `czvf`, `xvf` or `tvf` will be sufficient. `t`,`c` and `x` mean test (list the contents of an archive), compress and expand an archive, respectively. `v` indicates that tar should be verbose and list the files as it expands them. `f` is used to specify the filename of the archive.

From a terminal, type the following (all on one line) to download the code:

```
wget http://pypi.python.org/packages/source/s/simplejson/
simplejson-3.3.0.tar.gz
```

The `tar.gz` file extension tells you that the file is zipped to save space and is a *tar archive*. A tar archive is often used in Linux as it provides a convenient way to package multiple files and preserve file permissions and other attributes. It is possible to unzip the file and then untar a file as separate operations, but because so many tar archives are compressed, it is possible to do it in a single action. To unzip and untar the compressed archive, type the following on the command line:

```
tar xvf simplejson-3.0.7.tar.gz
```

As the command executes, it lists the files as they are expanded (unpacked) from the archive.

Change into the newly expanded directory by typing the following in a terminal:

```
cd simplejson-3.3.7
```

On Linux, most software that is supplied as source code shares a similar installation process of extract, build and install. Many Python modules follow this same pattern. Because a malicious user could insert a module that would cause harm, you need to use `sudo` to provide enough access privileges to install the module. Type the following to install the package:

```
python setup.py build
sudo python setup.py install
```

After the module has installed, return to the parent directory by typing the following:

```
cd ..
```

httplib2

Follow the same procedure to install the `httplib2` package from http://code.google.com/p/httplib2. In a terminal, type the following:

```
wget http://httplib2.googlecode.com/files/httplib2-0.8.tar.gz
tar xvf httplib2-0.8.tar.gz
cd httplib2-0.8
python setup.py build
sudo python setup.py install
cd ..
```

python-oauth2

python-oauth2 is hosted on GitHub at http://github.com/simplegeo/python-oauth2, so you obtain it through `git` rather than `wget`. In a terminal, type the following:

```
git clone "http://github.com/simplegeo/python-oauth2"
cd python-oauth2
python setup.py build
sudo python setup.py install
cd ..
```

The final step is to install the `python-twitter` module.

```
wget "http://python-twitter.googlecode.com/files/
python-twitter-0.8.2.tar.gz"
tar xvf python-twitter-0.8.2.tar.gz
cd python-twitter-0.8.2
python setup.py build
sudo python setup.py install
cd ..
```

If you have IDLE open, close all the windows and restart it so it can access the newly installed modules.

Talking to Twitter

Twitter requires programs that access it automatically to send authentication information with requests. This allows Twitter to prevent abuse by blocking programs that put too much load on its servers. Your program will authenticate itself to Twitter by sending secret tokens.

Getting Access Tokens for Twitter

You will need to get four pieces of information by signing into the developers' part of Twitter. This section explains how to get a `consumer_key`, `consumer_secret`, `access_token_key` and `access_token_secret`. The names sound confusing, but all they are is a secret code that will identify you and your program to Twitter.

Visit `https://dev.twitter.com/apps/new` and log in. (You will need to sign up to Twitter if you don't have an account.) If you're not old enough to have your own Twitter account you could ask a parent, guardian or teacher to do this for you.

Enter the name of your application, a brief description and a website. If you don't have a website you could enter `www.example.com` as a placeholder. Read the terms and click to indicate your acceptance of the terms. Fill in the CAPTCHA and then click Create Your Twitter Application. You may need to enter a different name for your application if it is already in use; you could try prefixing it with your Twitter username.

Upon success, scroll down and click Create My Access Token. Wait up to a minute and reload the page.

Make a note of the Consumer Key and Consumer Secret entries from the OAuth section, and Access Token and Access Token Secret from the Your Access Token section because you will need to include these in your program.

Writing Code to Talk to Twitter

With the Python modules installed, it is time to write the code that will talk to Twitter. Create a new file named `twitter_tag_listen.py`, containing the following code:

```
#!/usr/bin/env python

#twitter_tag_listen.py
#listens for new tweets containing a search term

import time
import sys
import twitter

DEFAULT_SEARCH_TERM = "chicken" #what we search twitter for
TIME_DELAY = 30 # seconds between each status check

def main():
#replace values for consumer_key, consumer_secret,                ↵
    access_token_key and access_token_secret with the values      ↵
    you made a note of from the Twitter website.
        api = twitter.Api(consumer_key='xxxxxxcxK9I3g',           ↵
              consumer_secret='xxxxxfLBmh0gqHohRdkEH891B2XCv00',  ↵
              access_token_key='xxxxx25-                          ↵
              Dw8foMCfNec2Gff72qxxxxxwMwomXYo',                   ↵
              access_token_secret='xxxxxxjIuFb88dI')

        previous_status = twitter.Status()

        # has user passed command line argument?
        if len(sys.argv) > 1:
                search_term = sys.argv[1]
        else:
                search_term = DEFAULT_SEARCH_TERM

        print "Listening to tweets containing the word '%s'."
              % search_term
```

```
        while True:
                # grab the first tweet containing the
                search_term
                current_status = api.GetSearch(term=search_term,
                count=1)[0]

                # if the status is different then
                # pass it to the chicken

        if current_status.id != previous_status.id:
                        #if the result we get from twitter is
                        #different from what we got last time,
                        # we know there's a new tweet
                        print current_status
                        previous_status = current_status

                # wait for a short while before checking again
                time.sleep(TIME_DELAY)

if __name__ == "__main__":
        main()
```

Run the code to test it. Obviously, you need to be connected to the Internet for it to work. The program should print the latest tweets that contain the word *chicken*. Press Ctrl + C to stop the program.

You are nearly ready to add in the code that controls the toy, but before you do, it is worth looking at how the code works.

By now you should be recognising statements in the program. First you import three modules that you need. The Time module provides the `sleep()` function that creates the delay between each time you check Twitter. The Sys module provides a way to get command-line arguments. Next constants are defined before the main function begins.

The Twitter module is written in OOP style, so you create an `api` object with the following statement:

```
api = twitter.Api(consumer_key='xxxxxxcxK9I3g',                    ↵
      consumer_secret='xxxxxfLBmh0gqHohRdkEH891B2XCv00',           ↵
      access_token_key='xxxxx25-                                   ↵
      Dw8foMCfNec2Gff72qxxxxxwMwomXYo',                            ↵
      access_token_secret='xxxxxxjIuFb88dI')
```

Command-Line Options and Arguments

When you run a Python program from the command line, you can follow it with a list of arguments. This provides an easy way to pass data into a program at startup and is typically used to pass names of files or configuration options. For many programs, if you run it with just the `–h` option it will display simple help and summarise the options and arguments available. To try this with espeak, type the following from a terminal to display a summary of command-line options:

```
espeak -h
```

In the `twitter_tag_listen.py` example, because your program only takes one argument, you read it from the list held by `sys.argv`. However, as you begin to master Python and your programs get more complicated, you may wish to use the `argparse` module that splits up the arguments and can automate generating usage information.

You check if any command-line arguments were used by checking the length (that is, the number of items) in `sys.argv`. If there's at least one, then you set the `search_term` to be the first argument:

```
# has user passed command line argument?
if len(sys.argv) > 1:
        search_term = sys.argv[1]
else:
        search_term = DEFAULT_SEARCH_TERM
```

Finally, you enter the main loop – a block of code that will keep going round in a loop, running again and again. This is a `while` loop, which is discussed in Appendix B.

In this program you use the condition `True`, to make it loop indefinitely. To stop the program, press Ctrl + C. In Linux, this keyboard combination sends a message from the operating system to interrupt the program, which in most cases will *terminate* it (cause it to stop running).

With all the components written and tested, it's the moment of truth: Will they work together?

Putting It All Together

Connect the wires from your toy to the common, and normally open, relay contact terminals on PiFace Digital as shown in Figure 2-5. The code example in this chapter uses relay 0, which is the bottom two terminals nearest to JP3 and the Raspberry Pi expansion pins. You can use the manual override button in the emulator as described in Chapter 1 to check that the toy moves when the relay is turned on.

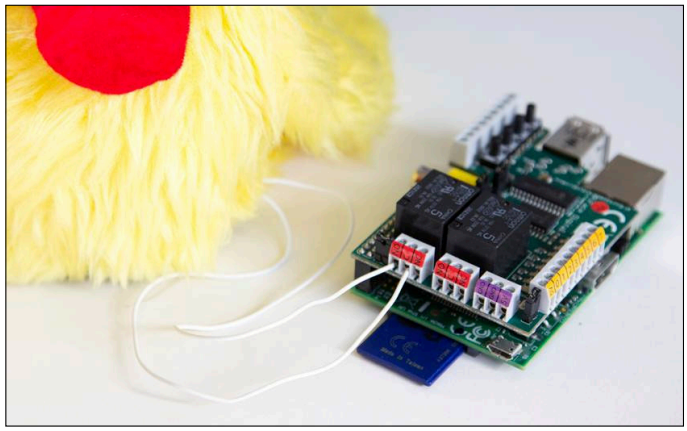


FIGURE 2-5: Wiring the toy up to the normally open relay contacts.

Update the `twitter_tag_listen.py` code as follows:

1. Import your chicken and `piface.pfio` modules.
2. Create a Chicken object called `chick` and pass in the number of the output pins wired up to the toy.
3. Instead of printing a tweet, start the chick wobbling, say the tweet, and then stop the chick from wobbling.

The code for `twitter_tag_listen.py` is shown in full in Listing 2-1.

Listing 2-1 `twitter_tag_listen.py`

```
#!/usr/bin/env python

#twitter_tag_listen.py
#listens for new tweets containing a search term
#and then wobbles a chicken

import time
import sys
import twitter

import chicken
import piface.pfio as pfio

DEFAULT_SEARCH_TERM = "chicken" #what we search twitter for
TIME_DELAY = 30 # seconds between each status check
```

continued

```
def main():
        api = twitter.Api(consumer_key='xxxxxxcxK9I3g',                    ↵
        consumer_secret='xxxxxfLBmh0gqHohRdkEH891B2XCv00',                 ↵
        access_token_key='xxxxx25-                                         ↵
        Dw8foMCfNec2Gff72qxxxxxwMwomXYo',                                  ↵
        access_token_secret='xxxxxxjIuFb88dI')
        previous_status = twitter.Status()
        chick = chicken.Chicken(pfio,0)

        # has user passed command line argument?
        if len(sys.argv) > 1:
                search_term = sys.argv[1]
        else:
                search_term = DEFAULT_SEARCH_TERM

        print "Listening to tweets containing the word '%s'." %           ↵
        search_term

        while True:
                # grab the first tweet containing the
                # search_term
                current_status = api.GetSearch(term=search_term,           ↵
                count=1)[0]

                # if the status is different
                # then pass it to the chick.say function
                if current_status.id != previous_status.id:
                        chick.start_wobble()
                        chick.say(current_status.text)
                        chick.stop_wobble()
                        previous_status = current_status

                # wait for a short while before checking again
                time.sleep(TIME_DELAY)

if __name__ == "__main__":
        main()
```

Try it! Run the code and you should have an animated, talking toy that responds to Twitter. Don't forget you can make the file executable by running

```
chmod a+x twitter_tag_listen.py
```

Try out different search terms by passing in arguments. For example, type the following to search for the hash tag `RaspberryPi`:

```
./twitter_tag_listen.py "#raspberrypi"
```

TIP

Note that you have to enclose the tag in quotes. This tells the command line to ignore the special meaning that # has, and to pass it through to your program as part of the argument.

Wrapping Up

By now you should have your own animatronic, twittering soft toy. You've also seen the advantages of breaking a program up into manageable parts and reusing other people's code. Becoming a programmer is sometimes like being a plumber – it's about connecting up functions, with data flowing between them. If you're struggling with your own program design, try splitting it up into a set of smaller problems and keep splitting until all the problems are really simple to code. Not only is it less daunting at the design stage, but by testing stage-by-stage, it's harder for bugs to hide and easier for you to find them. When writing functions, you've seen the need for good *observability for testing and debugging* – that is, you can see what is going on inside them so you can check that they work and fix them when they don't!

You've also seen the need for good documentation. If other people are going to reuse your code, then they need clear instructions about how to install the program, what other modules it depends on and what arguments each function takes and does.

Practically, you've also learned about using tar and untar for packaging up sets of files and how to build and install Python modules.

YOUR TURN!

There are lots of ways you can customise the twittering soft toy; why not try some of the following suggestions? Don't forget to film what you make, upload it to YouTube and tag it with `Raspberry Pi Project Book`.

More Ideas

There are many things you can do with your own toy. Here are some suggestions:

- Try changing the arguments passed to espeak. (Don't forget that `–h` will give you a list of options.) For example, you could try
  - Different languages
  - Different voices, both male and female

- Changing pitch
- Changing speed

Here's an example of how you could change `espeak.py`:

```
import subprocess

DEFAULT_PITCH = 50  # 0-99
DEFAULT_SPEED = 160 # words per min


def say(words, pitch=None, speed=None):
    if not pitch:
        pitch = DEFAULT_PITCH

    if not speed:
        speed = DEFAULT_SPEED

    devnull = open("/dev/null", "w")
    try:
        subprocess.call([
            "espeak",
            "-v", "en-rp", # english received pronunciation
            "-p", str(pitch),
            "-s", str(speed),
            words],
            stderr=devnull)
```

- You could change the speech parameters depending on who is talking or the content of the tweet.
- You could connect multiple soft toys, each with different voices, and have a conversation. (Hint: Create `chick0 = chicken.Chicken(pfio,0)` and `chick1 = chicken.Chicken(pfio,1)` and then wire a different chicken up to each relay.)
- You could control more motors in your soft toy and make it dance if it detects certain hash tags.
- If you're really adventurous you could replace the `espeak.py` module completely with a module that looks up and plays sound recordings of yourself saying particular words.

Chapter 3
Disco Lights

In This Chapter

- Using individual bits in variables to define individual LEDs
- Connecting external LEDs to your PiFace board
- Writing a Python program user interface that looks like any windowed application
- Customising the user interface to use your own choice of colours
- Getting input from music

IN MY YOUTH, during the late 60s, I answered an advertisement in the *Manchester Evening News* for someone to turn lights on and off in time to the music in an Ashton-under-Lyne night club. I went to the interview that Friday evening, which consisted of their showing me the lighting control rig and saying I had to be there by 7:30 p.m. on Saturday and Sunday. To be honest, I didn't fancy five hours of switching lights on and off for just £1.00, so I arrived the following evening on my Lambretta with a rucksack full of electronics. I had a large multiway switch used in telephone exchanges called a uniselector (you can still get these on eBay), which was wired up to make an on/off control of five circuits. I started hacking the lighting panel, and before long, I had five coloured spotlights flashing away while I put my feet up.

These days, you cannot go hacking about with mains like that – health and safety would have a fit. And with the Raspberry Pi, you have the opportunity to do something a lot more sophisticated. So in this chapter, you are going to see how to control those disco lights, and change the pattern with a click of a mouse. Not only that, but you will see how to drive the light sequence from the beat of the music.

In this chapter, you'll learn how to write a Python program to define a sequence of lights. You'll also learn about various aspects of electronics and control.

Defining Your Sequence

So far in this book, you have written programs that interact through the Python console. Now you are going to produce a proper desktop application. This would be quite a daunting prospect if it were not for the help that you can get from a Python package that does a lot of the under-the-hood hard work for you. This just leaves you to specify exactly what things should look like. This package also integrates the windows style selected for your whole desktop, so the result looks consistent with other applications.

This package is called `pygame` and comes preloaded in most Raspberry Pi distributions. It consists of a number of functions to create and update windows, draw in the windows, register a mouse click and read the keyboard. It will also handle sound and music. but you will not be looking at that function this time.

Start IDLE, and select a new window. For a start let's look at Listing 3-1, a very basic piece of code to open a window and close it down.

