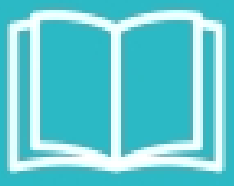
Ashwin Pajankar , Arush Kakkar,
Matthew Poole, Richard Grimmett

Raspberry Pi: Amazing Projects from Scratch

Learning Path

Explore the powers of Raspberry Pi and build your very own projects right out of the box





Packt

Table of Contents

Raspberry Pi: Amazing Projects from Scratch
Raspberry Pi: Amazing Projects from Scratch
Credits
Preface
- What this learning path covers
- What you need for this learning path
- Who this learning path is for
- Reader feedback
- Customer support
  - Downloading the example code
  - Errata
  - Piracy
  - Questions

1. Module 1
- 1. Introduction to Raspberry Pi and Python
  - Single-board computers
  - Raspberry Pi
    - Raspberry Pi models
  - Operating systems
    - Raspbian
  - Setting up the Raspberry Pi
    - Preparing MicroSD card manually
    - Booting up our Pi for the first time
    - Shutting down and rebooting Pi safely
    - Updating the Pi
  - Getting started with Python
    - Turtle programming with Python
  - Summary
- 2. Minecraft Pi
  - Introduction to Minecraft Pi
  - Playing Minecraft Pi
    - Movement control in Minecraft Pi
    - Action control in Minecraft Pi

Other controls in Minecraft Pi
Python programming for Minecraft Pi
Summary
3. Building Games with PyGame
Introducing PyGame
Installing PyGame
Drawing a binary fractal tree
Building a snake game
Summary
4. Working with a Webcam and Pi Camera
Working with webcams
Crontab
Creating a timelapse sequence using fswebcam
Webcam video recording and playback
Working with the Pi Camera and NoIR Camera modules
Using raspistill and raspivid
Using picamera in Python with the Pi Camera module
The Pi camera versus the webcam
Summary
5. Introduction to GPIO Programming
Introducing GPIO pins
Building an LED Blinker
Connecting a button
Installing PiGlow
Using PiGlow
Building a binary clock
Summary
6. Creating Animated Movies with Raspberry Pi
Introducing stop-motion animation
Setting up the prerequisites
Setting up and testing the camera
Adding the hardware button
Rendering the video
Summary
7. Introduction to Computer Vision
Introducing Computer Vision
Introducing OpenCV

- Setting up Pi for Computer Vision
  - Testing the OpenCV installation with Python
- Introducing NumPy
  - Array creation
  - Basic operations on arrays
  - Linear algebra
- Working with images
  - Using matplotlib
- Working with Webcam using OpenCV
  - Saving a video using OpenCV
  - Pi Camera and OpenCV
- Retrieving image properties
- Arithmetic operations on images
- Splitting and merging image color channels
  - Negating an image
- Logical operations on images
- Colorspaces and conversions
- Tracking in real time based on color
- Summary

8. Creating Your Own Motion Detection and Tracking System
- Thresholding images
  - Otsu's method
- Noise
  - Kernels for noise removal
  - 2D convolution filtering
  - Low pass filtering
- Morphological transformations on images
- Motion detection and tracking
- Summary

9. Grove Sensors and the Raspberry Pi
- Introducing the GrovePi
- Setting up the GrovePi
- Displaying the weather
- Intruder detection system
- Summary

10. Internet of Things with the Raspberry Pi
- Introducing the Internet of Things

Installing the Twitter API for Python
Using Tweepy
Setting up a SQLite database in Python
Building a tweeting weather station
Adding speech capabilities to our weather station
Summary
11. Build Your Own Supercomputer with Raspberry Pi
Introducing a Pi-based supercomputer
Installing and configuring MPICH2 and MPI4PY
Installing the MPICH library
Installing MPI4PY
Setting up the Raspberry Pi cluster
Setting up SSH access from the host to the client
Running code in parallel
Performance benchmarking of the cluster
Introducing N-Body simulations
Installing and running GalaxSee
Summary
12. Advanced Networking with Raspberry Pi
Introducing DHCP
A few networking concepts
Configuring a Raspberry Pi to act as a DHCP server
Introducing Domain Naming System (DNS)
Setting up a DNS server on the Pi
Configuring the setup for a web server
Automating node discovery in a network
Summary
13. Setting Up a Web Server on the Raspberry Pi
Introducing and installing Apache on Raspbian
Installing PHP and MySQL
Installing WordPress
Configuring the WordPress installation
Summary
14. Network Programming in Python with the Pi
The basics of sockets
The difference between TCP and UDP
The architecture and programming of UDP sockets

Sending and receiving data with UDP
UDP servers and NCAT
An echo server using Python UDP sockets
A UDP client
The architecture of TCP sockets
Creating a TCP socket
Connecting to a server with a TCP socket
Receiving data from the server
Programming socket servers
Binding a socket
Listening for incoming connections
Handling multiple connections
Looking back
A Telnet client in Python
A chat program
The chat server
The chat client
References
Exercise
Summary
15. Newer Raspberry Pi Models
The Raspberry Pi Zero
The Raspberry Pi 3
2. Module 2
1. Setting Up Your Raspberry Pi
Which flavor of Pi?
Raspberry Pi Model A
Raspberry Pi Model B
Raspberry Pi Model B+ and Model 2
Model comparison table
So which one?
Preparing the SD card
Downloading the Raspbian image
Using Microsoft Windows
Using Linux
Booting your Pi
Expanding the file system

Using the raspi-config utility
Setting up your Pi
Getting up to date
Getting the right time
fake-hwclock
ntp
Talking of security–
What is this sudo thing anyway?
Connecting via Wi-Fi
Summary
2. Connecting Things to Your Pi with GPIO
Prerequisites
Say hello to the GPIO
Digital I/O pins
The I2C bus
The SPI bus
The UART serial bus
USB ports
Power connections
Getting acquainted with the GPIO
Let there be light
Getting flashy–
Adding a switch
Pulling yourself together
The detection script
The most elaborate light switch in the world
The illuminating script
Summary
3. Extending Your Pi to Connect More Things
Prerequisites
The I2C bus
Just 2 wires
What's your address?
There is a parallel universe
Serial-to-parallel conversion
Give me power
Building an I2C expander

- The I2C port expander circuit
  - Let's walk through the circuit
- Building your expansion board
- Using ready-made expansion boards
- Hobbytronics MCP23017 expander port kit
- PiFace Digital I/O expansion board
- Gertboard
- Summary

4. Adding a Magnetic Contact Sensor
- Prerequisites
- The working of magnetic contact sensors
- Setting up the I2C port expander
- Enabling the I2C Bus
- Installing the I2C tools package
- Finding our devices
- Setting up the port expander
- Connecting our magnetic contact sensor
- Monitoring the sensor
- Anti-tamper circuits
- Getting into the zone
- Summary

5. Adding a Passive Infrared Motion Sensor
- Prerequisites
- Passive infrared sensors explained
- Setting up your PIR sensor
- Give me power (again)
- Connecting our PIR motion sensor
- 12V alarm zone circuits
- Alarm circuit protection
- How it works
- Wireless PIR motion sensors
- 433-MHz wireless alarm systems
- Connecting a 433-MHz receiver
  - The alternative approach (because we have no choice)
- The receiver wiring diagram
- Logging detection data
- Summary

6. Adding Cameras to Our Security System
- Prerequisites
- The Raspberry Pi camera module
  - Connecting the camera module
  - Setting up the camera module
    - Testing the camera module
- Be a video star
  - Caught on camera
- You have new mail
  - Setting up the e-mail sender client
  - Sending attachments
  - Where was that taken?
- Night vision
  - An illuminating experience
  - The Elaborate light switch re-visited
  - Is that a badger?
- Using USB cameras
  - Installing the webcam
  - Taking a snap
    - Snap snap snap
- The multicamera setup
  - The Slave driver
- Summary

7. Building a Web-Based Control Panel
- Installing the web server
  - Testing the PHP5 installation
- Being in control
  - Arming yourself
- The master configuration file
- Creating the web page
  - The control panel HTML template
  - Giving it some style
  - Making it dynamic
    - Getting a bit of help first
    - The main PHP code
    - I'm someone else
- Remote access to our control panel

- Setting up a dynamic DNS account
  - The Raspberry Pi dynamic DNS client
- Setting up a static IP on your Raspberry Pi
- Port-forwarding
- Summary

8. A Miscellany of Things
- Arming and disarming the system
- Driving inductive loads
- Beyond intrusion
  - A simple water detector
    - How it works
  - A simple temperature sensor
    - How it works
  - A carbon monoxide detector
- Remote administration for our Raspberry Pi
  - Getting Webmin
    - Updating the repository sources
    - Importing the signing key
    - Accessing Webmin locally
    - Remotely accessing Webmin
- Summary

9. Putting It All Together
- Alarm system diagram
  - Overview of the system elements
    - A +12V power supply
    - A +3.3V power supply
    - The opto-isolator input module
    - The port expander
    - An arm/disarm switch
    - Alarm outputs
- Designing the control scripts
- Building the control script
  - Exploring the script code
    - Declarations
    - Updating config settings
    - Setting up the GPIO
    - Setting up the I2C port expander

Decoding the zone inputs status
Initialization
The system monitoring loop
Arming the system
Monitoring the zones
Entry delay
Sounding the main alarm
Disarming and resetting the system
We're done (almost)–
Automatically starting the system
Preserving the SD card
Creating a RAM-based file system
Conclusion
Tips for building systems
Summary
3. Module 3
1. Configuring and Programming Raspberry Pi
Configuring Raspberry Pi – the brain of your robot
Installing the operating system
Adding a remote graphical user interface
Establishing wireless access
Programming on Raspberry Pi
Creating and running Python programs on the Raspberry Pi
An introduction to the C/C++ programming language
Summary
2. Building the Biped
Building robots that can walk
How servo motors work
Building the biped platform
Using a servo controller to control the servos
Communicating with the servo controller with a PC
Connecting the servo controller to the Raspberry Pi
Creating a program to control your biped
Summary
3. Motion for the Biped
A basic stable pose
A basic walking motion

A basic turn for the robot
Summary
4. Avoiding Obstacles Using Sensors
Connecting Raspberry Pi to an infrared sensor
Connecting Raspberry Pi to a USB sonar sensor
Summary
5. Path Planning and Your Biped
Connecting a digital compass to the Raspberry Pi
Accessing the compass programmatically
Dynamic path planning for your robot
Basic path planning
Avoiding obstacles
Summary
6. Adding Vision to Your Biped
Installing a camera on your biped robot
Installing a USB camera on Raspberry Pi
Installing RaspiCam on Raspberry Pi
Downloading and installing OpenCV – a fully featured vision library
Edge Detection and OpenCv
Color and motion finding
Summary
7. Accessing Your Biped Remotely
Adding a wireless dongle and creating an access point
Adding a joystick remote control
Adding the capability to see remotely
Summary
A. Bibliography
Index

Raspberry Pi: Amazing Projects from Scratch

Raspberry Pi: Amazing Projects from Scratch

Explore the powers of Raspberry Pi and build your very own projects right out of the box

A course in three modules



BIRMINGHAM - MUMBAI

Raspberry Pi: Amazing Projects from Scratch

Copyright © 2016 Packt Publishing All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: September 2016

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street Birmingham B3 2PB, UK.

ISBN 978-1-78712-849-1

www.packtpub.com

Credits

Authors

Ashwin Pajankar

Arush Kakkar

Matthew Poole

Richard Grimmett.

Reviewers

Nathan Johnson

Elliot Kermit-Canfild

Anna Torlen

Lihang Li

Cédric Verstraeten

Ashwin Pajankar

Werner Ziegelwanger

Content Development Editor

Sumeet Sawant

Production Coordinator

Arvindkumar Gupta

Preface

Machine learning and predictive analytics are becoming one of the key strategies for unlocking growth in a challenging contemporary marketplace .It is one of the fastest growing trends in modern computing and everyone wants to get into the field of machine learning. In order to obtain sufficient recognition in this field, one must be able to understand and design a machine learning system that serves the needs of a project. The idea is to prepare a Learning Path that will help you to tackle the real-world complexities of modern machine learning with innovative and cutting-edge techniques. Also, it will give you a solid foundation in the machine learning design process, and enable you to build customized machine learning models to solve unique problems

What this learning path covers

Module 1, *Raspberry Pi By Example*, provides you an introduction to the Raspberry Pi. It helps in building games with PyGame and creation of real-life applications with the Raspberry Pi. It further demonstrates the GPIO and cameras with advanced concepts in OpenCV. This module further delves with setting up a web server and creating network utilities.

Module 2, *Building a Home Security System with Raspberry Pi, lets you* explore the GPIO Port along with building input/ output expansion board which helps in overcoming the limitations on GPIO. It helps you to create log files based on events using Bash Scripts. This module further covers a miscellany of things for accessing home security control panel.

Module 3, *Raspberry Pi Robotics Essentials*, starts with configuring and programming Raspberry Pi, along with construction of biped platform. It further covers in-depth planning of biped with the details

of connecting webcam, hardware and software in order to use input visual data into our system.

What you need for this learning path

- Module 1, Raspberry Pi By Example, recommends you to use the following kit: Raspberry Pi Model B, B+ or 2 (Multiple boards for last two chapters) USB hub, powered preferably Networking hub PC for preparing SD card Webcam and/or Pi Camera

- Module 2, Building a Home Security System with Raspberry Pi,, will need the following software: Gparted dd fake-hwclock Win32 Disk Imager 0.9.5 PuTTY

  i2c-tools

- Module 3, Raspberry Pi Robotics Essentials, asks you to use Raspbian, putty, Image Writer for Windows, libusb-1.0-0-dev and VncServer.

Who this learning path is for

Novice programmers and hobbyists who want to understand how to use Raspberry Pi to build interesting projects and home automation systems, as well as for those who want to delve deeper into the world of Raspberry Pi

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at http://www.packtpub.com. If you purchased this course elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at https://github.com/PacktPublishing/Raspberry-Pi-Making-Amazing-Projects-Right-from-Scratch-

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Part 1. Module 1

Raspberry Pi By Example

Start building amazing projects with the Raspberry Pi right out of the box

Chapter 1. Introduction to Raspberry Pi and Python

One can learn about topics in computer science in an easy way with the Raspberry Pi and Python. The Raspberry Pi family of single-board computers uses Python as the preferred development language. Using Raspberry Pi and Python to learn programming and computer science-related concepts is one of the best ways to start your journey in this amazing world of computers that is full of creative possibilities. We will explore these possibilities in this book.

We will commence our journey in this chapter by getting ourselves familiar with the following topics:

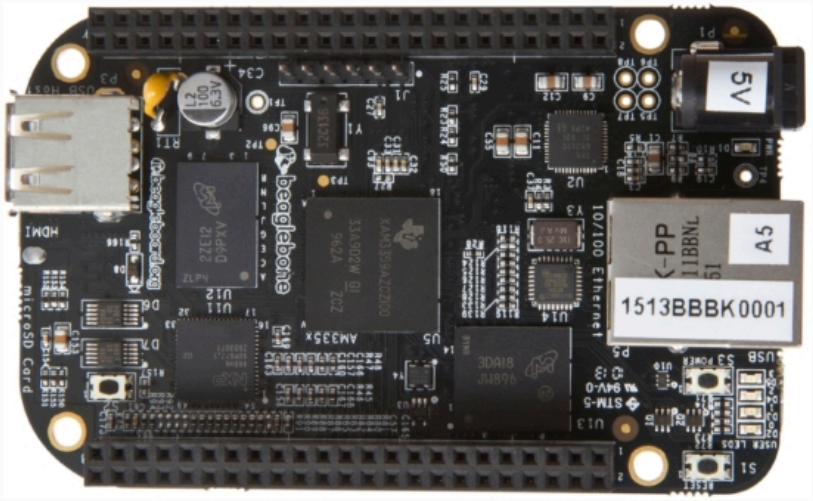
- Single-board computers
- Raspberry Pi
- Raspbian
- Setting up Raspberry Pi
- Basics of Python
- Turtle programming with Python

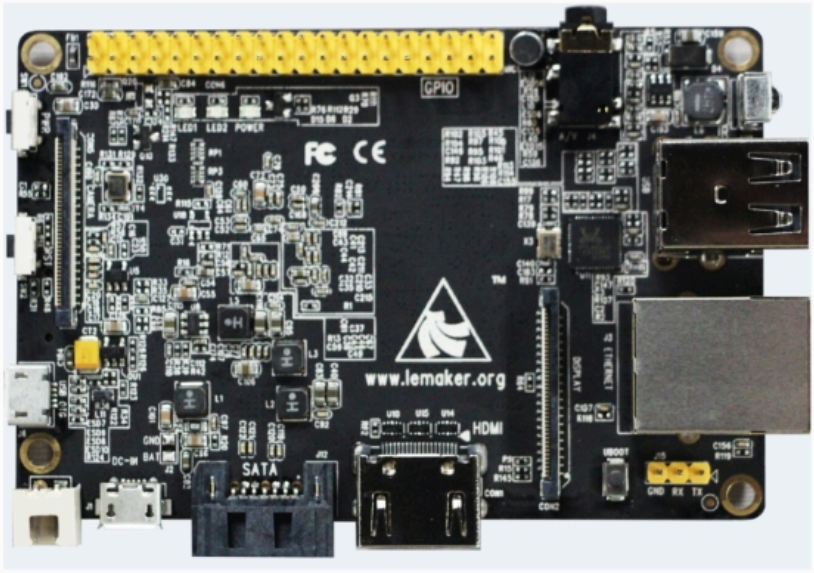
Single-board computers

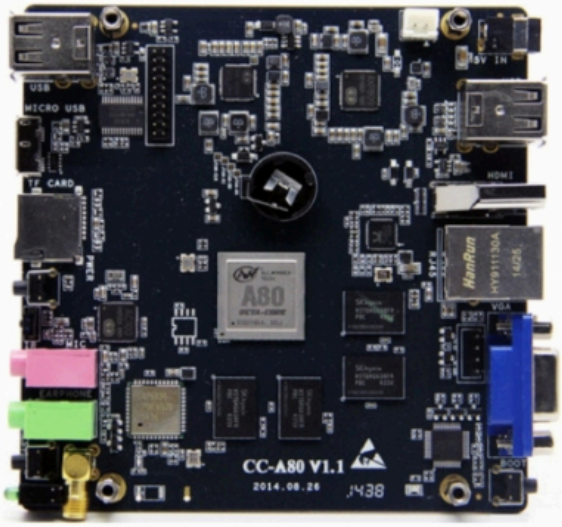
A single-board computer system is a complete computer on a single circuit board. The board includes a processor(s), RAM, input/output (I/O), and networking ports for interfacing devices. Unlike traditional computer systems, a single-board computer is not modular and its hardware cannot be upgraded as it is integrated on the board itself. Single-board computers are used as low-cost computers in academia, research, and embedded systems. The use of single-board computers in embedded systems is quite prevalent and many individuals and organizations have developed and released fully functional products based on single-board computers.

The Microcomputer Trainer MMD-1 designed by John Titus in 1976 is the first true single-board microcomputer that was based on the Intel C8080A. It was called **dyna -micro** in the prototyping phase, and the production units were called **MMD-1** (short for **Mini Micro Designer 1**).

Popular single-board computers available in the market include but are not limited to Raspberry Pi, Banana Pro, BeagleBone Black, and Cubieboard. The following images are of the front view of BeagleBone Black, Banana Pro, and Cubieboard 4, respectively:


beagleboard.org
10/100 Ethernet
microSD Card
HDMI
1513BBBK0001


GPIO
www.lemaker.org
HDMI
SATA


CC-A80 V1.1
2014.08.26
A80

Raspberry Pi

The Raspberry Pi is a series of low-cost, palm-sized single-board computers developed by Raspberry Pi Foundation in the UK. The intention behind the creation of the Raspberry Pi is to promote the teaching of basic computer skills in schools, which it serves very well. Raspberry Pi has expanded its footprint well beyond its intended purpose by penetrating the embedded systems market and computer science research.

Note

This is the home page of Raspberry Pi Foundation: http://www.raspberrypi.org.

The Raspberry Pi is manufactured with licensed agreements with Newark element14, RS Components, Allied Electronics, and Egoman. These companies manufacture and sell the Raspberry Pi. The hardware is the same across all manufacturers.

The following table displays the URLs of the manufacturers' websites, where you can shop for Pi and related items online:

Manufacturer	**Website**
Newark element14	http://www.newark.com
RS Components	http://uk.rs-online.com

Manufacturer	Website
Egoman	http://www.egoman.com.cn
Allied Electronics	http://www.alliedelec.com

You can also shop for Pi and the other third-party add-ons at the following links:

- http://shop.pimoroni.com
- http://www.adafruit.com

Raspberry Pi models

The following are, at the time of writing this, the major models of Raspberry Pi:

- Model A (not in production; discontinued in favor of the production of later and upgraded models)
- Model A+ (currently in production and available for purchase)
- Model B (available for purchase but not in production)
- Model B+ (currently in production and available for purchase)
- Raspberry Pi 2 Model B (currently in production and available for purchase)

Note

Check out the **Product** page of Raspberry Pi at http://www.raspberrypi.org/products/.

Additionally, Raspberry Pi is also available in a more flexible form factor intended for industrial and embedded applications. It is known as **Compute Module**. A Compute Module prototyping kit is also made available by the foundation.

Note

Check out the following URLs for the Compute Module and Compute Module development kit, respectively:

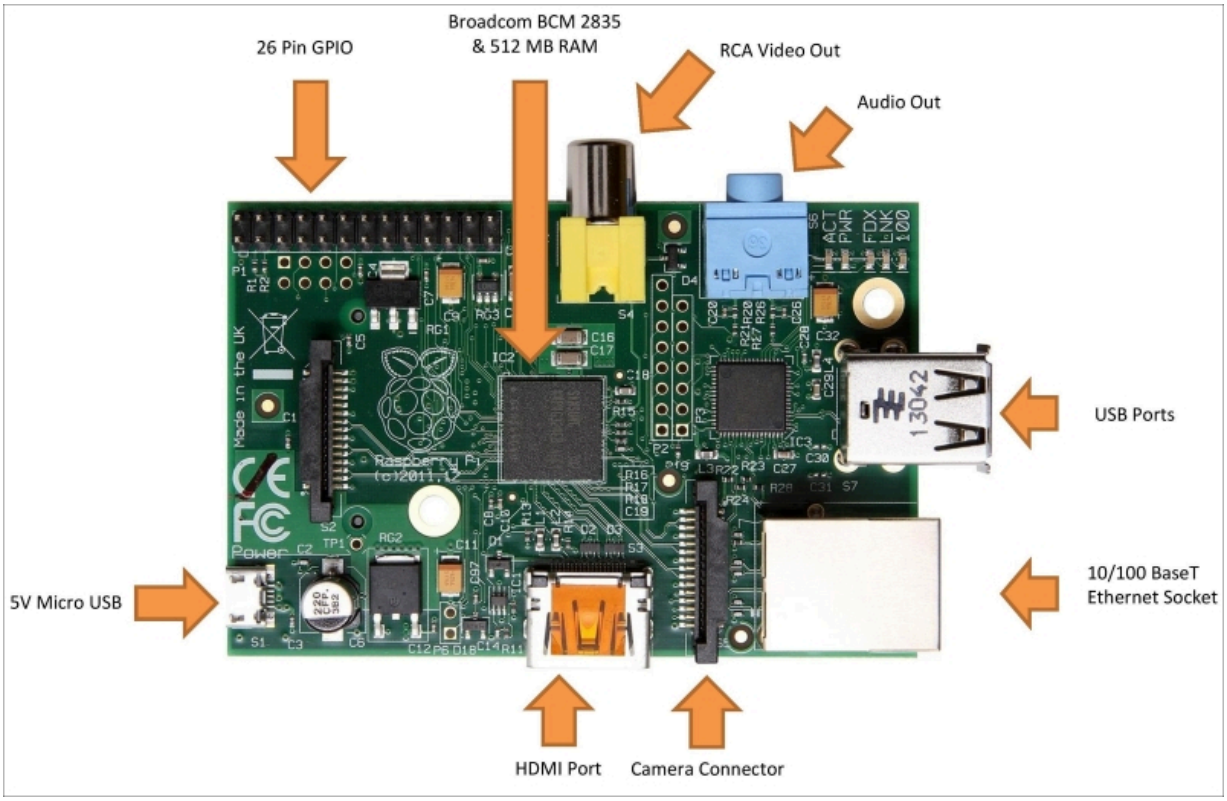
http://www.raspberrypi.org/products/compute-module/

http://www.raspberrypi.org/products/compute-module-development-kit/

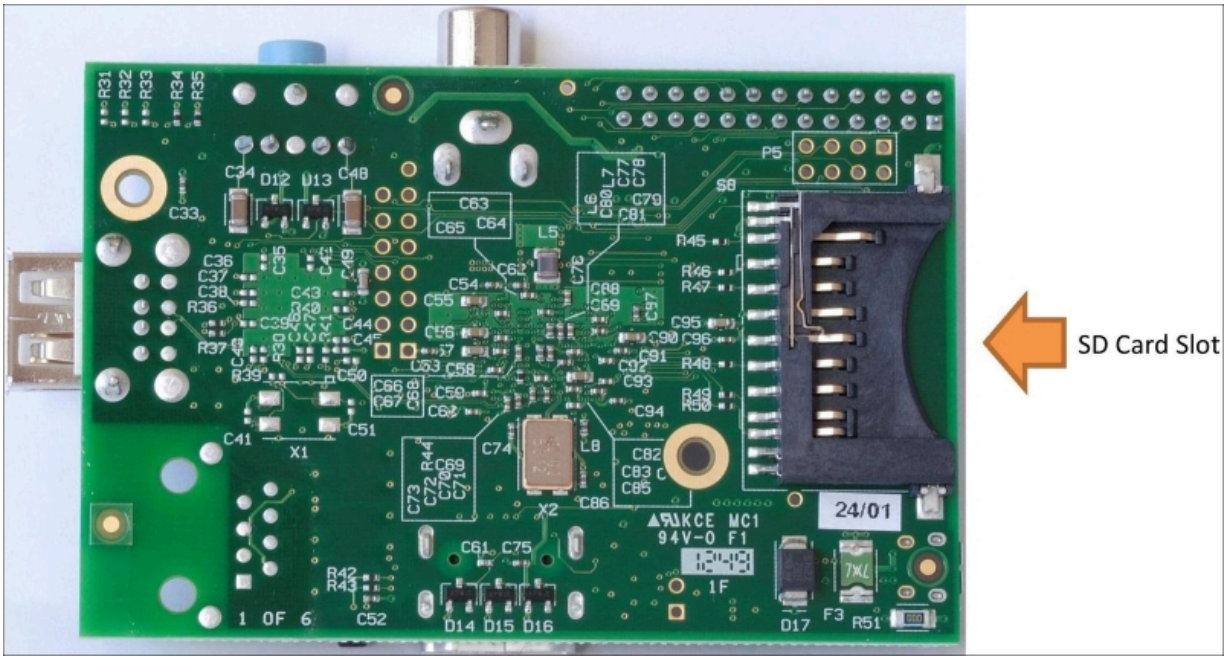
The following table compares the currently available models of Pi:

	Model A	Model B	Model B+	Pi 2 Model B
SOC (System on a chip)	Broadcom BCM2835(CPU, GPU, DSP, SDRAM, one USB port)			Broadcom.BCM2836 (CPU, GPU, DSP, SDRAM, one USB port)
CPU	700 MHz single-core ARM1176JZF-S			900 MHz quad-core ARM Cortex-A7
GPU	Broadcom videoCore IV @ 250 MHz			
Memory (shared with GPU)	256 MB SDRAM	516 MB SDRAM		1 GB SDRAM
USB 2.0 ports	1	2		4
On Board Storage	MicroSD slot	SD / MMC / SDIO card slot	MicroSD slot	
Networking	None	10/100.Mbit/s Ethernet, no Bluetooth or Wi-Fi		
Power Source	5V via Micro USB or GPIO Headers (Power supply through micro USB is recommended.)			
Power Ratings	200mA (1 W)	700mA (3.5 W)	600mA (3.5 W)	800mA (4 W)

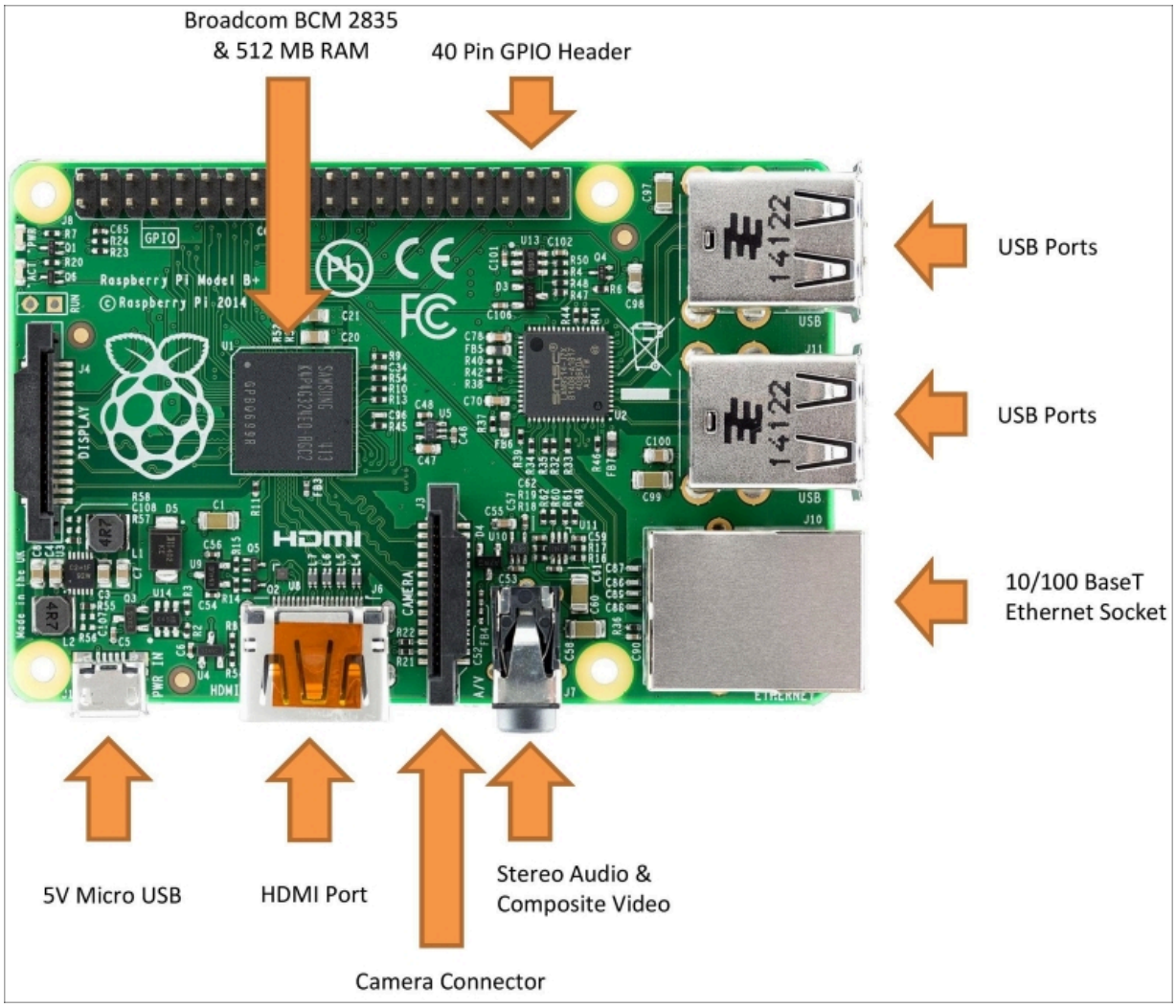
The following image shows the top view of the Raspberry Pi Model B front:



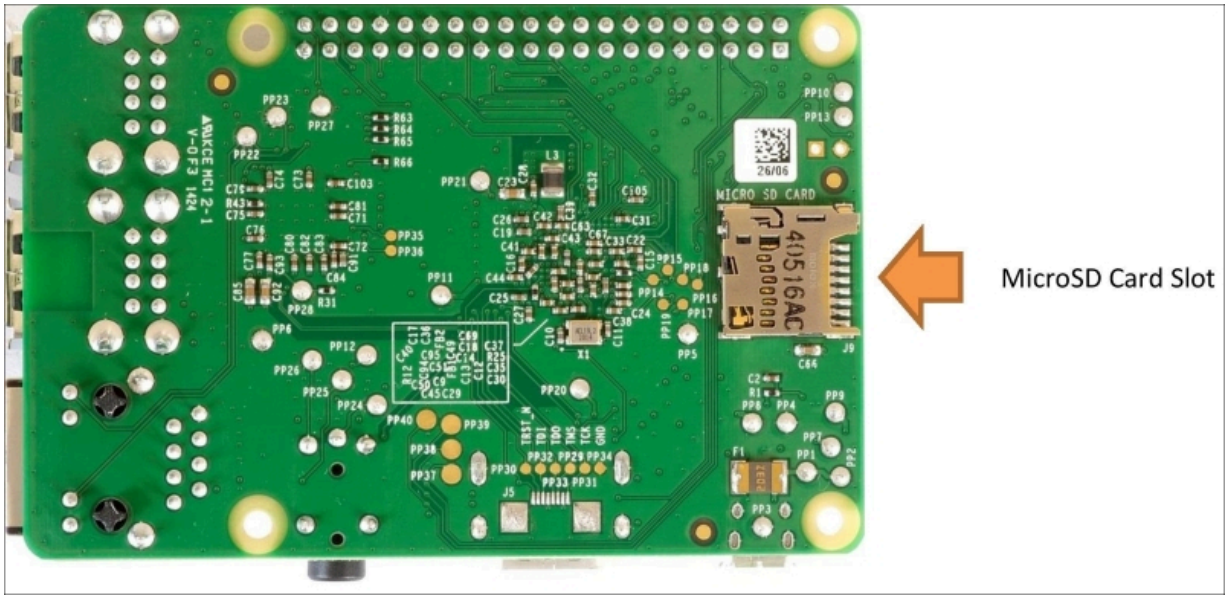
The following image shows the top view of the flip side of Raspberry Pi Model B:



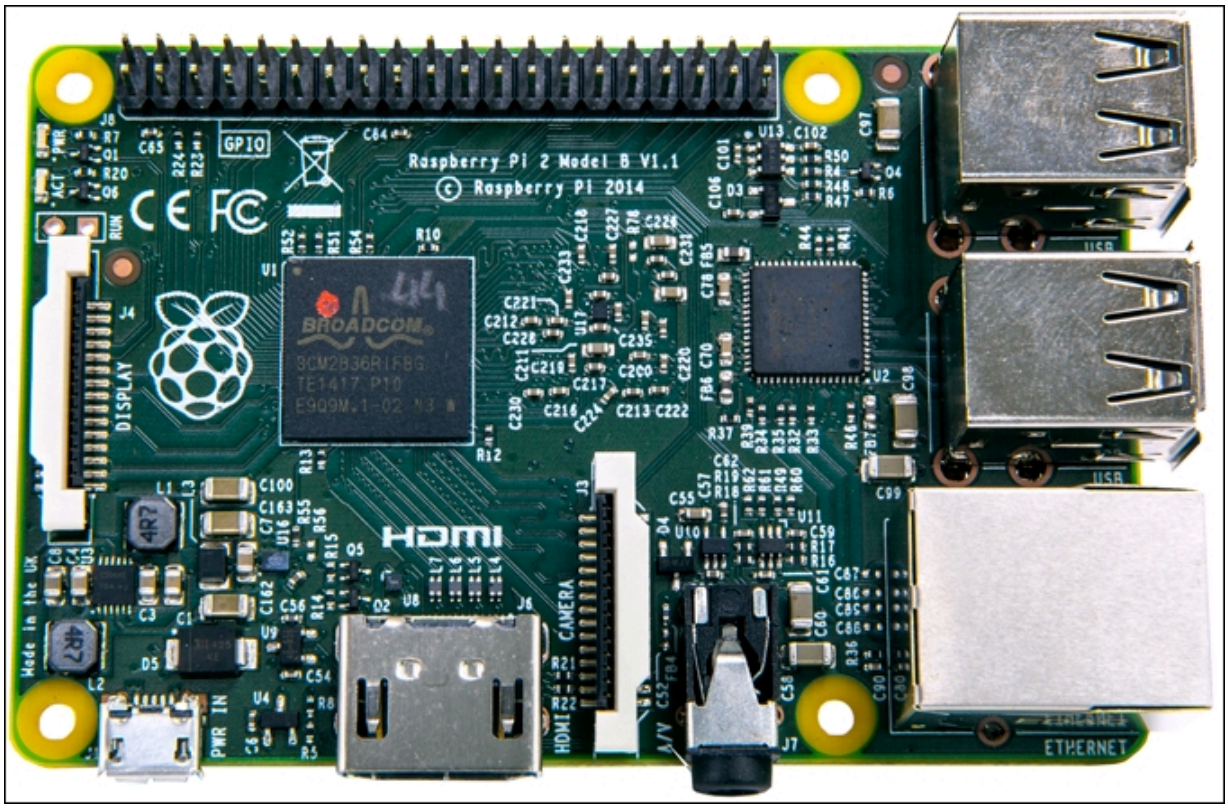
The following image shows the top view of the Raspberry Pi Model B+ front:



The following image shows the top view of the flip side of Raspberry Pi Model B+:



The following image shows the top view of the Raspberry Pi 2 Model B front. The location of the connectors and important ICs (integrated circuits) on the board is not different from Pi B+:



We will be using Raspberry Pi 2 Model B throughout this book. However, all the applications and programs in this book will work on all the models of Pi.

Operating systems

The Raspberry Pi primarily uses Unix-like Linux-kernel-based operating systems, such as variants of Debian and Fedora.

Raspberry Pi Models A, A+, B, and B+ are based on the ARM11 family chip, which runs on the ARMv6 instruction set. The ARMv6 instruction set does not support Ubuntu and Windows.

However, the recently launched Raspberry Pi 2 is based on ARM Cortex A7, which is capable of running Windows 10 and Ubuntu (Snappy Core). The following operating systems are officially supported by all the models of Raspberry Pi and are available for download at the download page:

- Raspbian: We will be using this with Raspberry Pi throughout the book
- OpenELEC
- Pidora (Fedora Remix)
- RASPBMC
- RISC OS

> **Note**
>
> Windows 10 and Ubuntu are only supported by the recently launched Pi 2.

Raspbian

Raspbian is an unofficial variant of Debian armhf (ARM Hard Float) compiled for *hard float* code that will run on Raspberry Pi computers. It is a free operating system based on Debian optimized for the Raspberry Pi hardware.

Note

To know more about Raspbian, visit http://www.raspbian.org/.

Setting up the Raspberry Pi

We need the following hardware to set up a Pi.

- Raspberry Pi 2 Model B (hereafter, this will be referred only as Pi).
- Power Supply: A micro USB power supply.

  Considering that we are going for slightly power-intensive usage of our Pi (such as connecting Pi Camera, webcam, and third-party sensors for Pi), a 5V 2A power supply is recommended. The micro USB pin is shown in the following image:



Note

> You can find a similar one online at http://www.adafruit.com/product/1995.

- A standard USB keyboard
- A MicroSD card and a MicroSD to SD card converter

  We need a minimum 4 GB Micro SD card.

- A USB mouse
- A monitor

  You can use either an HDMI monitor or a standard VGA monitor.

- A monitor connection cable and converter

  If you are using HDMI monitor, then an HDMI cable will be sufficient. If you are using a VGA monitor, then you need to use an HDMI to VGA converter with a VGA cable. Some special changes need to be made to the `/boot/config.txt` file if you're using a VGA monitor, which will be explained in the next section.



> **Note**
>
> You can find a similar one online at https://www.adafruit.com/products/1151.

- A Windows, Linux, or Mac OS computer with a MicroSD card reader and an Internet connection

Preparing MicroSD card manually

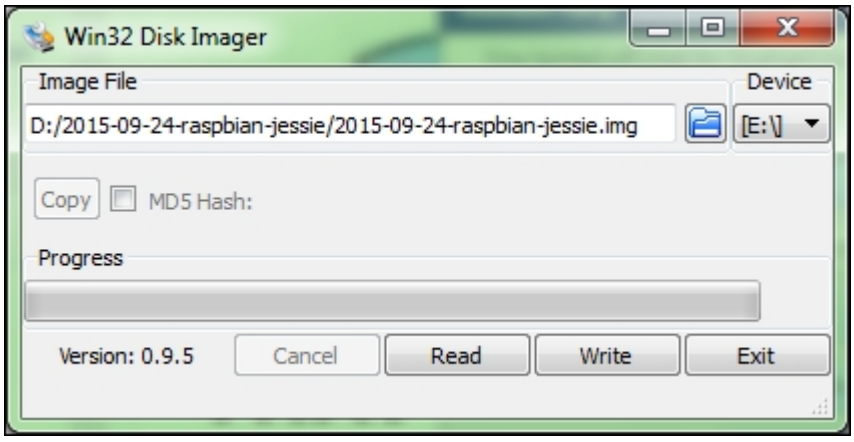
This is the original way to install an OS into a MicroSD card, and many users, including me, still prefer it. It allows the SD card to be prepared manually before it is used and it allows easier access to configuration files such as `/boot/config.txt`, which we might have to modify in a few cases before booting up. The default Raspbian image consists of only two partitions, `BOOT` and `SYSTEM`, which will fit into a 2 GB card. However, I recommend that you use a minimum 4 GB card to be on safe side. Choosing an 8 GB card will be adequate for most of the applications we are going to develop in this book.

The following are the instructions for Windows users:

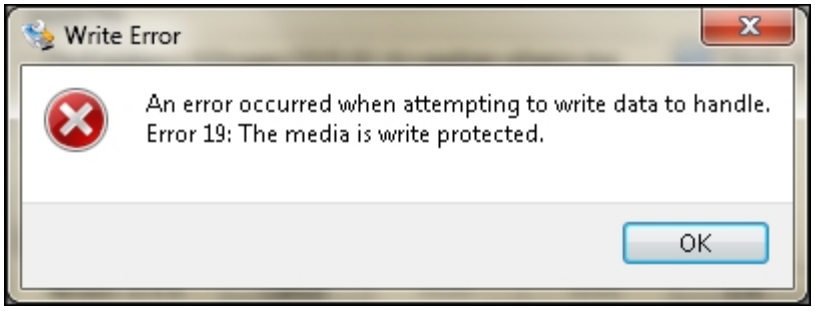
1. Download the Win32DiskImager installer, which is available at http://sourceforge.net/projects/win32diskimager/files/latest/download and then install it.
2. Download the installable version of WinZip, which is available at http://www.winzip.com/prod_down.html, and install it.
3. Go to http://www.raspberrypi.org/downloads and download the latest image of Raspbian. It will be a compressed file in the ZIP format and will need to be extracted.
4. Extract the ZIP file using WinZip. The extracted file will be in the `.img` format.
5. Insert the microSD card into the card reader and plug the card reader into the computer. Many computers nowadays have an inbuilt SD card reader. In this case, you will need to insert the microSD card into the microSD to SD card converter and insert

it into the computer's inbuilt SD card reader. MicroSD to SD card converters usually come bundled with microSD cards in the same package. If that's not the case, then you will have to procure it separately.

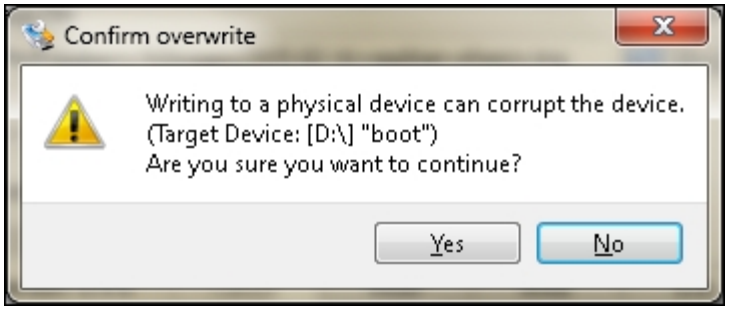
6. Run `Win32DiskImager.exe` and write the image onto the SD card:



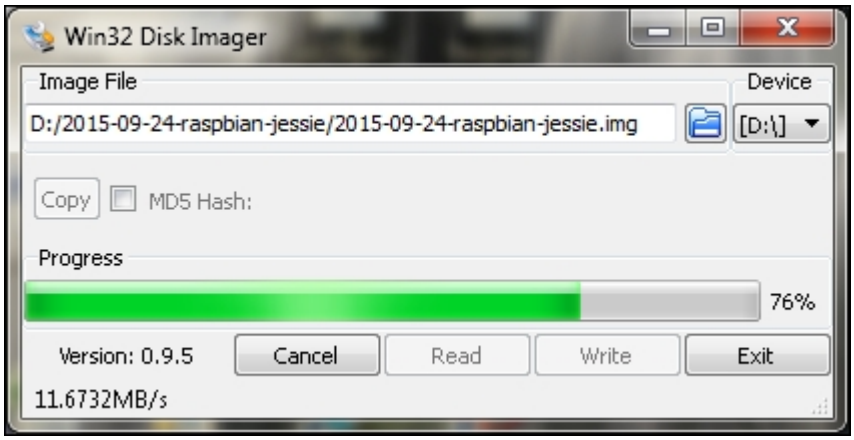
You might receive the following message if the card reader's write protection is on:



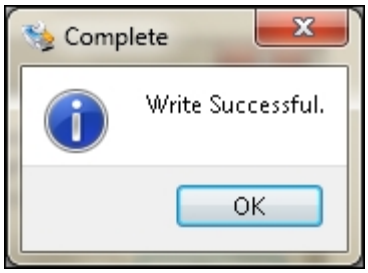
7. Toggle the write protection notch and try again. You will see the following message:



8. Click on **Yes** and it will start writing the image file to the microSD card:



9. Once the image is successfully written, it will display the following message:



Note

If you are using Linux, then you can find the instructions at https://www.raspberrypi.org/documentation/installation/installing-images/linux.md.

If you are using Mac OS, then you can find the instructions at https://www.raspberrypi.org/documentation/installation/installing-images/mac.md.

If you have an HDMI monitor, then skip this step. This additional step is required only if you are planning to use a VGA monitor in place of an HDMI monitor.

Browse the microSD card on the computer. Locate and open `config.txt`. We have to edit the file in order to enable proper display on the VGA monitor.

By default, the commented options (which have `#` at the beginning) are disabled. We are enabling this option by uncommenting this line, that is, by removing `#` from the beginning of the commented line. This is what you need to do:

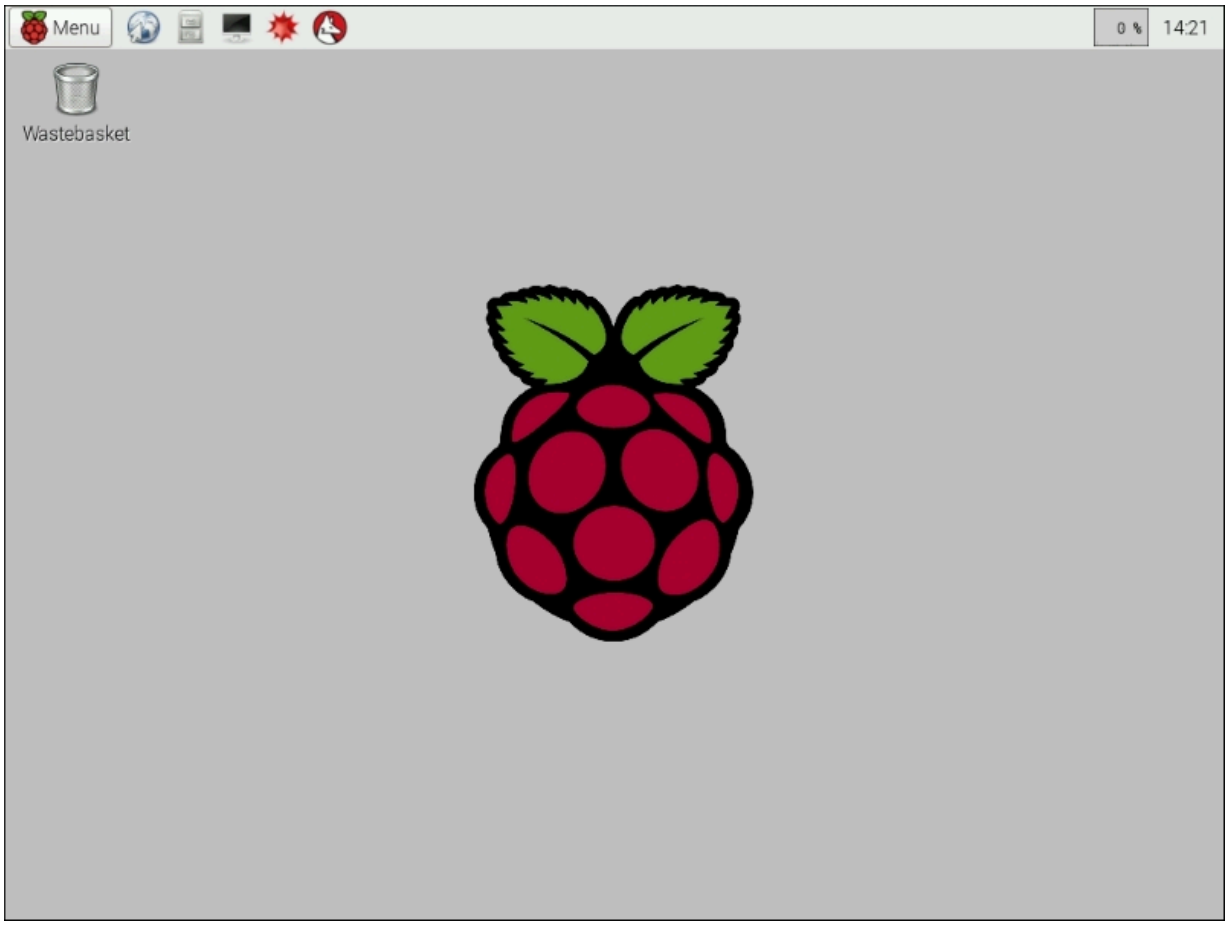
1. Change `#disable_overscan=1` to `disable_overscan=1`.
2. Change `#hdmi_force_hotplug=1` to `hdmi_force_hotplug=1`.
3. Change `#hdmi_group=1` to `hdmi_group=2`.
4. Change `#hdmi_mode=1` to `hdmi_mode=16`.
5. Change `#hdmi_drive=2` to `hdmi_drive=2`.
6. Change `#config_hdmi_boost=4` to `config_hdmi_boost=4`.
7. Save the file.

Booting up our Pi for the first time

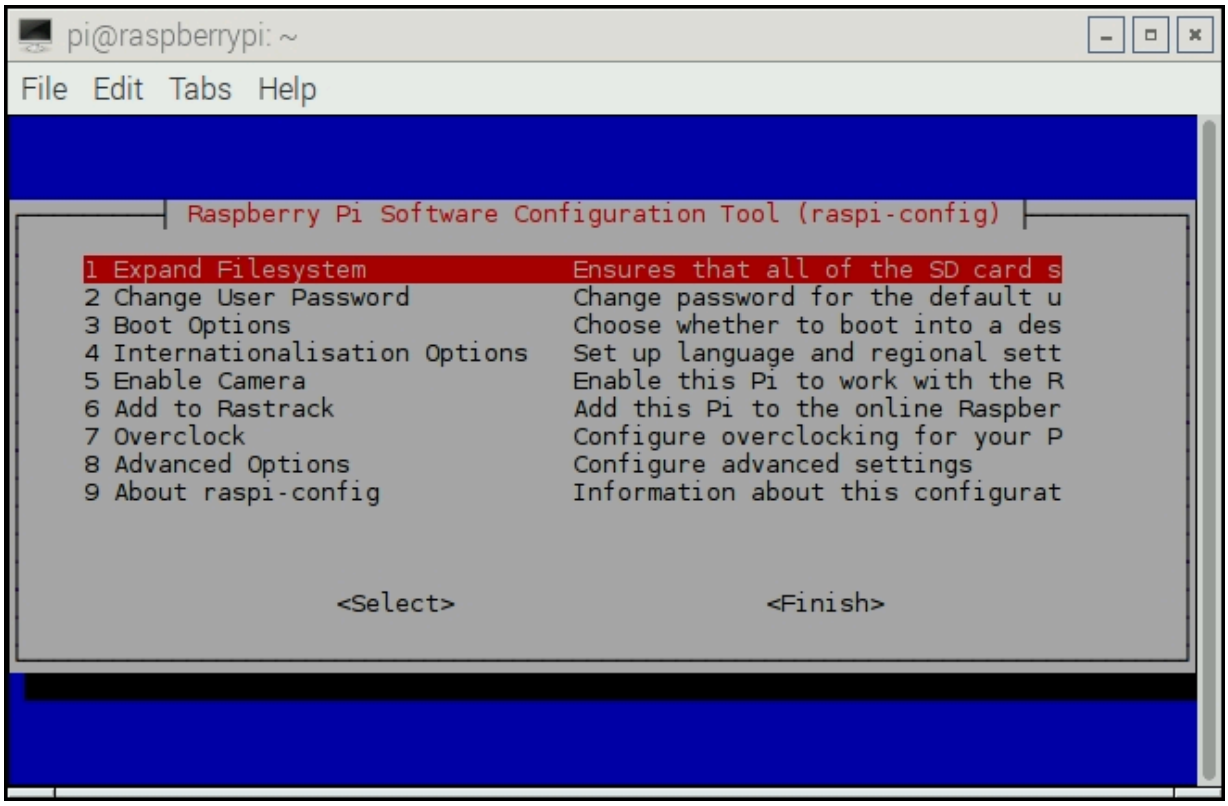
Let's boot up our Pi for the first time with the microSD card:

1. Insert the microSD card into the microSD card slot of the Pi.
2. Connect the Pi to the HDMI monitor. If you are connecting the VGA monitor, connect it using the HDMI to VGA converter.
3. Connect the USB mouse and the USB keyboard.
4. Connect the Pi to a power supply using the micro USB power cable. Make sure the power is switched off at this point.
5. Check all the connections once and then switch on the power supply of the Pi.

At this stage, our Pi will start booting up. You will see a green light on the Pi board blinking. This means that it's working! Now, there are few more things we need to do before we can really start using our Pi. Once it boots up, it will show the desktop as follows:

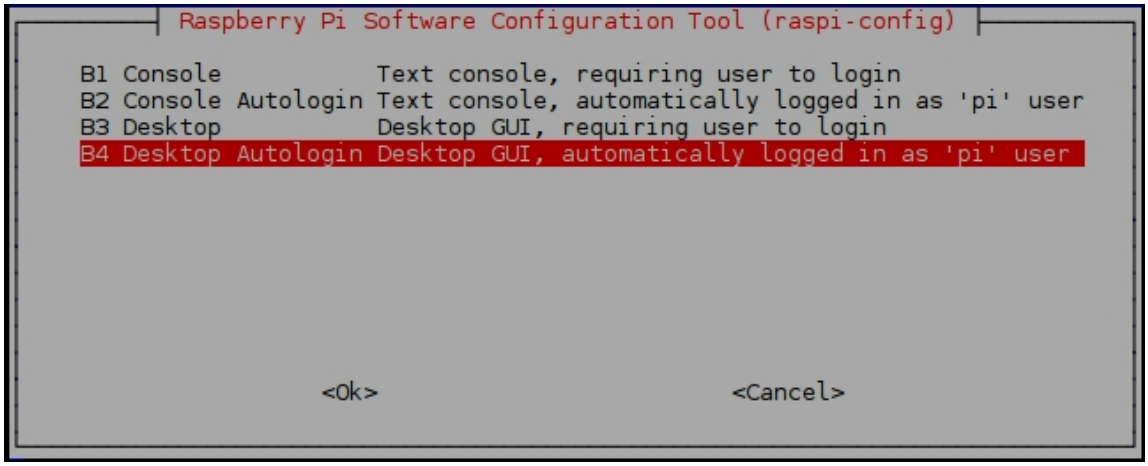


Once the desktop is visible, go to **Menu** | **Accessories** | **lxterminal**. Then, type `sudo raspi-config`. A text-based menu, such as the following, will appear:



Perform the following steps. We need to use arrow keys and the *Enter* key to select options in the text-based menu. Press *Enter* to select a menu item. Also, we can use the *Tab* key to directly go to the **Select** and **Finish** buttons:

1. Select **Expand Filesystem**.
2. In **Boot Options**, select **B4 Desktop Autologin**, as shown in the following screenshot:



Note

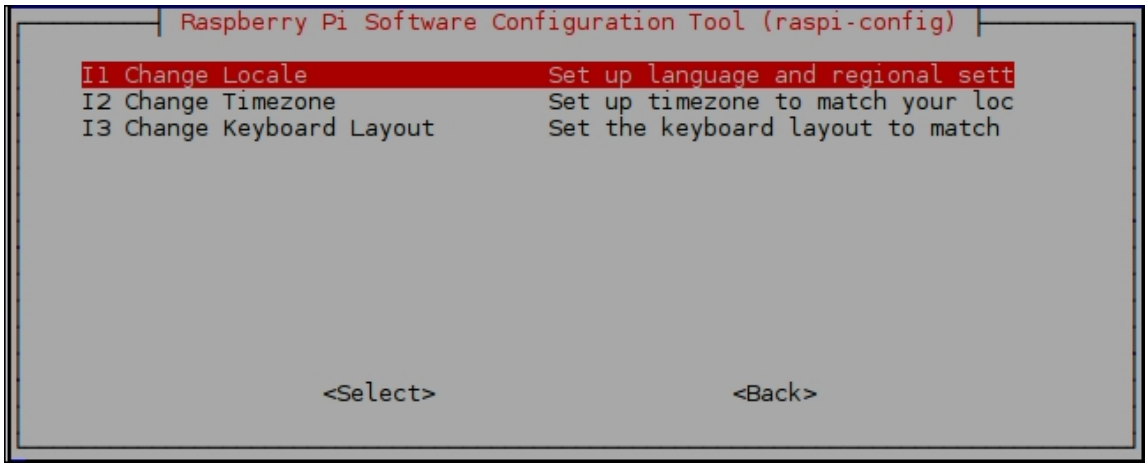
The default username is `pi` and the password is `raspberry`. We need it when we don't choose any of the preceding autologin options. We can change this password from the second option in the `raspi-config` menu.

We can also choose to boot to the console by selecting any of the first two options in the preceding menu. The default shell of Raspbian is Bash. We can confirm it by typing the following command:

```
echo $SHELL
```

We can always go to the graphical desktop from the Command Prompt by typing the `startx` command in the console.

3. Go to **Internationalisation Options** | **Change Timezone**.
4. Go to **Internationalisation Options** | **Change Keyboard Layout** | **Change it to US** (the default is **UK**).



5. Select **Enable Camera**.
6. Select **Advanced Options**.
7. Under this option, select **Memory Split** and enter `64MB` for GPU.

This option decides how much RAM is used by the **GPU** (**Graphics Processor Unit**). The more RAM is allocated to the GPU, the more intensive graphics processing can be done. 64 MB is a good value for most graphics purposes.

Once all these options are modified, select **Finish**. This will prompt for a reboot of the Pi. Choose **Yes** and let it reboot. Once rebooted, it will automatically take us to the Raspbian Desktop again.

You can always invoke the `raspi-config` tool from Command Prompt with the following command and change the settings:

```
sudo raspi-config
```

Shutting down and rebooting Pi safely

In the Raspbian menu, there are options to shut down and reboot the Pi.

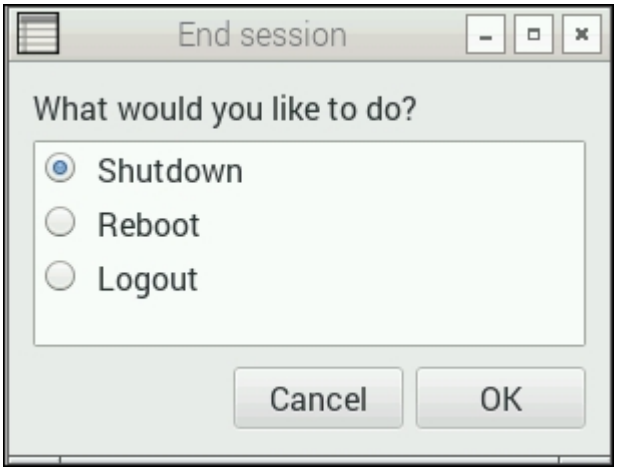
If we click on the following **Menu** button on the desktop, it will display multiple options:



The following image shows the last option:



If we click on the preceding option, the following window will appear:



Also, from Command Prompt LXTerminal, we can shut down Pi safely by issuing the following command:

```
sudo shutdown –h now
```

An alternative command for this is as follows:

```
sudo halt
```

You can reboot Pi with the following command:

```
sudo reboot
```

Updating the Pi

Now we have a working Pi running on the Raspbian OS. Let's update our Pi. Make sure you have a working wired or wireless Internet connection with reasonable speed for this activity:

1. Connect your Pi to an Internet modem or router with an Ethernet cable or plug in the Wi-Fi dongle to one of the USB ports.
2. Run the following command to restart the networking service:

```
sudo service networking restart
```

3. Make sure that your Raspberry Pi is connected to the Internet by typing the following command:

```
ping -c4 www.google.com
```

4. `apt` (Advanced Package Tool) is the utility used to install and remove software in Debian and its variants. We need to use it to update our Pi software.
5. Run the following commands in a sequence:

    - `sudo apt-get update`: This command synchronizes the package list from the source. Indexes of all the packages are refreshed. This command must be issued before we issue the `upgrade` command.
    - `sudo apt-get upgrade`: This command will install the newest versions of all the already installed software. Any obsolete packages/utilities are not removed automatically. If any software is in its newest version, then it's left as it is.
    - `sudo rpi-update`: This command is used to upgrade the firmware. The kernel and firmware are installed as a Debian package, and so they will also get updates. These packages are updated infrequently after extensive testing.
    - `sudo reboot`: This will reboot the computer.

Getting started with Python

Python is a high-level general-purpose programming language. It supports multiple programming paradigms, such as object-oriented programming, imperative programming, functional programming, procedural programming, aspect-oriented programming, and metaprogramming. It has a dynamic type system, automatic memory management, and a large standard library to carry out various tasks. It emphasizes code readability, and its syntax allows you to carry out tasks in fewer lines of code than other programming languages, such as C or C++.

Python was conceived and implemented by Guido van Rossum at CWI in the Netherlands as a successor to the ABC language, capable of exception handling and interfacing with the Amoeba operating system platform. Van Rossum is Python's principal author, and he continues to have the central role in deciding the direction of Python. He has been endowed with the title **benevolent dictator for life** (**BDFL**) by the worldwide Python community.

The core philosophy of the Python programming language is mentioned in this URL: https://www.python.org/dev/peps/pep-0020/; its first few lines are as follows:

"Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

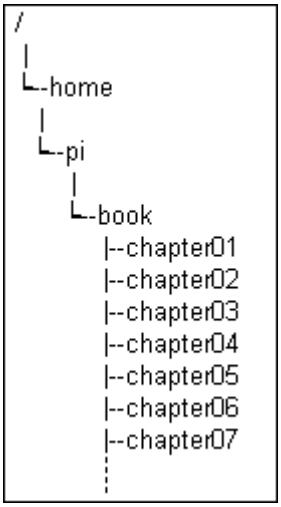
Complex is better than complicated.