Listing 3-1: Windows1 Test Program

```
#!/usr/bin/env python
"""
Window1 to open up a window on the desktop
"""
import os, pygame, sys
```

```
pygame.init()                    # initialise graphics interface
os.environ['SDL_VIDEO_WINDOW_POS'] = 'center'
pygame.display.set_caption("Test Window 1")
screen = pygame.display.set_mode([788,250],0,32)

def main():
        while True :
            checkForEvent()

def terminate(): # close down the program
    print ("Closing down please wait")
    pygame.quit()
    sys.exit()

def checkForEvent(): # see if we need to quit or
                     # look at the mouse
    #print "checking for quit"
    event = pygame.event.poll()
    if event.type == pygame.QUIT :
           terminate()
    elif event.type == pygame.KEYDOWN                                    ↵
      and event.key == pygame.K_ESCAPE :
            terminate()

if __name__ == '__main__':
    main()
```

When you run this, you should get just a black window in the middle of the screen. It won't do much, but it is a real window. You can drag it around the screen, and clicking the minimise icon at the top-right corner will fold up the window and put it on the task bar at the bottom of the screen. Clicking the close cross will quit the program as will pressing the Esc key. When the program quits, you will get a message printed out in blue in the console window along with several lines of red debug information telling you where the program quit.

If you look at the anatomy of the program, you will see things are quite simple. The first few lines tell `pygame` to create a window, of a certain size, with a certain title and put it in the middle of the screen. The main part of the program is an infinite loop that constantly checks to see if an event has occurred.

In programming terms, an *event* is something happening, which is normally how user interaction gets input to the program. You are looking for a close event or a key up event on the Esc key. A *close event* is either the user clicking the close cross on the window or the operating system telling the program to quit because it is going to shut down. If your program sees any of those events, it calls a `terminate` function that prints out a message. Then it quits `pygame` to release any memory it grabbed, and it exits to the operating system.

Getting the Code to Do More

Well, that was not very exciting, was it? Let's get the code to do a little more. Take a look at Listing 3-2:

Listing 3-2: Windows2 Test Program

```
#!/usr/bin/env python
"""
Window2 to open up a window on the desktop, draw something in it
   and read the mouse position upon a click
"""

import piface.pfio as pfio      # piface library
import os, pygame, sys

pygame.init()                   # initialise graphics interface
pfio.init()                     # initialise pfio
os.environ['SDL_VIDEO_WINDOW_POS'] = 'center'
pygame.display.set_caption("LED controll")
screen = pygame.display.set_mode([190,160],0,32)
box = False

def main():
        drawBox(box)
        while True :
            checkForEvent()

def drawBox(state):
        boxNum = 0
        # first draw the box
        # - fill colour depends on sequence bit state
        if state :
                pygame.draw.rect(screen,(255,0,0),                ↵
                  (50, 70, 40,40), 0)
        else :
                pygame.draw.rect(screen,(180,180,180),
                  (50, 70, 40,40), 0)
        #now draw the outline of the box
        pygame.draw.rect(screen,(0,0,180),(50, 70, 40,40), 3)
        pygame.display.update() # refresh the screen
        pfio.write_output(state)

def mouseGet() : # see where we have clicked
        global box
        x,y = pygame.mouse.get_pos()
```

```
        print "The mouse has been clicked at ",x,y
        if x in range(50,90) and y in range(50,110) :
                box = not box # toggle the state of the box
                drawBox(box)

def terminate(): # close down the program
    print ("Closing down please wait")
    pygame.quit()
    sys.exit()

def checkForEvent(): # see if we need to quit
                     # or look at the mouse
    #print "checking for quit"
    event = pygame.event.poll()
    if event.type == pygame.QUIT :
           terminate()
    elif event.type == pygame.MOUSEBUTTONDOWN :
            mouseGet()
    elif event.type == pygame.KEYDOWN                                    ↵
      and event.key == pygame.K_ESCAPE :
            terminate()

if __name__ == '__main__':
    main()
```

When you run this, you will get a much smaller window on the desktop with a single square in it. Click in the square, and four things will happen. First the square will turn from grey to red, and then you will hear the relay on the PiFace board come on and see one of the LEDs come on. Finally you will see the position of the mouse when it was clicked printed out in window coordinates. That means that the coordinate value will be the same when clicking the same spot within the window, irrespective of where that window is positioned on the screen. Let's see what's been done here.

This time you add in the call to import the `piface` library, which is going to control the relay and lights. You set a variable called `box` to be false, which is called a *logic* or *Boolean value* and can only take on one of two values. You can call these values one and zero or true and false. The main function calls a `drawBox` function and then enters an endless loop that simply checks the events.

Take a closer look at the `drawBox` function. It takes in a variable, called `state`, that defines what colour the box is going to be. It is tested and you use the draw rectangle command from `pygame`. At first this looks complex, with lots of parameters or numbers in it, but it is quite simple. The first parameter in the command tells where you are going to draw the rectangle, in this case in an area called `screen` you defined at the start of the program. The next three

numbers define the colour you will use in red, green and blue values – these are in brackets because they are one entity that could be replaced by a suitable variable later on called a *tuple*. Next you have four values bracketed as a tuple that define the X and Y coordinates of the rectangle, followed by how wide and how high to draw it. The final value tells the computer how wide a pen to draw this with. A zero is a special case and fills in the whole rectangle. Finally after drawing the rectangle, you have to tell the computer to update what is being shown on the screen.

This way of working means that no matter how complicated or time consuming it is to draw, the user always sees the screen change in a flash. The technical name for this is *double buffering,* because one buffer, or area of memory, is being used to display the picture, and the other is being used to construct the next picture. The display update call copies the construction buffer into the display buffer. Note that at this point the display and construction buffers both contain the same thing. Finally in this function the variable state is written out to the PiFace board. As this Boolean variable can only be a zero or a one, then the least significant LED is turned on or off, and all the other LEDs are turned off.

The last thing to look at in this program is the `mouseGet` function, which is called by the `checkForEvent` function when it detects a mouse button down event. The `mouseGet` function first recovers the coordinates of the mouse pointer when it was clicked. Then the compound `if` statement checks if both the `x` and `y` fall within the coordinates of the box. If it does, then you toggle or invert the state of the variable box and then call the function that draws it and writes to the outputs.

So with a mouse click, you can control a light.

A Small Detour into Theory

Now you've got a program that doesn't just act like a real window application; you can click in the window and control an output. However, before you can go on to looking at a complete sequencer you need to look a little bit about how the LEDs on the PiFace board are related to the value you write to the interface.

The basic unit of storage in a computer is the byte. A *byte* consists of eight bits, each bit being a separate logic level. Rather than thinking in bit terms, it is easier if you group these bits and consider storage in terms of bytes. However, as you will see you sometimes want to manipulate individual bits in that byte. In the last program you saw that a Boolean variable could only have one of two possible values; however it takes a byte to store that one variable, so all the other bits in it are wasted. If you take a byte you can store the state of eight LEDs in it. The relationship between the byte, the bits and the LEDs is shown in Figure 3-1.

So by using a byte variable to store the state of all eight LEDs you can then use a list of these variable to store a sequence of LED patterns. To output each pattern all you have to do is to write out the next variable in the list to the PiFace board.

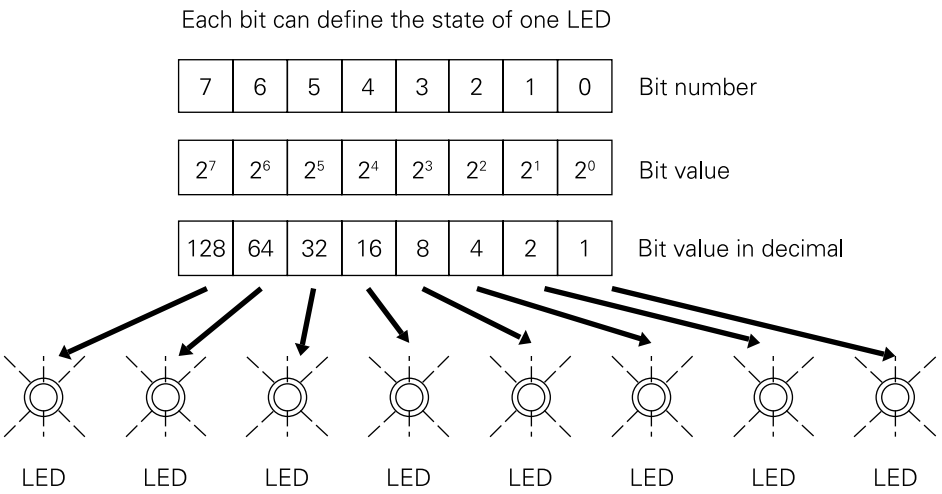


FIGURE 3-1: The relationship between bits and bytes.

Designing the Sequencer

Now that you have all the elements in place you can begin to think about how you want this to look and operate. This is called *top-down design* because you start with a top-level view of what you want the software to look like.

I envisaged a grid of squares, each one representing an LED and its position in the sequence. A column of eight squares would represent the states of the LEDs at any instance in the sequence. A marker under the column will indicate what LEDs are being lit at any time. A mouse click in one of these squares will toggle the LED.

In order to help set up the sequence there should be some control buttons, one to clear or turn off all the LEDs in all sequence positions, and another to toggle or invert all the LEDs. There should be control buttons to determine the speed of the sequence and finally one to select where to take the trigger to advance the sequence from. This last point is important if you want to synchronise the changing of the lights to the beat of the music. The sequence can either be stepped at a regular rate determined by the speed controls, or locked into the beat of the music. This last feature will require a little bit of extra hardware and is optional – you can skip it for now and add it later if you like.

Finally it would be good if all the colours used in the sequence software were customizable; that is, it should be easy to change by changing a single value at one point of the code only. This means that whenever you use a colour you do not hard code it in by putting the colour numbers into a function call, but rather use a variable to define that colour. Those variables for all the colours should be grouped in one place in the code for easy access.

Implementing the Sequencer

After designing the sequencer from the top down, when it comes to implementing the design it is better to write the code in what is known as a *bottom-up* implementation. That means starting at the lowest possible function and working your way up. Of course, if you just look at the finished code you don't see that. I started by taking the window test program and writing the functions that showed the columns representing the LEDs in the sequence. Then I expanded it so that I could click on each LED to turn the box on or off. Next came the controls to clear and invert, followed by the step indicator. This was followed by the speed controls and at this point I added the code to actually output something to the LEDs. Finally the auto/external step control was added and the code tidied up. This might not be the sequence of building up a program that first springs to the mind of a beginner, but the idea is to do a little coding and then test. So you are always looking to do something that can be instantly tested, even if it means writing the odd line of code that is not going to make it in the final mix. It also means that if something goes wrong with the test you know you have just written the code with the error in it.

Listing 3-3 shows the sequencer application.

Listing 3-3: The Sequencer Application

```
#!/usr/bin/env python
"""
Disco LED sequence display on the PiFace board
"""
import time                     # for delays
import piface.pfio as pfio      # piface library
import os, pygame, sys

pfio.init()                     # initialise pfio
pygame.init()                   # initialise graphics interface
os.environ['SDL_VIDEO_WINDOW_POS'] = 'center'
pygame.display.set_caption("LED sequence controller")
screen = pygame.display.set_mode([788,250],0,32)
background = pygame.Surface((788,250))

# define the colours to use for the user interface
cBackground =(255,255,255)
cLEDon = (255,0,0)
cLEDoff = (128,128,128)
cOutline = (255,128,0)
cText = (0,0,0)
cTextBack = (220,220,220)
cStepBlock = (0,255,255)
```

```
background.fill(cBackground) # make background colour
font = pygame.font.Font(None, 28)
seq = [ 1 << (temp & 0x7) for temp in range (0,32)]
   # initial sequence
timeInc = 0.3
stepInt = True # getting the step signal from inside the Pi
step = 0 # start point in sequence
nextTime = time.time()
lastSwitch = 0

def main():
        setupScreen()
        while True :
            checkForEvent()
            checkStep()

def checkStep() :
        global step, nextTime, lastSwitch
        if stepInt :
  # if we are getting the step command from the internal timer
            if time.time() > nextTime :
  # is it time to do a next step
                  updateSeq(step)
                  step += 1
                  if step >31 :
                          step = 0
                  nextTime = time.time() + timeInc
        else: # if not look at lowest switch
                  switchState = pfio.read_input() & 1
                  if switchState !=                                    ↵
                     lastSwitch and lastSwitch == 0:
                          updateSeq(step)
                          step += 1
                          if step >31 :
                                  step = 0
                  lastSwitch = switchState

def updateSeq(n) :
        pygame.draw.rect(screen,cBackground,                           ↵
           (10, 202,768 ,10), 0) # blank out track
        pygame.draw.rect(screen,cStepBlock,                            ↵
           (14 + n * 24, 202,10 ,10), 0) # draw new position
        pygame.display.update()
        pfio.write_output(seq[n])
```

continued

```
def setupScreen() : # initialise the screen
        screen.blit(background,[0,0]) # set background colour
        drawControl(10,58,"Clear")
        drawControl(86,62,"Invert")
        drawControl(168,68,"Faster")
        drawControl(250,74,"Slower")
        drawControl(350,132,"Auto Step")

        for x in range(0,32) :
             drawCol(x,seq[x])
        pygame.display.update()

def drawControl(xPos,xLen,name) :
        pygame.draw.rect(screen,cTextBack,                          ↵
                        (xPos, 216, xLen,32), 0)
        pygame.draw.rect(screen,cOutline,                           ↵
                        (xPos, 216, xLen,32), 2)
        text = font.render(name, True, cText, cTextBack )
        textRect = text.get_rect()
        textRect.topleft = xPos+4, 220
        screen.blit(text, textRect)

def drawCol(x,value):
        boxNum = 0
        x = 10 + x*24
        y = 10
        for bit in range(0,8):
                # first draw the box -
                # fill colour depends on sequence bit state
                if ((value >> boxNum) & 1) != 1 :
                        pygame.draw.rect(screen,cLEDoff,            ↵
                        (x, y + 24*boxNum, 20,20), 0)
                else :
                        pygame.draw.rect(screen,cLEDon,             ↵
                          (x, y + 24*boxNum, 20,20), 0)
                #now draw the outline of the box
                pygame.draw.rect(screen,cOutline,                   ↵
                 (x, y + 24*boxNum, 20,20), 2)
                boxNum +=1

def mouseGet() : # see where we have
                 # clicked and take the appropriate action
```

```
        global timeInc, stepInt
        x,y = pygame.mouse.get_pos()
        if y in range(10, 202) and x in range(10, 778 ) :
                bit = (y -10) / 24
                byte = (x- 10) / 24
                seq[byte] ^= 1 << bit
                drawCol(byte,seq[byte])
                pygame.display.update()
        elif y in range(216,248) :
                if x in range(10,58) : # the clear control
                                for a in range(0,32):
                                        seq[a] = 0
                                        drawCol(a,seq[a])
                                pygame.display.update()
                if x in range(86,148) : # the invert control
                                for a in range(0,32):
                                        seq[a] ^= 0xff
                                        drawCol(a,seq[a])
                                pygame.display.update()
                if x in range(168,236) : # the faster control
                                timeInc -= 0.05
                                if timeInc <= 0 :
                                        timeInc = 0.05
                if x in range(250,324) : # the slower control
                                timeInc += 0.05
                if x in range(350,482) :
                    # the step source control
                                stepInt = not stepInt
                                if stepInt :
                                     drawControl(350,132,          ↵
                                     "Auto Step")
                                else:
                                     drawControl(350,132,          ↵
                                     "External Step")
                                pygame.display.update()
        else:
                #print "mouse ",x,y

def terminate(): # close down the program
    print ("Closing down please wait")
    pfio.deinit()                               # close the pfio
    pygame.quit()
    sys.exit()
```

continued

```
def checkForEvent():
  # see if we need to quit or look at the mouse
    #print "checking for quit"
    event = pygame.event.poll()
    if event.type == pygame.QUIT :
           terminate()
    elif event.type == pygame.MOUSEBUTTONDOWN :
            mouseGet()
    elif event.type == pygame.KEYDOWN                                    ↵
      and event.key == pygame.K_ESCAPE :
            terminate()

if __name__ == '__main__':
    main()
```

So let's walk through the major sections of the code. It starts off by importing the required libraries and then initialising them and the program's window. Next comes the section where you can customise the colours for the program. I found a black background looks best when you are using the program but a white background looks a lot better when viewed in a book. The next section defines the few global variables needed by the program.

The `main` function is simple, just four lines. Set up the screen, and then loop forever checking for events to quit or mouse clicks to change what is happening. Finally check if you need to advance the sequence. This sequence advance function follows next, although as you know, the order of the function definitions is not important.

The `checkStep` function first looks at the variable that defines where the trigger to the next step is coming from. If this is from the internal timer, the time now is compared to when the next step should occur and if it is time the `updateSeq` function is called, and the `step` variable is incremented and tested to see if it has not gone over the maximum number of steps. If it has then the `step` variable is reset to zero. This is known as *wrapping around the counter*. Finally, the time for the next change is set up by adding the time now to the time increment variable. If the system is set up so that the sequence is advanced on a hardware input then that input is looked at to see if it is different from last time. This indicates a level change or edge has been detected, and if the input level is now high it is time to advance the sequence in the same way as before. One line that might puzzle beginners is this:

```
switchState = pfio.read_input() & 1
```

What the `&` operator does is to perform a bitwise `AND` operation between what is read and the number 1. The result of this is that the variable `switchState` just contains the least significant bit of the byte that is read from the PiFace board's input lines. This means you can

advance the sequence from the lowest switch or the music by attaching the special beat following circuit to it. I will describe that circuit later in this chapter.

The `updateSeq` function basically does two jobs; first it updates the position of the sequence indicator square by drawing a long thin rectangle in the background colour to erase the old square, and then drawing a new one. Finally it outputs the next pattern in the sequence with the following line:

```
pfio.write_output(seq[n])
```

This takes the list called `seq` and extracts the value that is next in the list given by the variable in the square braces and then writes it out to the PiFace board. This single line is what actually does the turning on and off of the lights; everything else just supports this one line.

The `setUpScreen` function simply calls other functions that draw the basic framework of the screen. So after wiping out everything in the screen buffer and setting it to the background colour there are five calls to draw control boxes. This call takes the parameters of the words in the box, its location in the x axis and the width of the box. Finally the `drawCol`, or draw column, function is called in a loop 32 times, one for each step in the sequencer. The two parameters it takes is what step in the sequence it is and what bit pattern it is to set it at.

Drawing text under the `pygame` module is a bit complex. First you have to define your font, which was done at the start of the code; the `None` parameter is the name of the default font file and the number used is the font size. You then have to render the font into a bitmap, which is a little bit of a screen buffer that contains only the font characters you want. You then define a rectangle that encompasses the whole of this small bitmap. Then you have to position this rectangle to the correct part of the screen. Finally you transfer that small screen buffer to the main one with the `screen_blit` call giving it the parameters of the screen buffer and where you want it put. See if you can follow those steps in the `drawControl` function.

The `drawCol` function draws a column of boxes, with one colour if that corresponds to a lit LED in the sequence or another colour if it is unlit. In order to do this you have to separate out all the bits from the sequence value. This is done by this line:

```
if ((value >> boxNum) & 1) != 1 :
```

What is happening here is that the variable called `value` is shifted to the left a number of times, defined by what box you are drawing. The `AND` operation then separates out just the least significant bit of this, as you saw before, and then makes the decision of what to draw based on this bit.

Finally the `mouseGet` function does all the work of dealing with clicks. First it gets the location of the mouse and checks to see if it is in the range of out array of LEDs in the sequence.

If it is, it works out what byte or value this location represents in the sequence and what bit within that byte it is. Then this line toggles the bit:

```
seq[byte] ^= 1 << bit
```

It will look a bit odd and so calls for some explanation. The right side of the equals sign makes a number with a one in the bit position that you want to change. This is done by taking the value 1 and shifting it to the left the number of times you calculated when you worked out what bit was clicked. The equals sign is preceded by a caret symbol ^ and means the exclusive OR operation. So this number you have created by shifting is exclusive ORed with the value of the sequence at this point, and it is then put pack into the sequence list. It is a shorthand way of saying this:

```
seq[byte] = seg[byte] ^ (1 << bit)
```

When you exclusive OR, or XOR as it is sometimes called, two numbers, the result is that you set bits that are only set to a logic one in one of the numbers and you invert the bits that are a set to a logic one in both numbers. So doing this operation simply inverts the bit corresponding to the bit you have clicked. You can then go and draw the whole column; again you update the screen image when you have finished all the drawing.

Next, the `mouseGet` function looks to see if the mouse has been clicked in any of the control boxes. If it has, it does the appropriate action. Clearing the sequence writes zero in every value in the sequence, while inverting applies an exclusive OR operation to all the bits in every value. The number 0xff is simply a byte with all bits set to one. This notation is called *hexadecimal* and believe it or not is simpler to think about than decimal when it comes to creating bit patterns. The faster and slower control boxes change the value to add to the `nextTime` variable. There are also some checks which stop the value from going below zero. Finally the last control changes the variable that determines where the sequence advance is coming from. In order to inform you where the advance trigger is coming from the text in this control box is changed when you click it. Figure 3-2 shows the sequencer as it appears on the screen.

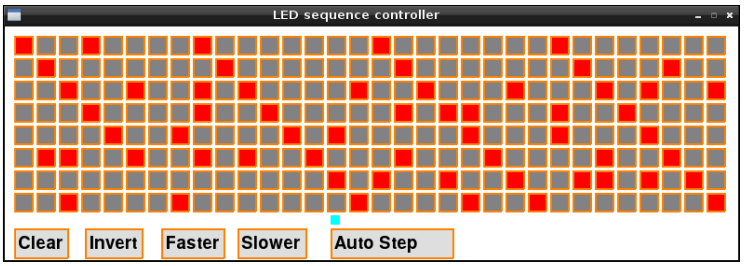


FIGURE 3-2: The sequencer application.

Notice the structure of the code. There is a data structure in a list called `seq`; it is the values in this list that control what is displayed on the screen, and what is output to the lights. Any changes are made to this list and then the list is used to change the display as well as providing the output. Note that the screen display is not used to hold data – only reflect it. This is an important principle and is used whenever you try and write a nontrivial piece of code.

The Lights

The next step is to control some lights rather than the LEDs that are on the PiFace board. While these LEDs are good for testing they are not going to be very impressive in a disco.

The buffer on the PiFace board is capable of switching voltages up to 40V with currents up to half an amp. Now while it can do this on any output it cannot do this on all the outputs at the same time. That is, there is a collective sum total of current the buffer can switch without getting too hot; this is about 650mA. This works out at about 80mA per output if you are to allow for all outputs to be on at once. What you are going to do is drive an LED strip off each output by using a 12V external power supply. There are two types of LED strips, those that have electronics embedded along the strip so that you can control individual lights in the strip, and those where the whole strip lights up at the same time when you apply voltage to it. You are going to use the latter type, which fortunately is cheaper as well.

These LED strips can be cut up at a point every third light, and every three lights consumes 20mA. Therefore you can tailor the amount of current drawn by simply cutting the appropriate length of strip. Some places sell these by the meter and others by the group of three. The absolutely cheapest place to get them is from the Far East through eBay, although the quality you get can be a bit hit and miss. There will be plenty of stockists that carry them in your home country.

In this project you have two options when it comes to powering these strips. The first is where the length of strip is restricted to 12 LEDs – that is about 130mm. The second is where you can power a strip length up to 0.7 of a meter, but more on that later. First you will look at the 130mm option.

Before you start you will have to configure the PiFace board by removing some of the links. This involves removing jumpers JP4, JP5, JP6 and JP7; this disconnects the internal 5V supply from the PiFace board's output devices and disables the relays. See Figure 3-3 for the position of these on the PiFace board. It is important you do this before connecting anything else up.

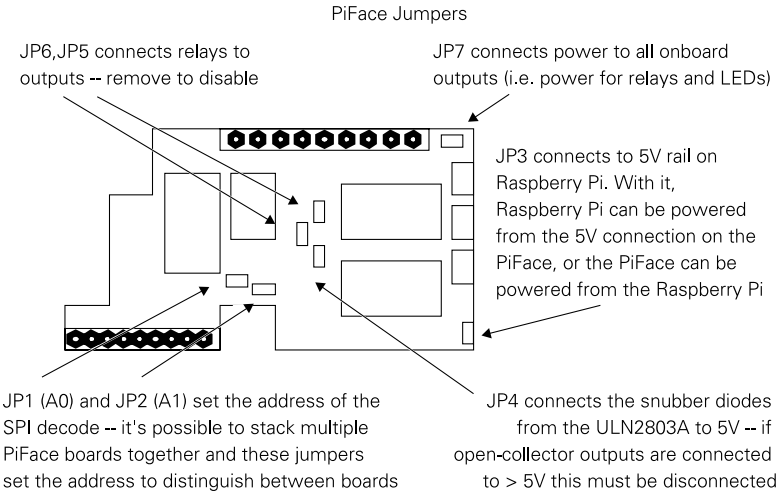


FIGURE 3-3: PiFace jumpers.

Now the LED strips come in different colours. Normally these are white, red, green, blue and amber, so no doubt you will be wanting some of each. You need to cut up each strip you want to light into smaller strips of 12 LEDs. Figure 3-4 shows you where to cut; you will need a sharp hobby knife or better still a scalpel. Every nine LEDs there is a copper soldering area; however this will not appear on the end of every strip of twelve lights. Not to worry – it is very easy to scrape the green solder mask off the board with a scalpel. If you don't fancy that you can always use the solder areas in the middle of the strips. You will end up with 8 strips all the same length but you might want a good mix of colours; the white ones do produce the most light however.

Then, you wire them up so that the positive for each strip is wired to the positive of your 12V power supply. The LED strips are marked with a + and - on the strips at every soldering area. The negative for each strip goes into a separate input of the PiFace board and finally the right-most connector on the input strip is connected to the negative wire of your power supply. This is shown in Figure 3-5, and it is vital that you get the positive and negative wires from your power supply the right way around. Check this before wiring the strips to the PiFace board. When you have wired up the strip's positive leads to the positive of the power supply, just take the negative lead from the strip and touch it against the negative lead of the power supply; if all is well the strip should light.

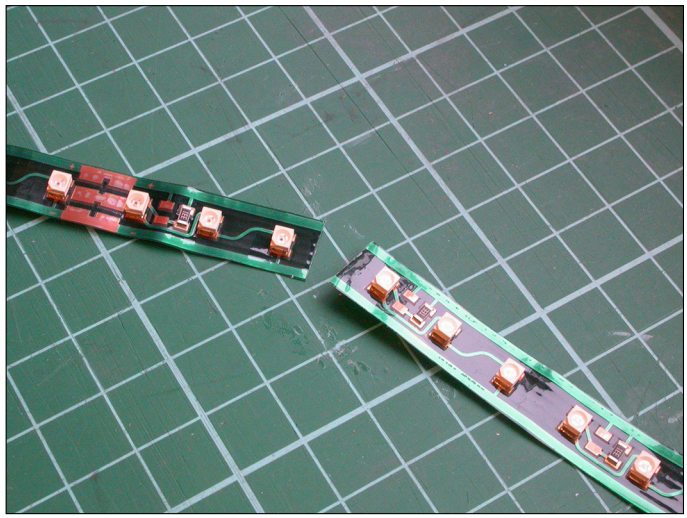


FIGURE 3-4: Where to cut the LED strip.

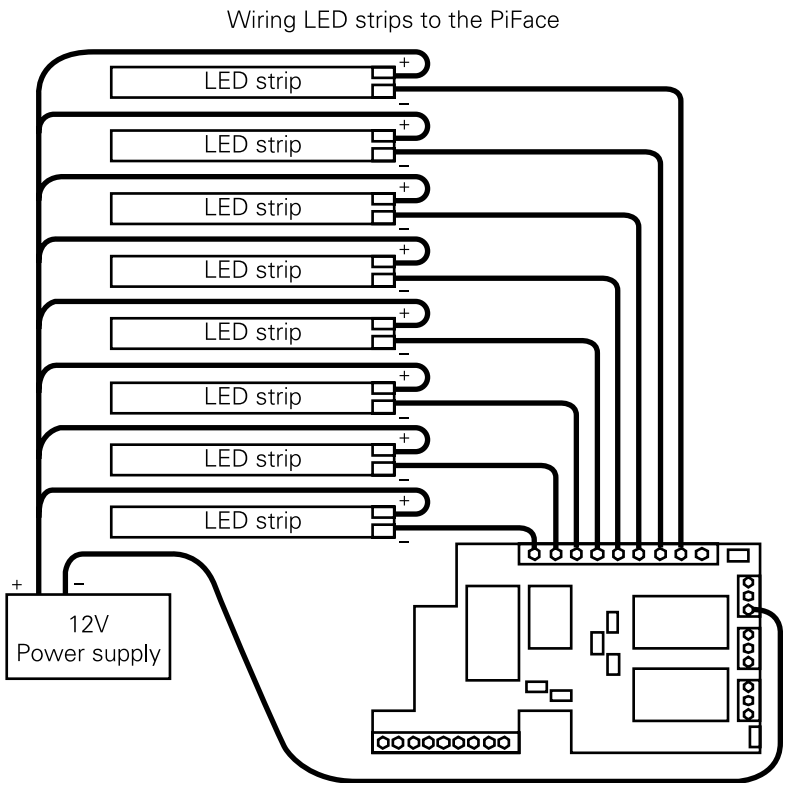


FIGURE 3-5: Wiring the LED strip to the PiFace board.

Now remove the power and touch the two negative wires again just to discharge the power supply before wiring it up to the PiFace board. Then attach the PiFace board to the Raspberry Pi and boot it up, plug in the 12V power supply and run the software. A note of caution: Never connect anything to the Raspberry Pi when it is powered up; it is easy to have an accident, and you can damage things with incomplete or partial wiring.

Using Longer Strip Lights

Now what about the longer strip light I mentioned at the start of the last section? You can draw up to 450mA from each of the outputs from the PiFace board so you can have longer strips of LEDs. This amount of current will drive 66 LEDs or 22 groups of three – this is a strip of 0.7 meters long. However, the down side is that you can't have more than one LED strip lit at any one time. With the existing software it is too easy to make a mistake and set two or more LEDs to come on in each column, but with the changing of just one line in the code you can make the software act as a safety watch dog and only allow one strip light to be on at any one time. The line is in the `mouseGet` function six lines in, and it is one that has been discussed already:

```
seq[byte] ^= 1 << bit
```

Now take that line and change it to

```
seq[byte] = (seq[byte]  &  (1 << bit) ) ^ (1 << bit)
```

You might also want to change the title of the window and the colour scheme at the start so you can distinguish between the two programs. Also, save it under a different name. What this line now does is clear out the sequence value for all bits except the bit you have clicked, and then it toggles that bit. So if any other bit has been set in that step it is cleared and the bit you have clicked is toggled. This prevents you from setting more than one strip to be lit at any one time. However, there is still a slight danger because if you click the invert control, all the outputs but one will be on. To be on the safe side you should remove the following line (10 lines down from the one you just changed):

```
seq[a] ^= 0xff
```

To tidy up the screen display you should remove the call to `drawControl` that sets up the invert control in the `setupScreen` function. You don't have to buy a strip 0.7 meter long; if you want you can join strips together if you cannot buy them in the length you want.

Now all you need to do is mount your light strips in some way – maybe a display board above the decks, or hanging down from the ceiling. I mounted the eight strips on a 8 x 10-inch piece of MDF painted black. I arranged them in a fan shape over half a circle. This would stand up nicely under my monitor. The display is startlingly different depending on what you put in front of the LEDs. If you use nothing, they are very raw but do shed a lot of light. A thin styrene sheet of 0.5mm or less thickness acts as a good diffuser if placed close to the LEDs. However, if you set it just a few inches in front of them the diffusion is much greater and you no longer see the individual lights but bars of colour. Finally another good diffuser is a few layers of clear bubble wrap, the round bubbles in it nicely complementing the individual round LEDs. Your imagination and design skill coupled with your venue will allow you to put these strips, be they short or long, into many a pleasing configuration. However, if you want to cover the dance floor with them you will have to install them behind acrylic sheets to prevent their being stamped on.

Making the Lights Move

Now so far you have looked at stepping the sequence along using the internal timers or an external push button, and if that is as far as you want to take this project, then fine. However, the next step is to have the music drive the change in sequence. Unfortunately this may not work as well as you might be expecting, but you can make a good stab at things relatively easily.

An audio signal, the sort that comes out of an MP3 player or from record decks, is a very complex waveform, consisting of lots of very rapid changes. The speed of the rapid changes carry the frequency content information of sound. The size of the waveform – that is over what range of voltage values they cover – is the amplitude or loudness information. However, the amplitude is varying rapidly to convey the frequencies. What you need to do is to isolate the loudness factor – that is to measure just the size of the peak of the waveform, but it is not quite as simple as that. With a loud sound you get a large positive value and a symmetrically large negative one, so in order to get a measure of loudness you have to ignore the negative value and hold the positive value at its peak. Such a circuit is possible and is called, rather unsurprisingly, a *peak detector*.