Flat is better than nested.

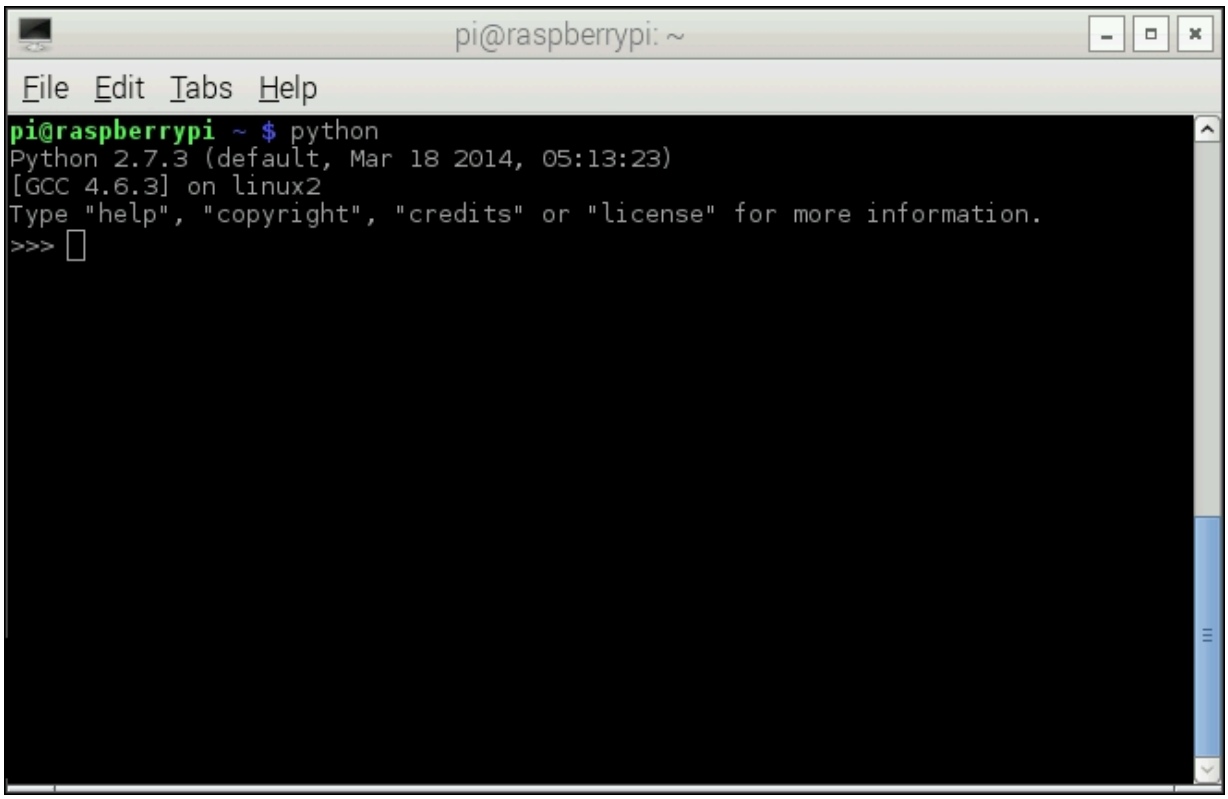
Sparse is better than dense.

Readability counts."

Python is the preferred programming language for the Raspberry Pi family of computers. Its interpreter comes preinstalled with Raspbian, and there is no need for any additional installation to get started with code. It is recommended that you have chapter wise directories for the code examples presented in this book, as shown in the following diagram:



Let's get started with Python. Open Raspbian's Command Prompt LXTerminal. It is located as a shortcut on the taskbar. Alternately, we can find it by navigating to **Menu** | **Accessories** | **Terminal**. We can start Python in interactive mode by typing `python` in the prompt and then pressing the *Enter* key. It will take us to the Python interactive shell, as follows:



Now type the following lines and press *Enter*:

```
print "Hello World!"
```

The output will be as follows:

```
Python 2.7.3 (default, Mar 18 2014, 05:13:23)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> print "Hello World!"
Hello World!
```

Congrats! We have started with Python programming. The interactive mode of Python is suitable for small programs. Press *Ctrl + D* to exit the interactive shell. For large and code-intensive projects, it's recommended that you use Python in script mode. In this book, we will be using Python in script mode unless specified explicitly. Let's look at how to use Python in script mode.

Create a subdirectory, `book`, in the `/home/pi` directory for the code examples in this book. We can do this with `mkdir book` (by default, we are be in the `/home/pi` directory). Then, navigate to this recently created directory with the `cd book` command. We can verify our current directory with the `pwd` command. It returns the current directory as follows:

```
pi@raspberrypi ~ $ mkdir book
pi@raspberrypi ~ $ cd book
pi@raspberrypi ~/book $ pwd
/home/pi/book
pi@raspberrypi ~/book $
```

As discussed earlier, we need to keep the code of each chapter in separate directories for better organization. Now, create the `chapter01` directory for this chapter in the current `book` directory using `mkdir chapter01`. Navigate to this with `cd chapter01`. At the beginning of each chapter, we are required to create a directory for the chapter under `/home/pi/book` for the code examples of that chapter.

We will now create a script file for our code and run it.

Use the Nano text editor to create and edit script files. If we type `nano prog1.py`, then Nano will open `prog1.py` for editing if it already exists in the current directory; otherwise, it will create a new file with the name `prog1.py`. You can exit the Nano editor by pressing *Ctrl* + *X*.

Note

You can find more information about nano at http://www.nano-editor.org/.

Alternatively, you can use the Leafpad text editor. We can find it by navigating to **Menu** | **Accessories**. Or, we can invoke it from Command Prompt with the `leafpad prog1.py` command.

Finally, you can also use `vim`, but you will need to install it by running the following command:

```
sudo apt-get install vim
```

> **Note**
>
> This is the link for an interactive tutorial on `vim`: http://www.openvim.com/.

Let's write the same Python code and run it as a script. Write the following code with Nano, Leafpad, or vim, and save it as `prog1.py`:

```
print "Hello World!"
```

To run the preceding program, use the `python prog1.py` command, and the output will be as follows.

We will run all the other Python programs in this book in the same way.

Let's try some more examples to have more hands-on Python.

The following is the iterative program to calculate the factorial of a given positive integer:

```
def fact(n):
    num = 1
    while n >= 1:
        num = num * n
        n = n - 1
    return num

print (fact(10))
```

In the preceding program, `def fact(n)` is a user-defined function that accepts an argument. The logic used to calculate the factorial of a

positive integer follows the definition and the function returns a calculated factorial. The last line of the program calls the factorial function and prints the returned output as follows:

```
pi@raspberrypi ~/book/chapter01 $ python prog2.py
3628800
```

The following is an iterative program for the Fibonacci series:

```
def fib(n):
    a=0
    b=1
    for i in range(n):
        temp=a
        a=b
        b=temp+b
    return a

for i in range (0,10):
    print (fib(i))
```

The preceding program prints the first 10 numbers in the Fibonacci series. A more "Pythonic" way of writing the same program, which eliminates the use of a temporary variable, is as follows:

```
def fib(n):
    a,b = 0,1
    for i in range(n):
        a,b = b,a+b
    return a

for i in range(0,10):
    print (fib(i))
```

The output of both of the preceding programs is the same and is as follows:

```
pi@raspberrypi ~/book/chapter01 $ python prog4.py
0
1
1
```

```
2
3
5
8
13
21
34
```

Turtle programming with Python

Turtle graphics is one of the best ways to learn programming for beginners. Originally, it was part of the Logo programming language, which was primarily used to introduce programming in schools. Python has the `turtle` module, which is an implementation of the same functionality provided by the original turtle. We can write programs with this module in a procedural as well as object-oriented way.

In Python, when we need to access a module that is not part of the current code, we need to import it. Over the course of the book, we will be importing various modules as and when needed, which will provide us with specific functionalities.

Let's get started with importing the `turtle` module, as shown here:

```
import turtle
```

The following code creates objects for turtle and the screen classes, respectively:

```
t=turtle.Turtle()
disp=turtle.Screen()
```

We will use the `t.color()` function with which we can set the pen and fill color, as follows:

```
t.color("black","yellow")
```

We will call the `t.begin_fill()` and `t.end_fill()` functions to have our shape filled with a fill color:

```
t.begin_fill()
t.end_fill()
```

The code to draw an actual shape we need will be in between these two function calls, as follows:

```
t.begin_fill()
while 1:
    t.forward(100)
    t.left(190)
    if abs(t.pos())<1:
        break
t.end_fill()
disp.exitonclick()
```

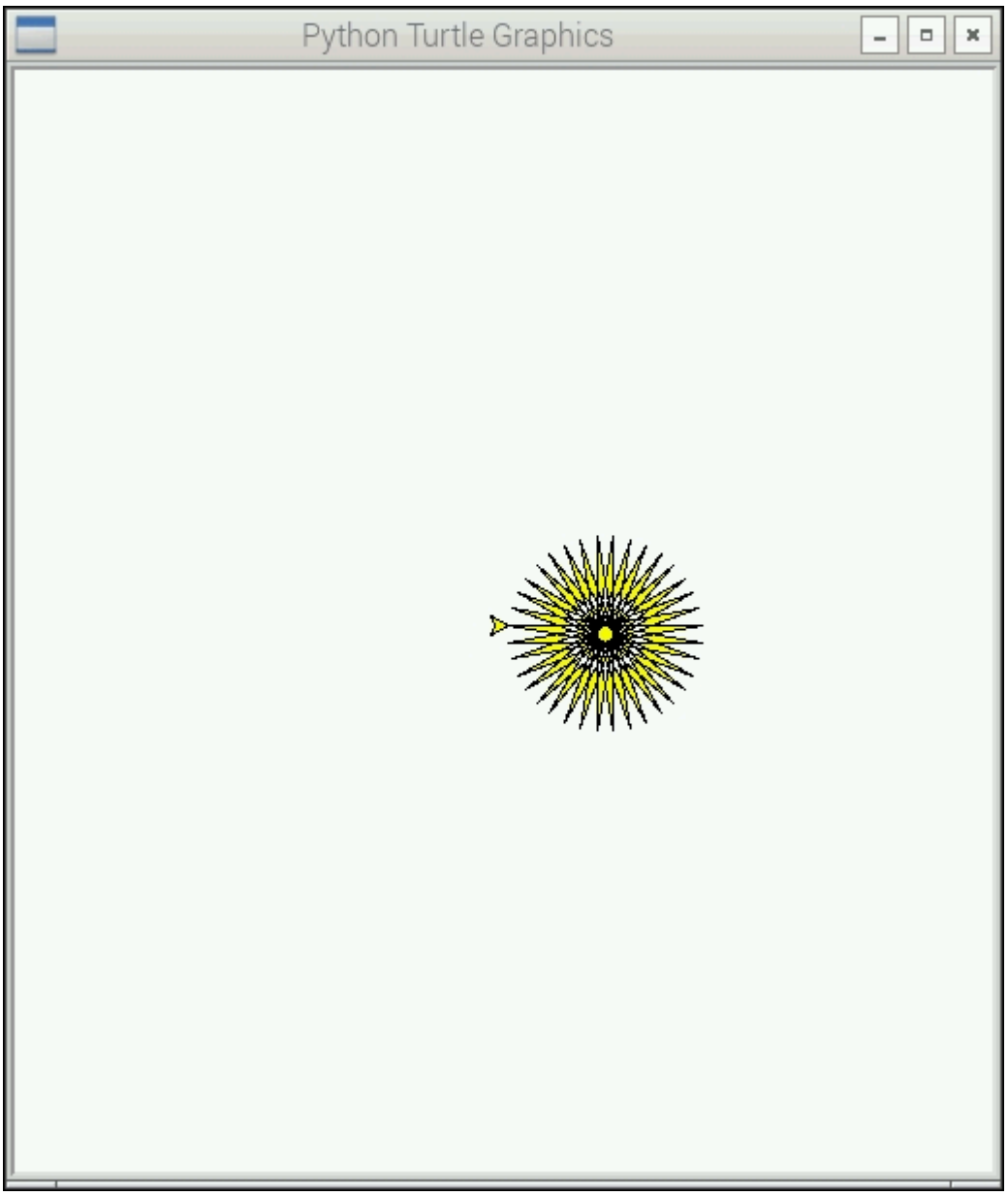
In the preceding code, `t.forward()` is used to move the turtle forward a specified distance, and `t.left()` is used to rotate the turtle left by 190 degrees. `t.pos()` returns the current coordinates of the turtle.

Finally, we use `disp.exitonclick()` to close the current output window when we click on the exit button.

Note

At the start, the turtle cursor is at (*0,0*) and is pointed toward the positive direction of the *x* axis (facing right).

The output of the program will be a cursor drawing the desired shape progressively, and it helps the programmer understand how the program is actually working. The final output of the preceding program is as follows:



Note

Detailed documentation for the turtle API can be found at https://docs.python.org/2/library/turtle.html.

Next, we will learn the concept of recursion. In terms of programming, recursion means calling the same block of code within itself. For a procedural and modular style of programming, this

stands for calling a function or method within itself. Usually, this is done to break a big problem into similar problems with smaller input sizes and then collect the output of all these smaller problems to derive the output of the big problem. One of the best ways to see recursion at work is to visualize it using a turtle. We will now write a program to draw a fractal tree using recursion.

First, we start by importing the required libraries, as follows:

```
import turtle
import random
```

We need the `random` library for the `randint()` function, which returns a random integer in the provided range. This is needed to make our generated tree seem different every time. Then, we will define a function to draw a part of the tree recursively:

```
def fractal_tree(b_len,t):
    if b_len > 5:
        temp=random.randint(1, b_len)
        temp_angle = random.randint(1, 25)
        t.forward(temp)
        t.right(temp_angle)
        fractal_tree(b_len-10,t)
        t.left(2 * temp_angle)
        fractal_tree(b_len-10,t)
        t.right(temp_angle)
        t.backward(temp)
```

In the preceding program, we are calling the same function twice in order to draw the further branches of the tree. If the `b_len` parameter is less than or equal to 5, then it will be a leaf (which means that the function will not be called again); else, the recursion will continue. We are randomizing the angle and length of the movement of the turtle while drawing the branches here; otherwise, the tree will be symmetrical, which is very unlikely in real life. The combination of `t.forward()`, `t.backward()`, `t.left()`, and `t.right()` ensures that at the end of each function call, the turtle cursor is at the same position as where it started.

Finally, we write the routine to call this recursive function:

```
t=turtle.Turtle()
disp=turtle.Screen()
t.left(90)
t.up()
t.backward(100)
t.down()
t.color("green")
fractal_tree(120,t)
disp.exitonclick()
```

The cursor does not draw the movements between the `t.up()` and `t.down()` function calls. In the preceding code, we are moving the cursor downward by 100 positions so that the tree should fit in the turtle graphics window. When we call `fractal_tree()` with `120` as the argument, it takes more than 30 minutes due to the high degree of recursion. The output of the preceding program is as follows:



Summary

In this chapter, we learned about the background of Raspberry Pi and Python. We understood the details of the different models of Pi. We learned how to set up the Raspberry Pi for programming. We also performed some hands-on Python programs. Then, we learned some graphics programming with the turtle library. This got us started with the Pi and the Python programming language, which we will be using throughout the rest of the book.

In the next chapter, we will learn how to play and program `minecraft-pi`.

Chapter 2. Minecraft Pi

In the previous chapter, we learned how to set up the Pi and looked at Python and turtle modules briefly. In this chapter, we are going to explore the Pi version of a popular computer game, *Minecraft*. We will also learn how to program Minecraft with Python as the programming interface and use it to script user actions and build wonderful things.

We will cover the following topics in this chapter:

- Introducing Minecraft Pi
- Playing Minecraft Pi
- Programming Minecraft Pi with Python

Introduction to Minecraft Pi

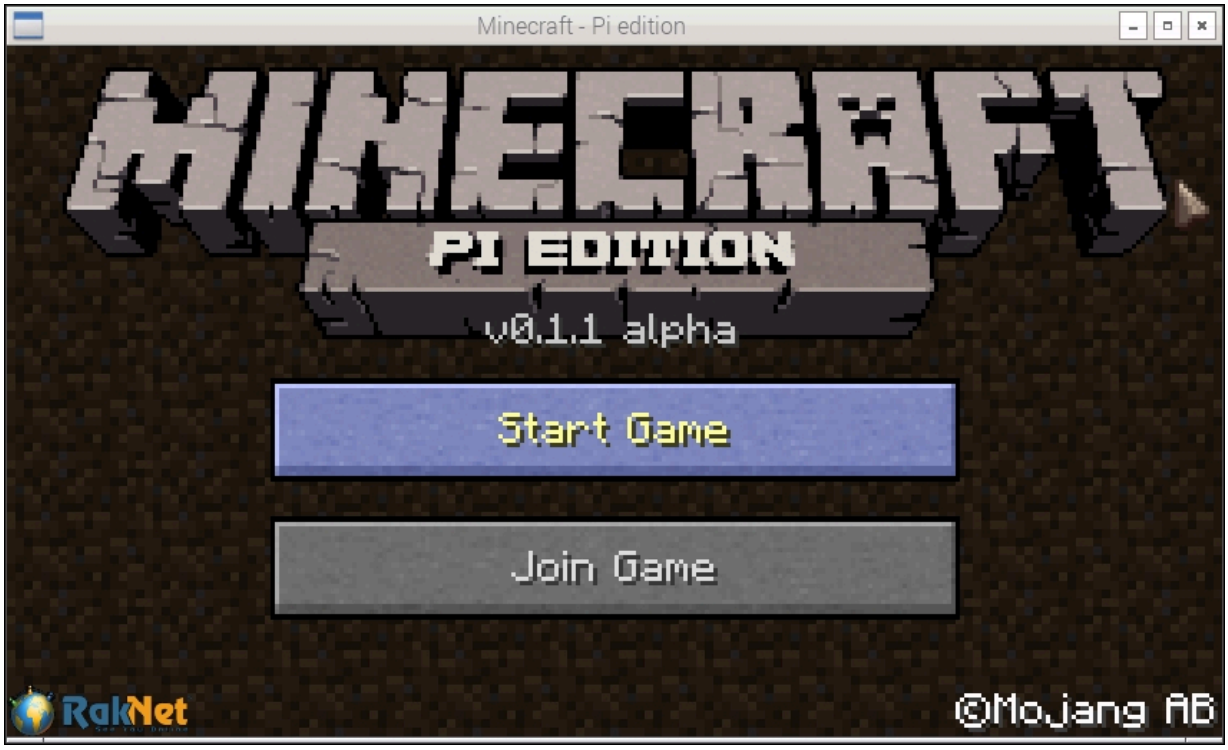
Minecraft is a very popular open world game. Like all other open world games, the player can freely explore the virtual world in a Minecraft Pi game. Minecraft was created by Markus Persson, and the game was later developed and published by the company Mojang, which is a Microsoft Game Studios subsidiary now. Jens Bergensten is the current lead designer and lead developer for the game. The alpha version of the game was released for PCs in 2009, and the complete version was released in 2011. The game is available for various platforms, which include PCs, Linux, Mac OS, iOS, Android, PlayStation, Xbox, and Raspberry Pi.

> **Note**
>
> You can get more information on Minecraft at https://minecraft.net/.

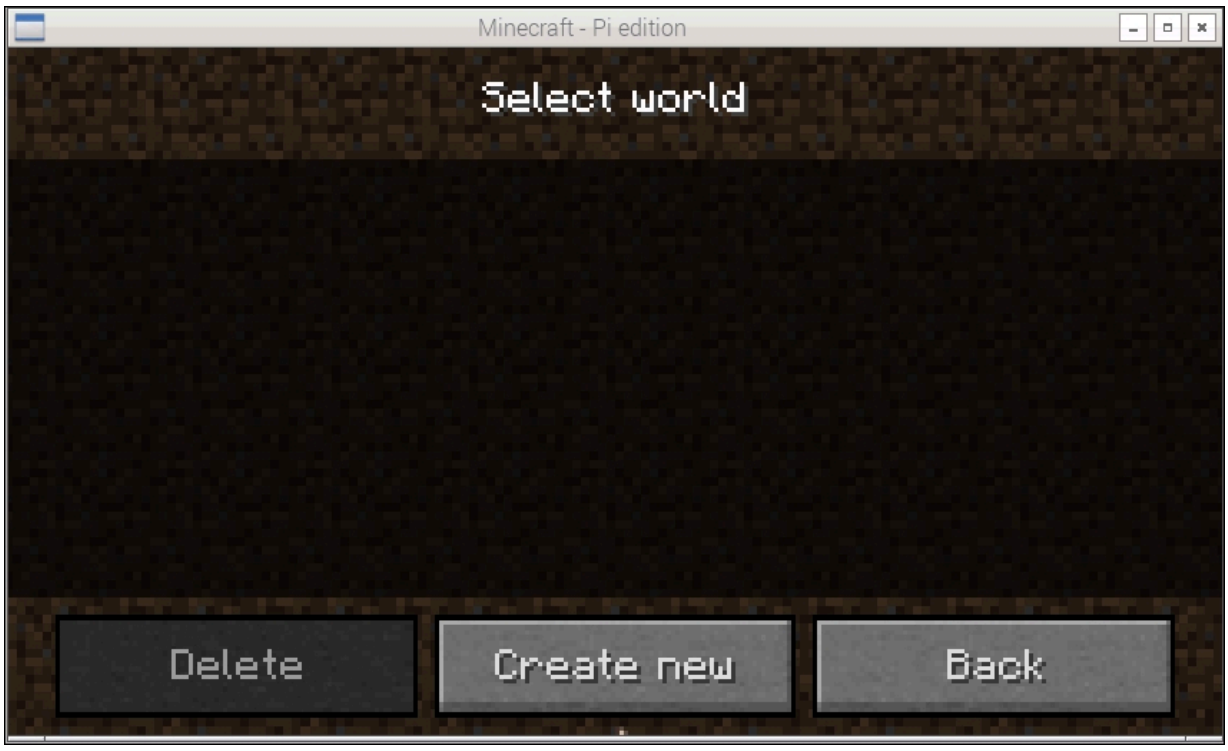
The version of this game for the Raspberry Pi is known as Minecraft Pi, and it was released in 2013. It was developed by Aron Nieminen and Daniel Frisk. Minecraft Pi focuses on the creative and building aspect of the game and does not include gathering resources and combat.

Playing Minecraft Pi

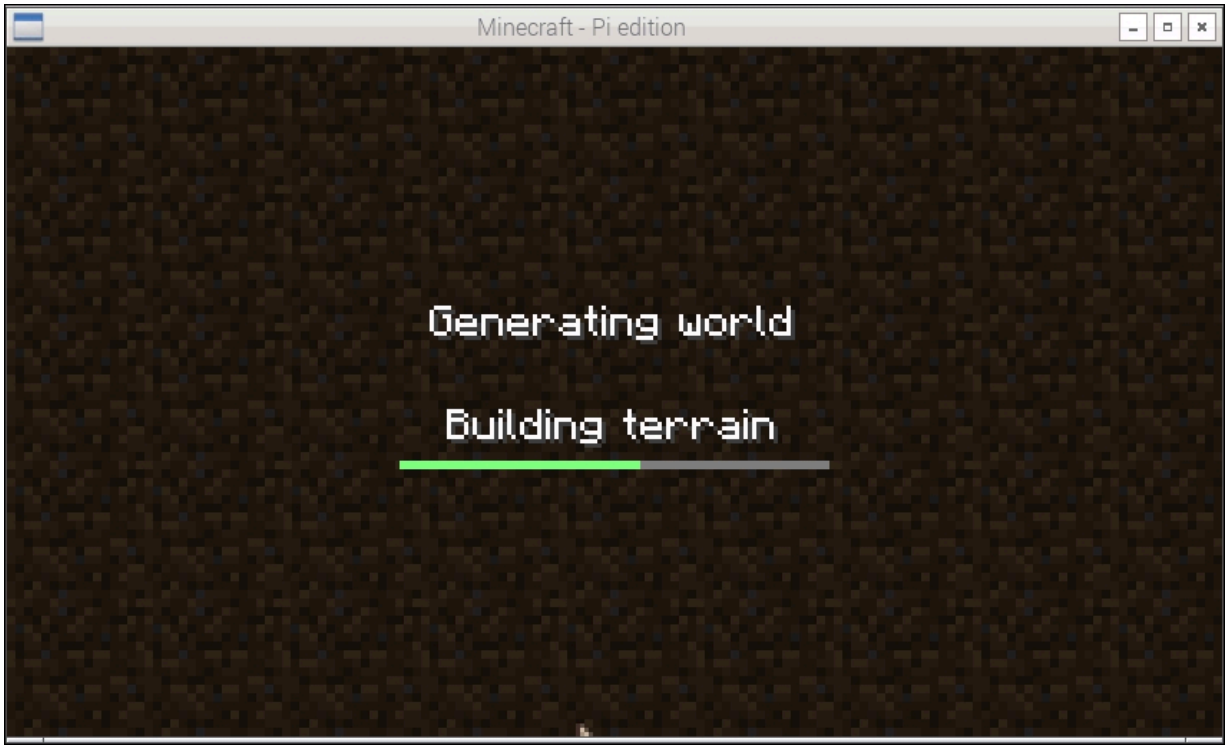
Minecraft Pi comes preinstalled in the latest version of Raspbian. So, there is no need for an additional installation. Minecraft Pi can be found by navigating to **Menu** | **Games**. Alternatively, we can start it by typing `minecraft-pi` in `lxterminal`. The following screen will appear once we start the game:



Click on the **Start Game** button. Then, the following window will appear:



Click on **Create New** and it will start generating a new world for the gameplay:



Once the new world for the gameplay is generated, the player character is placed in the virtual world. The default view in the game is the first-person view. It will look as follows:



Note

In Minecraft Pi, a new world is randomly generated in a procedural manner. This means that the world is randomly created with the algorithm rather than using predetermined components. So, no two worlds in Minecraft Pi will be the same. You can learn more about procedural generation from https://en.wikipedia.org/wiki/Procedural_generation.

We can switch from the first-person view to the third-person view by pressing the *Esc* key or by clicking on the following button for a view change:



Once this button is clicked on, it will change to the following button:



At this point, once we return to the game by pressing the *Esc* key or clicking on the **Back to game** button, we will see our Minecraft Pi character in the third person, as follows:



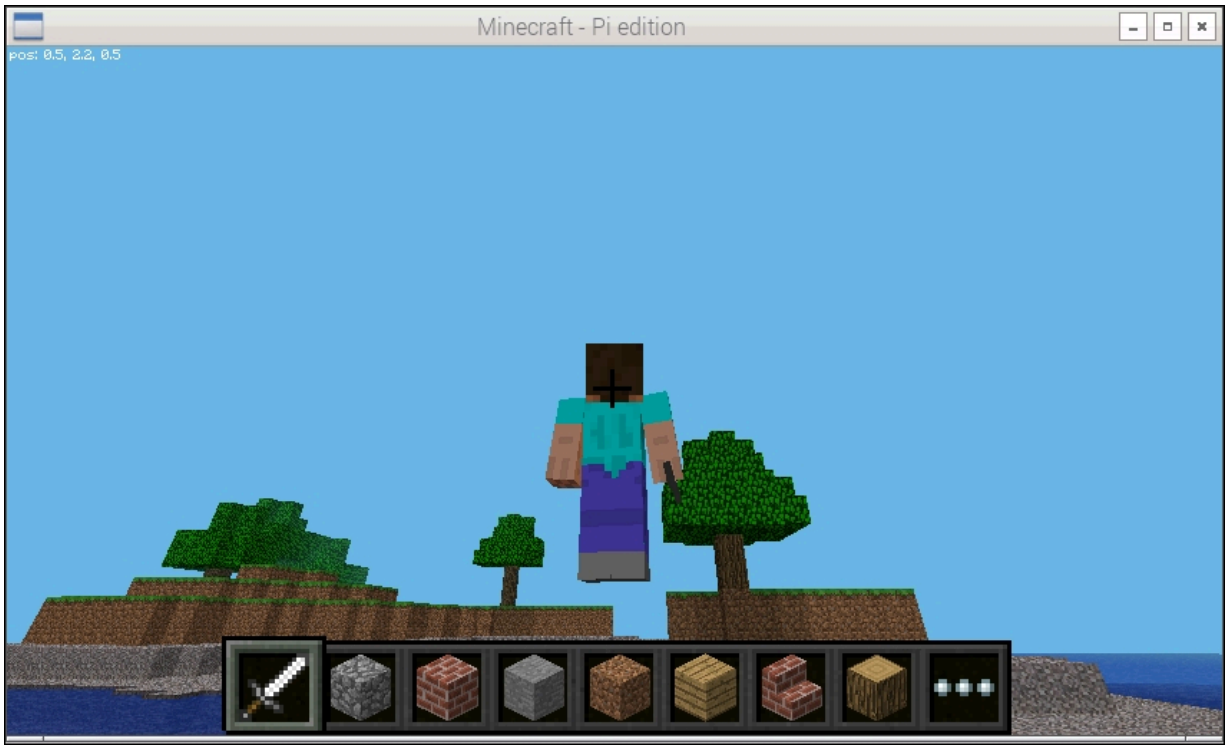
We can change this view to the first person view again by following the preceding steps.

Movement control in Minecraft Pi

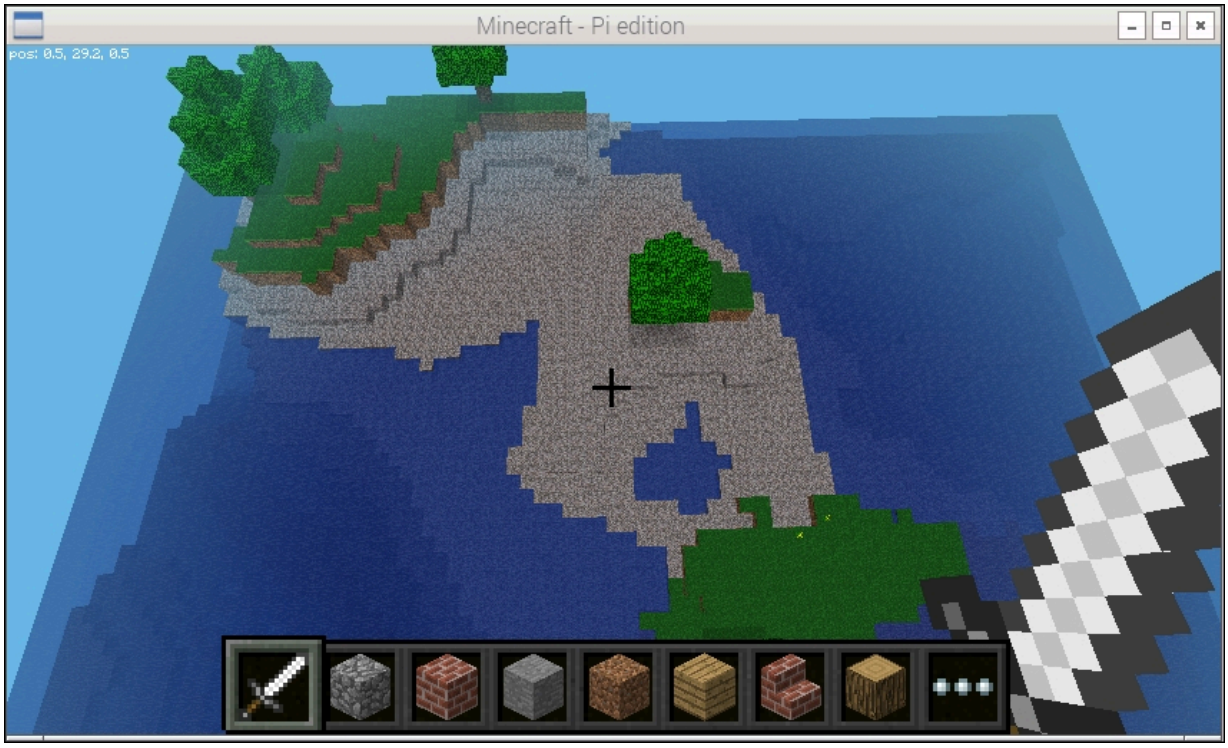
We can use a mouse to look around and also make use of the following keys for the movement:

Key	**Action**
W	Forward
S	Backward
A	Left
D	Right
Spacebar	Jump
Double spacebar	Fly or fall

Like most first/third-person games, movement is controlled by *WSAD* keys on the keyboard. The character jumps if spacebar is pressed once. If we hit spacebar twice, the character is lifted in the air, as shown in the following screenshot:



At this point, *WSAD* keys can be used to fly in the air. We can increase the altitude by pressing spacebar while flying. The following screenshot shows the game world view from a higher altitude:



If spacebar is pressed twice while flying, then the character will fall on the ground. While the character is falling, if we press spacebar twice, it will stop falling.