Now the beat of music is normally carried in the low frequencies. There are electronic circuits that will separate or filter a mishmash of frequencies so that only a specific range of frequencies get through. The two simplest type are known as *high pass* and *low pass*. In a low-pass filter only the low frequencies can pass through it. Exactly how low is low depends on what is known as the filter's *break frequency*. This is defined as the frequency where the output is cut down by half compared with the input. By correct choice of components you can make this break frequency any value that you want. Most of the low frequencies in music are between 200Hz and 30Hz – any lower and you tend to feel it more than hear it. So to get at the beat

of the music you must filter it with a low-pass filter at a break frequency of 200Hz. The key to filters is the capacitor component, which acts a bit like a frequency dependent resistor. The higher the frequency, the lower its resistance is to AC signals or its capacitive reactance.

The final piece in the jigsaw is called a *comparator,* which compares two voltages and gives a logic one output if one output is higher than the other or a logic zero if it is lower. By varying one voltage with a control knob or potentiometer and feeding a varying signal into the other, you can trigger a digital input when the varying voltage exceeds that set by the knob. If you feed the output of a peak detector into this you can, by turning the knob, set the level that will trigger the sequencer to advance to the next step.

Designing the Circuit

So to implement all that you need a couple of components called *operational amplifiers,* or *op amps* for short. These are very simple on the outside but quite complex on the inside. Basically there are two inputs marked + and – with a single output. The way it works is that the output is equal to the difference in voltage between the two inputs multiplied by a big number called the *open loop gain* which is typically 100,000. So you might think that if you put one volt into the amplifier you will get 100,000 volts out. Well you would if you powered it with a 100,000 volt power supply and you could find an op amp that would work at that level. What happens in practice is that the output will only go as high as the power supply. Also the open loop gain is too high to be useful most of the time and so when you design a circuit that you want to use as an amplifier some negative feedback is applied, as shown in Figure 3-6. Don't confuse this with positive feedback, sometimes known just as *feedback* or *howl around,* when an amplifier's output is fed into an input, like a microphone picking up the amplified sound.

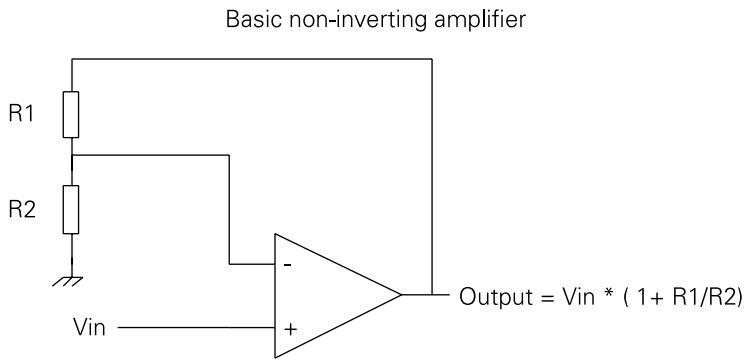


FIGURE 3-6: A non-inverting amplifier.

Negative feedback means feeding a proportion of the output back into the – (negative) input so that the input in effect gets turned down. Consider the circuit in Figure 3-6 and assume there is zero volts on Vin, and also zero volts on the output. Also imagine that the two

resistors have the same value. Now suddenly Vin is changed to 1V so the difference between the two inputs is also 1V. So the output sets off to amplify this difference into 100,000V. However, as the output rises to one volt then the voltage on the negative input will have risen to half a volt, because the two resistors act as a potential divider and feed half the voltage of the output back into the negative input. At this stage the difference between the two voltages is only half a volt, so the output tries to amplify this by 100,000 to give an output of 50,000V. But the higher the voltage gets on the output the more of it is fed back to the negative input. Eventually a balance point is reached when the voltage on the two inputs is exactly the same and so the amplifier's output will not get any higher. This balance point, in this case, happens when the output is exactly twice the input, so in effect the whole circuit has a gain of 2. You can make the gain anything you want, within reason, by simply altering the ratio of the two resistors. So if you feed a tenth of the output back into the negative input, you will have a gain of ten. The actual formula for calculating the gain is shown in Figure 3-6.

So armed with that information you can set about to design the beat extracting circuit whose schematic is shown in Figure 3-7.

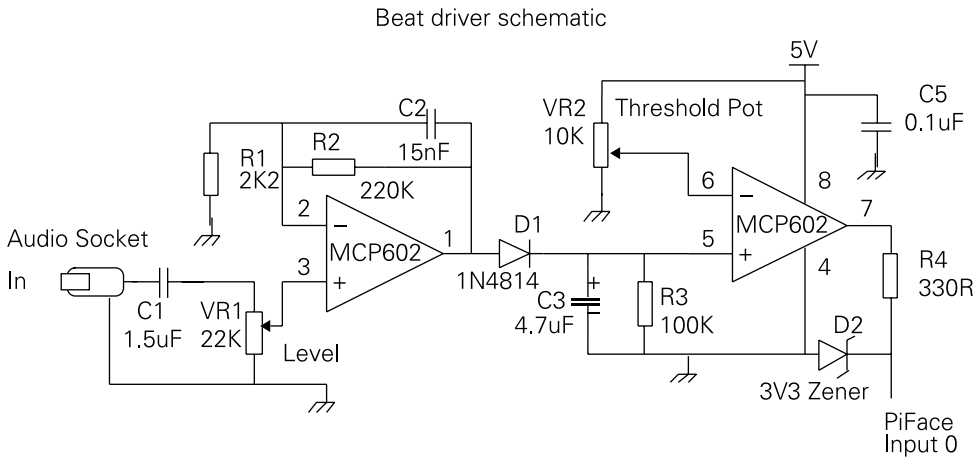


FIGURE 3-7: The schematic of the beat driver.

This uses two op amps which conveniently come in one package. The signal passes through C1 to remove any DC component and then into a pot so that you can set the level into the amplifier. This first op amp is configured just like the previous example as a non-inverting amplifier, only this time there is a capacitor across the feedback resistor. This means that for low frequencies the gain is determined by the value of resistor R2. But as the frequency increases, the capacitive reactance of the capacitor shorts out the feedback resistor to lower the gain. So in this section you have combined a low-pass filter with an amplifier.

The output of this amplifier is passed through a diode. This is a component that will only let electricity flow in one direction, in this case from the amplifier into capacitor C3. So as the

waveform goes up and down rapidly it will start to charge up C3; it only gets more charge when the output of the first amplifier exceeds the voltage on C3 so this capacitor remembers the peak voltage of the audio signal. That is all well and good but you need some way of forgetting a peak signal that happened some time ago and so R3 discharges the capacitor at a slower rate. The result is that the voltage on C3 represents the peaks of the signal or, as we say, it is an envelope follower. The value of R3 determines how quickly the envelope decays.

This envelope voltage is fed into the second op amp. Here you have no feedback, and you just use the open loop gain. The negative input is fed by a voltage set by a knob or pot VR2; this is a threshold voltage. If the envelope voltage is above this then the output goes crashing up to the supply rail of 5V. If, however, the envelope voltage is below this threshold then the output gets put firmly at zero volts or ground. This digital signal is too big to be fed into the PiFace board so it needs cutting down with R4 and D2 to make it a 3.3V signal suitable to dive the sequencer. D2 is a special sort of diode known as a *zener diode;* it starts to conduct at a set voltage. You can get these diodes that conduct at all sorts of voltages; you want one here to conduct at 3.3V or, as it is often written, *3V3*. The ground is shown by the hatched symbol at the end of R1. All the points with this symbol must be connected together.

Building the Circuit

So what you need to do as the final step is construct this circuit. I much prefer making circuits on strip board and not solderless breadboard. The problem with breadboard is that it can make poor or intermittent contact which means that you could appear to have wired it up correctly but it is not. Therefore you can waste a lot of time just jiggling the components around hoping this will make it work. A small piece of veroboard or prototyping strip board is all you need. You can also use sockets for the integrated circuits, which means you can reuse them or replace them if they are damaged. The physical layout of the circuit is shown in Figure 3-8.

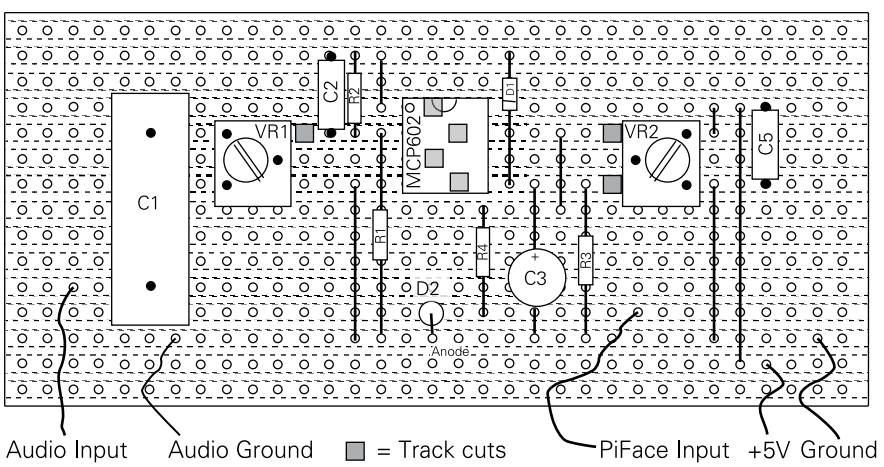


FIGURE 3-8: The physical layout of the beat driver circuit.

This shows the view of the board looking from the top or component side. The dotted lines indicate the copper strips on the underside of the board.

Running the Circuit

After you have built the circuit you need to attach it to your audio input and wire the output, 5V and ground into the PiFace board. Adjust the threshold knob until it is at the mid-point, and boot up your Raspberry Pi. Then run the sequence program and switch to the external step. Start off the music and adjust the volume until you start to see the sequence advance. If it won't turn up far enough then you might have to increase the value of R2; try changing it to 470K. Then adjust the threshold until you see the sequence pick up the beat of the music. You might have to go back and adjust the volume. It works better on some types of music than others.

Over to You

Well that comes to the end of my bit but not of your bit. You can extend and improve this in many ways. You can use transistors or FETs to drive longer LED strips and have many of them on all the time. You can extend the software to save your sequence in a file. Then make it so you can save different patterns in different files. You can implement a shift function where you can concatenate several sequences to make a much longer one and even display an extended sequence by drawing the new pattern when the old one is done. Better yet you could have the display scrolling. Or make the window bigger and the boxes smaller to fit more steps in.

You can add some software that keeps the sequence kicking over if you have switched to an external input and have not had a trigger for a certain amount of time. You could add a small delay after an external trigger to stop them from happening too rapidly. You can add an extra control button to set the sequence to a random pattern.

On the hardware side, you might have noticed that the dynamic range of some music makes it drop out of the trigger zone. There are special amplifiers called *gated compressors;* they are made so that things like walkie-talkies have a constant audio signal into the transmitter. The gain of the amplifier is adjusted automatically to keep the output constant. The SSM2165 is one example of such an amplifier.

You might want to replace the envelope follower's discharge resistor with a pot, something like 220K. Finally you might want to adjust the filter capacitor, or even have a more sophisticated second or fourth order filter on the input. However, whatever you do, keep on dancing.

Appendix A
Getting Your Raspberry Pi Up and Running

In This Appendix

- What the operating system is for
- How to put the operating system on an SD card for the Raspberry Pi
- How to connect up your Raspberry Pi
- A bit about the boot process
- Basic troubleshooting if your Raspberry Pi doesn't start

THIS APPENDIX IS a beginner's guide to your first steps with the Raspberry Pi. It goes from getting it out of the box to getting something on the screen. Even if you already have your Raspberry Pi up and running, it's worth a quick skim as you'll discover how a 21-year-old student changed the world and a bit about how the operating system for your Raspberry Pi works.

The Operating System

The Raspberry Pi uses Linux for its operating system (OS) rather than Microsoft Windows or OS X (for Apple). An *operating system* is a program that makes it easier for the end user to use the underlying hardware. For example, although the processor (the chip at the centre of the Raspberry Pi that does the work) can do only one thing at a time, the operating system gives the impression the computer is doing lots of things by rapidly switching between different tasks. Furthermore, the operating system controls the hardware and hides the complexity that allows the Raspberry Pi to talk to networks or SD cards.

The most popular operating system for the Raspberry Pi is Linux. The widescale use of Linux (just think how many Raspberry Pis there are, not to mention Android phones, web servers, and so on) shows how much an idea can grow. After you start tinkering with the Raspberry Pi, one of your ideas might grow to be as big (or bigger) than Torvalds's or those of the founders of the Raspberry Pi, and you too will make a real impact on the world. So let's get started!

Linux

Part of the success of the Raspberry Pi is thanks to the enthusiastic community that is behind it. Linux is a testament to what can be achieved with the support of volunteers around the world. In 1991, Linus Torvalds began work on an operating system as a hobby while he was a 21-year-old student at the University of Helsinki. A year later, his hobby operating system for desktop PCs (80386) was available online under the name *Linux*. Crucially, the code for the operating system was available as well. This allowed volunteers around the world to contribute; to check and correct bugs; to submit additional features; and to adapt and reuse other's work for their own projects. If you master the projects in this book and learn more about computing, then who knows – one of your hobby projects could be as successful as Linus Torvalds's is.

The popularity of Linux grew, and in addition to its use as a desktop operating system, it is now used for the majority of web servers, in Android devices and in the majority of the world's supercomputers. Most importantly for us, it is used on the Raspberry Pi.

Linux Distributions

Because Linux code is publically available, different organisations have made slight changes to it and distributed it. This has led to different *distributions* (versions), including Red Hat, Fedora, Debian, Arch, Ubuntu and openSUSE. Some companies sell their distributions and provide paid-for support, whereas others are completely free. Raspbian is based on the Debian distribution with some customisations for the Raspberry Pi and is what is used in this book.

Getting the OS on an SD Card

The Raspberry Pi doesn't know how to coordinate its hardware without an OS. When it is powered up, it looks on the SD card to start loading the OS. As such, you're going to need an SD card with an OS on it.

You can either buy an SD card that already has an OS on it, or you can copy an OS to your own SD card with a PC. A premade card is simplest, but more expensive. Creating your own isn't too difficult, but it is slightly more involved than just copying a file.

Premade Cards

Premade cards are bundled in kits or available to purchase from element14, RS or other online stores. A 4GB card should be big enough for getting started and cost less than £10.

Creating Your Own SD Card

There are two ways to create your own SD card for the Raspberry Pi, using NOOBS or by transferring an image yourself.

Using NOOBS

New Out Of Box Software (NOOBS) was created for the Raspberry Pi to automate transferring SD card images. NOOBS boots your Raspberry Pi from a FAT formatted card and then repartitions and clones the filesystem for you. Using NOOBS should be as simple as formatting a card and unzipping a file download from `www.raspberrypi.org/downloads`. Some operating systems do not format cards properly, so it is sometimes necessary to download a program to format the card. Although NOOBS can be simple, it doesn't always work, and it can be slower. Anyway, it's more satisfying to use the do-it-yourself approach.

Filesystems

Computer storage like SD cards, USB memory sticks and hard disks essentially contain millions of separate compartments that store small (too small to even store most files) amounts of data in large grids. The individual compartments, called *blocks,* are addressed by a coordinate system – you can think of them as a piece of squared paper the size of a sports field. The sports field is partitioned into areas that are handled by the operating system to provide *filesystems.* It is the OS's job to manage how data is written to this massive storage area, so that when a user refers to a file by name, all the tiny blocks of data are combined in the correct order. There are different ways in which the blocks are formatted, with different features. As such, an identical file will be stored differently on the underlying grid by different filesystems.

Typically, Microsoft Windows uses FAT or NTFS, OS X uses HFS Plus and Linux uses ext. Most blank SD cards are formatted as FAT by default. Because the Raspberry Pi runs Linux, it uses the ext filesystem, which must be set up and populated with files.

Transferring an Image Yourself

You need an SD card larger than 2GB to fit the OS on it. A 4GB card is ideal.