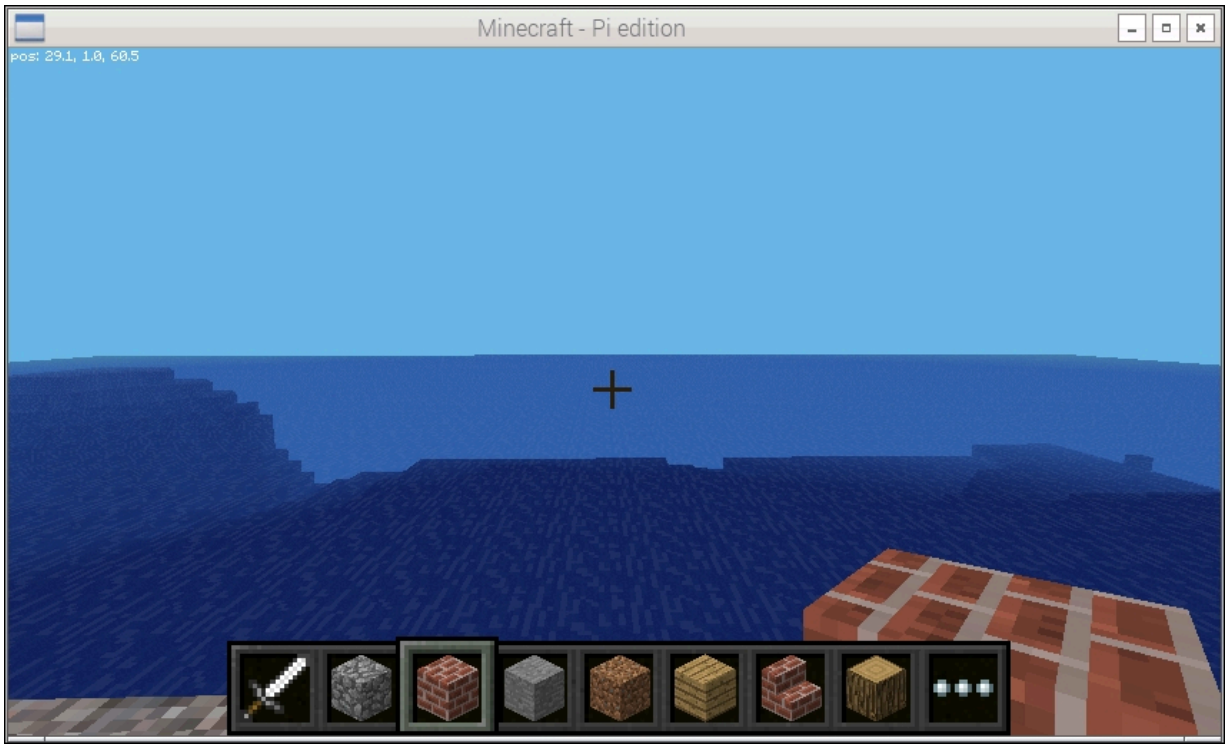
Action control in Minecraft Pi

By default, the player character starts with a sword in hand. If we right-click with the sword in hand, it will remove the block in front of the character.

There is a quick draw panel at the bottom of the screen, as follows:



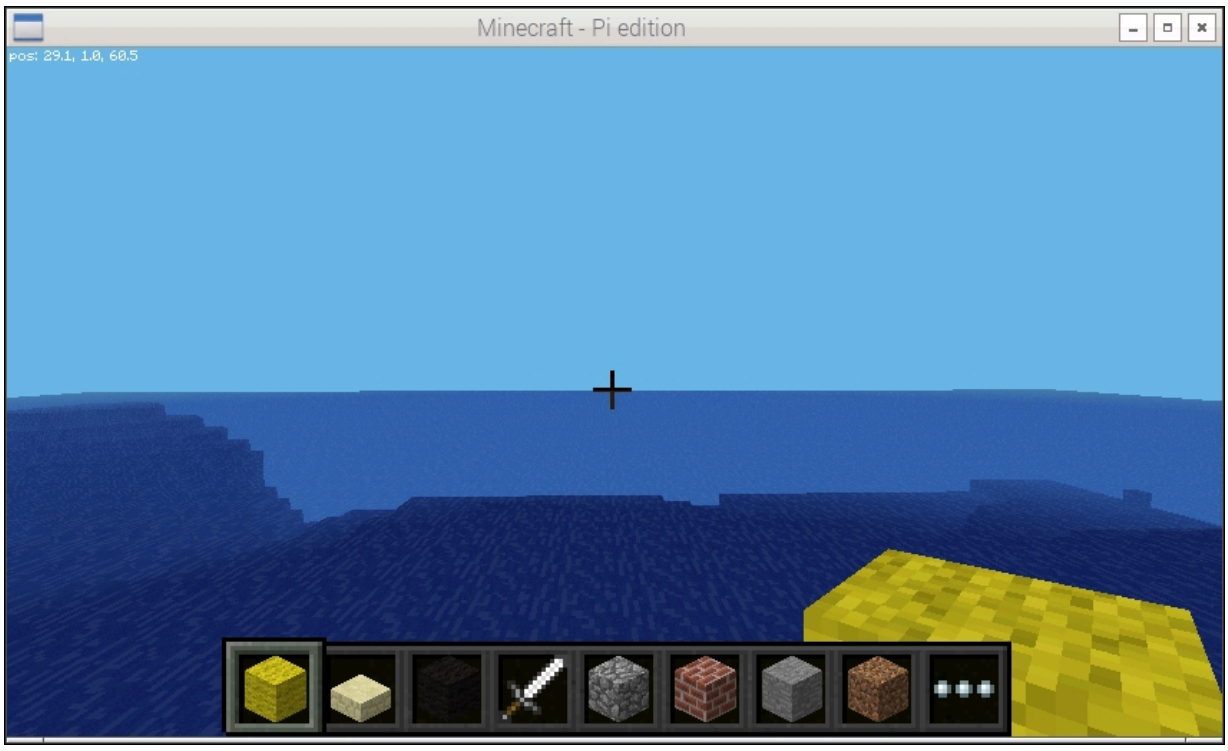
Any item can be selected by scrolling the mouse wheel or using the numbers on the keyboard. The quick draw panel always hold eight items, and the current item in the hand is highlighted as follows:



More items can be accessed by pressing the *E* key. This will open the inventory, as follows:



Items in the inventory can be navigated by *WSAD* keys. While the inventory is open, if the *Esc* key is pressed, it will cancel the inventory without choosing an item. Items in the inventory can be chosen by clicking on the item, and the item will be added to the quick draw panel as well as the character's hand, as follows:



With a block in the character's hand, right-clicking will place the block in front of the character and left-clicking will remove it. The number of blocks available to the player character is infinite.

Minecraft Pi features a creative mode, so the character does not take environmental damage (for example, when the character falls, it does not die) and is not affected by hunger. This game mode helps players focus on building and creating large projects.

Unlike other editions of Minecraft, Minecraft Pi does not feature any other gameplay modes.

Other controls in Minecraft Pi

When a game is running, pressing the *Tab* key will take the focus away from the game, and the mouse cursor will be freed to enable interaction with other windows on the desktop. Pressing *Esc* will pause the game and will take us to the game menu, where we can toggle views (as seen earlier), enable/disable sound, and quit the main window.

Note

Now, with all this information on the movements and the actions in the Minecraft Pi virtual game world, try to create a few things before continuing with the remainder of this chapter.

Python programming for Minecraft Pi

Minecraft Pi comes with the Python programming interface. This means that it's possible to script the action of a character in the game using Python. In this section, we will learn how to use Python programming to script the character actions as well as create amazing effects in the Minecraft world.

Start Minecraft Pi and create the world. Once it's done, while the game is running, free the mouse by pressing the *Tab* key and opening `lxterminal`. Create the `/home/pi/book/chapter02` directory. Navigate to the directory and write the following code in the `prog1.py` file:

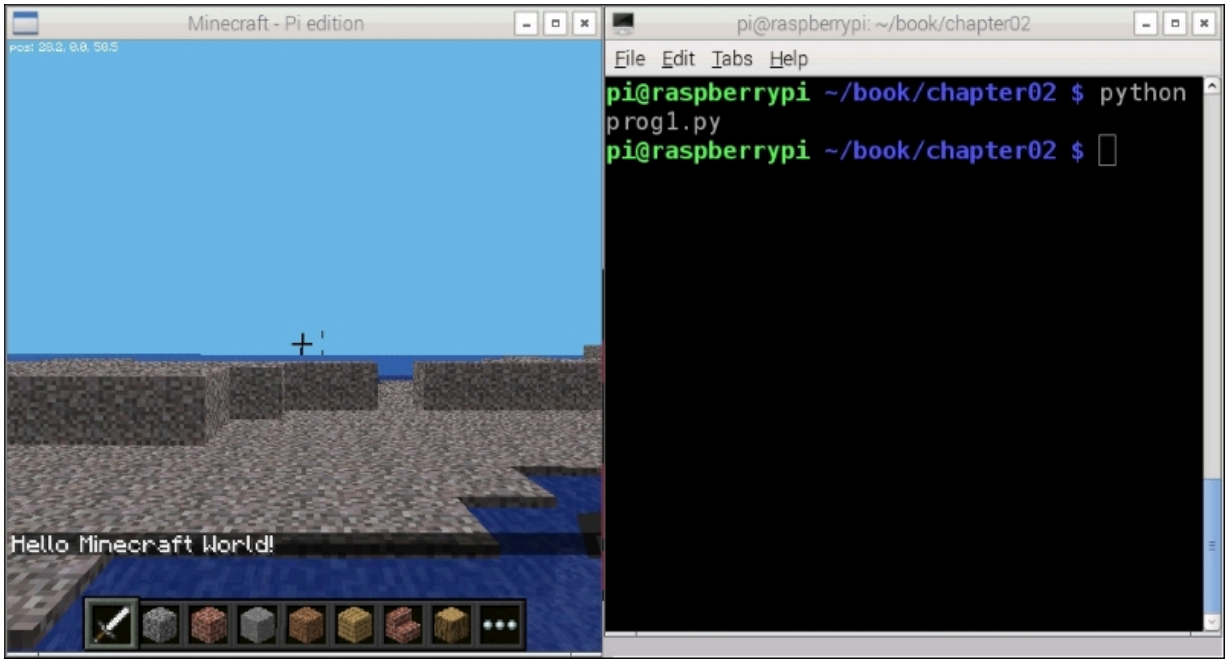
```
import mcpi.minecraft as minecraft

mc = minecraft.Minecraft.create()

mc.postToChat("Hello Minecraft World!")
```

The preceding program first imports the Minecraft Pi Python API. The second statement creates the connection to Minecraft, and the third statement posts the message to the game chat.

Run the preceding code; the following is its output:



Let's take a look at some of the most important functions of the Minecraft Pi Python API.

We can see the player's current coordinates in the game window in the top-left corner of the window. We can retrieve these coordinates with `getPos()`. The following is the code to retrieve the player character's current coordinates and print them:

```
import mcpi.minecraft as minecraft
mc = minecraft.Minecraft.create()
cur_pos = mc.player.getPos()
print cur_pos.x ; print cur_pos.y ; print
cur_pos.z
```

This will print the current coordinates to the screen. Alternatively, the following syntax can be used to achieve this:

```
import mcpi.minecraft as minecraft
mc = minecraft.Minecraft.create()
cur_x , cur_y , cur_z = mc.player.getPos()
print cur_x ; print cur_y ; print cur_z
```

The `setPos()` function can be used to set the character's position. We need to pass the desired coordinates of the character to this function, as follows:

```
import mcpi.minecraft as minecraft
mc = minecraft.Minecraft.create()
cur_x , cur_y , cur_z = mc.player.getPos()
mc.player.setPos(cur_x+10, cur_y, cur_z)
```

The preceding code will displace the character by 10 blocks in the *x* axis.

The following code will set the character position 100 blocks higher than the current position, and as a result, the character will start falling:

```
mc.player.setPos(cur_x, cur_y + 100 , cur_z)
```

We can place the blocks of our choice with the `setBlock()` function. We need to pass the coordinates of the block (the first three arguments) and the type of the block (the fourth argument) we need to set to this function as follows:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
mc = minecraft.Minecraft.create()
cur_x , cur_y , cur_z = mc.player.getPos()
mc.setBlock(cur_x + 1 , cur_y , cur_z ,
block.ICE.id )
```

You will find an ice block placed in front of you. If you do not find the block, then try to look around; it will be just beside or behind you.

Some blocks (such as wool and wood) have additional properties. We can pass this additional property as the fifth argument to the function for the wool and wood block types. The following code creates a column of a wool block with all the possible colors:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
mc = minecraft.Minecraft.create()
cur_x , cur_y , cur_z = mc.player.getPos()

for i in range (0 , 15):
  mc.setBlock(cur_x + 1 , cur_y + i , cur_z ,
block.WOOL.id, i )
```

Run the preceding code and check the output for yourself.

We can use `setBlocks()` to place multiple blocks for a given volume. The following example places multiple gold blocks:

```
import mcpi.minecraft as minecraft
import mcpi.block as block
mc = minecraft.Minecraft.create()
cur_x , cur_y , cur_z = mc.player.getPos()

mc.setBlocks(cur_x + 1 , cur_y + 1 , cur_z + 1 ,
cur_x + 6 , cur_y + 6 , cur_z  + 6 ,
block.GOLD_BLOCK.id )
```

We can also use mathematical equations to draw geometric shapes in the game world. Now we know that `setBlock()` is used to place a single block. We can use this function in a single `for` loop to create a line of blocks. Calling this function in a double `for` loop will set a two-dimensional geometric shape. We can further extend this by introducing one more `for` loop. This will create a three-dimensional shape. The following code places a golden sphere near the player's position. We will place the gold block in the positions where the coordinates satisfy the equation of the sphere with a radius of 10 blocks:

```
import mcpi.minecraft as minecraft
import mcpi.block as block

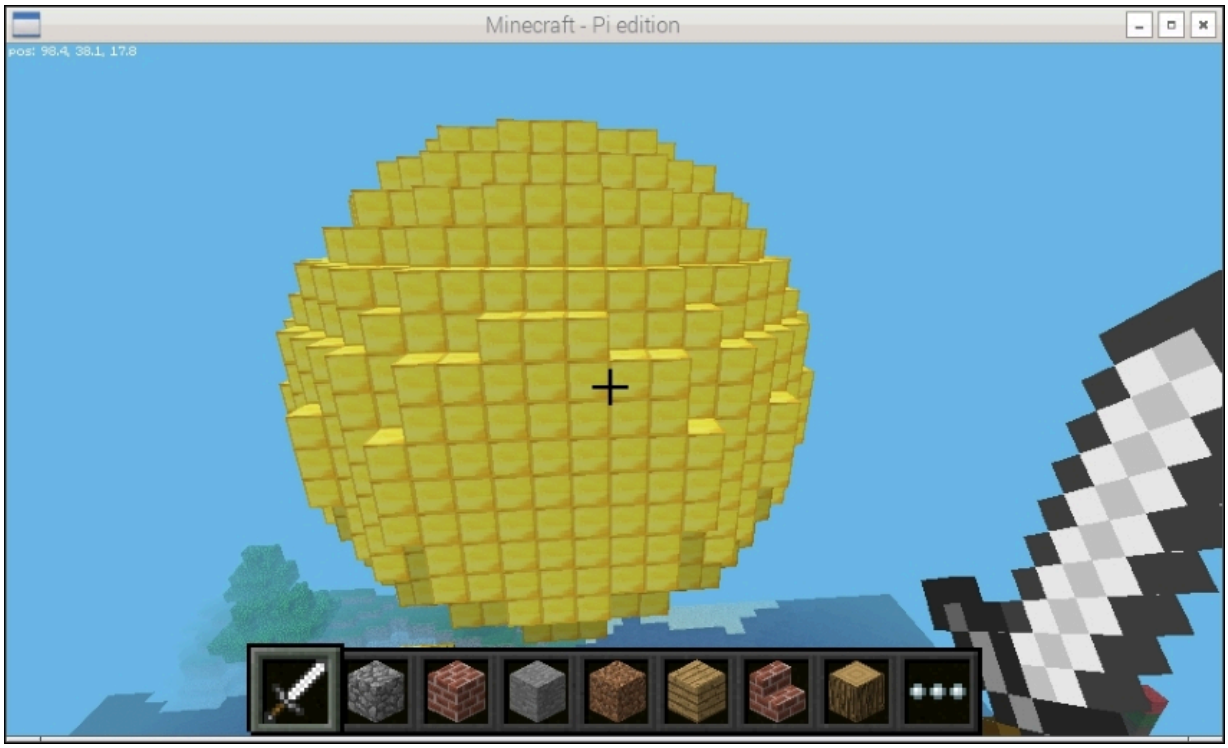
mc = minecraft.Minecraft.create()

r = 10
```

```
cur_x , cur_y , cur_z = mc.player.getPos()

for x in range(r*-1,r):
  for y in range(r*-1, r):
    for z in range(r*-1,r):
      if x**2 + y**2 + z**2 < r**2:
        mc.setBlock(cur_x + x, cur_y + ( y + 20 )
, cur_z - ( z + 20 ) , block.GOLD_BLOCK)
```

The following is the output of the preceding program:



The following code places a gold block below the player's current position until the execution of the code is terminated by pressing *Ctrl + C*:

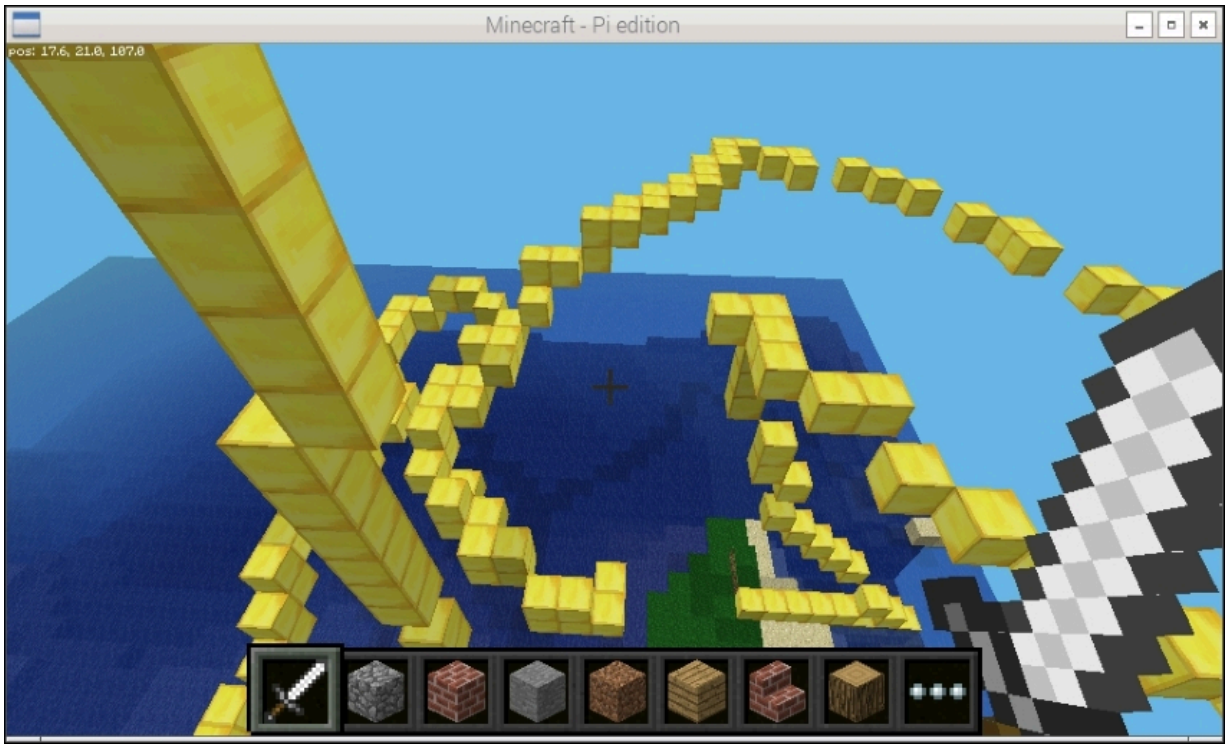
```
import mcpi.minecraft as minecraft
import mcpi.block as block
import time

mc = minecraft.Minecraft.create()

while 1:
```

```
  cur_x, cur_y , cur_z = mc.player.getPos()
  mc.setBlock(cur_x,cur_y-
1,cur_z,block.GOLD_BLOCK.id)
  time.sleep(0.1)
```

The following is the output of the preceding code:



Now, we will conclude the chapter with some explosions. We can place a TNT block near us. It can be activated by left-clicking while holding the sword in hand. It will explode in a few seconds. The explosion will destroy some blocks in the blast radius:

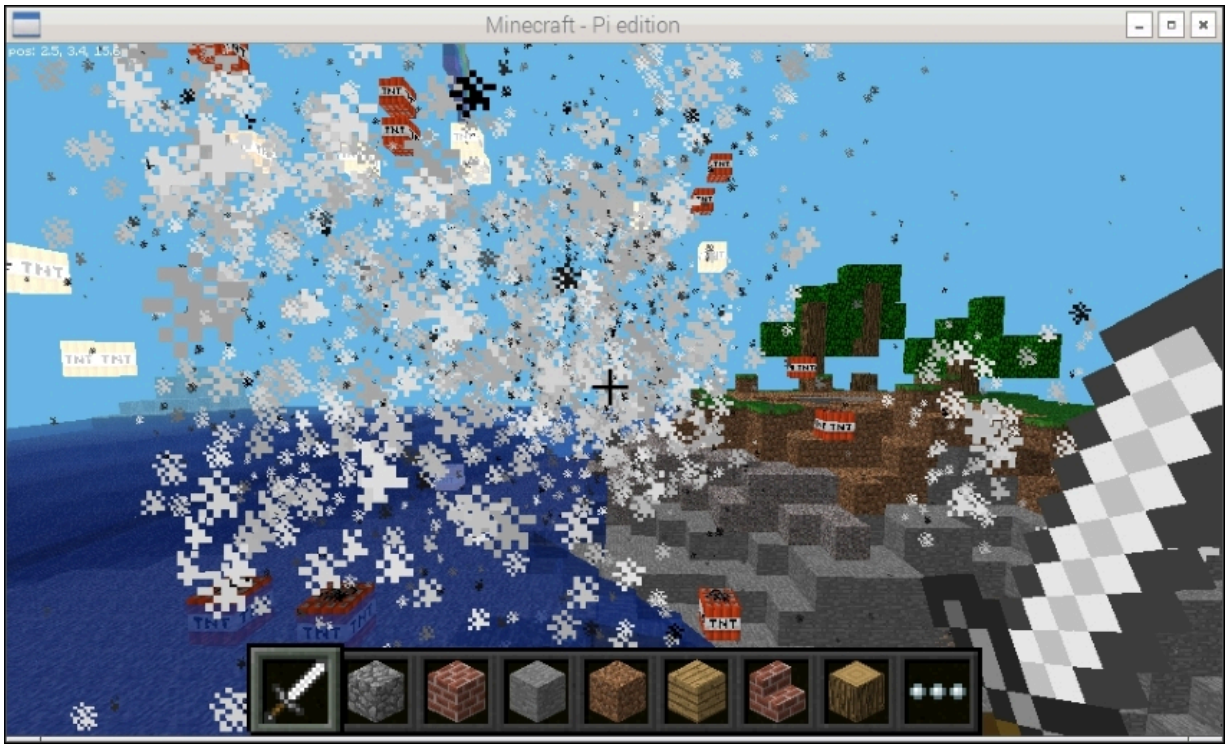
```
cur_x, cur_y , cur_z = mc.player.getPos()
mc.setBlocks(cur_x+1,cur_y,cur_z, block.TNT.id,1)
This can be made more spectacular by placing
multiple TNT blocks, as follows:
import mcpi.minecraft as minecraft
import mcpi.block as block

mc = minecraft.Minecraft.create()

cur_x, cur_y , cur_z = mc.player.getPos()
```

```
mc.setBlocks(cur_x+1,cur_y,cur_z,cur_x+4,cur_y+3,c
ur_z+3,block.TNT.id,1)
```

Once blocks are placed, activate the TNT block by the sword. The following is the screenshot of an explosion after activating the TNT blocks placed by the preceding code:



Note

The detailed Minecraft Pi Python API can be found at http://www.stuffaboutcode.com/p/minecraft-api-reference.html.

Summary

In this chapter, we got ourselves familiarized with the Minecraft Pi gameplay and learned how to program it using the Python API. We learned how to place and remove a block. Also, we implemented the code required to create some wonderful geometric shapes. In the end, we learned how to activate and explode TNT blocks to blow the large areas in the in-game world.

In the next chapter, we will learn how to use the Pygame library in order to create a game for the Pi.

Chapter 3. Building Games with PyGame

In the previous chapter, we learned how to get started with Minecraft on Raspberry Pi and how to play it and use Python to manipulate things. Continuing with the same theme, we will now take a look at a gaming library in Python called PyGame and also learn how to create simple games with it. In this chapter, we will go through the following topics:

- Introducing PyGame
- Drawing a fractal tree
- Building a simple snake game

Introducing PyGame

PyGame is a set of Python modules designed for the writing of video games. It is built on top of the existing **Simple DirectMedia** (**SDL**) library, and it works with multiple backends, such as OpenGL, DirectX, X11, and so on. It was built with the intention of making game programming easier and faster without getting into the low-level C code that was traditionally used to achieve good real-time performance. It is also very flexible and comes with many operating systems. It is very fast as it can use multiple core CPUs very easily and also use optimized C and assembly code for core functions.

PyGame was built to replace PySDL after its development was discontinued. Originally written by Pete Shinners, it is a community project from 2004 and is released under the open source free software GNU's lesser general public license. Since it is very simple to use and is open source, it has a lot of members in the international community and so it enjoys access to a lot of resources that other libraries may lack. There are many tutorials that can build different games with PyGame. It contains the following modules

Module	The description
cdrom	Manages CD-ROM devices and audio playback
cursors	Loads cursor images and includes standard cursors
display	Controls the display window or screen
draw	Draws simple shapes onto a surface
event	Manages events and the event queue
font	Creates and renders Truetype fonts
image	Saves and loads images
joystick	Manages joystick devices
key	Manages the keyboard
mouse	Manages the mouse
movie	Used for the playback of MPEG movies

Module	The description
sndarray	Manipulates sounds with Numeric
surfarray	Manipulates images with Numeric
time	Controls timing
transform	Scales, rotates, and flips images

Note

For more information on the PyGame library, you can visit www.pygame.org.

Installing PyGame

PyGame usually comes installed with the latest Raspbian distribution, but if it isn't you can use the following command to install it:

```
sudo apt-get install python-pygame
```

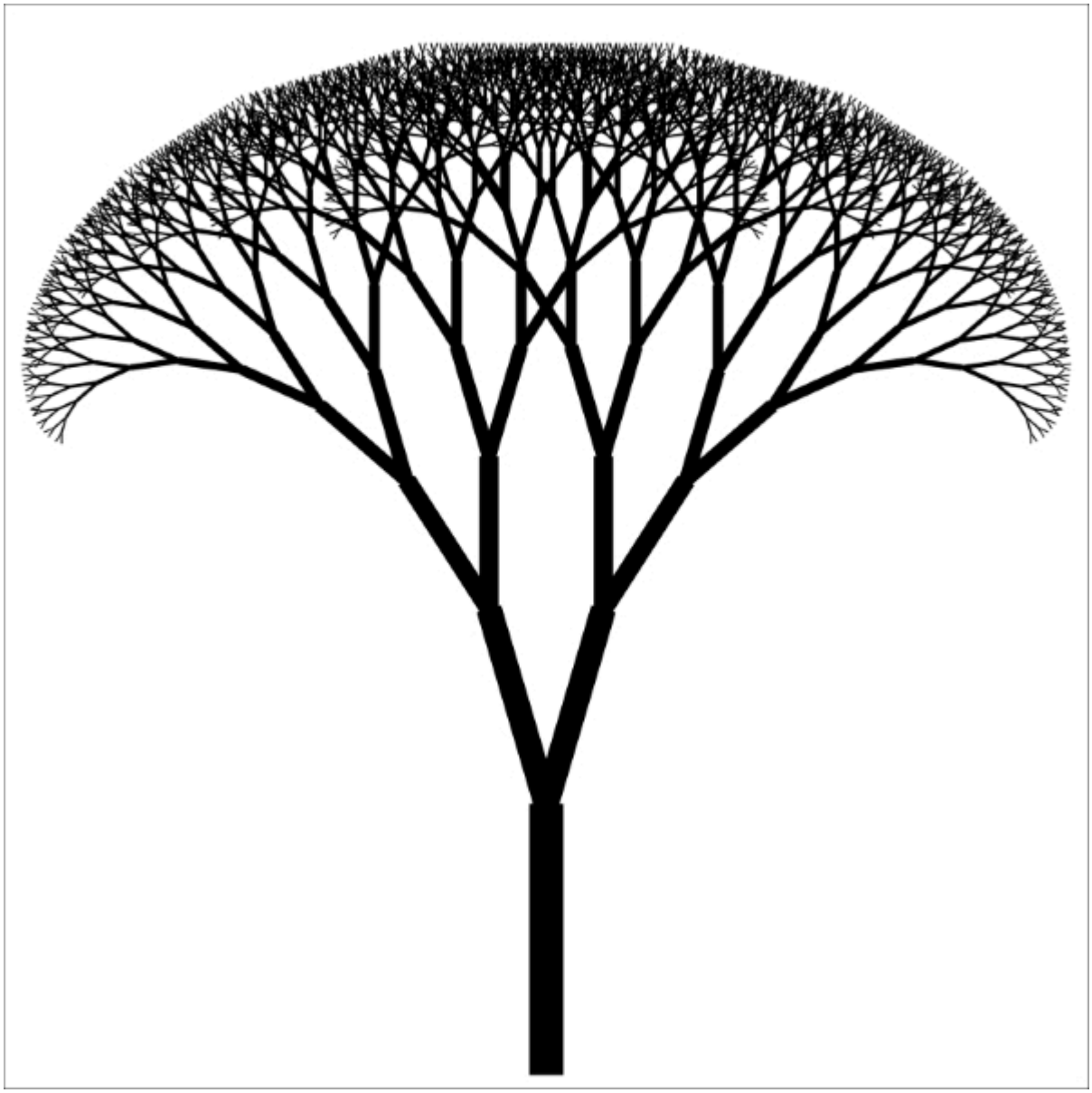
Test your installation by opening a Python terminal by entering `python` in a regular terminal and pressing *Enter*. Now, execute the following command:

```
import pygame
```

Now that you have your system set up and you have hopefully checked out the PyGame website to explore its complete functionalities, we will move on to build the binary fractal tree to introduce you to the workings of PyGame. Let's begin!