Visit `www.raspberrypi.org/downloads` and follow the links to download the latest version of Raspbian. You are looking for a filename containing the word `raspbian` and a date, and that ends in `.zip`. Make a note of the letters and numbers that are shown as SHA-1 checksum. Because of the speed of development, new versions are released frequently, so the exact name will differ from the one that's used in the following instructions. The location you download the file to may also be slightly different, so you should use your location accordingly when completing the instructions.

The download page has links to other distributions and other operating systems that you can try later, but for now it's best to stick with Raspbian because it is reliable, has a good selection of software for beginners and is consistent with the examples in this book.

The instructions for creating an SD card are different depending on which OS you're using. Refer to the appropriate section for Windows, Linux and OS X.

Creating an SD Card with Windows

It is hard to check checksums in Windows, so the following instructions assume that the downloaded image file is correct. After the download is complete, follow these steps to uncompress it and transfer the data to the SD card:

1. Unzip the downloaded file `2012-10-28-wheezy-raspbian.zip`.

2. Insert an SD card and make a note of the corresponding drive letter (for example, E:). Make sure that the card does not contain any data you want to save because it will be completely overwritten.
3. Go to `https://launchpad.net/win32-image-writer` and download the binary version of `Win32DiskImager` from the Downloads section on the right side of the web page. Unzip the program.
4. Start `Win32DiskImager.exe` as Administrator. Depending on how your system is set up, this may require you to double-click the program name, or require you to hold down the Shift key, right-click the program icon and select Run As.
5. In the Win32DiskImager window, select `2012-10-28-wheezy-raspbian.img`.
6. In the Device drop-down on the right, select the drive letter you noted in step 2 (see Figure A-1).

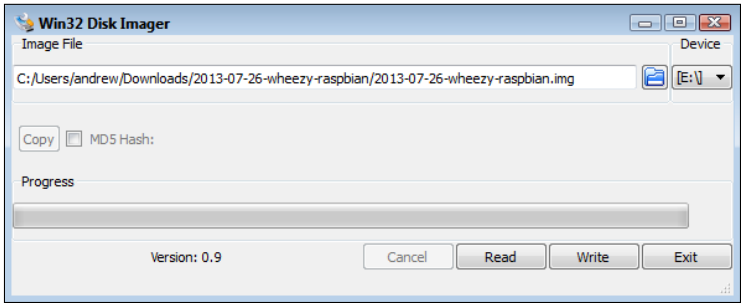


FIGURE A-1: The Win32 DiskImager window.

Images

When talking about downloading the OS for the Raspberry Pi, you may hear it called an *image,* which may be slightly confusing. It is an image of the underlying storage. (Imagine an aerial photo of the entire sports field of storage blocks, even the blank ones, rather than separate files! If you were to print this photo at the same size on another sports field, you'd have an exact copy of all the files stored on the original one.)

It is possible to store an image as a single file in another filesystem, but this arrangement is not suitable for a running Raspberry Pi. As such, a Raspberry Pi will not work if you just copy an image onto a FAT formatted card. Instead, you must tell your OS that you want to transfer it at the block level, so that every block on your card matches those of the person who made the image. That way, Linux interprets these underlying blocks on the disk to provide a filesystem that is identical to the person who made the image.

In summary, filesystem images provide an easy way of cloning an entire filesystem such that all the files, their permissions, attributes, dates, and so on are identical.

7. Click Write and wait for the imaging process to complete. (This step could take about 15–30 minutes, so be patient.)
8. Exit Win32DiskImager and eject the SD card that should now contain your OS.

Creating an SD Card with Linux

With Linux, it's easiest to create the SD card image from the command line, as detailed in the following steps.

1. Start a terminal and use the `cd` command to change to the directory containing the file you downloaded (for example, `cd Downloads`).
2. Unzip the downloaded file by typing `unzip` followed by the downloaded filename (for example, `unzip 2012-10-28-wheezy-raspbian.zip`).
3. List the image files in the current directory by typing `ls *.img` and make sure that the extracted image file is listed.
4. Calculate the checksum to ensure that the downloaded file is not corrupt or hasn't been tampered with. To do this, type the following:

   ```
   sha1sum 2012-10-28-wheezy-raspbian.zip
   ```

   Make sure that the result matches with the SHA-1 checksum given on the `http://raspberrypi.org/download` page. Although it is unlikely that they will differ, if they do, try downloading and unzipping again.
5. Insert an SD card. Make sure there's no data on it that you want to save, because it will be completely overwritten.
6. Type `dmesg` and find the device name that Linux uses to refer to the newly inserted card. It will usually be named `sdd`, `sde`, `sdf` or something similar. Alternatively, it may be in the form `mmcblk0`. Use this name wherever you see `sdX` in the following steps.

Checksums

A *checksum* is an easy way to check whether data has been corrupted. A checksum is a mathematical sum that is performed by the supplier of the data. When you receive the data, you perform the same sum and, in most cases, if the answer is the same, you can be almost certain that the data is the same, without comparing bit by bit. Checksums are used extensively in computing – in network communications, when processing credit cards and even in barcodes. Although they are not infallible, they make it much easier to be fairly confident data is correct.

Linux Permissions and sudo

Linux restricts some actions that might cause damage to other users. As such, some commands will not work unless you have the appropriate privileges. On some distributions, you need to switch to being the user *root* (the administrator account) before running the command requiring more privileged access. Other distributions will allow selected users to prefix the command with `sudo`. The following instructions assume that your user account has been set up to use `sudo`. If not, type `su` in the terminal first to become root.

7. If Linux has automounted the card, you need to unmount it first by typing `sudo umount /dev/sdX`.
8. Double-check that you have the correct device by typing `sudo fdisk -l /dev/sdX`. Check that the size displayed matches the size of the card that you inserted.
9. When you are absolutely sure you have the right label for the card, type the following (replacing `sdX` with the name you found in step 6) to copy the image across to the card. (This step could take about 15–30 minutes, so be patient.)

   ```
   dd if=2012-10-28-wheezy-raspbian.img of=/dev/sdX
   ```

10. Type `sudo sync` before removing the card to ensure all the data is written to the card and is not still being buffered.

Creating an SD Card with OS X

With OS X, it's easiest to create the SD card image from the command line.

Although the Macintosh normally uses drag and drop for many operations, there is a way to get "under the hood" to perform unusual operations. Your gateway to doing this is an application called *Terminal*. This is usually found in the Utilities folder, within the Applications folder. A quick way to find it is to hold down the ⌘ key and press the spacebar. This will open the Spotlight search window. Type `terminal` and then press Enter to open the Terminal application.

To create an SD card, follow these steps:

1. Start a terminal.
2. Use the `cd` command to change to the directory containing the file you downloaded. A quick way to do this is to type `cd` followed by a space and then drag the folder containing the file into the Terminal window. This will automatically fill in the rest of the command with the pathname of that folder. Then press Enter to perform the command.

3. Unzip the downloaded file by typing `unzip` followed by the downloaded filename (for example, `unzip 2012-10-28-wheezy-raspbian.zip`).

   You won't see a progress bar during this process, so you might think the computer has frozen – but don't worry. It could take a minute or two before all of the files are unzipped.

4. List the image files in the current directory by typing `ls *.img` and make sure that the extracted image file is listed.

5. To make sure everything is fine, you can calculate the checksum for the file; however, you can omit this step if you want. Calculating the checksum ensures that the downloaded file is not corrupt. To do this, type the following:

   ```
   shasum
   2012-10-28-wheezy-raspbian.zip
   ```

   Make sure that the result matches with the SHA-1 checksum on the `http://raspberrypi.org/download` page. It is unlikely that they will differ, but if they do, try downloading and unzipping again.

6. Type `diskutil list` to display a list of disks.

7. Insert an SD card. Make sure that it doesn't contain any data that you want to save because it will be completely overwritten.

8. Run `diskutil list` again and note the identifier of the new disk that appears (for example, `/dev/disk1`). Ignore the entries that end with `s` followed by a number. Use the disk identifier wherever `disk`*X* appears in the following steps.

9. Type `sudo diskutil unmountdisk /dev/disk`*X*.

10. Type `sudo dd bs=1m if=2012-10-28-wheezy-raspbian.img of=/dev/disk`*X*. (This step could take about 15–60 minutes, so be patient.)

11. Type `sudo diskutil eject /dev/disk`*X* before removing the card.

Connecting Your Raspberry Pi

Now that you have your OS for your Raspberry Pi, it's time to plug it together.

Remove the Raspberry Pi from the box and, to make it easier to follow these instructions, position the same way around as shown in Figure A-2 (so the words *Raspberry Pi* appear the correct way up).

Plug the USB keyboard into one of the USB sockets, as shown in Figure A-3.

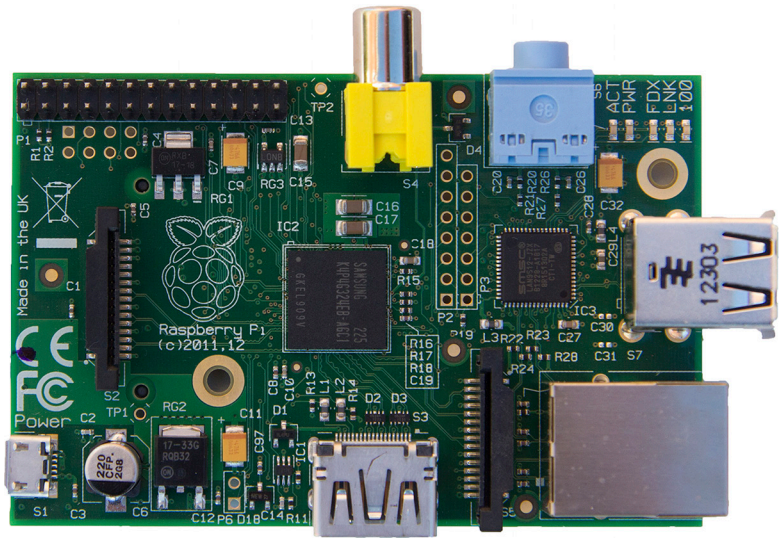


FIGURE A-2: The Raspberry Pi, the size of a credit card and a miniature marvel of engineering.

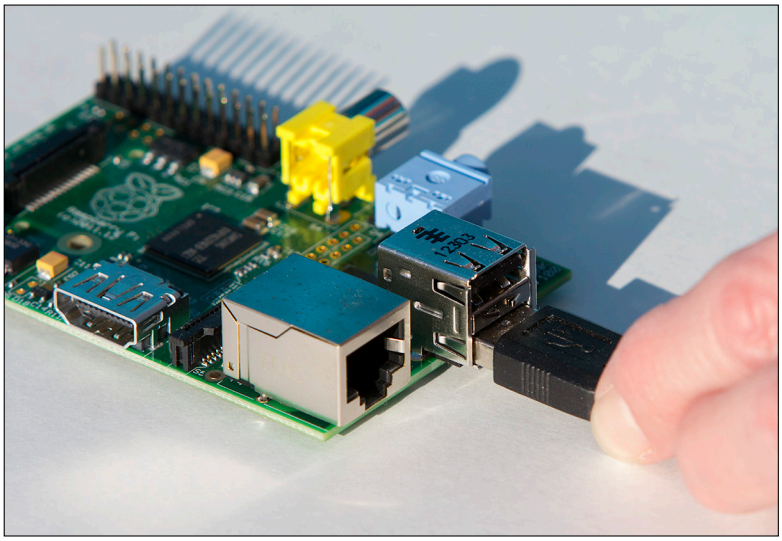


FIGURE A-3: Inserting the USB keyboard.

Older PS/2 keyboards will not work. You'll have to buy (or borrow) a USB keyboard, but they're not expensive.

TIP

Plug the mouse in next to the keyboard, as shown in Figure A-4.

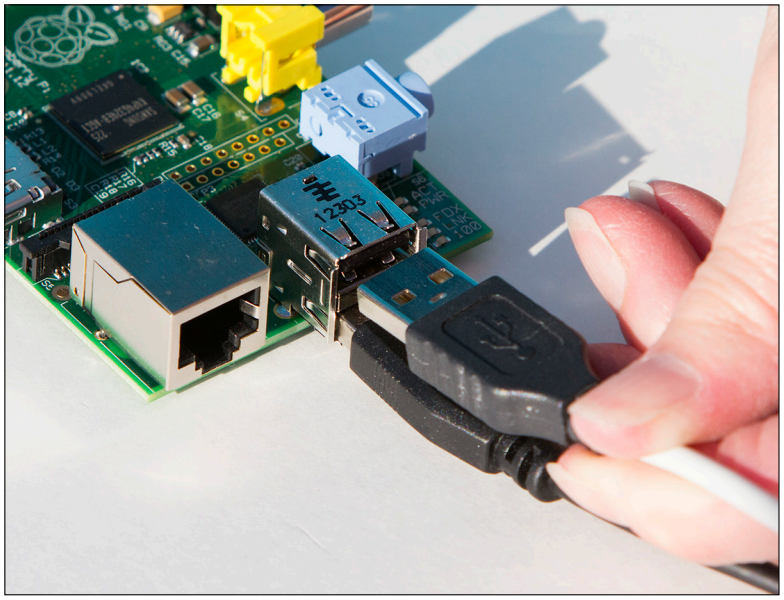


FIGURE A-4: Inserting the USB mouse.

Connecting a Display

The Raspberry Pi can be connected by HDMI or composite video directly. With the use of an adapter you can connect it by DVI or VGA. You should use HDMI or DVI whenever possible because they give a better picture.

Look at the sockets on your display to determine how to connect your Raspberry Pi.

Connecting via HDMI

If your display has an HDMI input, as shown in Figure A-5, then connect your Pi with a HDMI-HDMI cable. This is the only type of video connection that can also be used to carry audio from the Pi to your display. The HDMI socket on the Pi is at the bottom as shown in Figure A-5.

Connecting via DVI

If your display has a DVI input as shown in Figure A-6, you will need an adapter. HDMI and DVI have very similar electrical signals, so adapters are *passive* – that is, they don't contain any electronics, just two sockets with wires in between. You can buy cables with an HDMI and DVI connector or adapters as shown in Figure A-6 for less than £5.

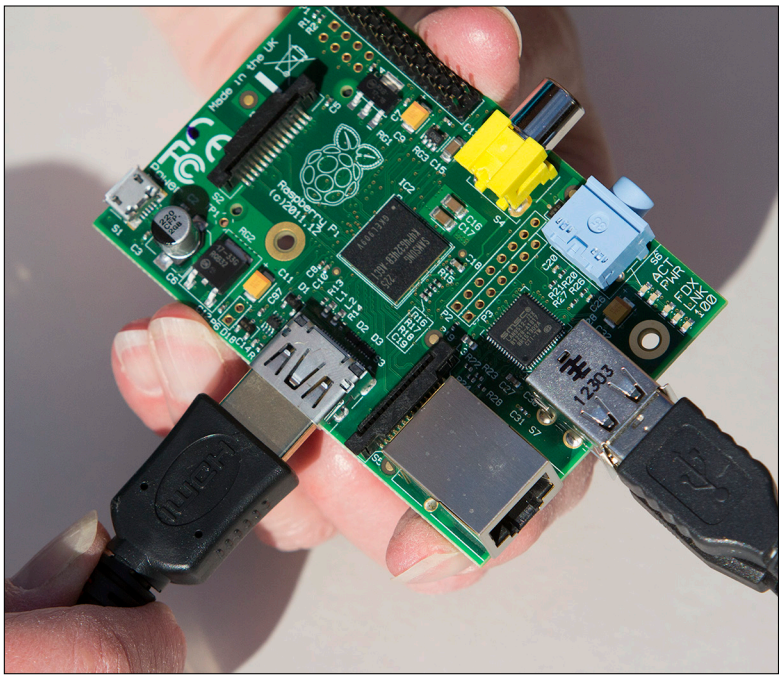


FIGURE A-5: HDMI connection on the Raspberry Pi.