Drawing a binary fractal tree

A binary fractal tree is defined recursively by binary branching. Typically, it consists of a trunk of length 1, which splits into two branches of decreasing or equal length, each of which makes an angle Q with the direction of the trunk. Furthermore, both of these branches are divided into two branches, each making an angle Q with the direction of its parent branch, and so on. Continuing in this way, we can infinitely make branches, and the collective diagram is called a **fractal tree**. The following diagram visually shows what such a fractal tree might look like:



Now, let's move on to the code and take a look at how such a fractal tree can be constructed with PyGame. Following this paragraph is the complete code, and we will go through it statement by statement in further paragraphs:

```
import pygame
import math
import random
import time

width = 800
```

```
height = 600

pygame.init()
window = pygame.display.set_mode((width, height))
pygame.display.set_caption("Fractal Tree")
screen = pygame.display.get_surface()

def Fractal_Tree(x1, y1, theta, depth):
    if depth:
        rand_length=random.randint(1,10)
        rand_angle=random.randint(10,20)
        x2 = x1 +
int(math.cos(math.radians(theta)) * depth *
rand_length)
    y2 = y1 + int(math.sin(math.radians(theta)) *
depth * rand_length)
        if ( depth < 5 ):
            clr = ( 0 , 255 , 0 )
        else:
            clr = ( 255, 255 , 255 )
    pygame.draw.line(screen, clr , (x1, y1), (x2,
y2), 2)
    Fractal_Tree(x2, y2, theta - rand_angle, depth
- 1)
    Fractal_Tree(x2, y2, theta + rand_angle, depth
- 1)

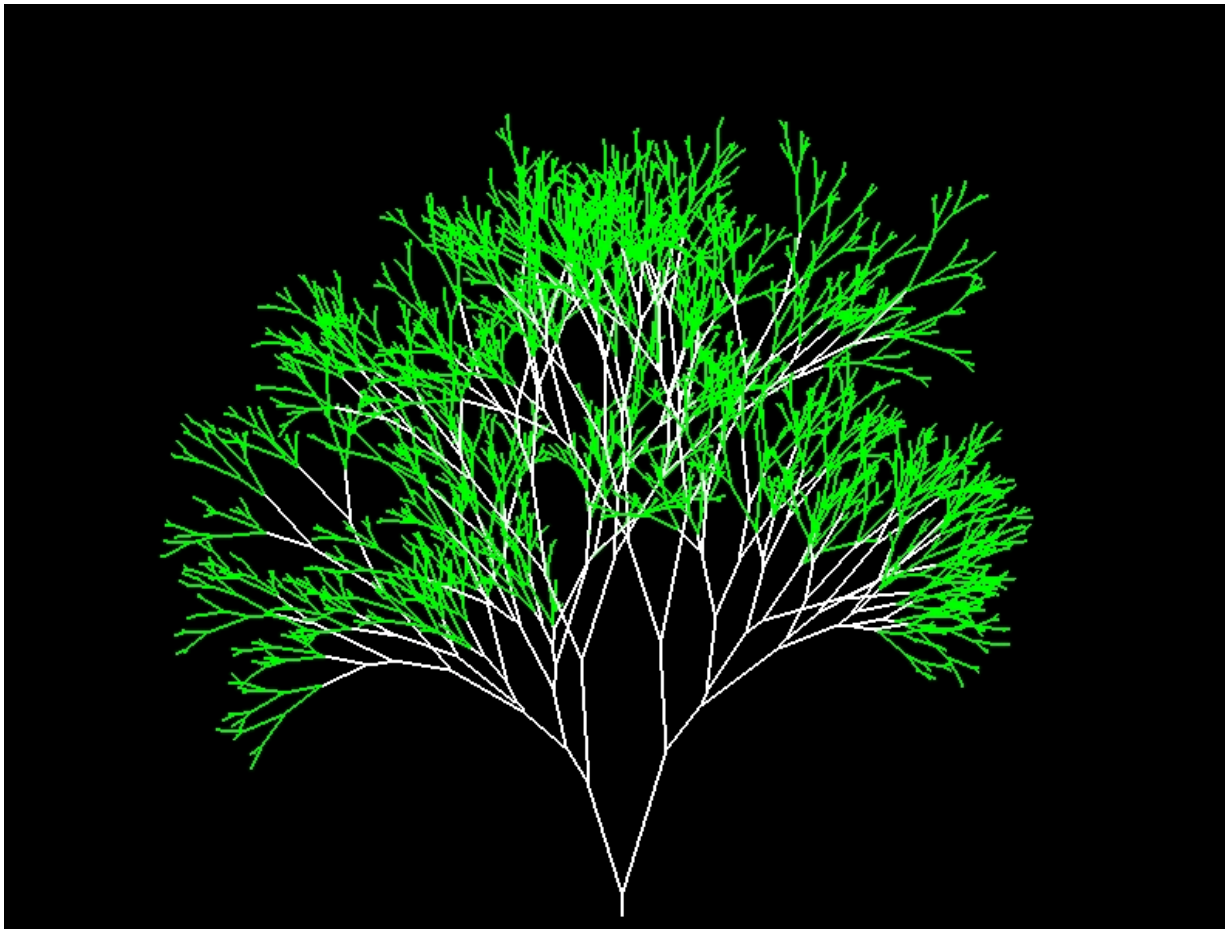
Fractal_Tree( (width/2), (height-10) , -90, 12)
pygame.display.flip()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT or event.type
== pygame.KEYDOWN:
            pygame.quit()
            exit(0)
```

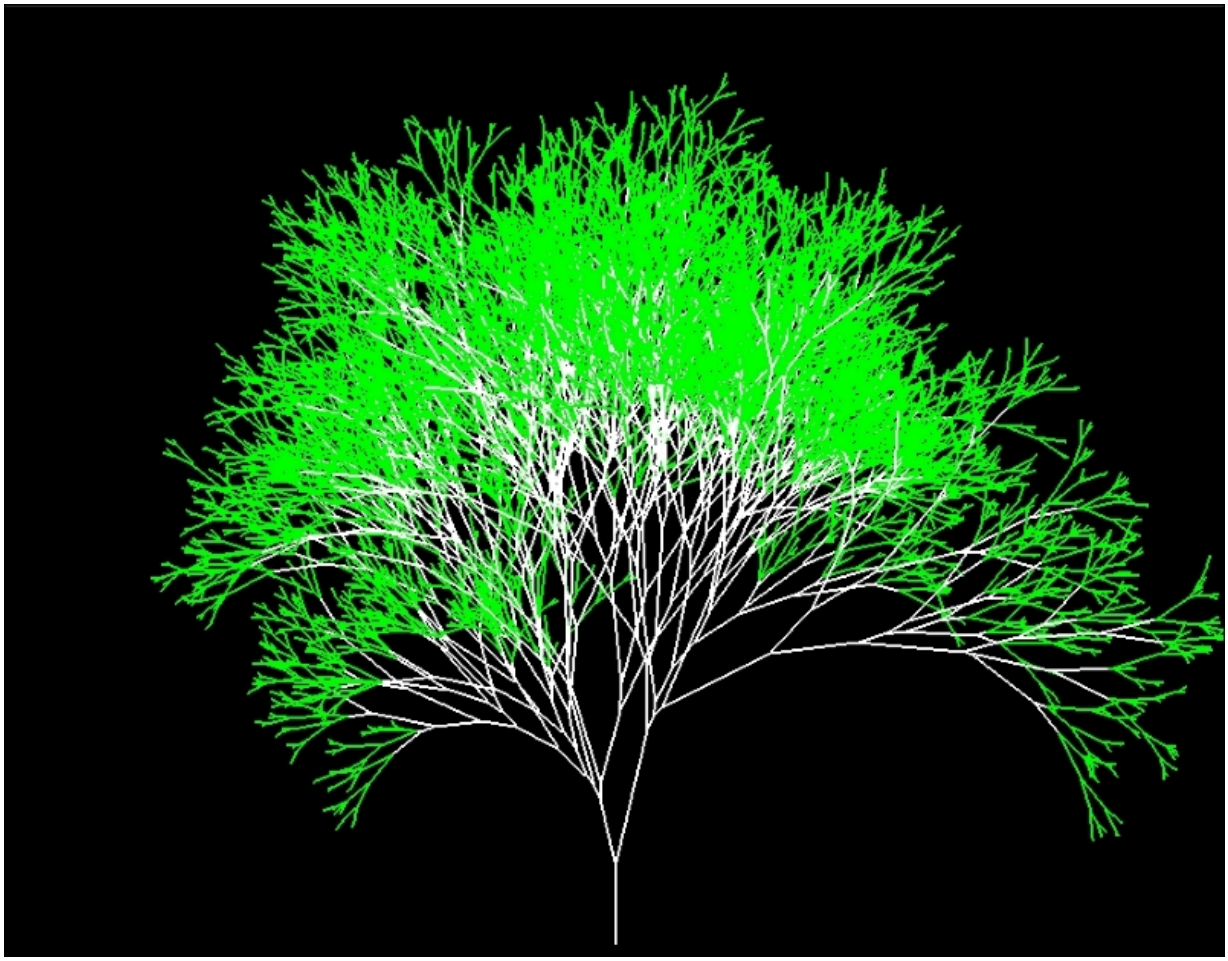
Save the preceding code as `prog1.py` and then run the following:

```
python prog1.py
```

You will now get the following output:



If you increase the depth by 2 with a slight increase in the canvas area, the fractal tree now looks like this:



The new tree looks considerably denser and more branched out than the original tree.

Now, since you know what the output looks like, let's grab our magnifying glasses and sift through the program to understand how it works!

The first four lines are there to satisfy the dependencies required to build the program. These are PyGame, a math library, a library to generate random numbers, and a library to keep track of time for delay functions. Next, we specify the dimensions for the screen space required for our program.

`The pygame.init()` method initializes all the modules that were loaded as part of importing pygame in the first statement. It is required to be executed if you want to be able use any functionality

of PyGame. The `display.set_mode()` method creates a new `Surface` object, which represents the area on the screen that is visible to the user. It takes a tuple consisting of the dimensions of the window as the argument: the `width` and `height` variables in this case. It is literally the canvas on which you can draw. Anything you do to this object will be shown to the user. Images and other objects are represented as PyGame objects, and you can overlay them on the the main surface. Then, we set the title of the window using the `caption()` method, and finally, the `screen` variable actually gets the object that the display is stored in. So now, our canvas is stored in the `screen` variable, and we will use it to make any changes to the canvas.

The `Fractal_Tree` function should be fairly easy to understand. It takes four arguments: the starting point of the branch (`X1`, `Y1`), the angle of the branch with respect to the positive *x* axis—which is a horizontal line going to your right when you look at the computer screen, and the depth of the fractal tree (which indicates the levels in the tree). It is called for the first time in the 28th line with appropriate arguments. You might notice that towards the end of the function, it calls itself using different arguments. These kinds of function are called recursive functions and are very useful in tasks where there is repetition and where the same task needs to be performed with minor differences.

If the depth is positive, it selects a random length and angle for the next branch by selecting a random integer from the `randint()` method of the random package. It then specifies the coordinates of the end of that branch, which are labeled `X2` and `Y2`. If the depth is less that `5`, it selects the color of the line as green; otherwise, it is white. Here, it is important to understand that the color is represented in the RGB format. Hence, (`0`, `255`, `0`) means green. Similarly, (`255`, `0`, `0`) will be red. The three numbers in the tuple represent the intensity of RGB colors; hence, we can select any color using a mixture of these three intensities.

Finally, there is a recursive call to itself (`Fractal_tree`), and the program then draws the second level of the fractal tree and so on until the depth becomes zero. There are two recursive calls: one to draw the left branch and the other to draw the right branch. If you've noticed, the function isn't actually executed until the 28th line. And once it is executed, the complete pattern is drawn at once due to the recursiveness of the function, but it still isn't displayed. The next line, `pygame.display.flip()`, is responsible for displaying the drawn shapes on screen:

```
while True:
    for event in pygame.event.get():
If event.type == pygame.QUIT or event.type ==
pygame.KEYDOWN:
            pygame.quit()
            exit(0)
```

This block of code is there to ensure that PyGame quits properly and specifies how to shut down the program. The `event.get()`method clears the event queue so that you always get the last event that occurred. An event queue consists of all the key presses and mouse clicks that happen, and they are stored in a **Last In First Out** (**LIFO**) fashion. You will see this `while` loop in almost every PyGame program as it handles the exit of the program properly. If you are using IDLE, then not shutting down PyGame properly can cause it to hang. In this case, PyGame quits when `pygame.quit()` is executed. Finally, with `exit(0)`, Python also quits and closes the application.

As we are randomizing the length and the angle every time the function calls itself, no two branches will be exactly the same, giving it the appearance of the irregularity of real-life trees. By induction, no two trees will be same.

You can modify parts of this code to see for yourself how the tree behavior changes on changing a few variables, such as `theta` and `depth`. Now that we have completed all the basics of PyGame and can implement fairly complex problems, we are now ready to move on to a real challenge: making an actual game.

Building a snake game

Who doesn't remember the classic game called Snake, which involves a snake chasing a morsel of food? It is probably the very first game that you played as a child. The basic premise of the game is that you control a snake and lead it to a morsel of food. Every time the snake consumes that food, it grows by one unit length, and if the snake hits a boundary wall or itself, it dies. Now as you can imagine, the more you play the game, the longer the snake grows, which, consequently, makes it more difficult to control the snake. In some versions of the game, the speed of the snake also increases, making it even more difficult to control. There comes a point where you simply run out of screen space and the snake inevitably hits a wall or itself, and the game is over.

Here, we will learn how to build such a game. The basic logic of playing the game will be to have a moving rectangle, of which we know the leading point coordinates. This will be our snake. It will be controlled by the four arrow keys. The piece of food is initialized randomly on the screen. At each point of time, we will check whether the rectangle has hit the boundary wall or itself since we know the position of the snake at every point of time. If it has, then the program will exit. Let's now look at the code sectionwise; the code will be explained after each section:

```
from pygame.locals import *
import pygame
import random
import sys
import time

pygame.init()

fpsClock = pygame.time.Clock()

gameSurface = pygame.display.set_mode((800, 600))
pygame.display.set_caption('Pi Snake')

foodcolor = pygame.Color(0, 255, 0)
```

```
backgroundcolor = pygame.Color(255, 255, 255)
snakecolor = pygame.Color(0, 0, 0)
textcolor = pygame.Color(255, 0, 0)

snakePos = [120,240]
snakeSeg = [[120,240],[120,220]]
foodPosition = [400,300]
foodSpawned = 1
Dir = 'D'
changeDir = Dir
Score = 0
Speed = 5
SpeedCount = 0
```

Now, we will learn what each block of code does, but for brevity the very basics are skipped as we have already learned about them in the previous sections.

The first few lines before the `finish()` function initialize PyGame and set the game parameters. The `pygame.time.Clock()`function is used to track time within the game, and this is mostly used for frames per second, or FPS. While it seems somewhat trivial, FPS is very important and can be tweaked. We can increase or decrease the FPS to control the speed of the game. Going further into the code, we can choose options such as the screen size, the color of the snake, the starting position, the starting speed, and so on. The `snakePos` list variable has the head of the snake, and `snakeSeg` will contain the initial coordinates of the segment of the snake in a nested list. The first element contains the coordinates of the head, and the second element contains the coordinates of the tail. This block of code also defines the initial food position, the state of the food, the initial direction, the initial speed, and the initial player score:

```
def finish():
    finishFont = pygame.font.Font(None, 56)
    msg = "Game Over! Score = " + str(Score)
    finishSurf = finishFont.render(msg, True,
textcolor)
    finishRect = finishSurf.get_rect()
    finishRect.midtop = (400, 10)
    gameSurface.blit(finishSurf, finishRect)
```

```
    pygame.display.flip()
    time.sleep(5)
    pygame.quit()
    exit(0)
```

The preceding block of code defines the finishing procedure for the game. As we will see in the following code, `finish()` is called when the snake either hits the walls or itself. In this, we first specify the message that we want to display and its properties, such as the font, and then we add the final score to it. Then, we render the message via the `render()` function, which operates on the `finishFont` variable. Then, we get a rectangle via `get_rect()`, and finally we draw those via the `blit()` function. When using PyGame, `blit()` is a very important function that allows us to draw one image on top of the other. In our case, this is very useful because it allows us to draw a bounding rectangle over the message that we show as a part of the ending of the game. Finally, we render our message on screen via the `display.flip()` function and after a delay of `5` seconds, we quit the game:

```
while 1:
    for event in pygame.event.get():
if event.type == QUIT:
pygame.quit()
            exit(0)
elif event.type == KEYDOWN:
if event.key == ord('d') or event.key == K_RIGHT:
changeDir = 'R'
if event.key == ord('a') or event.key == K_LEFT:
changeDir = 'L'
if event.key == ord('w') or event.key == K_UP:
changeDir = 'U'
if event.key == ord('s') or event.key == K_DOWN:
changeDir = 'D'
if event.key == K_ESCAPE:
pygame.event.post(pygame.event.Event(QUIT))
        pygame.quit()
        exit(0)

if changeDir == 'R' and not Dir == 'L':
Dir = changeDir
if changeDir == 'L' and not Dir == 'R':
```

```
Dir = changeDir
if changeDir == 'U' and not Dir == 'D':
Dir = changeDir
if changeDir == 'D' and not Dir == 'U':
Dir = changeDir

if Dir == 'R':
snakePos[0] += 20
if Dir == 'L':
snakePos[0] -= 20
if Dir == 'U':
snakePos[1] -= 20
if Dir == 'D':
snakePos[1] += 20
```

Then, we move on to the infinite `while` loop, which contains the bulk of the game's logic. Also, as mentioned earlier, the `pygame.event.get()`function gets the type of event; according to what is pressed, it changes the state of some parameters. For example, pressing the *Esc* key causes the game to quit, and pressing the arrow keys changes the direction of the snake. After that, we check whether the new direction is directly opposite to the old direction and change the direction only if it isn't. In this case, `changedDir` is only an intermediate variable. We then change the position of the snake according to the direction that's selected. Each shift in position signifies a shift of 20 pixels on screen:

```
snakeSeg.insert(0,list(snakePos))
if snakePos[0] == foodPosition[0] and snakePos[1]
== foodPosition[1]:
foodSpawned = 0
Score = Score + 1
SpeedCount = SpeedCount + 1
if SpeedCount == 5 :
SpeedCount = 0
Speed = Speed + 1
else:
snakeSeg.pop()

if foodSpawned == 0:
x = random.randrange(1,40)
y = random.randrange(1,30)
foodPosition = [int(x*20),int(y*20)]
```

```
foodSpawned = 1

gameSurface.fill(backgroundcolor)
for position in snakeSeg:
pygame.draw.rect(gameSurface,snakecolor,Rect(posit
ion[0], position[1], 20, 20))
pygame.draw.circle(gameSurface,foodcolor,
(foodPosition[0]+10, foodPosition[1]+10), 10, 0)
pygame.display.flip()
if snakePos[0] > 780 or snakePos[0] < 0:
finish()
if snakePos[1] > 580 or snakePos[1] < 0:
finish()
for snakeBody in snakeSeg[1:]:
if snakePos[0] == snakeBody[0] and snakePos[1] ==
snakeBody[1]:
finish()
fpsClock.tick(Speed)
```

It is important to keep in mind that at this point, nothing is rendered on screen. We are just implementing the logic for the game, and only after we are done with that will anything be rendered on the screen. This will be done with the `pygame.display.flip()`function. Another important thing is that there is another function named `pygame.display.update()`. The difference between these two is that the `update()` function only updates specific areas of the surface, whereas the `flip()` function updates the entire surface. However, if we don't give any arguments to the `update()` function, then it will also update the entire surface.

Now, since we changed the position of the head of the snake, we have to update the `snakeSeg` variable to reflect this change in the snake body. For this, we use the `insert()` method and give the position of object we want to append and the new coordinates. This adds the new coordinates of the snake head into the `snakeSeg` variable. Then comes the interesting part, where we check whether the snake has reached the food. If it has, we increment the score, set the `foodSpawned` state to `False`, and increase `SpeedCount` by one. So, once the speed count reaches five, the speed is increased by one unit. If not, then we remove the last coordinate with the `pop()`

method. This is interesting because if the snake has eaten the food, then its length will increase; consequently, the `pop()` method in the `else` statement will not be executed, and the length of the snake will be increased by one unit.

In the next block of code, we check whether the food is spawned; if not, we randomly spawn it, keeping in mind the dimensions of the screen. The `randrange()`function from the random package allows us to do exactly that.

Finally, we get to the part where the actual rendering takes place. Rendering is nothing but a term for the process that is required to generate and display something on screen. The first statement fills the screen with our selected background color so that everything else on the screen is easily visible:

```
for position in snakeSeg:
pygame.draw.rect(gameSurface,snakecolor,Rect(posit
ion[0], position[1], 20, 20))
```

The preceding block of code loops through all the coordinates present in the `snakeSeg` variable and fills the space between them with the color specified for our snake in the initialization code. The `rect` function takes three inputs: the window name on which the game will be played, the color of the rectangle, and the coordinates of the rectangle that are given by the `Rect` function. The `Rect` function itself takes four arguments: the `x` and `y` position and `height` and `width` of the rectangle. This means that for every coordinate contained in the `snakeSeg` variable, we draw on a rectangle that has dimensions of `20 x 20` pixels. So, we can see that we do not have to keep track of the snake as a whole; we only have to keep track of the coordinates that describe the snake.

Next, we draw our food using the `circle()` method from the `draw` module in the PyGame package. This method takes five arguments: the `window name`, the `color`, the `centre` of the circle, the `radius`, and the `width` of the circle. The centre of the circle is given by a tuple that contains the `x` and `y` coordinates that we selected previously. The

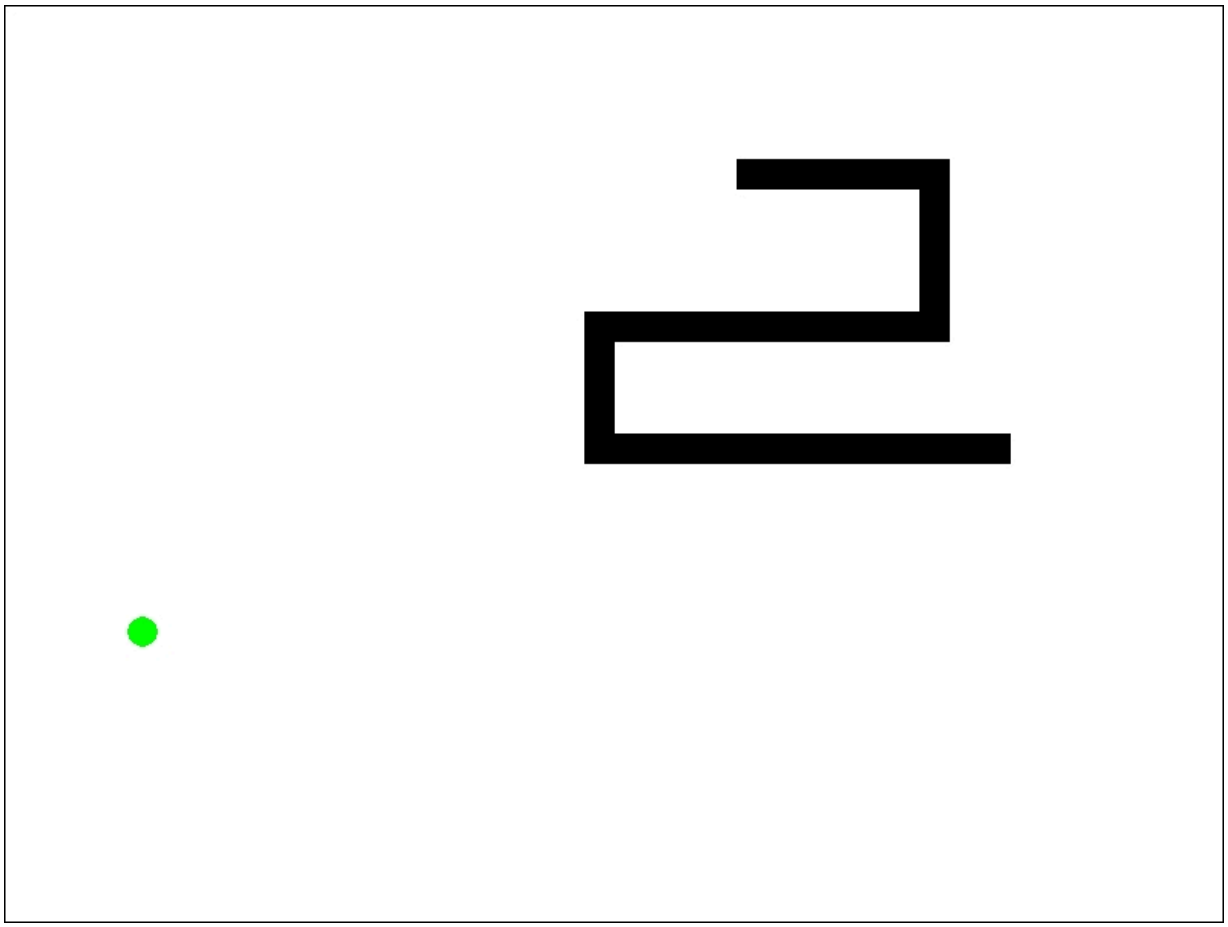
next statement, `pygame.display.flip()`, actually displays what we have just drawn.

Then, we check for the conditions in which the game can end: hitting the wall or itself. When it hits an exit condition, the `finish()`function is called. The first two lines of the function are self-explanatory. In the third line, `render()` basically makes the text displayable. But it is not displayed yet. It will only be displayed once we call the `pygame.display.flip()` function. The next two lines set the position of the textbox that will be displayed on the window. And, finally, we quit the PyGame window and the program after a delay of `5` seconds.

Save this program in a file named `prog2.py` and run it using the following command:

```
python prog2.py
```

This is what you will be greeted with:



We can play the game for as long as we want (and we should because it's our creation) and when we exit, the game will be greeted by the same message that was defined in the `finish()` function!



As you may recognize, in the preceding screenshot, the black shape is the snake and the green circle is its food. You can play around with it and try to get an idea of the logic behind this game as to how it might be programmed.

With this, we complete the implementation of the snake game, and you should try out the program for yourself. An even better way to fully understand how the program works is to change some parameters and see how that affects the playing experience.

Summary

In this chapter, we learned about PyGame and its capabilities. We also learned how to build a binary fractal tree with random branches.

Furthermore, we built a snake game and used it to gain further experience in programming games using PyGame. You can also modify the game to gain any additional features you like.

Using these examples as an inspiration, you can also try to build games of your own. You can combine your knowledge of the chapter on Minecraft to build your own simple Minecraft clone!

In the next chapter, we will learn about the basics of Pi Camera and its webcam and how to capture images using them. We will also build some real-life examples using the Python language.

Chapter 4. Working with a Webcam and Pi Camera

In the previous chapter, we learned how to set up and write our own games using `pygame`. In this chapter, we will learn how to use different types and uses of cameras with our Pi. Let's take a look at the topics we will study and implement in this chapter:

- Working with a webcam
- Crontab
- Timelapse using a webcam
- Webcam video recording and playback
- Pi Camera and Pi NoIR comparison
- Timelapse using Pi Camera
- The `PiCamera` module in Python

Working with webcams

USB webcams are a great way to capture images and videos. Raspberry Pi supports common USB webcams.

> **Note**
>
> To be on the safe side, here is a list of the webcams supported by Pi: http://elinux.org/RPi_USB_Webcams.

I am using a Logitech HD c310 USB Webcam as shown in the following image:



Note

You can purchase it online, and you can find the product details and the specifications at http://www.logitech.com/en-in/product/hd-webcam-c310.

Attach your USB webcam to Raspberry Pi through the USB port on Pi and run the **lsusb** command in the terminal. This command lists all the USB devices connected to the computer. The output should be similar to the following output depending on which port is used to connect the USB webcam:

```
pi@raspberrypi ~/book/chapter04 $ lsusb
Bus 001 Device 002: ID 0424:9514 Standard
Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation
2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard
Microsystems Corp.
Bus 001 Device 004: ID 148f:2070 Ralink
Technology, Corp. RT2070 Wireless Adapter
Bus 001 Device 007: ID 046d:081b Logitech, Inc.
Webcam C310
Bus 001 Device 006: ID 1c4f:0003 SiGma Micro HID
controller
Bus 001 Device 005: ID 1c4f:0002 SiGma Micro
Keyboard TRACER Gamma Ivory
```

Then, install the **fswebcam** utility by running the following command:

```
sudo apt-get install fswebcam
```

fswebcam is a simple command-line utility that captures images with webcams for Linux computers. Once the installation is done, you can use the following command to create a directory for output images:

```
mkdir /home/pi/book/output
```

Then, run the following command to capture the image:

```
fswebcam -r 1280x960 --no-banner
~/book/output/camtest.jpg
```

This will capture an image with a resolution of 1280 x 960. You might want to try another resolution for your learning. The **--no-banner** command will disable the timestamp banner. The image will be saved with the filename mentioned. If you run this command multiple times with the same filename, the image file will be overwritten each time. So, make sure that you change the filename if you want to save previously captured images. The text output of the command should be similar to the following output:

```
--- Opening /dev/video0...
Trying source module v4l2...
/dev/video0 opened.
No input was specified, using the first.
--- Capturing frame...
Corrupt JPEG data: 2 extraneous bytes before
marker 0xd5
Captured frame in 0.00 seconds.
--- Processing captured image...
Disabling banner.
Writing JPEG image to
'/home/pi/book/output/camtest.jpg'.
```

Crontab

Cron is a time-based job scheduler in Unix-like computer operating systems. It is driven by a **crontab** (cron table) file, which is a configuration file that specifies shell commands to be run periodically on a given schedule. It is used to schedule commands or shell scripts to run periodically at a fixed time, date, or interval.

The syntax for **crontab** in order to schedule a command or script is as follows:

```
1 2 3 4 5 /location/command
```

Here are the definitions:

- 1: Minutes (0-59)
- 2: Hours (0-23)

- 3: Days (0-31)
- 4: Months [0-12 (1 for January)]
- 5: Days of the week [0-7 (7 or 0 for Sunday)]
- **/location/command**: The script or command name to be scheduled

The **crontab** entry to run any script or command every minute is as follows:

```
* * * * * /location/command 2>&1
```

We will be using **crontab** in many chapters in this book. In the next section, we will learn how to use **crontab** to schedule a script to capture images periodically in order to create the timelapse sequence.

> ## Note
>
> You can refer to this URL for more details on `crontab`: http://www.adminschoice.com/crontab-quick-reference.

Creating a timelapse sequence using fswebcam

Timelapse photography means capturing photographs in regular intervals and playing the images with a higher frequency in time than those that were shot. For example, if you capture images with a frequency of one image per minute for 10 hours, you will get 600 images. If you combine all these images in a video with 30 images per second, you will get 10 hours of timelapse video compressed in 20 seconds. You can use your USB webcam with Raspberry Pi to achieve this. We already know how to use the Raspberry Pi with a webcam and the fswebcam utility to capture an image. The trick is to

write a script that captures images with different names and then add this script in **crontab** and make it run at regular intervals.

Begin with creating a directory for captured images:

```
mkdir /home/pi/book/output/timelapse
```

Open an editor of your choice, write the following code, and save it as **timelapse.sh**:

```
#!/bin/bash

DATE=$(date +"%Y-%m-%d_%H%M")
fswebcam -r 1280x960 --no-banner
/home/pi/book/output/timelapse/garden_$DATE.jpg
```

Make the script executable using:

```
chmod +x timelapse.sh
```

This shell script captures the image and saves it with the current timestamp in its name. Thus, we get an image with a new filename every time as the file contains the timestamp. The second line in the script creates the timestamp that we're using in the filename. Run this script manually once, and make sure that the image is saved in the **/home/pi/book/output/timelapse** directory with the **garden_<timestamp>.jpg** name.