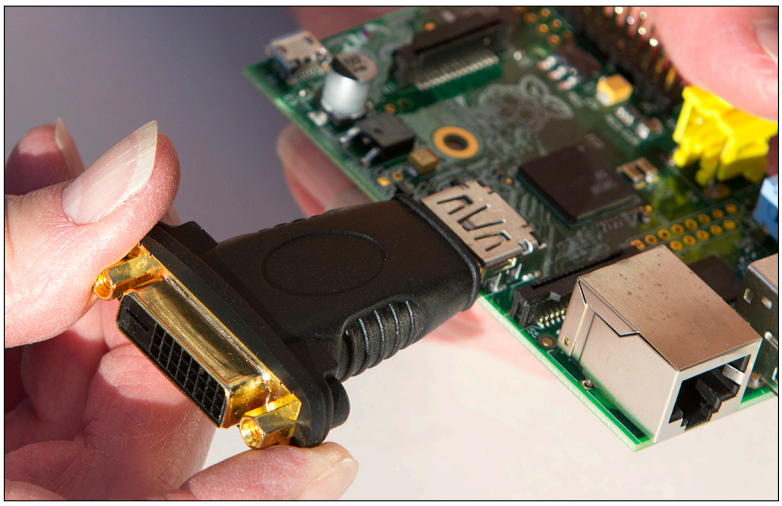


FIGURE A-6: HDMI-DVI adapter.

Connecting via VGA

DVI and HDMI both work with *digital* signals and are only found on newer monitors. Older monitors with VGA use *analogue* signals and as such need some sort of electronic circuit to convert between them. You can buy adapters that convert between HDMI and VGA for

about £20 online. The Pi-View device shown in Figure A-7 is designed specifically for the Raspberry Pi and is available through element14.

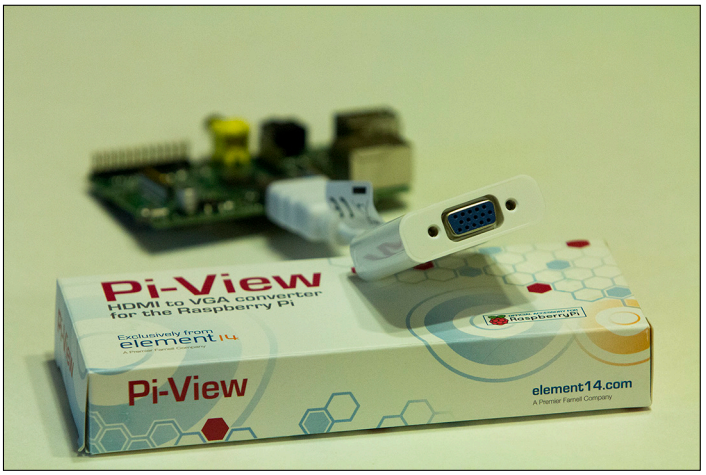


FIGURE A-7: HDMI-VGA adapter, Pi-View.

Connecting via Composite

If your display only has a connector for composite video, you need a phono-to-phono cable that plugs in to the yellow connector on the top of the Raspberry Pi as shown in Figure A-8. Be aware that composite is an old technology and may produce a poor quality display.

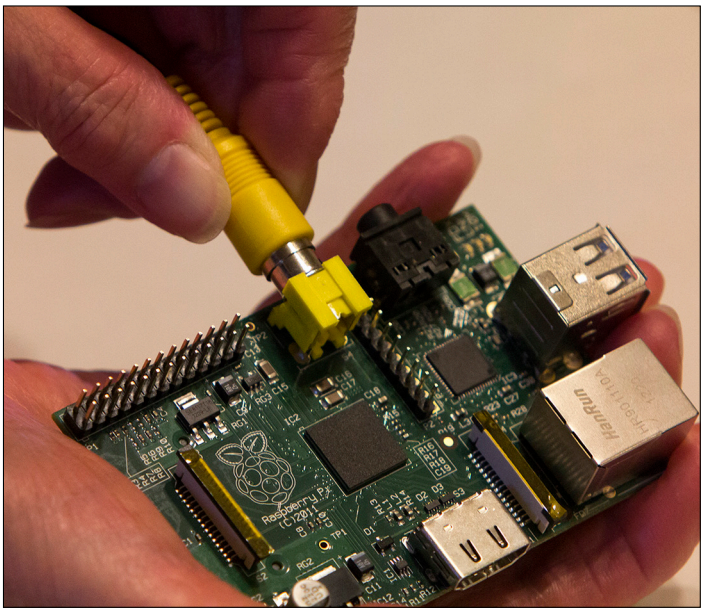


FIGURE A-8: Phono connector for composite video.

Analogue and Digital

Inside most computers you will find digital signals – that is, signals where it only matters if they are on or off. Usually there is a difference of a few volts between a signal being on or off. Data is sent by a code of ons and offs, typically referred to as 1s and 0s. A small change in voltage due to radio or magnetic interference is usually not large enough to change the meaning.

Analogue signals tend to only be used in modern computers where they have to connect with something physical. An analogue signal typically represents data as a range of voltages. As such, a small change in voltage means a different value will be read. This means the data can be changed by electrical interference.

VGA monitors represent different colours with different voltages. Consequently, any interference will affect what is shown on the screen, and the image is degraded! Small amounts of interference will have no effect on digital data for HDMI. However, if the interference is strong enough, then all data will be corrupted and no image will be transmitted.

Connecting to a Network

The Raspberry Pi has an Ethernet socket that allows your Pi to connect to the Internet or your home network. You can download new software and updates, or browse the web. You could even run your own web server!

If you will be using a network, connect a network cable on the right side as shown in Figure A-9. Although the Raspberry Pi uses the network to set its clock and to download updates and new programs, it will work without a network connection.

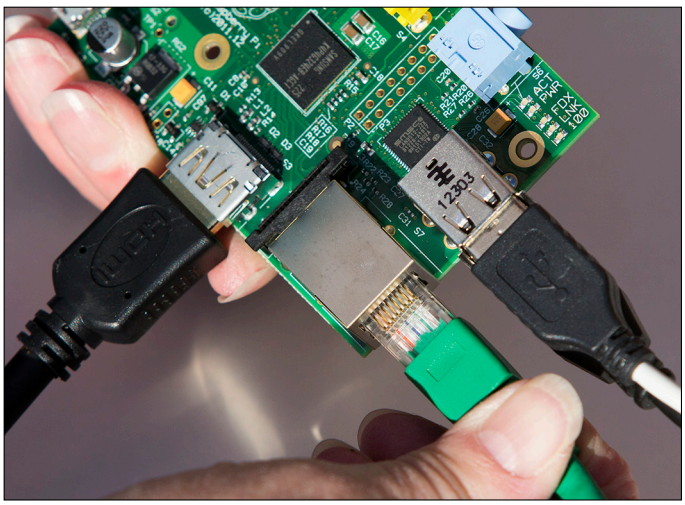


FIGURE A-9: Network connection.

Booting the Operating System

You will need an SD card with the OS already installed on it. You can either buy one pre-installed or follow the instructions earlier in this appendix to make your own.

Insert the SD card in the slot on the underside of the Raspberry Pi, on the left, as shown in Figure A-10. Take care to keep the card parallel with the Raspberry Pi when you slide it in or out so as not to break the edge of the retaining slots (shown in Figure A-11).

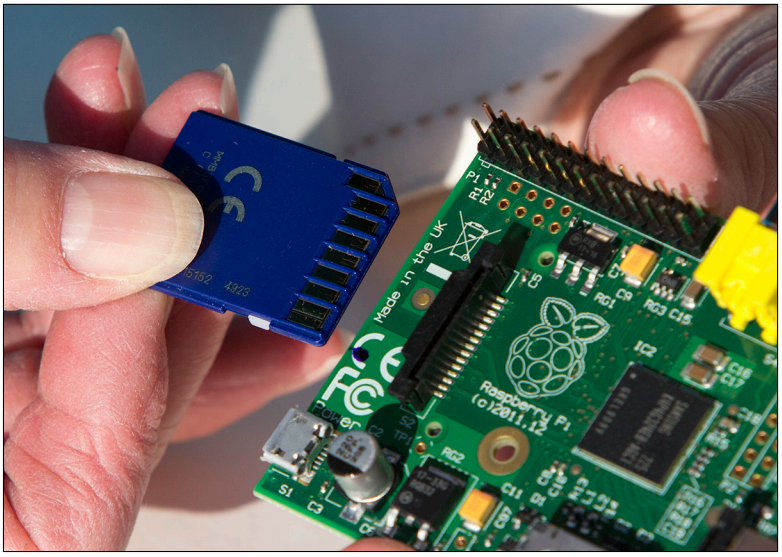


FIGURE A-10: Insert the SD card carefully.

FIGURE A-11: Take care not to snap off the plastic that keeps the SD card from falling out.

Powering Up!

WARNING

Before connecting power, get into the habit of checking that there is nothing conductive in contact that could cause a short circuit with your Raspberry Pi. A quick check that there's nothing metallic nearby could save you from damaging your Pi!

Plug in the power supply to the bottom left of the Raspberry Pi as shown in Figure A-12. On the top-right corner, you should see a green light (labelled *PWR*) come on and another one (labelled *ACT*) flash.

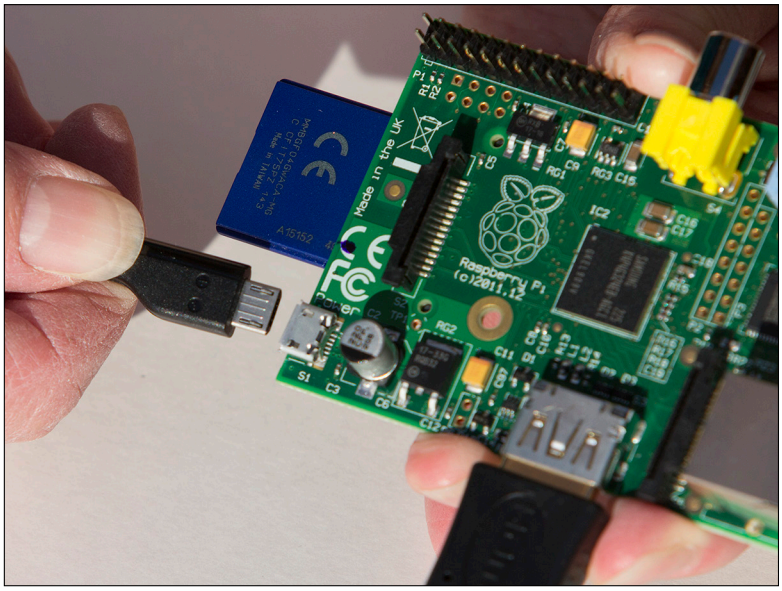


FIGURE A-12: Insert a micro USB for power.

The Raspberry Pi needs a power supply that can supply 5V 700mA (3.5W). Most decent-quality mobile phone chargers will work fine – many have the output marked on them, so it's easy to check. If your power supply can't deliver enough power, your Raspberry Pi may not start, or it may freeze when it does something computationally more demanding. For more information, see the "Troubleshooting" section later in this appendix.

The Boot Process

After you've connected everything, have a correctly imaged SD card, and powered up your Raspberry Pi, it will quickly flash a colourful square to test the graphics. After a few seconds, the Raspberry Pi logo will appear in the top-left corner of the screen, and many lines of text will scroll past.

The text reveals some of the work the OS is doing. You may see messages as the various drivers are loading, such as the keyboard driver, sound driver and network driver. After the drivers have loaded, the OS runs any startup programs and displays the login prompt.

By default, the username is `pi` and the password is `raspberry`.

Type `pi` and press Enter.

Now type `raspberry` and press Enter. Linux doesn't display anything when you type passwords, which can be a bit unfamiliar if you are used to other OSs.

You should see the command line prompt, where you can type commands and run programs. In the next section, you're going to start the program that allows you to use the Raspberry Pi graphically.

Starting the Graphical Desktop

If you are familiar with Windows or OS X, you are used to a friendly graphical desktop that is loaded automatically with icons you can click. On the Raspberry Pi, however, in order to show that a graphical desktop doesn't have to be integral to a computer, it isn't loaded automatically.

To start the graphical display on the Raspberry Pi, type `startx`.

After a few seconds the X server will start, and you will be able to use a graphical desktop. If you can see the Raspberry Pi logo in the background, then congratulations – you have successfully connected your Raspberry Pi! The projects in this book assume that you're starting from here, with the desktop displayed.

The X Server

The design of Linux means that the graphical desktop runs on top of the OS as a separate program called the *X server*. This opens up additional possibilities, such as controlling one computer with the display being shown on another computer over a network connection. This means that you can control the Raspberry Pi without having a monitor plugged into it, which is useful if you put it in a remote location.

Starting a Terminal under X

Linux makes greater use of the text-based command line, often known as a *terminal*. This can be very powerful and quicker for some tasks than using a mouse. To start a terminal in a window under X, double-click the LXTerminal icon on the desktop, or select it from the menu by clicking Accessories and then clicking LXTerminal.

Troubleshooting

Hopefully, you'll never need this section, but even if you think you've followed all the instructions, you might discover that something doesn't work. Finding and debugging problems are important aspects of computing. The general approach is to be logical and eliminate parts until you can isolate where the problem is. It's a good idea to simplify to the simplest possible configuration first – unplug the keyboard, mouse and/or display to see if the Pi shows signs of life – and then add things one by one. When you are suspicious of what might be at fault, try borrowing a known working replacement from a friend or try the suspected faulty part in theirs. This way, you can eliminate parts until the fault is found.

Common Problems

The majority of problems in getting the Raspberry Pi to work are easy to fix. The following sections describe some of the issues that you might encounter with the Raspberry Pi and how to troubleshoot them.

No Lights Come On

If none of the lights come on when you power up your Raspberry Pi, the power supply may not be providing the required 5V. If you have a meter, you can measure the output; if not, plug a phone in and see if it starts charging. If the power supply manages to power a phone, it still might not provide enough for a Raspberry Pi. Try borrowing a friend's that you know works.

Only the Red Light Comes On

If just the red light comes on, then the Raspberry Pi is getting some power, but it isn't booting the OS. Make sure that the SD card is correctly inserted, and then check that it is correctly imaged. Even if the card is correctly imaged, it may be that the card isn't compatible with the Raspberry Pi. If possible, try another card that is known to work, either from a friend or by buying a premade card. Also check that the power supply is providing enough power.

No Monitor Output

Check that the connector to the monitor hasn't come loose and that if your monitor has a choice of inputs that the correct one is selected. Normally, there is a switch on the front that cycles through the input sources. With some monitors it is necessary to have connected the monitor to the Raspberry Pi before powering it up. If you are still having trouble, try a different monitor and cable.

Intermittent Problems

If the Raspberry Pi freezes or resets, particularly when you do something that demands more power (such as graphics-intensive work or adding a peripheral), then it's likely the power supply isn't providing enough power.

Power Problems

The Raspberry Pi needs more power than some micro USB power adapters can provide. It is certainly more than what's provided by the output of most computer USB ports. As the Raspberry Pi does different tasks, the amount of power it needs varies. Consequently, with some adapters, it may work some of the time, but then stop when it needs more power.

Your power supply should provide a minimum of 700mA at 5V or at least 3.5W. Most power supplies will have a label that details the output power or current it can provide, such as the one shown in Figure A-13. However, some power supplies have overly optimistic labeling and don't deliver what they claim! If your Raspberry Pi partly works and suddenly stops working, particularly when you ask it to do something more intensive such as graphics, then the power supply is probably not up for the job. In some cases it is not the power supply itself that is at fault, but the cable connecting it to the Raspberry Pi. Some cables can have a relatively high resistance and so can drop the voltage getting to the computer.

The power adapter also has to supply any peripherals plugged onto the Raspberry Pi. If a peripheral takes too much power, then your Raspberry Pi will stop working.

If you know how to use a multimeter, you can check the voltage supplied by the power supply under load. You can find information about how to do this in the *Raspberry Pi User Guide* (Wiley, 2012). If you measure less than 4.3V at the test points, then it might be worth changing the cable before you change the power supply.

Alternatively, you can try using a different adapter. You could buy one that is recommended by the supplier of your Raspberry Pi.

FIGURE A-13: A power supply showing its output rating.

If You Need More Help

If you're still struggling with your Raspberry Pi, then you may need other sources of assistance. A major benefit of the huge popularity of the Raspberry Pi is the support offered from an enthusiastic, helpful community. If you are still having difficulty, see if you can find a solution at http://elinux.org/R-Pi_Troubleshooting, or check the Raspberry Pi forums at www.raspberrypi.org/forum.

You can often get help in person by attending a user group or local meeting, commonly referred to as a *Raspberry Jam*. It's a worldwide network, so just check http://raspberryjam.org.uk to find the nearest location.

The *Raspberry Pi User Guide* also provides suggestions for troubleshooting and configurations to work with specific hardware.

Let the Fun Begin!

Now that you've got your Raspberry Pi powered up, it's time to start having fun with the projects. The Insult Generator project in Appendix B is a good one to start with because it introduces how to program the Raspberry Pi in Python – and more importantly, it can be used to insult your friends and family!

Appendix B
Introductory Software Project: The Insult Generator

In This Appendix

- Storing data to variables
- Strings
- Printing messages on the screen
- Functions
- Getting input from the user
- Numbers – integers and floats
- `if` statements
- Loops

THIS PROJECT IS just a bit of fun for you to get going with your first program. The program generates a comedy insult by combining a verb, an adjective and a noun at random. In other words, you'll make your highly sophisticated Raspberry Pi display something like "Your brother is a big old turnip!"

By beginning with something simple, you can start having fun without having to write too much code, and after you've got something running, you can change it to make it more sophisticated. In fact, professional computer programmers often take a similar approach: They write something simple and test it, and then add more and more features, testing as they go.

It's also useful to look at sample code, work out what it is doing and then change it to suit your requirements. Most professional programmers work this way too. Feel free to experiment and customise the projects in the book. Just remember to keep a copy of the original program so that you can go back to it if it doesn't work.

This appendix helps to get you started programming the Raspberry Pi and, as such, it has the most theory in. Do stick with it, and at the end of the appendix, you'll have the knowledge to make the program your own. There's a lot in this appendix, but you needn't do it all in one go; sometimes it's better to come back after a break. Programming is no less creative than painting a picture or knitting, and like these hobbies, you need to spend an hour or two covering the basics before producing a masterpiece!

In this appendix, you will learn how enter a Python program and run it. You'll also learn about various aspects of the Python language.

Running Your First Python Program

Many people use a word processor to produce documents with a computer because it provides features such as spelling and grammar checkers. However, there is nothing to stop you from using a simple text editor like Notepad in Windows, TextEdit on an Apple Mac or LeafPad on the Raspberry Pi. Similarly, when writing code, you can just type it in a text editor or you can use an Integrated Development Environment (IDE). Similar to a spell checker in a word processor, an IDE checks the syntax (to ensure that it will make sense to the computer) and has other helpful features to make writing code a pleasure!

Just as there are lots of different word processors, there are a number of IDEs for Python. For the simple example in this appendix, you are going to type your first Python program into IDLE. This is good for beginners because it is simple and can often be found wherever Python is installed, including on the Raspberry Pi.

To start IDLE, click the menu in the bottom-left corner of the screen (where the Start button is in Microsoft Windows), choose Programming, and then click IDLE, as shown in Figure B-1.

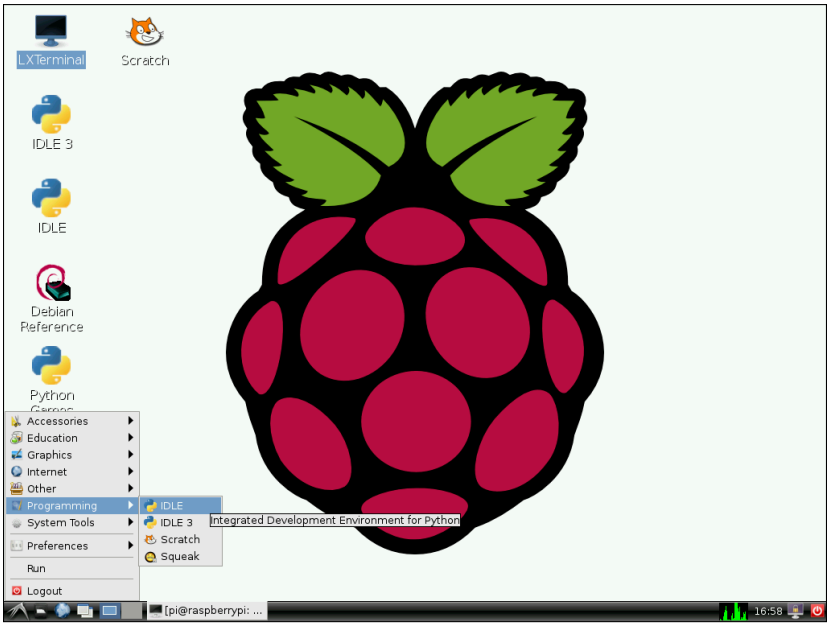


FIGURE B-1: Starting IDLE.

TIP

The Raspberry Pi comes with two versions of IDLE. IDLE 3 uses Python version 3.0, which contains more functionality and has subtle changes to parts of the language. The examples in this appendix are written for Python 2 (that is, IDLE). If you use the examples in the book without changing them you'll receive errors.

You will see the IDLE window appear in interactive mode. In this mode, what you type is interpreted as soon as you press Return, which is a great way to try out your Python code. To see how this works, follow these steps:

1. Type the following code:

```
print ("Hello World")
```

NOTE

Computers are less forgiving of mistakes than humans, so make sure that you type the code exactly as it appears in this and other examples in this book.

2. Press Return. You should see Python run your first line of code and display the greeting shown in Figure B-2.

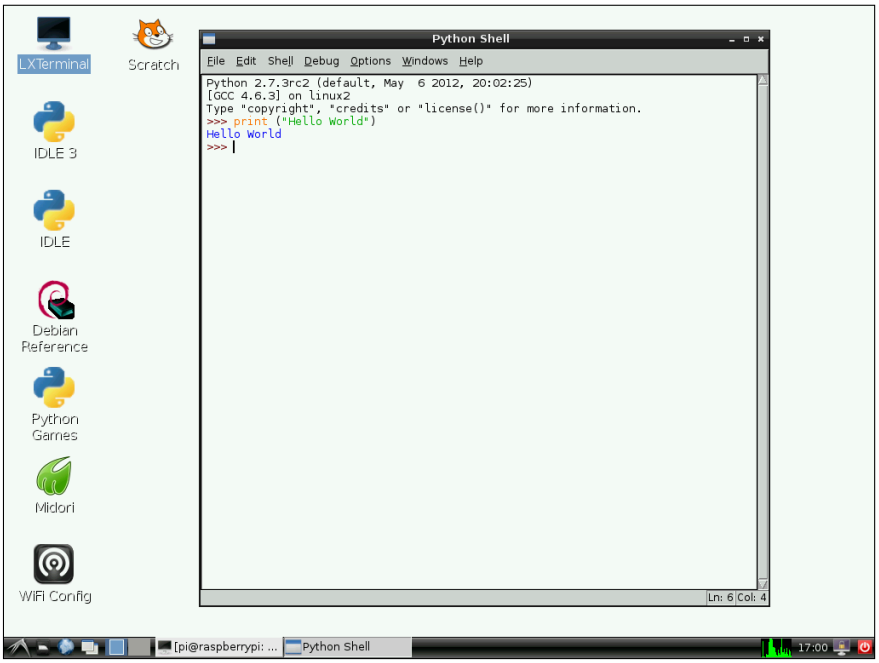


FIGURE B-2: Python says "Hello World".

NOTE Many programmers write a "Hello World" program whenever they learn a new language. It is about the simplest program and is a good way to check that it is possible to write some code and then run it. It dates back to the first tutorials of how to program in the 1970s. There's even an equivalent in hardware to "Hello World" in Chapter 1, "Test Your Reactions".

If you got the result shown in Figure B-2, then welcome to the club – you're now a computer programmer! If not, go back and make sure that you typed the code *exactly* as shown in the example (sometimes even the number of spaces matter in Python), because computers need to be told precisely what to do. This strict rule means that unlike English, a statement can only be interpreted with one meaning.

Saving Your Program

IDLE allows you to save your code so that you don't have to re-enter it each time you want to run it. Just follow these steps:

1. Create a blank file for your program by selecting New Window from the File menu, as shown in Figure B-3.

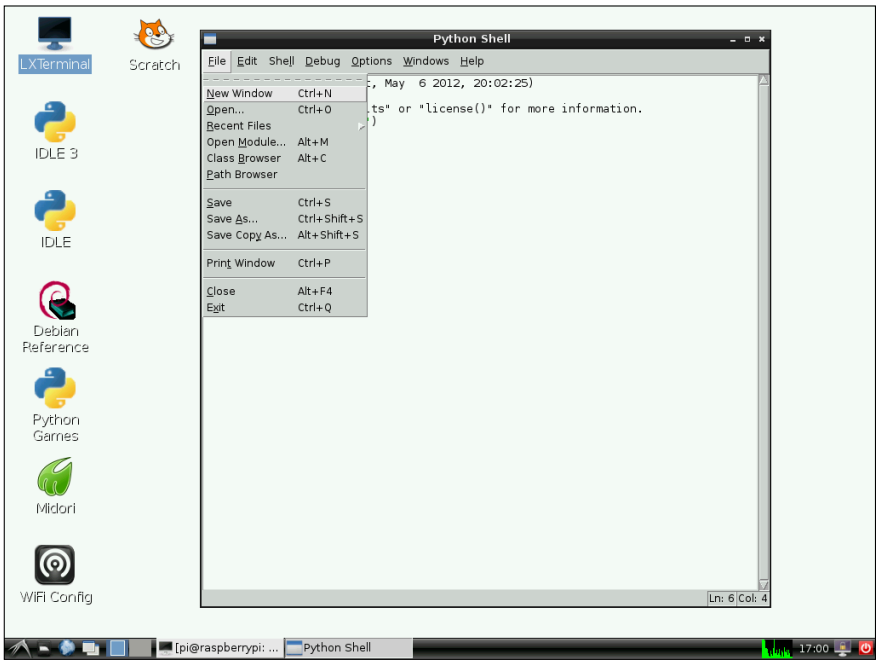


FIGURE B-3: Creating a new file.

2. Enter the following code and then click the Run menu and choose Run Module, or press F5.

```
message = "hello world from a saved file"
print (message)
```

3. Python displays a message that says, "Source Must Be Saved", as shown in Figure B-4. Click OK.

Source is an abbreviation for *source code,* which is another way of saying the program you've entered.

NOTE

4. Type in a filename (for this example, you can just call it `Hello`), and then click Save as shown in Figure B-5.

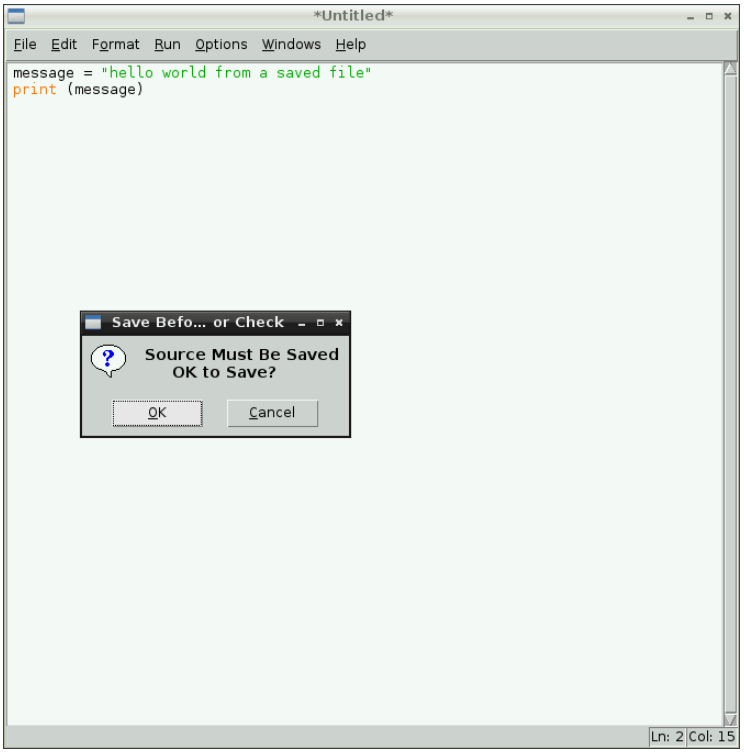


FIGURE B-4: Python prompts, "Source Must Be Saved".

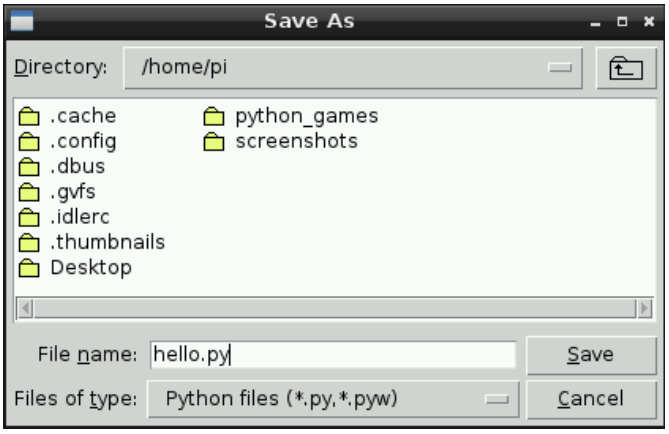


FIGURE B-5: The Save As dialogue box in IDLE.

5. After IDLE has saved your code, you will see a message saying RESTART (Python does this so you know you're always starting from the same consistent point), and then your code will run in the Python Shell window, as shown in Figure B-6. If you've made a mistake, you'll see an error – correct it and then choose Run Module again.

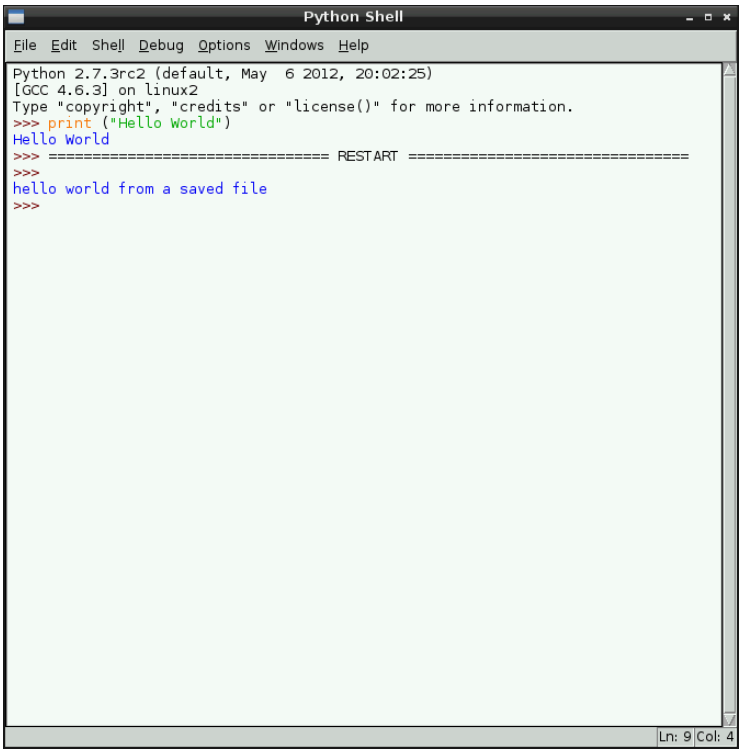


FIGURE B-6: IDLE running Python code from a file.

TIP When you save your code, IDLE adds `.py` to the end of the filename. This is the file extension for Python source files (just as Word adds `.doc` to documents).

Generating an Insult

Now that you've successfully run your first program, it's time to write something more interesting – in this case, the computer will generate its own message to print. Type the following code in a new file and then run it:

```
from random import choice
adjectives = ["wet", "big"]
nouns = ["turnip", "dog"]
print  ("You are a")
adjective = choice (adjectives)
print (adjective)
print (choice (nouns))
```

When you run this program, you should see a message similar to "You are a big turnip" displayed. Run the program a few times and you should see a variety of insults, built at random!

You can use the keyboard shortcut F5 to run the program. However, ensure the editor window containing your program has focus (is active) by clicking on it before pressing F5 so IDLE knows the correct code to run.

You'll be changing the program to display a personalised message later in this appendix, but before you do, it's worth examining the code more closely. The following subsections describe what the different lines of the program do.

TIP Looking at how other people's code works is useful when you're learning to program, and the World Wide Web is a good source of many examples.

Variables

Variables are used to store data. Creating a variable is like getting a cardboard box to reserve some storage space and writing a unique label on it. You put things in the box for storage and then get them out again later. Whenever you access the box you use its unique label.

Let's start with something simple to illustrate variables:

```
message = "hello"
```

The equals sign means *assignment*, and tells Python to assign (or store) what is on the right-hand side in the variable named on the left – in this case, the characters h, e, l, l and o are stored in the variable named `message`.