To run this script at regular intervals, we need to schedule it in **crontab**.

The **crontab** entry to run our script every minute is as follows:

```
* * * * * /home/pi/book/chapter04/timelapse.sh
2>&1
```

Open the **crontab** of the Pi user with **crontab –e**. It will open **crontab** with **nano** as the editor. Add the preceding line to **crontab**,

save it, and exit it.

Once you exit **crontab**, it will show the following message:

```
no crontab for pi - using an empty one
crontab: installing new crontab
```

Our timelapse webcam setup is now live. If you want to change the image capture frequency, then you have to change the crontab settings. To set it every 5 minutes, change it to ***/5 * * * ***. To set it for every 2 hours, use **0 */2 * * ***. Make sure that your MicroSD card has enough free space to store all the images for the time duration for which you need to keep your timelapse setup.

Once you capture all the images, the next part is to encode them all in a fast playing video, preferably 20 to 30 frames per second. For this part, the **mencoder** utility is recommended. The following are the steps to create a timelapse video with **mencoder** on a Raspberry Pi or any Debian/Ubuntu machine:

1. Install **mencoder** using **sudo apt-get install mencoder**
2. Navigate to the output directory by using:

```
cd /home/pi/book/output/timelapse
```

3. Create a list of your timelapse sequence images using:

```
ls garden_*.jpg > timelapse.txt
```

4. Use the following command to create a video:

```
mencoder -nosound -ovc lavc -lavcopts
vcodec=mpeg4:aspect=16/9:vbitrate=8000000 -vf
scale=1280:960 -o timelapse.avi -mf
type=jpeg:fps=30 mf://@timelapse.txt
```

This will create a video with name `timelapse.avi` in the current directory with all the images listed in `timelapse.txt` with a 30 fps frame rate. The statement contains the details of the video codec, aspect ratio, bit rate, and scale. For more information, you can run

man mencoder on Command Prompt. We will cover how to play a video in the next section.

Webcam video recording and playback

We can use a webcam to record live videos using **avconv**. Install **avconv** using **sudo apt-get install libav-tools**. Use the following command to record a video:

```
avconv -f video4linux2 -r 25 -s 1280x960 -i
/dev/video0 ~/book/output/VideoStream.avi
```

It will show following output on the screen:

```
pi@raspberrypi ~ $ avconv -f video4linux2 -r 25 -s
1280x960 -i /dev/video0
~/book/output/VideoStream.avi
avconv version 9.14-6:9.14-1rpi1rpi1, Copyright
(c) 2000-2014 the Libav developers
  built on Jul 22 2014 15:08:12 with gcc 4.6
(Debian 4.6.3-14+rpi1)
[video4linux2 @ 0x5d6720] The driver changed the
time per frame from 1/25 to 2/15
[video4linux2 @ 0x5d6720] Estimating duration from
bitrate, this may be inaccurate
Input #0, video4linux2, from '/dev/video0':
  Duration: N/A, start: 629.030244, bitrate:
147456 kb/s
    Stream #0.0: Video: rawvideo, yuyv422,
1280x960, 147456 kb/s, 1000k tbn, 7.50 tbc
Output #0, avi, to
'/home/pi/book/output/VideoStream.avi':
  Metadata:
    ISFT            : Lavf54.20.4
    Stream #0.0: Video: mpeg4, yuv420p, 1280x960,
q=2-31, 200 kb/s, 25 tbn, 25 tbc
Stream mapping:
  Stream #0:0 -> #0:0 (rawvideo -> mpeg4)
Press ctrl-c to stop encoding
frame=  182 fps=  7 q=31.0 Lsize=     802kB
```

```
time=7.28 bitrate= 902.4kbits/s
video:792kB audio:0kB global headers:0kB muxing
overhead 1.249878%
Received signal 2: terminating.
```

You can terminate the recording sequence by pressing *Ctrl* + *C*.

We can play the video using **omxplayer**. It comes with the latest raspbian, so there is no need to install it. To play a file with the name **vid.mjpg**, use the following command:

```
omxplayer ~/book/output/VideoStream.avi
```

It will play the video and display some output similar to the one here:

```
pi@raspberrypi ~ $ omxplayer
~/book/output/VideoStream.avi
Video codec omx-mpeg4 width 1280 height 960
profile 0 fps 25.000000
Subtitle count: 0, state: off, index: 1, delay: 0
V:PortSettingsChanged: 1280x960@25.00 interlace:0
deinterlace:0 anaglyph:0 par:1.00 layer:0
have a nice day ;)
```

Try playing timelapse and record videos using **omxplayer**.

Working with the Pi Camera and NoIR Camera modules

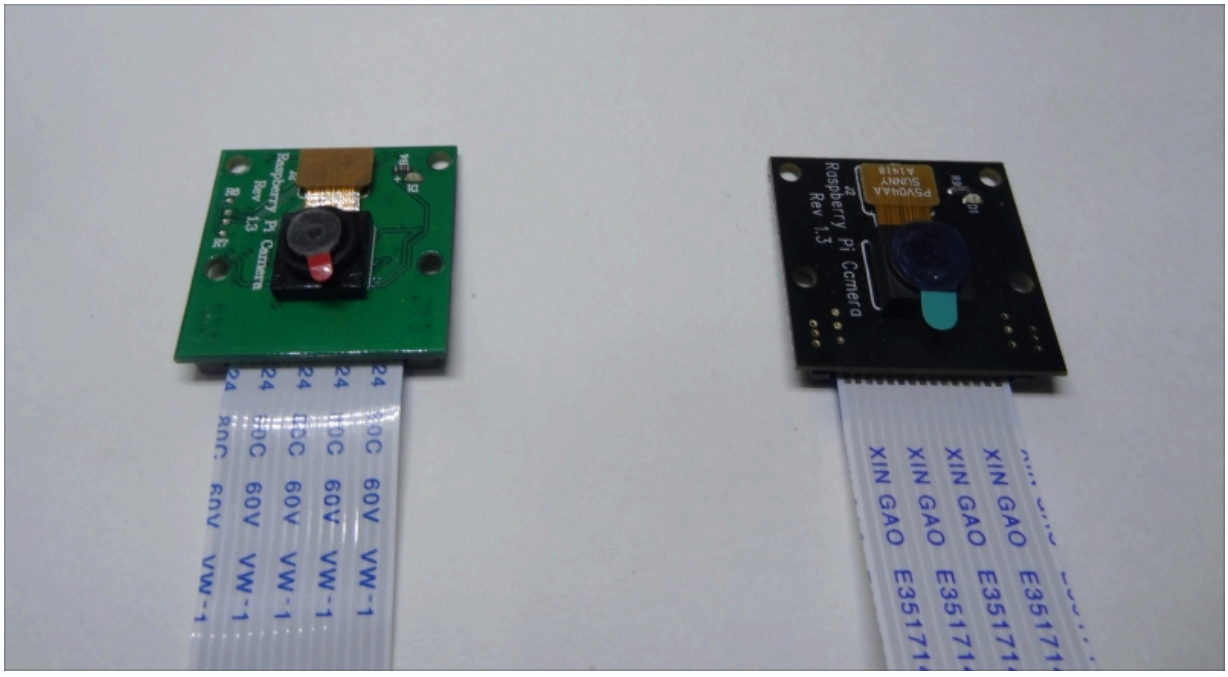
These camera modules are specially manufactured for Raspberry Pi and work with all the available models. You will need to connect the camera module to the CSI port, located behind the Ethernet port, and activate the camera using the **raspi-config** utility if you haven't already.

> ## Note
>
> You can find the video instructions to connect the camera module to Raspberry Pi at http://www.raspberrypi.org/help/camera-module-setup/.

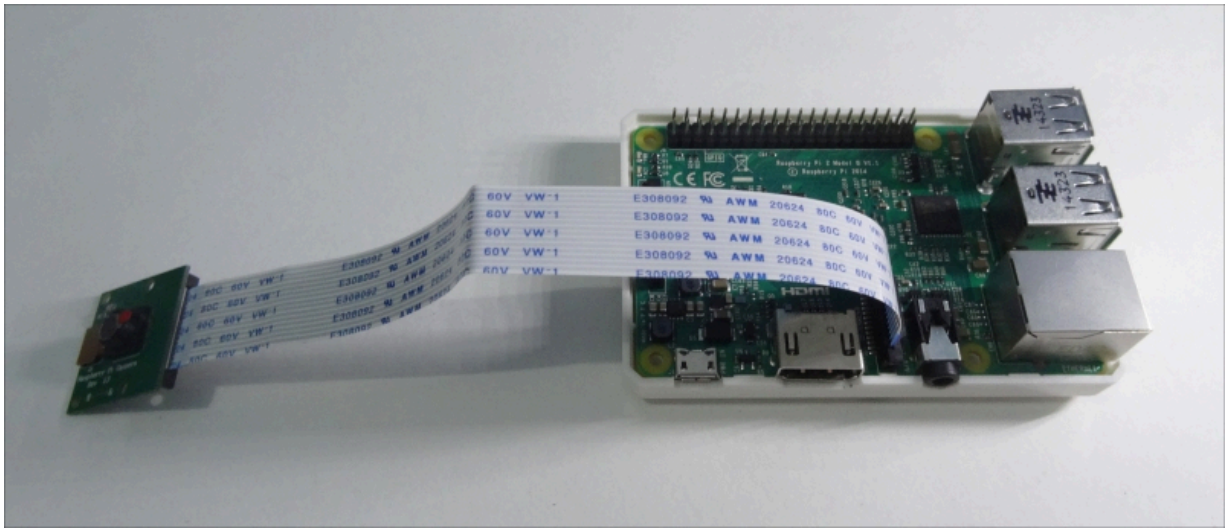
This page lists the types of camera modules available: http://www.raspberrypi.org/products/.

Two types of camera modules are available for the Pi. These are Pi Camera and Pi NoIR camera, and they can be found at https://www.raspberrypi.org/products/camera-module/ and https://www.raspberrypi.org/products/pi-noir-camera/, respectively.

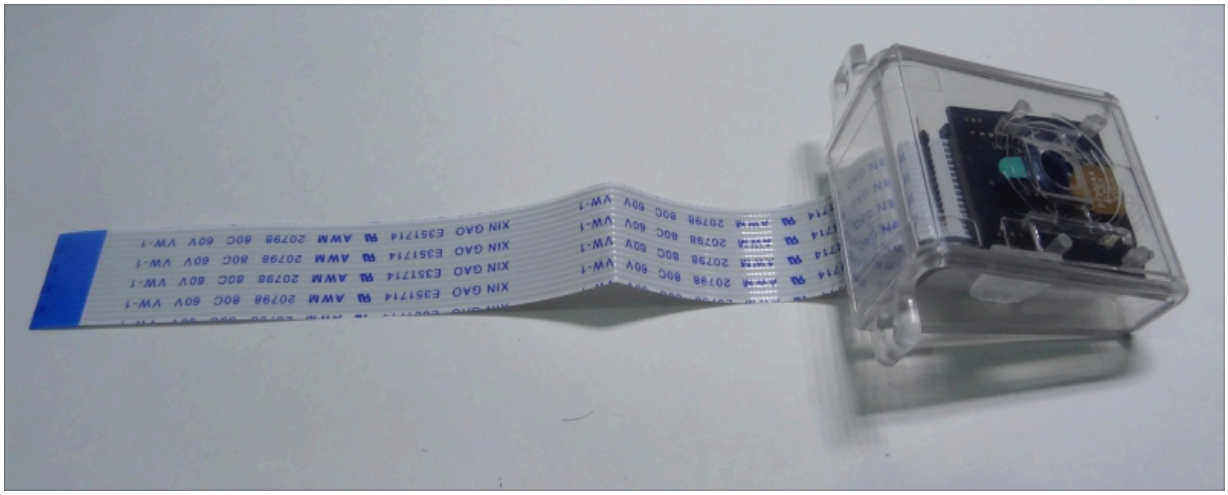
The following image shows Pi Camera and Pi NoIR Camera boards side by side:



The following image shows the Pi Camera board connected to the Pi:



The following is an image of the Pi Camera board placed in the camera case:



The main difference between Pi Camera and Pi NoIR Camera is that Pi Camera gives better results in good lighting conditions, whereas Pi NoIR (NoIR stands for No-Infra Red) is used for low light photography. To use NoIR Camera in complete darkness, we need to flood the object to be photographed with infrared light.

Note

This is a good time to take a look at the various enclosures for Raspberry Pi Models. You can find various cases available online at https://www.adafruit.com/categories/289.

An example of a Raspberry Pi case is as follows:



Using raspistill and raspivid

To capture images and videos using the Raspberry Pi Camera module, we need to use `raspistill` and `raspivid` utilities.

To capture an image, run the following command:

```
raspistill -o cam_module_pic.jpg
```

This will capture and save the image with name **cam_module_pic.jpg** in the current directory.

To capture a 20 second video with the camera module, run the following command:

```
raspivid –o test.avi –t 20000
```

This will capture and save the video with name **test.avi** in the current directory. Unlike **fswebcam** and **avconv**, **raspistill** and **raspivid** do

not write anything to the console. So, you need to check the current directory for the output. Also, one can run the **echo $?** command to check whether these commands executed successfully. We can also mention the complete location of the file to be saved in these command, as shown in the following example:

```
raspistill -o
/home/pi/book/output/cam_module_pic.jpg
```

Just like **fswebcam**, **raspistill** can be used to record the timelapse sequence. In our timelapse shell script, replace the line that contains **fswebcam** with the appropriate **raspistill** command to capture the timelapse sequence and use **mencoder** again to create the video. This is left as an exercise for the readers.

Now, let's take a look at the images taken with the Pi Camera under different lighting conditions.

The following is the image with normal lighting and the backlight:



The following is the image with only the backlight:



The following is the image with normal lighting and no backlight:



For NoIR camera usage in the night under low light conditions, use IR illuminator light for better results. You can get it online. A typical off-the-shelf LED IR illuminator suitable for our purpose will look like the one shown here:



Using picamera in Python with the Pi Camera module

The **picamera** is a Python package that provides a programming interface to the Pi Camera module. The most recent version of raspbian has `picamera` preinstalled. If you do not have it installed, you can install it using:

```
sudo apt-get install python-picamera
```

The following program quickly demonstrates the basic usage of the `picamera` module to capture an image:

```
import picamera
import time

with picamera.PiCamera() as cam:
    cam.resolution=(1024,768)
    cam.start_preview()
    time.sleep(5)
    cam.capture('/home/pi/book/output/still.jpg')
```

We have to import time and `picamera` modules first. **cam.start_preview()** will start the preview, and **time.sleep(5)** will wait for 5 seconds before **cam.capture()** captures and saves image in the specified file.

There is a built-in function in **picamera** for timelapse photography. The following program demonstrates its usage:

```
import picamera
import time

with picamera.PiCamera() as cam:
    cam.resolution=(1024,768)
    cam.start_preview()
    time.sleep(3)
    for count, imagefile in
enumerate(cam.capture_continuous('/home/pi/book/ou
tput/image{counter:02d}.jpg')):
        print 'Capturing and saving ' + imagefile
        time.sleep(1)
        if count == 10:
            break
```

In the preceding code, **cam.capture_continuous()** is used to capture the timelapse sequence using the Pi camera module.

> **Note**
>
> Checkout more examples and API references for the picamera module at http://picamera.readthedocs.org/.

The Pi camera versus the webcam

Now, after using the webcam and the Pi Camera, it's a good time to understand the differences, the pros, and the cons of using these.

The Pi camera board does not use a USB port and is directly interfaced to the Pi. So, it provides better performance than a webcam in terms of the frame rate and resolution. We can directly use the **picamera** module in Python to work on images and videos. However, the Pi camera cannot be used with any other computer.

A webcam uses an USB port for interface, and because of that, it can be used with any computer. However, compared to the Pi Camera its performance, it is lower in terms of the frame rate and resolution.

Summary

In this chapter, we learned how to use a webcam and the Pi camera. We also learned how to use utilities such as **fswebcam**, **avconv**, **raspistill**, **raspivid**, mencoder, and omxplayer. We covered how to use **crontab**. We used the Python **picamera** module to programmatically work with the Pi camera board. Finally, we compared the Pi camera and the webcam. We will be reusing all the code examples and concepts for some real-life projects soon.

In the next chapter, we will learn about programming Pi GPIO with PiGlow, which is a third-party add-on board for the Pi.

Chapter 5. Introduction to GPIO Programming

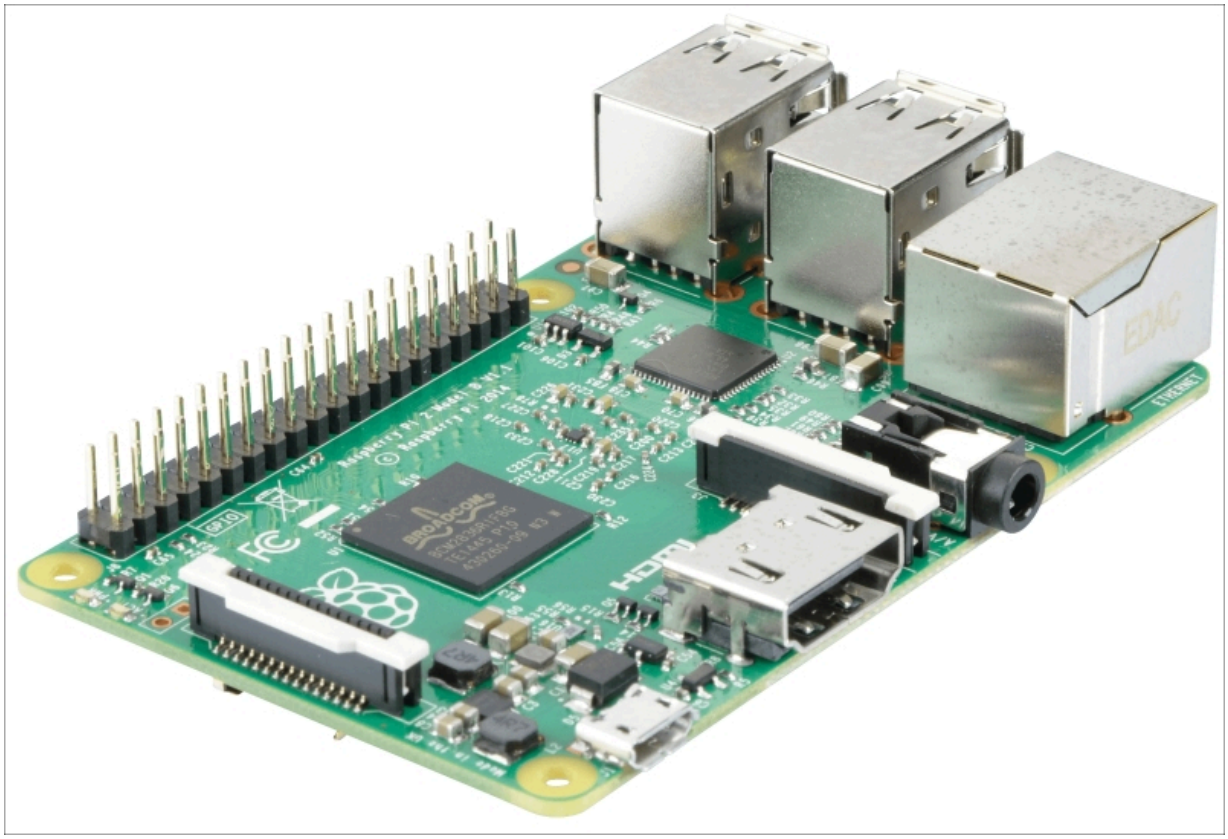
In the previous chapter, we learned how to use a webcam and Pi Camera to capture images and videos. We also built a timelapse photography box to get experience about the real-life uses of what we learned.

In this chapter, we are going to adopt a hardware-focused approach and use Raspberry Pi's most talked about feature: its **GPIO** (short for **General Purpose Input Output**) pins. Specifically, we will learn how to use GPIO pins on Raspberry Pi B+ and Raspberry Pi 2. Once we get familiar with the use of the pins, we will also build a real-life project to further our knowledge of their use. The following topics will be covered in the chapter:

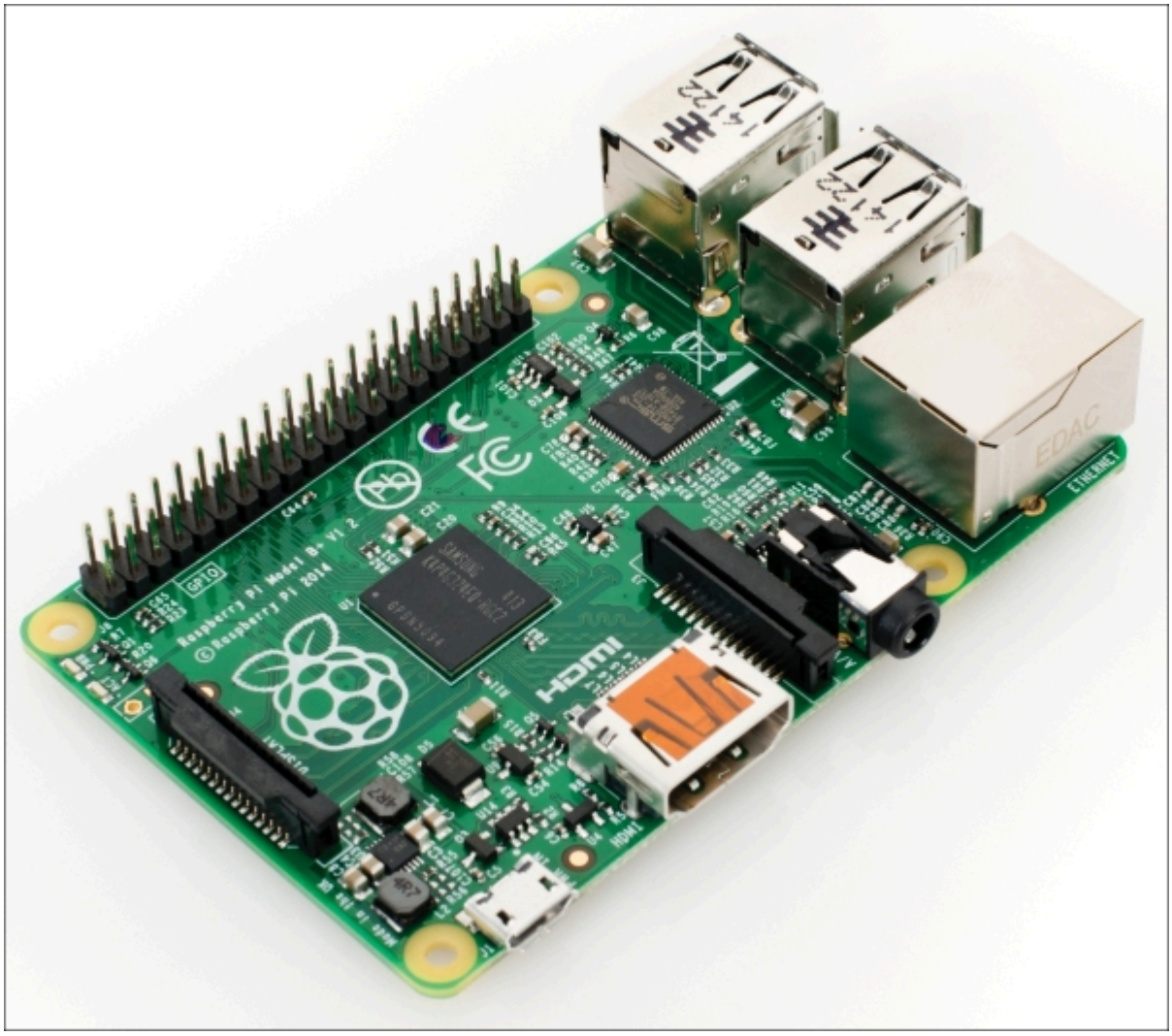
- Introducing GPIO pins
- Blinking an LED with GPIO pins
- Adding push button controls
- Learning about PiGlow

Introducing GPIO pins

Most of you may know where GPIO pins are located on the Raspberry Pi. If not, the following illustration will make it clear:



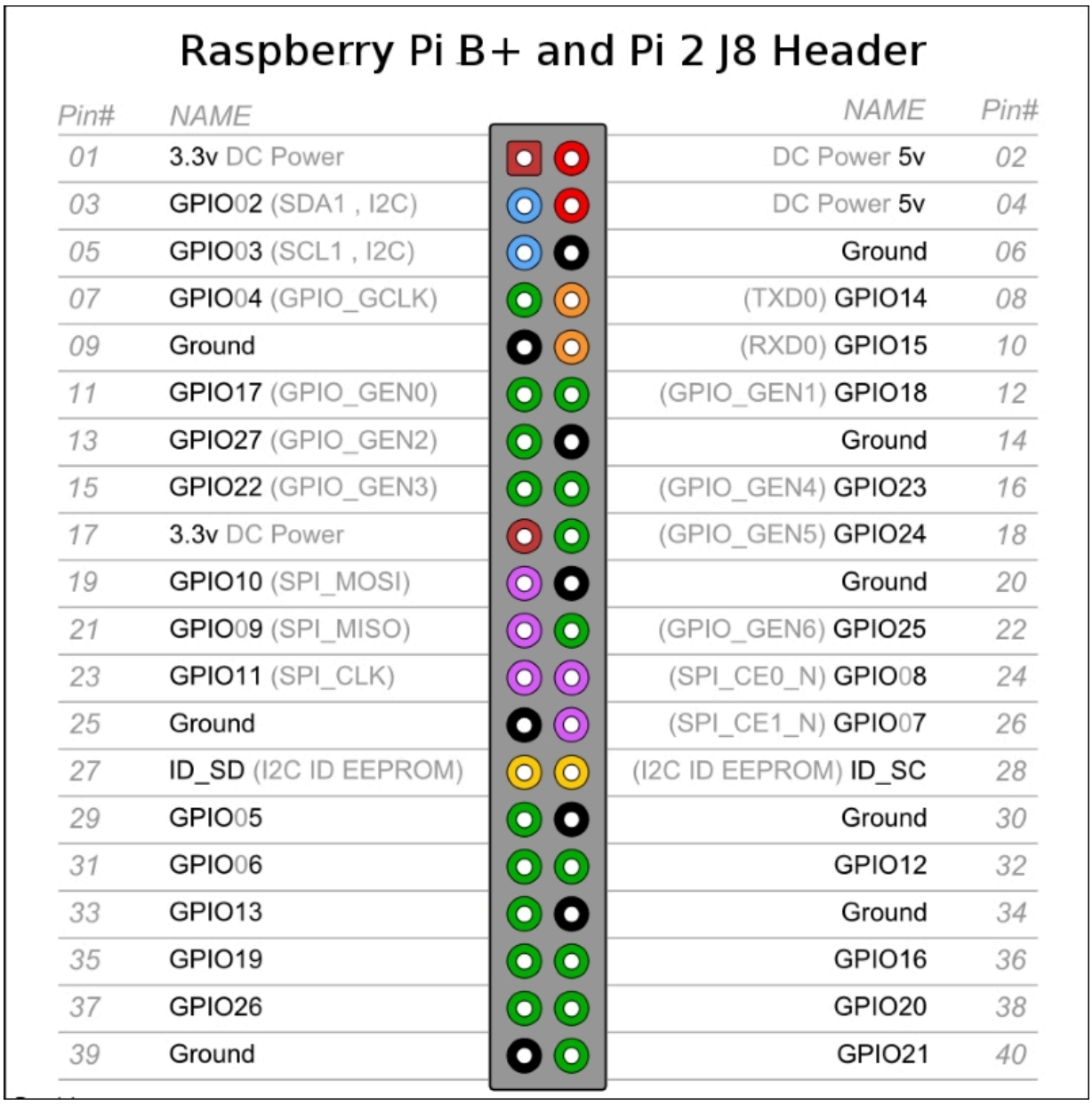
The following is a top view of the Raspberry Pi B+ board, which will help you see the components even more clearly:



The following diagram of the GPIO pins gives information about the naming convention and the function of each of the pins:

Note

Note that the GPIO pins of Raspberry Pi B+ and Pi 2 are the same.



As you can see from the preceding diagram, there are four power pins, two for 3.3V and two for 5V, 8 ground pins distributed across the rail, 26 GPIO pins—some of which also provide protocols such as UART, SPI, PCM, PWM, I2C—and two pins reserved for accessing the EEPROM via I2C.

GPIO pins can be ON or OFF and HIGH or LOW. When a 3.3V pin is high, it outputs 3.3V, and when it is low, it outputs 0V. A GPIO pin can act as an input as well as an output but not both at the same

time. To use it as an output is as simple as setting the pin state to ON or OFF.

To use GPIO pins, we will use a simple Python library called `RPi.GPIO`, which takes out all the effort and makes it easy to configure and use the pins. Configuring a pin to be an output or an input requires only a single statement. We will now see how to manipulate GPIO pins to use this library.

Take care to use Broadcom chip names for the GPIO pins. You can look this up in the diagram given earlier. If our pin has been set to the output mode, we can also set its activation or voltage level to HIGH or LOW. Keep in mind that in the case of the Raspberry Pi, high voltage is represented by 3.3V, and low voltage is represented by 0V. It will be damaging for the Raspberry Pi if you connect a 5V device to its 3.3V GPIO pins. However, you can use two 5V pins to power a small add-on, such as an LED or a buzzer. If you plan to use anything that requires more current, ensure that you use a separate power supply as the Pi can supply only a limited current. So, directly powering a motor from the Pi is not a good idea.

You can do amazing things with the input pins. Not only can you take inputs from a button, but you can also connect sensors, such as a light sensor, smoke sensor, and temperature sensor to build all kinds of amazing projects. You can also use an Internet-connected Raspberry Pi to control your outputs from anywhere in the world! But the networking part is outside the scope of this chapter.

A simple project at this point would be to build a simple LED blinking program. So let's see how that works!