Strings

The `"` speech marks (also known as quotation marks in some parts of the world) tell Python to treat the enclosed characters as a *string* of letters rather than trying to understand the word as an instruction.

To display text on the screen, you use the `print` command followed by what you want displayed after it – for example:

```
message = "hello"
print (message)
```

This will display the contents of the variable `message` on the screen, which in this case is `hello`.

If, on the other hand, you enter the following code:

```
print ("message")
```

The word `message` will be displayed on the screen, because the speech marks tell Python to treat text within them as a string of characters and not a variable name.

TIP `print` is slightly confusing in that it displays characters on the screen and has nothing to do with sending it to a printer to appear on paper.

Lists

To store multiple pieces of data in Python together, you can use lists. Lists are specified as items separated by commas within square brackets. Reconsidering the example of the cardboard box, a list can be considered as a named box with internal dividers to store separate items.

YOUR TURN! Looking back at the insult generator code, lists of strings are used to store multiple adjectives and nouns. Because you now know about strings and lists, you can try adding some more words of your own. Remember to enclose them in quotes (`""`) and separate them with a comma.

Functions

A *function* can be thought of as a little machine that may take an input, perform some sort of processing on it, and then produce an output as shown in Figure B-7. You can create your own functions, or you can use functions that are included in Python or written by other people. To use a function, you *call* it by entering its name.

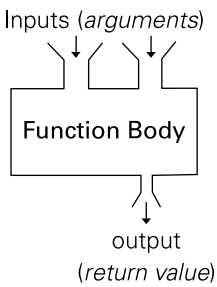


FIGURE B-7: Functions are little machines that process inputs to produce an output.

Structuring Your Programs

Functions are a way to structure programs. Computers and computer programs can quickly become very complicated. The best way to deal with this is to break things down into simple, manageable chunks. It's something we do in everyday life: If you ask someone to make you a cup of tea, you don't give them a long list of instructions about filling the kettle with water, turning it on, waiting for it to boil, adding a tea bag to the teapot, and so on. We don't think about all the details – after someone has been told how to make a cup of tea, they don't need to be told all the steps each time. When you tell someone to make a cup of tea you assume they have already been shown how to fill the kettle and that you don't have to tell them how to use a tap!

If we broke every task down to the simplest steps every time, things would become unmanageable. Programming computers is just the same – tasks are broken down into manageable chunks. Knowing exactly how and where to break a program into chunks comes with experience, but with this approach, it becomes possible to program a computer to make it do just about anything.

Functions may take *arguments* (sometimes called *parameters*), which are a way of supplying data to them. Think of them as the raw materials into, or the controls that adjust, the function machine. Imagine a machine that makes different pasta shapes; its arguments might be raw pasta and a setting that determines what shape it produces. Arguments can be a variable (which may change as a program runs) or something *hard-coded* (written directly into the program by the programmer and never changed) in the program itself.

The `print` command that you used in the preceding code is a function in Python 3 that displays its parameter on the screen. The arguments to a function are often contained in brackets after the function name.

`choice` is another function that you have been using, perhaps without realising it. Its argument is a list of items, and the processing it does is to select one at random. Its output is an item from the list, which it returns.

TIP If you find yourself writing the same code in multiple parts of a program, or using copy and paste, you should think about putting the repeated code into your own function.

There are so many functions that if all of them were available at once, it would be overwhelming to the programmer. Instead, Python contains only a few essential functions by default, and others have to be *imported* from packages of functions before they can be used.

`choice` is an example of a function that needs to be imported from the `random` package. In the earlier example, the line `import choice from random` performs this role. You only need to import a function once in a program, but you can use it multiple times.

Insult Your Friends by Name!

The programs so far have produced an output, but when run, have not taken any input from the user. The next example asks the user for a name and then prints a personalised greeting. To try this out, enter the following code:

```
name = raw_input("What is your name?")
print ("Hello " + name)
```

TIP `raw_input` became the `input` function in Python 3. If you're using IDLE 3, remember to type `input` wherever you see `raw_input` in the examples in this book.

The `raw_input` function (renamed to `input` in Python 3) takes a message to print as its argument and returns the data the user entered. In this example, the variable `name` is assigned the input from the user when the program is run.

This example also introduces how to join strings together. Strings are joined together, or *concatenated* as a programmer may say, by placing + between the strings. It's important to note that because the computer treats strings as just characters and not words, when strings are concatenated, it does not automatically insert spaces. Therefore it is up to the programmer to add any spaces needed. In the preceding example, there is a space after `Hello` in the quotes – without this, the computer would print something like `HelloFred`.

Help with Functions

When you type a function like `choice` in IDLE, a tooltip pops up telling you what arguments the function takes and what it returns. This is a useful quick reference so you don't have to remember exactly what parameters a function takes, and it's easier than looking up the full reference online.

If you press the Ctrl key and the spacebar simultaneously, IDLE will attempt to autocomplete what you've typed thus far, which is useful if you can't remember exactly what a function is called. To try this out, type `pri` and then press Ctrl + space. You should see a list of functions with `print` highlighted, as shown in Figure B-8. Press the spacebar again to have IDLE finish off the typing for you.

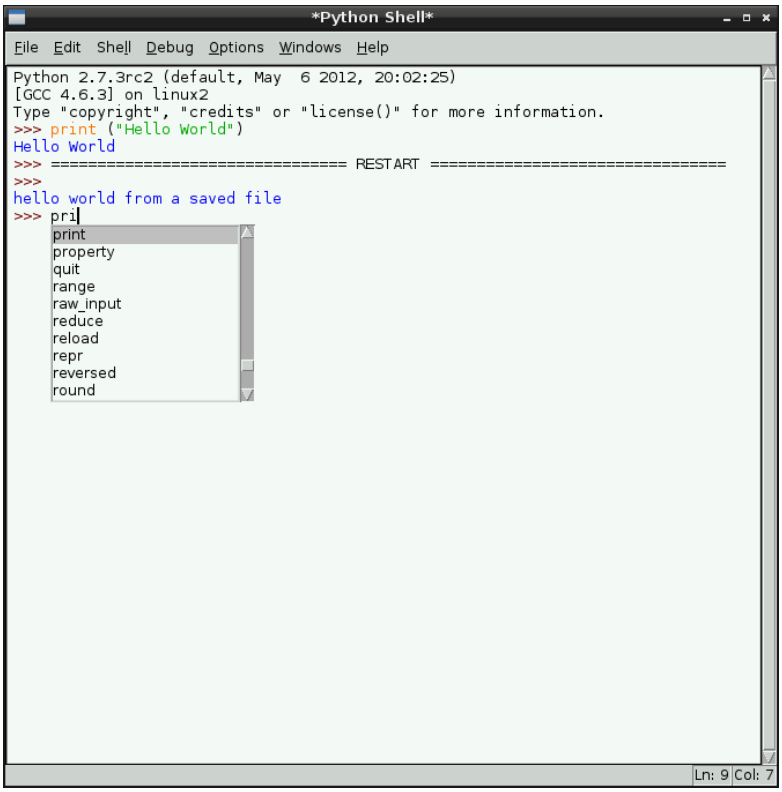


FIGURE B-8: Autocomplete of the `print` function in IDLE.

Conditional Behaviour

Computer programs would be very dull if they always executed the same statements. Luckily, programs can do different things depending on the data.

In this example, you'll make your insult generator change what it prints depending on the age of the user. To achieve this conditional behaviour, you'll tell the program to do one thing if something is true, or if not true, do something else. As a quick test of conditional behaviour enter the following code in an empty file:

```
age = 12
if (age < 16):
    print ("young")
else
    print ("old")
```

Run the program and you should find it prints `young`. Change the `age` variable to be larger than 15 and run the program again. This time it should print `old`.

Create a Stream of Insults!

In the next part of this project, you're going to change the program to produce multiple insults, which is a good example of the use of functions. You're going to define your own function that you can call whenever you want an insult, and then create a loop that calls the function multiple times.

Making Your Own Functions

You define functions in Python by writing `def` (for definition) followed by the name of the function and the parameters it takes and a colon (:), followed by the *body* of the function.

As a simple example, enter the following in an interactive Python window to define a simple function that will print a personalised greeting:

```
def printHelloUser (username):
    print ("Hello " + username)
```

Note that the body of the code is indented. This shows that it is still part of the function on the previous line. Also note that there are no spaces in the function names. Including spaces would confuse the computer, so programmers separate words by using capitals in the middle (like `printHelloUser` in the example). Some programmers call this *camel case* because the capital letters in the middle of a word are like humps on the back of a camel.

TIP Python doesn't care what you call your functions, but other programmers will! So if you want other people to use your code, you should follow conventions.

Now enter the following to call the function you just defined:

```
printHelloUser("Fred")
```

You're now ready to use what you've learned in this appendix to write a `printInsult` function. To begin, enter the following code in an interactive Python window:

```
from random import choice
def printInsult (username, age):
    adjectives = ["wet", "big"]
    nouns = ["turnip", "dog"]
    if (age < 16):
        ageAdjective = "young "
    else:
        ageAdjective = "old "
```

continued

```
    print  (username + ", you are a " +
            ageAdjective + choice(adjectives) +
            " " + choice(nouns))
```

Now, whenever you need a personalised insult you can just call `printInsult`, with your victim's name and their age, and it will produce one on demand! So, to insult 10-year-old Fred, you would write the following code line:

```
printInsult("Fred",10)
```

And Python would print something like this:

```
Fred, you are a young wet turnip
```

YOUR TURN! Call the `printInsult` function with the names and ages of some of your friends and family!

Creating Loops

Loops are great when there's repetition in a task. To try this out, you're going to create a program that loops around a few times, each one producing another insult. You'll look at two variations of loop: a `for` loop and `while` loop. Both have a test that determines if the code in the loop body should be run again, or if the program should skip the loop body and continue. A `for` loop is typically used as a counting loop, where code needs to be run *for* a particular number of times – for example, for six times. A `while` loop tends to be used *while* a condition is true – for example, while a user wants to continue running the program.

for Loop

Type the following code to loop for each item in the `adjectives` list and print it out:

```
adjectives = ["wet", "big"]
for item in adjectives:
    print (item)
```

The indented code, `print (item)`, is the body of the loop that is repeated on each *iteration* (loop). `for item in adjectives`: sets up the loop and tells Python to loop for each item in the `adjectives` variable. On each iteration, the next value from the list is placed in the `item` variable.

So, to print "hello world" three times you write this:

```
for count in [1, 2, 3]:
    print ("loop number ",count)
    print ("hello world")
```

TIP

You can use commas to separate multiple items for printing.

You can use `range` instead of typing every number in a list. Type

```
list(range (10))
```

TIP

In Python 2 `range` returns a list. In Python 3 you need to tell it specifically when you want it to return a list – this is not necessary within a `for` statement.

Python returns the list of numbers from 0 to 9:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also give `range` two arguments – a start and an end – like this:

```
range (5,15)
```

You will use a `for` loop with a range to test the `printInsult` function. In the same interactive Python window where you defined `printInsult()`, enter the following code:

```
for testAge in range (14,17):
    print ("age: " + str(testAge)

    printInsult("Monty",testAge)
```

Python prints the following:

```
age: 14
Monty, you are a young big turnip
age: 15
Monty, you are a young big dog
age: 16
Monty, you are a old big dog
age: 17
Monty, you are a old wet dog
```

while Statement

Let's look at an example `while` loop. Type the following code and run it. It will loop and keep printing an insult until you type no.

```
userAnswer=""
while (userAnswer!="no"):
    printInsult("Eric",20)
    userAnswer = raw_input("can you take any more?")
```

`!=` means not equal, so the `while` loop repeats whilst the variable `userAnswer` is not equal to no. After printing an insult the code gets input from the user and updates the `userAnswer` variable ready for the test before the start of the next loop.

TIP `raw_input` was renamed `input` in Python 3.

Consider what the loop would look like if you didn't create a function – you'd have to include all the code inside the loop body. This makes the code harder to read, and means you'd have to retype it in each of these examples!

TIP If your program gets stuck in an *infinite loop*, a loop that never ends, you can stop your program by pressing Ctrl + C.

Putting It All Together

You should now have a program that generates a torrent of insults! This appendix has covered the basics. Look at each line and see if you understand what each part does before running the program. Then, to personalise your program, you could make it produce different insults depending on the user's name. For example, you could make it say something nice only if your name is entered, or you could change the number of insults it generates depending on the user or their age (such as a younger brother). You could print "really old" for people over a certain age, or if you're clever, you could use a loop to print an additional "really" for every decade someone has been alive.

The main thing is to not be afraid of changing things to see what happens. Just as you can't learn to paint without practicing, you won't learn how to program without experimenting. Throughout this book, there are ideas to change the projects to make them your own and to make the computer do what you want it to do.

Still hungry for more Raspberry Pi®?

The full book of Raspberry Pi Projects publishes December 2013 in print and e-book

£14.99/ $24.99

Raspberry Pi Projects

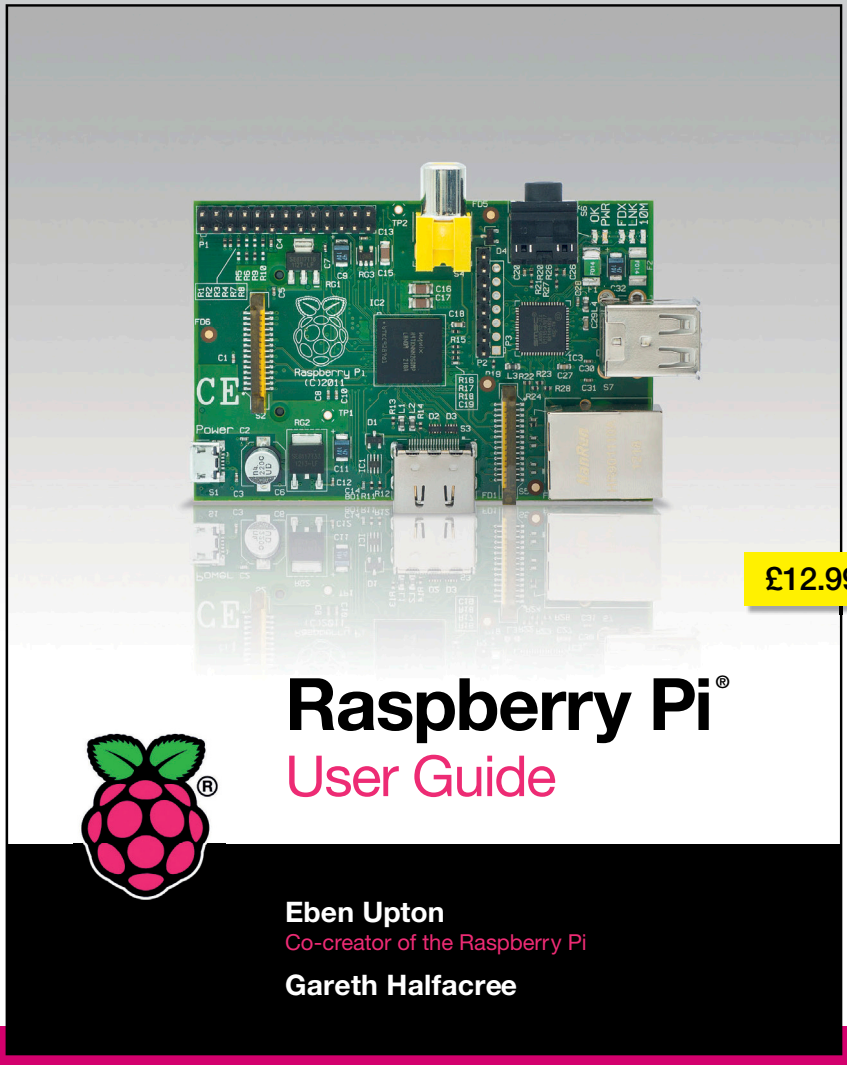
Dr. Andrew Robinson
Mike Cook

Available wherever books are sold.

For more information or to order direct go to **www.wiley.com** or call **+44 (0) 1243 843291**

13-59223

The best-selling guide to getting started with your Raspberry Pi®



£12.99 / $19.99

Available wherever books are sold.

For more information or to order direct go to **www.wiley.com** or call **+44 (0) 1243 843291**

13-59223