Building an LED Blinker

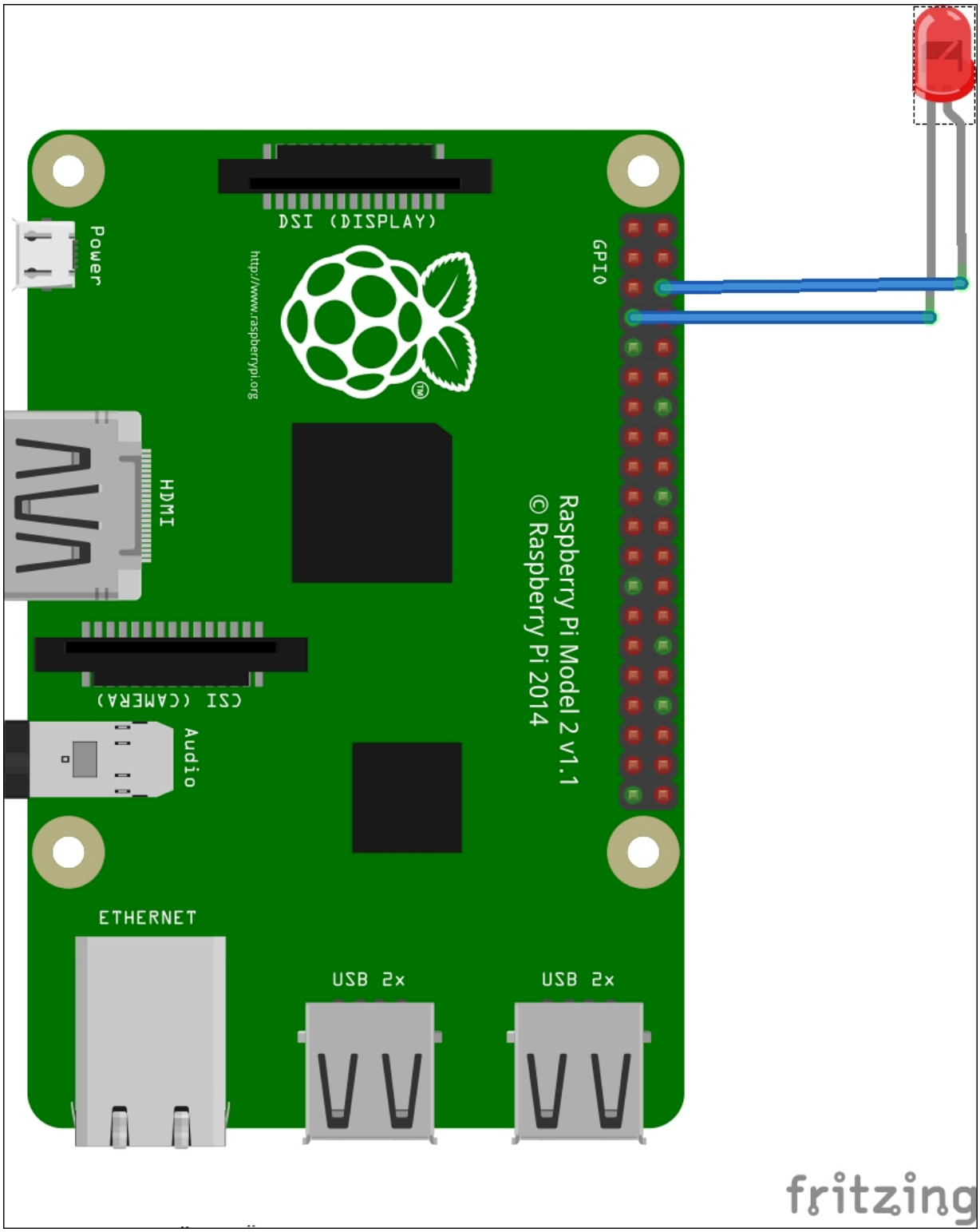
For this project, you will require a humble low-power LED with the color of your choice, which can be obtained at your local hardware store or can be ordered online. The Raspberry Pi will act as both the switch and the power supply. In fact, we will power and switch the LED from the same output pin. The complete code and the wiring diagram has been given here. We will learn the code line by line following the diagram. Connect your LED to the Raspberry Pi, as shown. In an LED, the longer leg is the positive pin by convention, and the shorter leg has negative polarity. So take care to connect it the right way, or it might get damaged:

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)
GPIO.setup(7, GPIO.OUT)

def Blink(speed):
    GPIO.output(7,True)
    time.sleep(speed)
    GPIO.output(7,False)
    time.sleep(speed)
    GPIO.cleanup()

Blink(1)
```



Once you've connected the LED as shown, go ahead and save the code as `prog1.py`, and execute the following command:

```
python prog1.py
```

If everything goes according to the plan, you should now see your LED blinking. Now we will analyze the code to see what exactly was done.

As usual, we begin the Python program by importing the required dependencies. `RPi.GPIO` is the Python library that provides the API that allows us to use GPIO pins. It is preinstalled on the Raspbian operating system. Then, we import the time module that will provide us with the delay for the blink.

`GPIO.setmode()` sets the addressing mode of the GPIO pins. There are two modes, namely BOARD and BCM. The BOARD mode addresses the pins by the numbering given on the board. For example, in the preceding GPIO diagram, the GPIO04 is pin 7. The other mode is BCM and it defines the pins according to the Broadcom Firmware on the Raspberry Pi. So, the GPIO04 pin will be referred to as 4 rather than 7. The next statement, `GPIO.setup()`, configures the specified pin to be an output or input depending on the argument.

Now, we'll move on to the main chunk of the code, which is the `Blink()` function. The `GPIO.output()` function takes two arguments: the pin number and the logical operation to be performed. In the first output statement, we set pin 7 to HIGH, and then in the second output statement, we set it to LOW. In between these functions, we have the `sleep()` function from the time package, which takes only one argument: the time in seconds. It does nothing but delay the execution of the next statement by the input number of seconds, which could also be fractions such as 0.5 or 0.2. The last statement in the `Blink()`function is the `cleanup()` function, which resets all the ports you have used back to the input mode. This prevents the pin from being damaged when you have set HIGH as an output and then accidentally connect it to ground, which would short-circuit the port. So, it's safer to leave the ports as inputs.

Connecting a button

As we saw in the previous section, connecting an LED to a GPIO pin was easy, and the code was simple. In this section, we will use the same general code in order to program a GPIO pin to take an input from the button and if the button is pressed, light the LED that we have already connected in the previous section. We will now take a look at the code and then explain those lines that were not encountered in the previous section:

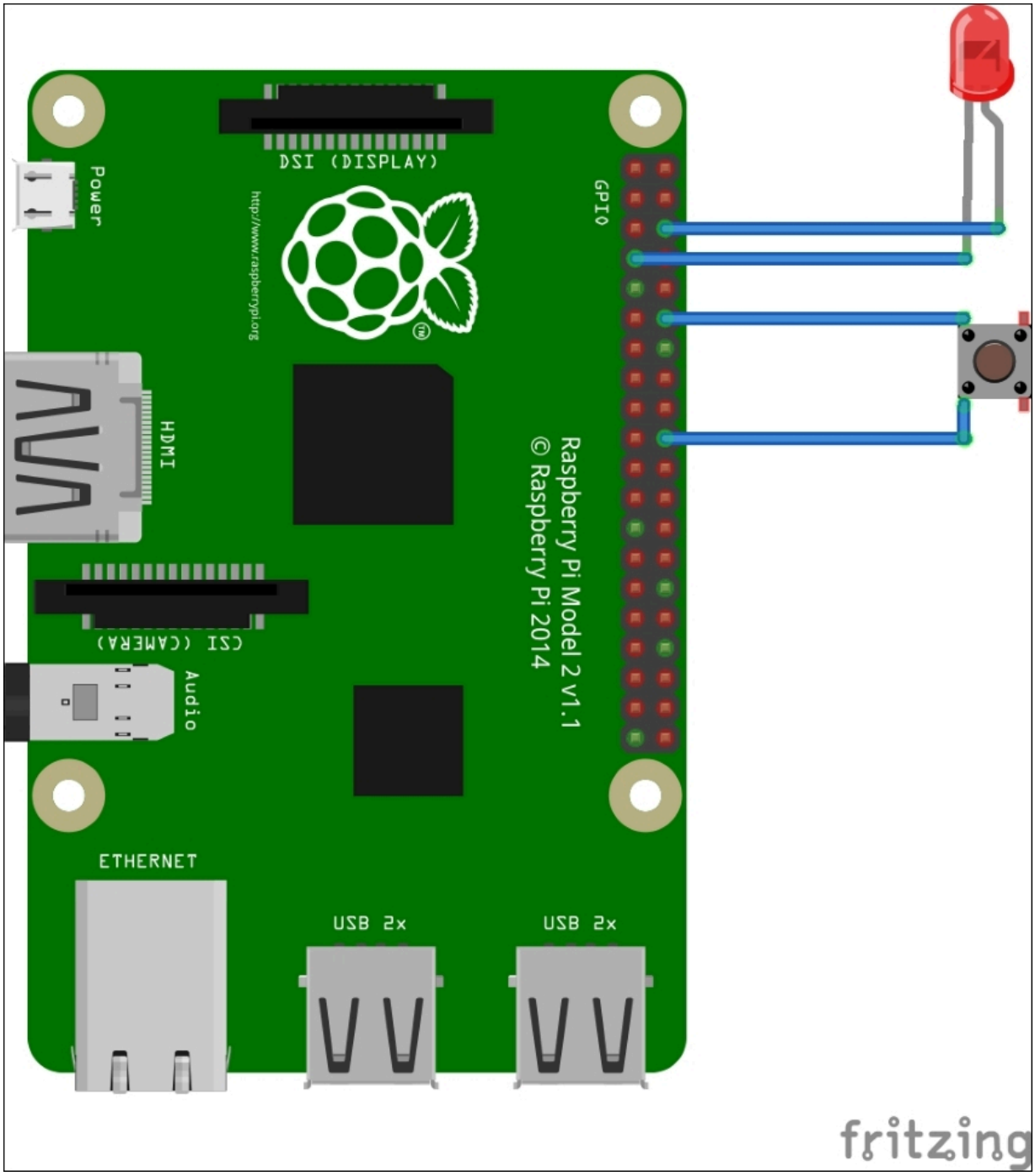
```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BOARD)

GPIO.setup(12, GPIO.IN, pull_up_down =
GPIO.PUD_UP)
GPIO.setup(7, GPIO.OUT)

while True:
    input_state = GPIO.input(12)

if input_state == False:
    print('Button Pressed')
GPIO.output(7, True)
else :
    print('Button Not Pressed')
GPIO.output(7, False)
```



Now, save the code as `prog2.py` and run it with the following command:

```
python prog2.py
```

In this section, we configure pin #12 (by board numbering) to be an output via the same `GPIO.setup()` statement that we used in the previous section. But here, we have an extra argument, `pull_up_down=GPIO.PUD_UP`. This means that the internal resistor for the GPIO pin is being pulled up to 3.3V, and it will return `False` when the button is pressed. This is a little counter-intuitive, but without pulling up or down, the input will be left "floating", which means that the Pi will not be able to determine a button press simply because it does not have anything to compare the voltage level to!

Now we enter the infinite `while` loop, where it checks for the button state using the `GPIO.input()`function. As mentioned earlier, if the button is pressed, `input_state` will return `False` and the LED will be lit up. If, instead, the button is not pressed anymore, the LED will go off!

Simple, isn't it? Let's move on a more complicated add-on.

Installing PiGlow

The following figure shows the PiGlow board:



Since we have learned how to control an LED and a button via Raspberry Pi GPIO pins, we will now move on to a module called

PiGlow, which is an add-on board for the Raspberry Pi and provides 18 individually controlled LEDs, and which you can use via a provided library. We will now proceed to install the software requirements for PiGlow by using a script provided by Pimoroni itself. To install PiGlow, run the following command in a new terminal:

```
curl get.pimoroni.com/piglow | bash
```

To check whether PiGlow has been installed properly, open a Python terminal and execute the following command:

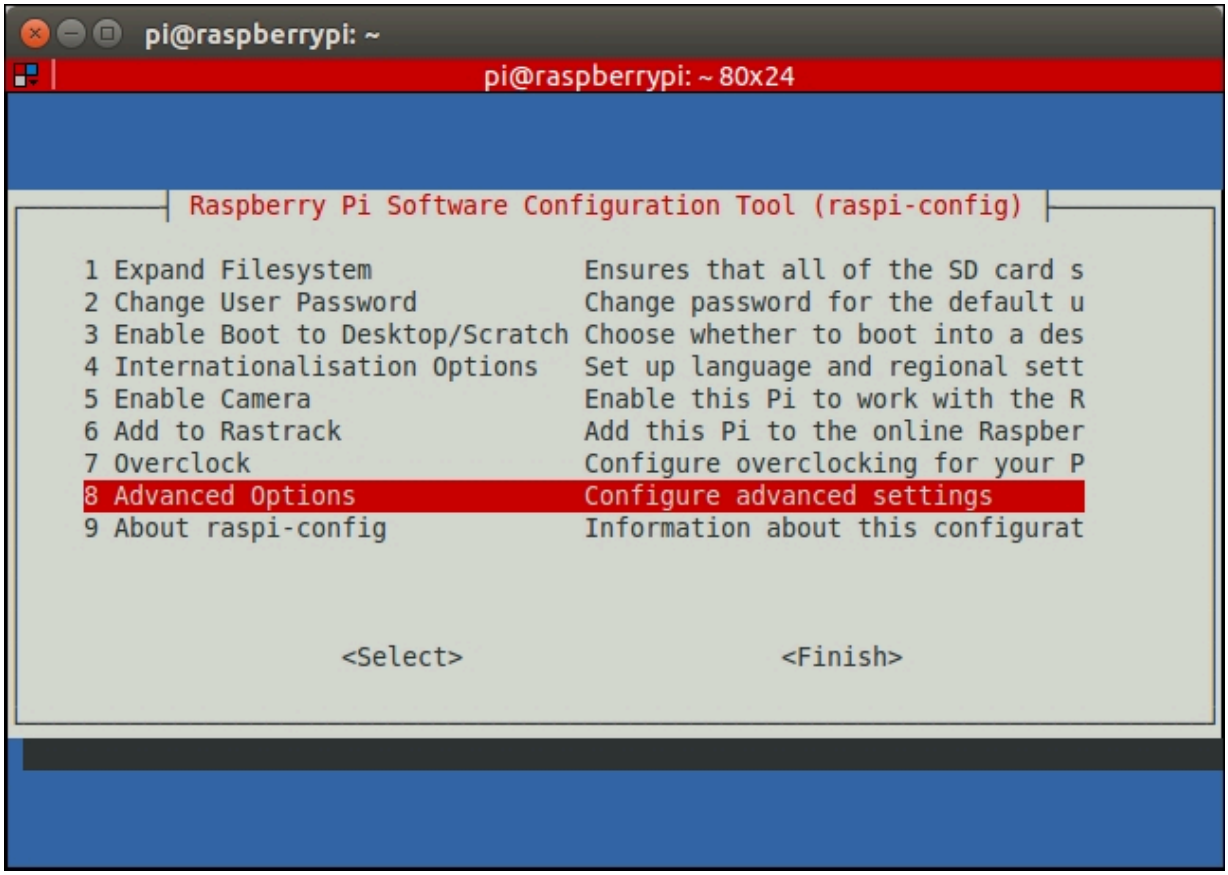
```
import piglow
```

If it is successful, then continue.

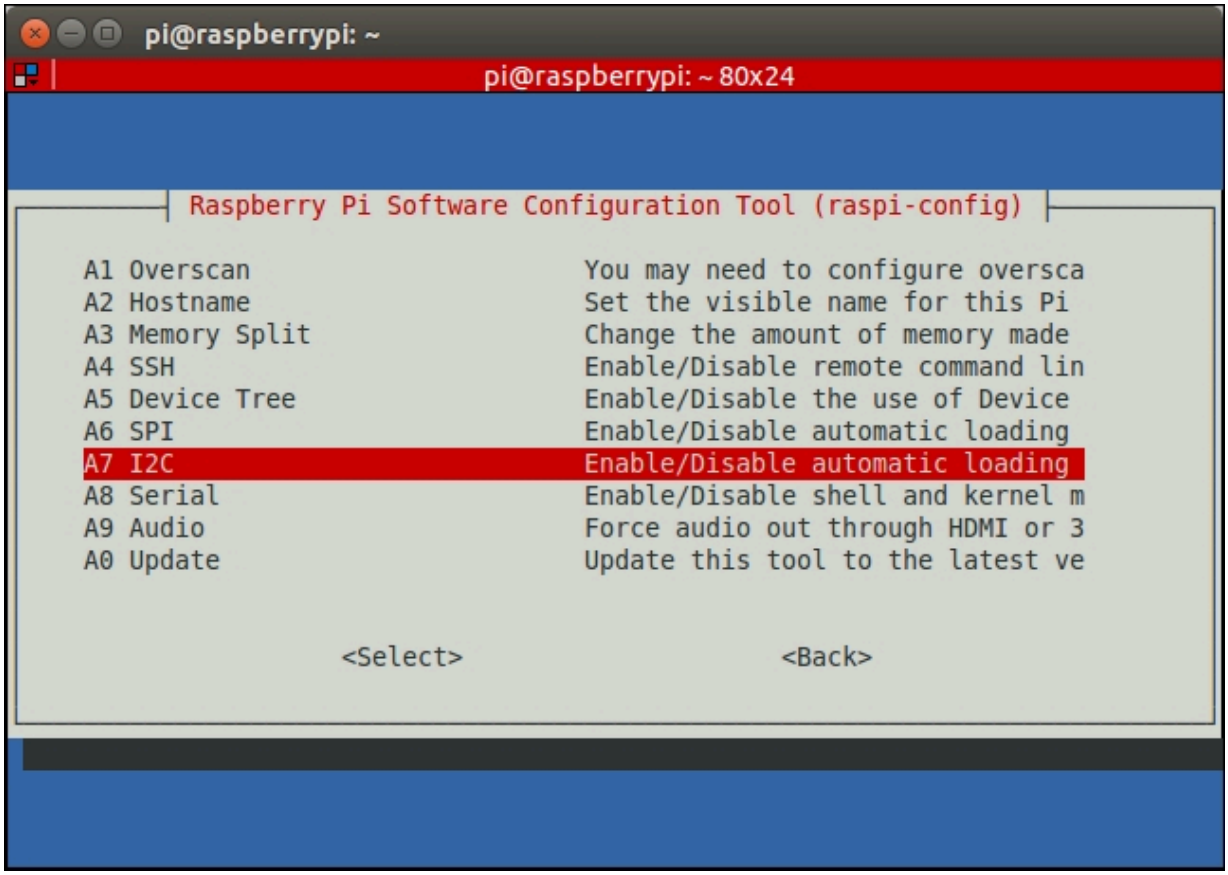
To confirm whether I2C has been enabled, open the Raspberry Pi configuration tool by entering the following command:

```
sudo raspi-config
```

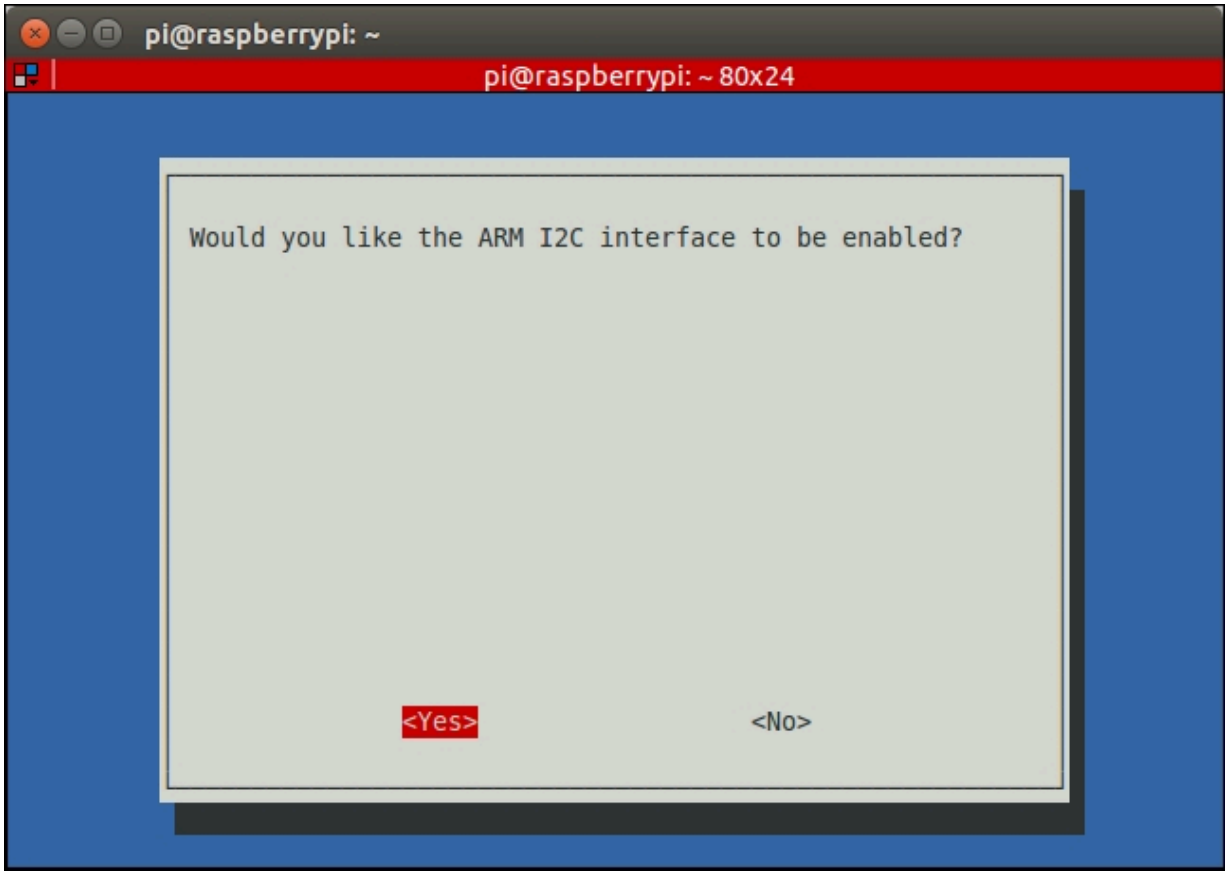
Then, select the **Advanced Options** menu from the displayed options.



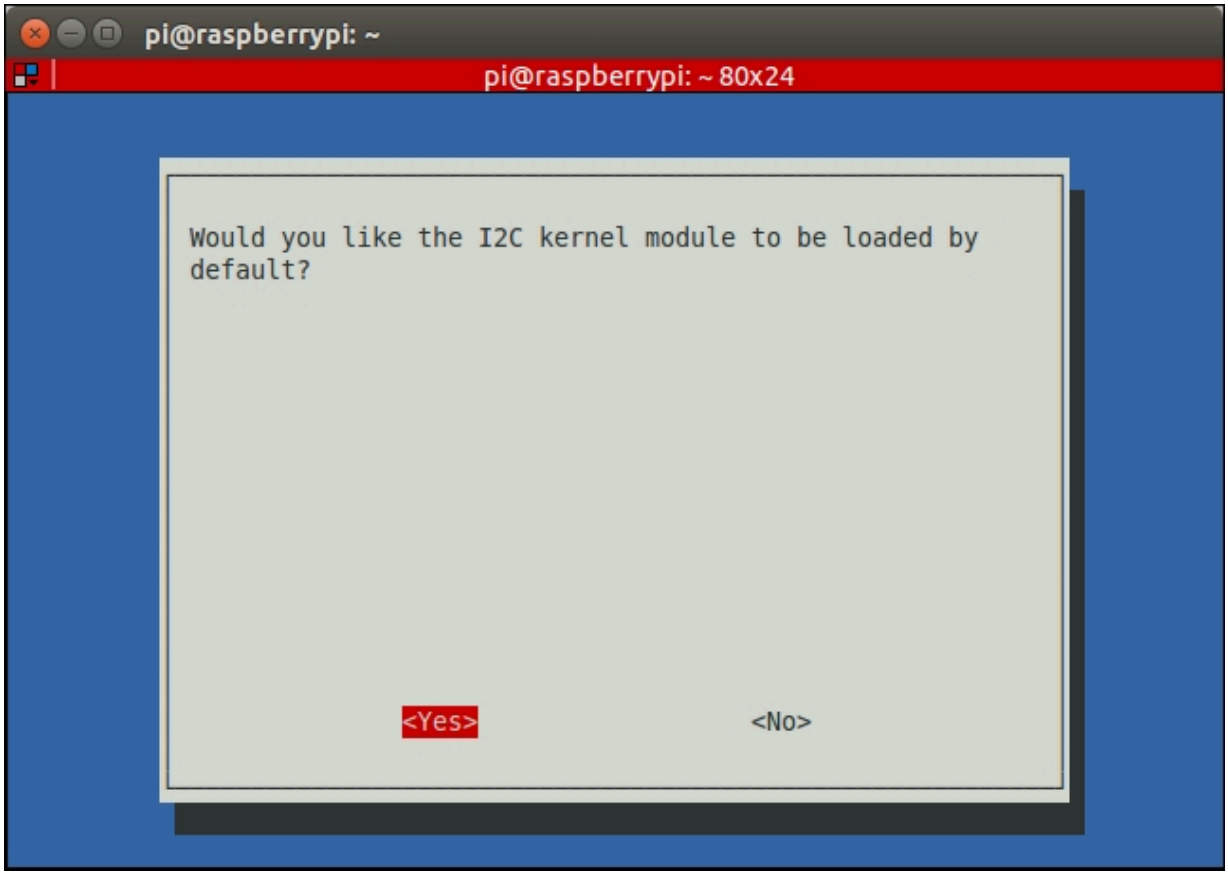
When we enter the **Advanced Options** menu of raspi-config, we are greeted by the following screen. We will now select the **A7** option, which is **I2C**.



Once we select the **I2C** menu, the following display will be shown:



Select **Yes** for both the previous screen and the following one:


pi@raspberrypi: ~
pi@raspberrypi: ~ 80x24
Would you like the I2C kernel module to be loaded by default?
<Yes>
<No>

Using PiGlow

PiGlow has a set of female GPIO railings, which cover the Raspberry Pi's male GPIO rails. In case of Raspberry Pi 2 and B+, the size of the railing in PiGlow is shorter, so we need to make sure that PiGlow is attached to only lower numbered pins on the Raspberry Pi.

> **Note**
>
> You can find out more about PiGlow on the Pimoroni website at https://shop.pimoroni.com/products/piglow.

We can now proceed to control the LEDs on the PiGlow board. Open up a Python terminal and execute the following statements in a sequential order:

```
import piglow
piglow.red(64)
```

Here, we import the particular library in Python first. The `red()` method selects all the red LEDs on the PiGlow board. It takes only one argument, the intensity of the LEDs, which can be a number from 0 to 255. You can also try to pass the number 255 to the `red()` method to see how it affects the intensity of the LED. In this case, passing 0 would mean that the LED is off. But once we execute the earlier two statements, nothing will happen. We have to update the PiGlow with our changes by issuing the following command:

```
piglow.show()
```

Why? Because if we are setting up a pattern, lighting the LED costs time and resources. However, we can use the show method to draw the changes instantaneously. If you find this strange, you can turn this off by turning auto-update on with the following command:

```
piglow.auto_update = True
```

Similar to the `red()` method used earlier, we also have functions to control different colors, namely white, blue, green, yellow, and orange.

But maybe we want to control each arm of the light rather than all the lights of a single color. Well, we can do that with the `arm()` function! Take a look:

```
piglow.arm(0, 128)
```

The arm function takes two arguments. The first is the number of the arm, and second is the intensity from 0 to 255. For example, in the earlier statement, we set the LEDs of the first arm to an intensity of value 128, which is half of the full level of brightness. Suppose we want to control arbitrary LEDs according to what we specify. We're in luck! The `set()` function lets us control individual LEDs. It can be used in the following ways:

- `set(0, 255)`: This sets LED 0 to full brightness
- `set([1, 3, 5], 200)`: This sets LEDs 1, 3, and 5 to a 200 level of brightness
- `set(0, [255, 255, 255])`: This sets three LEDs starting at 0 index to full brightness

Now that we have learned the basics of using PiGlow, we will move on to building real-life examples with it. First, we will build a very simple example that looks like the arms of PiGlow are like cycle's spokes. Cool, right? The code is given here, and an explanation is as follows:

```
import time
import piglow

i = 0

while True:
```

```
    print(i)
piglow.all(0)
piglow.set(i % 18, [15, 31, 63, 127, 255, 127, 63,
31, 15])
piglow.show()
i += 1
time.sleep(0.1)
```

Now save the program as `prog1.py` and run it with the following command:

```
python prog1.py
```

As with every Python program, we first import the dependencies required, which in this case are the time library and the PiGlow library we just installed. Next, we set up the counter to keep track of which LED we want to light up and to execute the bulk of the program we enter into an infinite `while` loop. This `while` loop will continue lighting the LEDs until we kill the program.

First, we print the status of the counter, and then we execute the `all()` function, which will refresh all the LEDs and ensure that they are turned off at the beginning. We then use the third form of the `set()` function we saw earlier. This means that starting from the zeroth LED, we light up nine LEDs in an ascending and then descending fashion. The `i % 18` will give the remainder of i divided by 18 so that the index of the LED remains within the range of 0 to 18 at all times. Then, we actually show the pattern that we set in the earlier statement via the `show()` function. Finally, we increment the counter and add a delay of 100 milliseconds.

This pattern gets repeated for every LED, and when done in quick succession, it appears like the spokes are cycling.

This was a very basic example, and now we're ready to move on to a more complicated one, which involves building a binary clock. Let's begin!

Building a binary clock

In this example, we will be building a binary clock using the club PiGlow in which each hour, minute, and second is represented by the three arms of PiGlow. The numbers are displayed in the binary format. We will now look at the code and learn what each part does:

```
import piglow
from time import sleep
from datetime import datetime

piglow.auto_update = True

show12hr = 1
ledbrightness = 10

piglow.all(0)

hourcount = 0
currenthour = 0
```

Like the previous example, we import PiGlow and the time libraries, but we also import an additional library, `datetime`, which gives us the current time. Then, we set the auto-update parameter to true so that PiGlow updates as soon as we make a change to it rather than pushing all the changes at once. We will now set some customization parameters, namely selecting between a 12-hour format or a 24-hour format and the brightness of the LEDs. With that completed, we refresh the LED to the 0 level and set the hour counters. Now we are ready to move on the meat of the code:

```
while True:
    time = datetime.now().time()
print(str(time))
hour = time.hour
min = time.minute
sec = time.second

if show12hr == 1:
    if hour > 12:
```

```
        hour = hour - 12

    if currenthour != hour:
        hourcount = hour
    currenthour = hour

    for x in range(6):
        piglow.led(13 + x, (hour & (1 << x)) *
    ledbrightness)

    for x in range(6):
        piglow.led(7 + x, (min & (1 << x)) *
    ledbrightness)

    for x in range(6):
        piglow.led(1 + x, (sec & (1 << x)) *
    ledbrightness)

    if hourcount != 0:
        sleep(0.5)
    piglow.white(ledbrightness)
    sleep(0.5)
    hourcount = hourcount - 1
    else :
        sleep(0.1)
```

The main code logic is executed inside an infinite while loop so that the program never stops.

We can now proceed to learn what it is that makes the program *tick*. First, we must determine the current time, which is done by the `datetime.now().time()` method. Then, we store each of current hour, minute, and time in a separate variable to display them on separate arms. In the next `if` statement, we merely check whether we want to display the 12-hour format, and if we do, we subtract 12 from the current hour if it is greater than 12. So, something like 15 will be represented as 3. We have two variables named `currenthour` and `hourcount`, which serve to keep the correct hour displayed on the PiGlow arm by decrementing `hourcount` whenever an hour passes. This way, the hour is decreased, which is necessary if we are using the 12 hour mode. To understand the next statements, which are

actually responsible for lighting up the LEDs, we need to understand the logic operators in Python. Some of these are given as follows:

Operation	**Description**
X and Y	Logical AND
X or Y	Logical OR
Not X	Logical NOT
X | Y	Bitwise OR
X ^ Y	Bitwise XOR
X & Y	Bitwise AND
X << Y	Bitwise Left Shift
X >> Y	Bitwise Right Shift

We are now greeted with a `for` loop. It has a very simple format as follows:

for (variable) in range(upper limit): The variable is a counter that starts from 0 and gets incremented after each iteration even though we don't explicitly do it. The upper limit, specified as the argument of the `range()` method, tells the `for` loop how many iterations are to be executed.

So, for x in range(6), we will set up x as a counter and the loop will be executed exactly six times. This is preferred since each arm has six LEDs, and each of them can be set without having six separate statements.

To understand what the following piece of code does, we will need to delve into the world of binary numbers and bitwise operations. You will also notice that in the argument of the `led()` function, we are doing a bitwise operation of the numbers. Why so? In the first iteration of the `for` loop, the value of *x* is 0, and the binary representation of 1 is 00000001. So, the bitwise shift operation will introduce no change. In the second iteration, however, the value of *x* will be 1. So, a 1-bit left shift will result in the binary number 00000010, which is the representation of integer 2. Similarly, the third iteration will result in 00000100, which is binary for 4. Now, there is also the bitwise AND operation on the variables holding the values of hours, minutes, and seconds. Consider, for example, that the current hour is 3 p.m. The binary representation will be 00000011. If we do a bitwise AND operation in the first iteration, we get 00000001. This is 1 and it is multiplied with the `ledbrightness` variable; we light up the 13 + 0 = 13th LED at a brightness of 10. This LED is the first LED of the arm that is represented by hours. Now, doing a bitwise AND in the second iteration, we have hour = 00000011 and 1 << *x* = 00000010. The result is 00000010, which is the integer 2. This means that the 14th LED will light with a brightness of 20. But in the third iteration, since 1 << *x* = 00000100, we get the integer 0 at the output, which means that the 15th LED will not light at all!

Applying the same logic if the time is 4 pm, the 13th LED will not light up, the 14th LED will not light up, but the 15th LED will light up with a

brightness of 40. Going in the same direction, we now have an intuitive understanding of how the hours, minutes, and seconds of the binary clock are represented.

Finally, we design the program to flash all the white LEDs once the hour finishes. We also add 500-ms delays on both sides of the flash so that it doesn't appear too sudden. Then, we decrement the hour count and the whole program repeats with the updated time.

Now, save the code as `prog3.py` and run it with the following command:

```
python prog3.py
```

You will now be able to see the hours, minutes, and seconds represented as binary numbers on PiGlow!

Summary

This chapter was particularly interesting and hands-on because we learned how to manipulate hardware from software! This is exactly the reason why the Raspberry Pi and other boards with GPIO pins have become so popular. They allow you to control real-life hardware applications with software.

We can create potentially unlimited applications for the Pi just by changing the sensors connected to it and tweaking the code. We can build stuff such as home automation systems, security applications, wearable devices, and many more with just this small board! In this chapter, we had the humble beginnings of such projects where we first learned to blink an LED, which wasn't very impressive by itself. Then, we learned how to control that same LED with a button, which was a bit more fun.

Then, we learned about PiGlow and how to connect it to the Raspberry Pi. We learned the various API commands that are used to control the LEDs on PiGlow, and using these commands, we built a binary clock.

With these solid foundations, we then moved on to an even more complicated undertaking, which involved a little theory and a bit of clever programming in Python, which greatly reduced our effort.

In our next chapter, we will learn how to create animated movies. We will use a more software-based approach and make animations such as stop motion.

Chapter 6. Creating Animated Movies with Raspberry Pi

In the previous chapter, we learned how to use GPIO pins to control LEDs and get input from buttons. We also learned how to control the PiGlow module with its provided Python API. In Chapter 4, *Working with Webcam and Pi Camera*, we learned how to operate Pi Camera or a general webcam to capture images and videos. In this chapter, we will combine the two concepts and learn how to create stop-motion animation. In this chapter, we will learn about the following:

- Creating a stop-motion animation using a push button
- Examining the theory behind stop-motion
- Revisiting some GPIO concepts
- Using the `ffmpeg` library to render the shots into a video

Introducing stop-motion animation

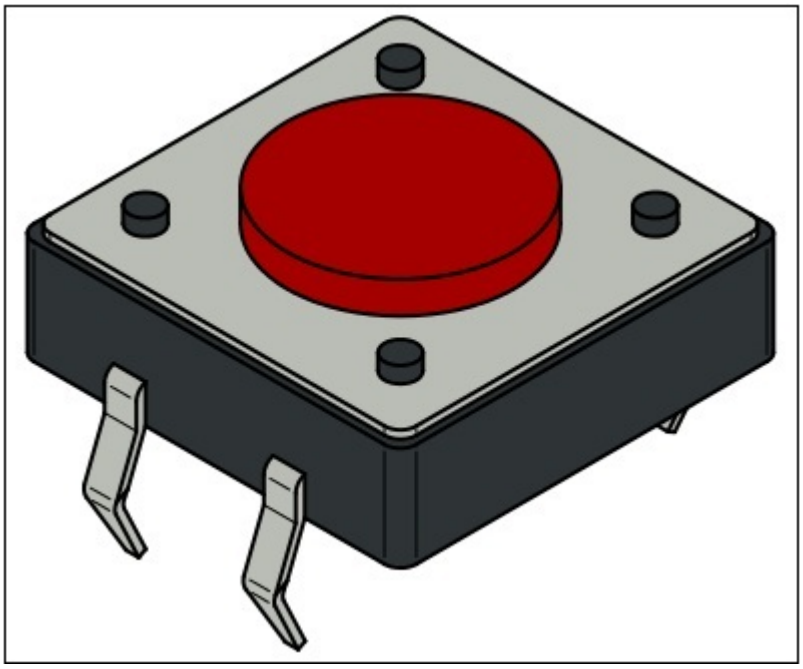
A **stop-motion animation** is basically a video of sequenced images, such as those where an object is physically manipulated so that it appears to move on its own when the video is played. The position of the object is changed incrementally after each shot so that over a course of multiple frames, it appears as though the object is moving. A common subject for stop-motion animations is a clay object, such as a bird, animal, and so on.

Setting up the prerequisites

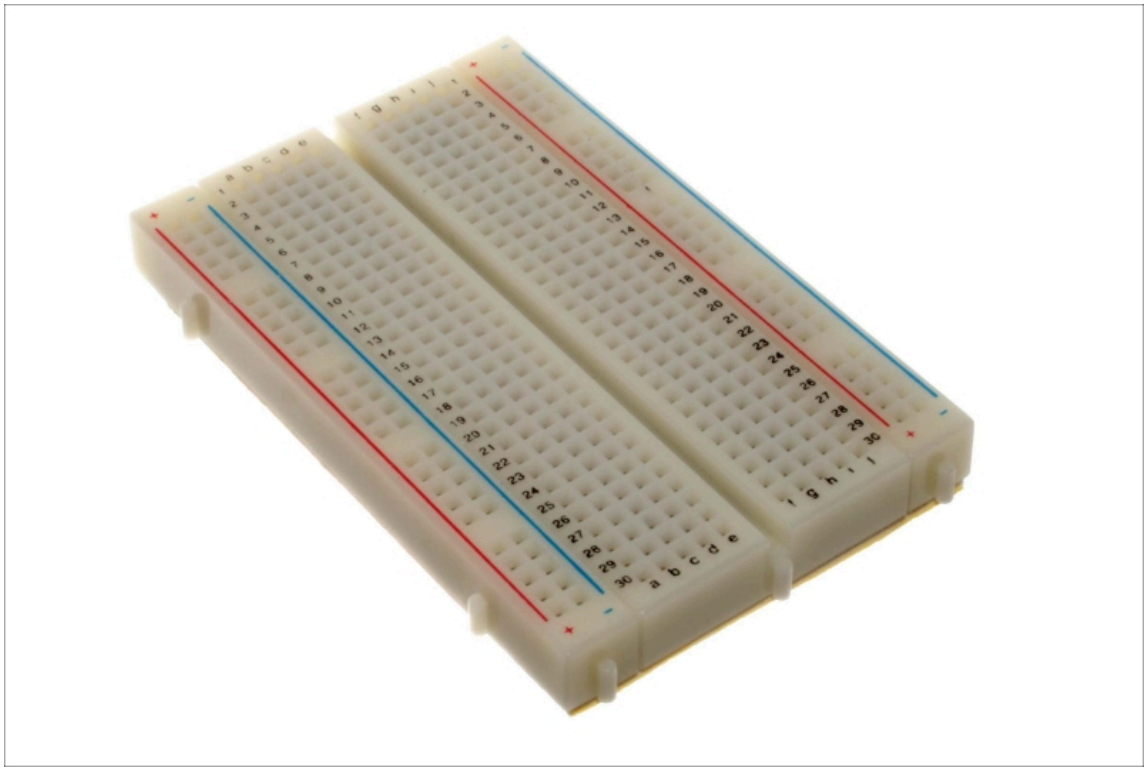
In addition to the Raspberry Pi, for this chapter we will require the following:

- Pi Camera
- Tactile push button
- Breadboard
- A few jumper cables

1. The tactile button can be purchased from your local hardware shop or online from sellers such as Adafruit. It might look something like the following:



2. The breadboard is basically a temporary connection mechanism for electronic components. Each pin in the 5-pin row is connected to each other, so they behave as a common terminal for each component connected to the same row. All the pins in the side rail are electrically connected and primarily serve as power lines. A breadboard looks like the following:



Jumper cables are nothing but wires that allow us to connect different rows in a breadboard and look like this:



We will require the `ffmpeg` library installed on our Pi. This can be installed with the following command:

```
sudo apt-get install ffmpeg
```

Once the installation is finished, you can test it by executing the following command:

```
avconv
```

If it is installed properly, then you should see some information about the `avconv` tool. If not, then try to install it properly and then run the command again.

Setting up and testing the camera

Before booting up the Pi, we need to connect Pi Camera to it. This is as simple as locating the camera port beside the Ethernet port, inserting the camera strip in it, and pushing the tab down to secure it.

You can now test to check whether this works. After booting up the Pi, issue the following command:

```
raspistill -o image.jpg
```

If everything went fine, you should now see an image in your home directory. Open it to check whether the camera took a satisfactory picture. Don't worry if it is upside down as we will correct this shortly. It will look something like this:



Now, we will look at the code to take a picture with Python. As we did earlier, we will first see the code completely and then learn what it

does statement by statement. Here it is:

```
import picamera
from time import sleep

with picamera.PiCamera() as camera:
    camera.start_preview()
    sleep(3)
    camera.capture('/home/pi/image.jpg')
    camera.stop_preview()
```

We first import the PI camera library, which is required to take pictures with PiCam. We also import the time library to provide delays in taking a picture. The next statement essentially ensures that we can access `picamera.PiCamera()` with the same camera. The `camera.start_preview()` method starts a preview screen for the camera, and after a delay of 3 seconds the `capture()` method takes a picture. The argument to the capture command gives the filename for the image file. Finally, the `stop_preview()` method stops the preview.

Save the file as `prog1.py` and run it with the following command as this is a `python3` program:

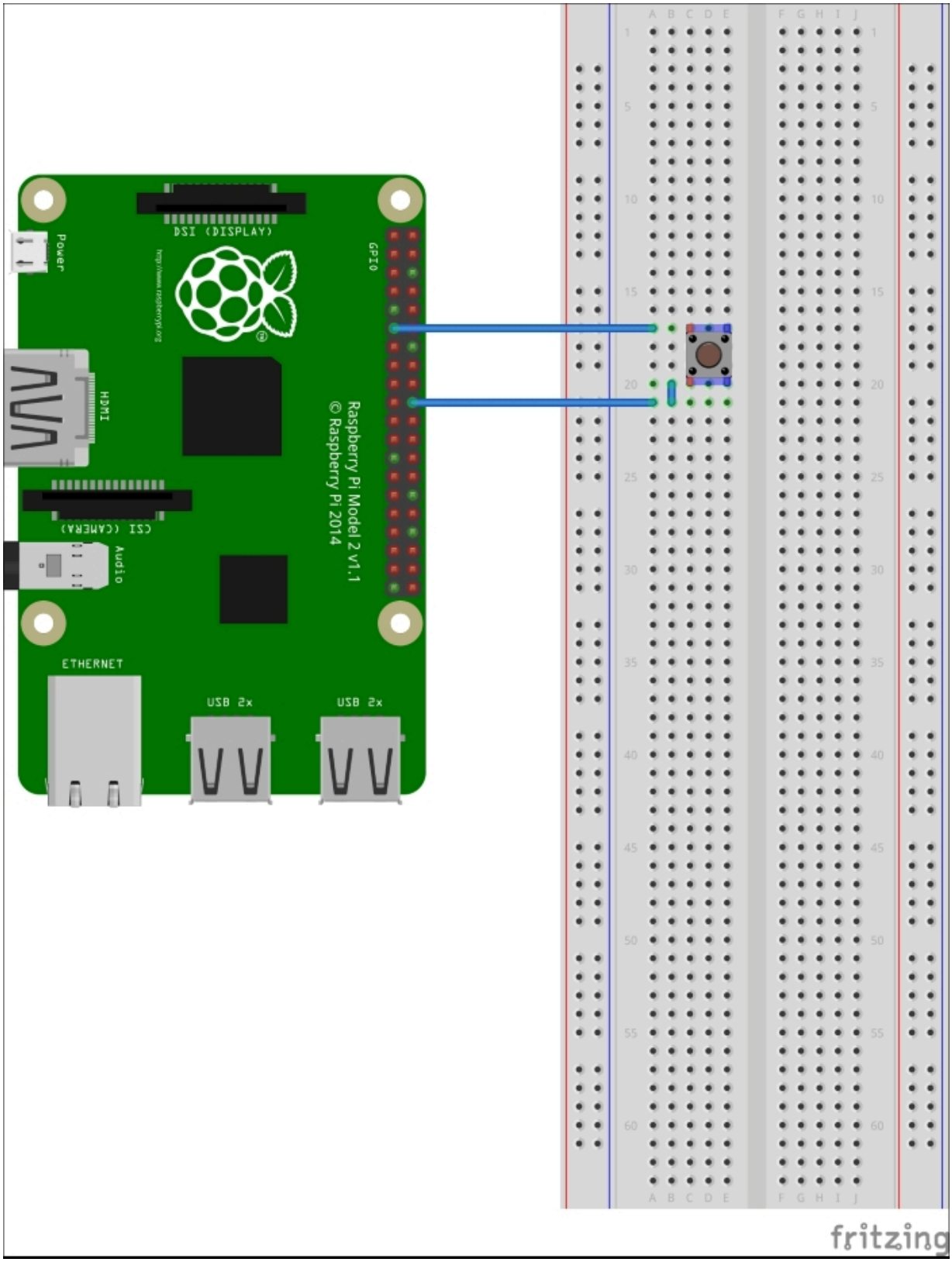
```
python3 prog1.py
```

Now, double-click on the new image file from the File Explorer and check whether it is upside down. If it is, then add the following two lines before the `start_preview()` method to correct it:

```
camera.vflip = True
camera.hflip = True
```

Adding the hardware button

To our current setup, we will add a hardware input button so that we can take pictures when we press that button. So, connect a tactile button to the Raspberry Pi, as shown in the following image:


DSI (DISPLAY)
Power
GPIO
HDMI
CSI (CAMERA)
Audio
ETHERNET
USB 2x
USB 2x
Raspberry Pi Model 2 v1.1
© Raspberry Pi 2014
fritzing

This time, we will use the BCM convention when specifying GPIO pins. We will connect the button to the GPIO17 pin. Then, we will modify the code so that it looks like the following:

```
import picamera
from RPi import GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN, GPIO.PUD_UP)

with picamera.PiCamera() as camera:
    camera.start_preview()
    GPIO.wait_for_edge(17, GPIO.FALLING)
    camera.capture('/home/pi/image.jpg')
    camera.stop_preview()
```

Save the preceding program as `prog2.py` and run it with the following command:

```
python3 prog2.py
```

We will now see what each line of code does, except those that we already saw in the previous program. To use the GPIO pins, we have to import the `RPi.GPIO` library, as we did in the previous chapter. Then, we set the GPIO addressing mode as BCM, which uses the Broadcom convention rather than numbering the pins on the board. This means that we can address the GPIO pin by its GPIO number rather than its board number. With the `setup()` method, we configure the pin to be an input and set it so that it is pulled up by default.

The function that actually does the magic is `wait_for_edge()`. It takes the pin number as the first argument and `GPIO_FALLING` as the second. This means that the function will be triggered only when the signal at pin 17 goes from high to low or when it falls. So, it will be triggered when the button goes from the pressed state to the unpressed state; that is, it will be triggered when we leave the button rather than when we press the button. The next statements capture and save the image and then exit the preview. If we want to take a selfie, then we can add a delay after the `wait_for_edge()` function in

order to enable us to get into position. Also, don't forget to add the two commands to correct an upside down picture if such a need arises.

Now that we have successfully taken individual photos without a camera, it's time to try taking a series of still images and combining them in order to make a stop-motion animation. Just so that your home folder does not just get messy, we can create a new folder and enter it with the following commands:

```
mkdir stopmotion
cd stopmotion
```

Modify the given code and add a loop to take a picture every time the button is pressed. The new code looks like the following:

```
import picamera
from RPi import GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN, GPIO.PUD_UP)

with picamera.PiCamera() as camera:
    camera.start_preview()
    frame = 1
    while True:
        GPIO.wait_for_edge(17, GPIO.FALLING)

camera.capture('/home/pi/stopmotion/frame%03d.jpg'
% frame)
        frame += 1
    camera.stop_preview()
```

What we have done is create an infinite while loop that keeps taking photographs as long as the button is pressed. The frame variable keeps track of the current frame and saves the corresponding image in the folder.

Note

But be careful about the space limitations of your SD card. If we hold down the button for long, the Pi could run out of space and crash once it run out of space, causing us to lose all the data and potentially corrupting the SD card.

Now, save the program as `prog3.py` and run it with the following command:

```
python3 prog3.py
```

Run the program and collect as many images as required for our stop-motion project. Press *Ctrl + C* to exit the program and open the File Manager to see your images.

Rendering the video

We are now ready to render the video from the sequence of images we just collected. We will begin rendering the video by entering the terminal window. Navigate to the stopmotion folder with the following command:

```
cd /home/pi/stopmotion
```

Then, execute the following command to begin rendering the video:

```
avconv -r 10 -qscale 2 -i frame%03d.jpg
animation.mp4
```

We will now see what each part of this command does:

- `-r` specifies the frame rate for the video, which is currently set at 10
- `-qscale` specifies the quality for the video and can range from 2-5
- `-i` specifies the input file

Once the rendering is complete, you can play it with the following command:

```
omxplayer animation.mp4
```

Congratulations! You have just created your own stop-motion animation creator! Go ahead and try it out. Build your own interesting animations and upload them on YouTube.

Summary

In this chapter, we revised how to use Pi Camera with the Raspberry Pi and take photos with it. We learned a neat trick to correct our photos if they are upside down. We also learned how to use a GPIO to use a tactile button to act as the trigger for our images.

Once we obtained the images required for our animation, we proceeded to combine those images in a video, where we specified the frame rate and quality. Finally, we rendered the video using the `ffmpeg` library.

In the next chapter, we will learn about the popular open source image processing library, OpenCV, and get it set up on the Raspberry Pi. We will also implement some interesting projects, such as video processing, logical operations on an image, colorspace conversions, and much more!

Chapter 7. Introduction to Computer Vision

In the previous chapter, we implemented a battery-operated portable Pi time-lapse box and a stop motion recording system. In this chapter, we will cover the basics of computer vision with Pi using the OpenCV library. OpenCV is a simple yet powerful tool for any computer vision enthusiast. One can learn about computer vision in an easy way by writing OpenCV programs in Python. Using a Raspberry Pi computer and Python for OpenCV programming is one of the best ways to start your journey in the world of computer vision. We will cover the following topics in detail in this chapter:

- Introducing computer vision
- Introducing OpenCV
- Setting up Pi for computer vision and NumPy
- Image basics in OpenCV
- Webcam video processing with OpenCV
- Arithmetic and logical operations on images
- Colorspace and the conversion of colorspace
- Object tracking based on colors

Introducing Computer Vision

Computer vision is an area of computer science, mathematics, and electrical engineering. It includes ways to acquire, process, analyze, and understand images and videos from the real world in order to mimic human vision. Also, unlike human vision, computer vision can also be used to analyze and process depth and infrared images. Computer vision is also concerned with the theory of information extraction from images and videos. A computer vision system can accept different forms of data as an input, including—but not limited to—images, image sequences, and videos that can be streamed from multiple sources to further process and extract useful

information from it for decision making. Artificial intelligence and computer vision share many topics, such as image processing, pattern recognition, and machine learning techniques.

Introducing OpenCV

OpenCV (short for **Open Source Computer Vision**) is a library of programming functions for computer vision. It was initially developed by the Intel Russia research center in Nizhny Novgorod, and it is currently maintained by Itseez.

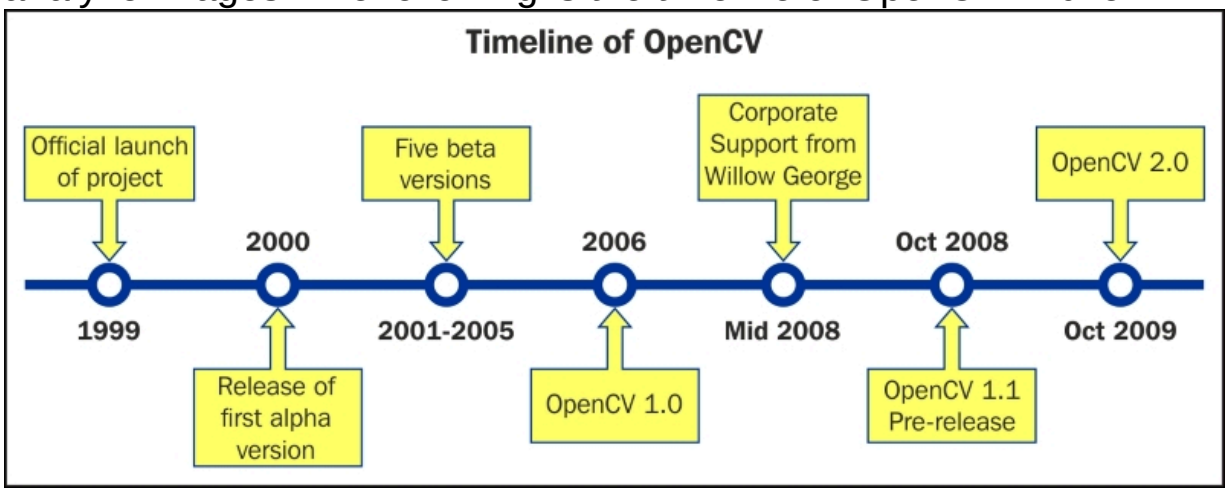
> ## Note
>
> You can read more about Itseez at http://itseez.com/.

This is a cross-platform library, which means that it can be implemented and operated on different operating systems. It focuses mainly on image and video processing. In addition to this, it has several GUI and event handling features for the user's convenience.

OpenCV was released under a **Berkeley Software Distribution** (**BSD**) license, and hence, it is free for both academic and commercial use. It has interfaces for popular programming languages, such as C/C++, Python, and Java, and it runs on a variety of operating systems, including Windows, Android, and Unix-like operating systems.

> ## Note
>
> You can explore the OpenCV homepage, www.opencv.org, for further details.

OpenCV was initially an Intel Research initiative to develop tools to analyze images. The following is the timeline of OpenCV in brief:



In August 2012, support for OpenCV was taken over by a nonprofit foundation, www.OpenCV.org, which is currently developing it further. It also maintains a developer and user site for OpenCV.

Note

At the time of writing this, the stable version of OpenCV is 2.4.10. Version 3.0 Beta is also available.

Setting up Pi for Computer Vision

Make sure that you have a working, wired Internet connection with reasonable speed for this activity. Now, let's prepare our Pi for computer vision:

1. Connect your Pi to the Internet through Ethernet or a Wi-Fi USB dongle.
2. Run the following command to restart the networking service:

```
sudo service networking restart
```

- Make sure that Raspberry Pi is connected to the Internet by typing in the following command:

```
ping –c4 www.google.com
```

If the command fails, then check the Internet connection with some other device and resolve the issue. After that, repeat the preceding steps again.

- Run the following commands in a sequence:

```
sudo apt-get update
sudo apt-get upgrade
sudo rpi-update
sudo reboot –h now
```

- After this, we will need to install a few necessary packages and dependencies for OpenCV. The following is the list of packages we need to install. You just need to connect your Pi to the Internet and type this in:

```
sudo apt-get install <package-name> -y
```

Here, `<package-name>` is one of the following packages:

libopencv-dev	libpng3	libdc1394-22-dev
build-essential	libpnglite-dev	libdc1394-22
libavformat-dev	zlib1g-dbg	libdc1394-utils
x264	zlib1g	libv4l-0
v4l-utils	zlib1g-dev	libv4l-dev
ffmpeg	pngtools	libpython2.6
libcv2.3	libtiff4-dev	python-dev
libcvaux2.3	libtiff4	python2.6-dev
libhighgui2.3	libtiffxx0c2	libgtk2.0-dev
libpng++-dev	libtiff-tools	libunicap2-dev
opencv-doc	libjpeg8	libeigen3-deva
libcv-dev	libjpeg8-dev	libswscale-dev

libcvaux-dev	libjpeg8-dbg	libjpeg-dev
libhighgui-dev	libavcodec-dev	libwebp-dev
python-numpy	libavcodec53	libpng-dev
python-scipy	libavformat53	libtiff5-dev
python-matplotlib	libgstreamer0.10-0-dbg	libjasper-dev
python-pandas	libgstreamer0.10-0	libopenexr-dev
python-nose	libgstreamer0.10-dev	libgdal-dev
v4l-utils	libxine1-ffmpeg	python-tk
libgtkglext1-dev	libxine-dev	python3-dev
libpng12-0	libxine1-bin	python3-tk
libpng12-dev	libunicap2	python3-numpy

For example, you have to install x264, then you will need to to type the following:

```
sudo apt-get install x264 -y
```

This will install the necessary package. Similarly, install all the previously mentioned packages. If a package is already installed on your Pi, then it will show the following message:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
x264 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 0
not upgraded.
```

In this case, don't worry. This package is already installed and comes with its newest version. Just proceed with installing all the other packages in the list one by one.

- Finally, install OpenCV for Python with this:

```
sudo apt-get install python-opencv -y
```

This is the easiest way to install OpenCV for Python; however, there is a problem with this. Raspbian repositories may not always contain the latest version of OpenCV. For example, at the time of writing this, Raspbian repository contains 2.4.1, while the latest OpenCV version is 2.4.10. With respect to the Python API, the latest version will always contain much better support and more functionality.

For the convenience of the readers, all these commands are included in an executable shell script, `chapter07.sh`, in the code bundle. Just run the script with the following command:

```
./chapter07.sh
```

This will install all the required packages and dependencies to get started with OpenCV on Pi.

> **Note**
>
> Another method to do the same is to compile OpenCV from the source, which I will not recommend for beginners as it's a bit complex and will take a lot of time.

Testing the OpenCV installation with Python

In Python, it's very easy to code for OpenCV. It requires very few lines of code compared to C/C++, and powerful libraries such as NumPy can be exploited for multidimensional data structures required for image processing.

Open a terminal and type `python`, and then type the following lines:

```
>>> import cv2
>>> print cv2.__version__
```

This will show us the version of OpenCV installed on the Pi, which is 2.4.1 in our case.

Introducing NumPy

NumPy is the fundamental package used for scientific computing with Python and it is matrix library for linear algebra. NumPy can also be used as an efficient multidimensional container of generic data. Arbitrary datatypes can be defined and used. NumPy is an extension to the Python programming language, adding support for large, multidimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. We will be using NumPy arrays throughout this book in order to represent images and carry out complex mathematical operations on them. NumPy comes with many built-in functions for all these operations so that we do not have to worry about all the basic array operations. We can directly focus on the concepts and code for computer vision. All OpenCV array structures are converted to and from Numpy arrays. So, whatever operations you can compute in Numpy, we can process them with OpenCV.

In this book, we will be using NumPy with OpenCV a lot. Let's start with some simple example programs that will demonstrate the real power of NumPy.

Open `python` in the terminal and try out the upcoming examples.

Array creation

Let's look at some examples of array creation. `array()method` is used very frequently in the remainder of the book. There are many ways to create arrays of different types. We will explore these ways as and when required throughout the remainder of this book:

```
>>> import numpy as np
>>> x=np.array([1,2,3])
>>> x
array([1, 2, 3])

>>> y=range(10)
```

```
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Basic operations on arrays

We are going to learn about a `linspace()`function now. It takes three parameters: `start_num`, `end_num`, and `count`. This creates an array with equally spaced points starting with `start_num` and ending with `end_num`. Try out the following example:

```
>>> a=np.array([1,3,6,9])
>>> b=np.linspace(0,15,4)
>>> c=a-b
>>> c
array([ 1., -2., -4., -6.])
```

The following is the code to calculate the square of every element in an array:

```
>>> a**2
array([ 1,  9, 36, 81])
```

Linear algebra

Let's explore some linear algebra examples. We will look at the `transpose()`, `inv()`, `solve()`, and `dot()` functions, which are useful for linear algebra:

```
>>> a=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a.transpose()
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])

>>> np.linalg.inv(a)
array([[ -4.50359963e+15,   9.00719925e+15,
-4.50359963e+15],
       [  9.00719925e+15,  -1.80143985e+16,
9.00719925e+15],
       [ -4.50359963e+15,   9.00719925e+15,
```

```
-4.50359963e+15]])

>>> b=np.array([3,2,1])
>>> np.linalg.solve(a,b)
array([ -9.66666667,  15.33333333,  -6.        ])

>>> c= np.random.rand(3,3)
>>> c
array([[ 0.69551123,  0.18417943,  0.0298238 ],
       [ 0.11574883,  0.39692914,  0.93640691],
       [ 0.36908272,  0.53802672,  0.2333465 ]])
>>> np.dot(a,c)
array([[ 2.03425705,  2.59211786,  2.60267713],
       [ 5.57528539,  5.94952371,  6.20140877],
       [ 9.11631372,  9.30692956,  9.80014041]])
```

Note

You can explore NumPy in detail at http://www.numpy.org/.

Working with images

Let's get started with the basics of OpenCV's Python API. All the scripts we will write and run will use the OpenCV library, which must be imported with the `import cv2` line. We will import few more libraries as required, and in the next sections and chapters, `cv2.imread()` will be used to import an image. It takes two arguments. The first argument is the image filename. The image should be in the same directory where the Python script is the absolute path that should be provided to `cv2.imread()`. It reads images and saves them as NumPy arrays.

The second argument is a flag that specifies that the mode image should be read. The flag can have the following values:

- `cv2.IMREAD_COLOR`: This loads a color image; it is the default flag
- `cv2.IMREAD_GRAYSCALE`: This loads an image in the grayscale mode
- `cv2.IMREAD_UNCHANGED`: This loads an image as it includes an alpha channel

The numeric values of the preceding flags are *1*, *0*, and *-1*, respectively.

Take a look at the following code:

```
import cv2 #This imports opencv
#This reads and stores image in color into
variable img
img =
cv2.imread('lena_color_512.tif',cv2.IMREAD_COLOR)
```

Now, the last line in the preceding code is the same as this:

```
img = cv2.imread('lena_color_512.tif',1)
```

We will be using the numeric values of this flag throughout the book.

The following code is used to display the image:

```
cv2.imshow('Lena',img)
cv2.waitKey(0)
cv2.destroyWindow('Lena')
```

The `cv2.imshow()` function is used to display an image. The first argument is a string that is the window name, and the second argument is the variable that holds the image that is to be displayed.

`cv2.waitKey()` is a keyboard function. Its argument is the time in milliseconds. The function waits for specified milliseconds for any keyboard key press. If 0 is passed, it waits indefinitely for a key press. It is the only method to fetch and handle events. We must use this for `cv2.imshow()` or no image will be displayed on screen.

`cv2.destroyWindow()` function takes a window name as a parameter and destroys that window. If we want to destroy all the windows in the current program, we can use `cv2.destroyAllWindows()`.

We can also create a window with a specific name in advance and assign an image to that window later. In many cases, we will have to create a window before we have an image. This can be done using the following code:

```
cv2.namedWindow('Lena', cv2.WINDOW_AUTOSIZE)
cv2.imshow('Lena',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Putting it all together, we have the following script:

```
import cv2
img = cv2.imread('lena_color_512.tif',1)
cv2.imshow('Lena',img)
cv2.waitKey(0)
cv2.destroyWindow('Lena')
```

To summarize, the preceding script imports an image, displays it, and waits for the keystroke to close the window. The screenshot is as follows:



The `cv2.imwrite()` function is used to save an image to a specific path. The first argument is the name of the file and second is the variable pointing to the image we want to save. Also, `cv2.waitKey()` can be used to detect specific keystrokes. Let's test the usage of both the functions in the following code snippet:

```
import cv2
img = cv2.imread('lena_color_512.tif', 1)
cv2.imshow('Lena', img)
```

```
keyPress = cv2.waitKey(0)
if keyPress == ord('q'):
    cv2.destroyWindow('Lena')
elif keyPress == ord('s'):
cv2.imwrite('output.jpg', img)
cv2.destroyWindow('Lena')
```

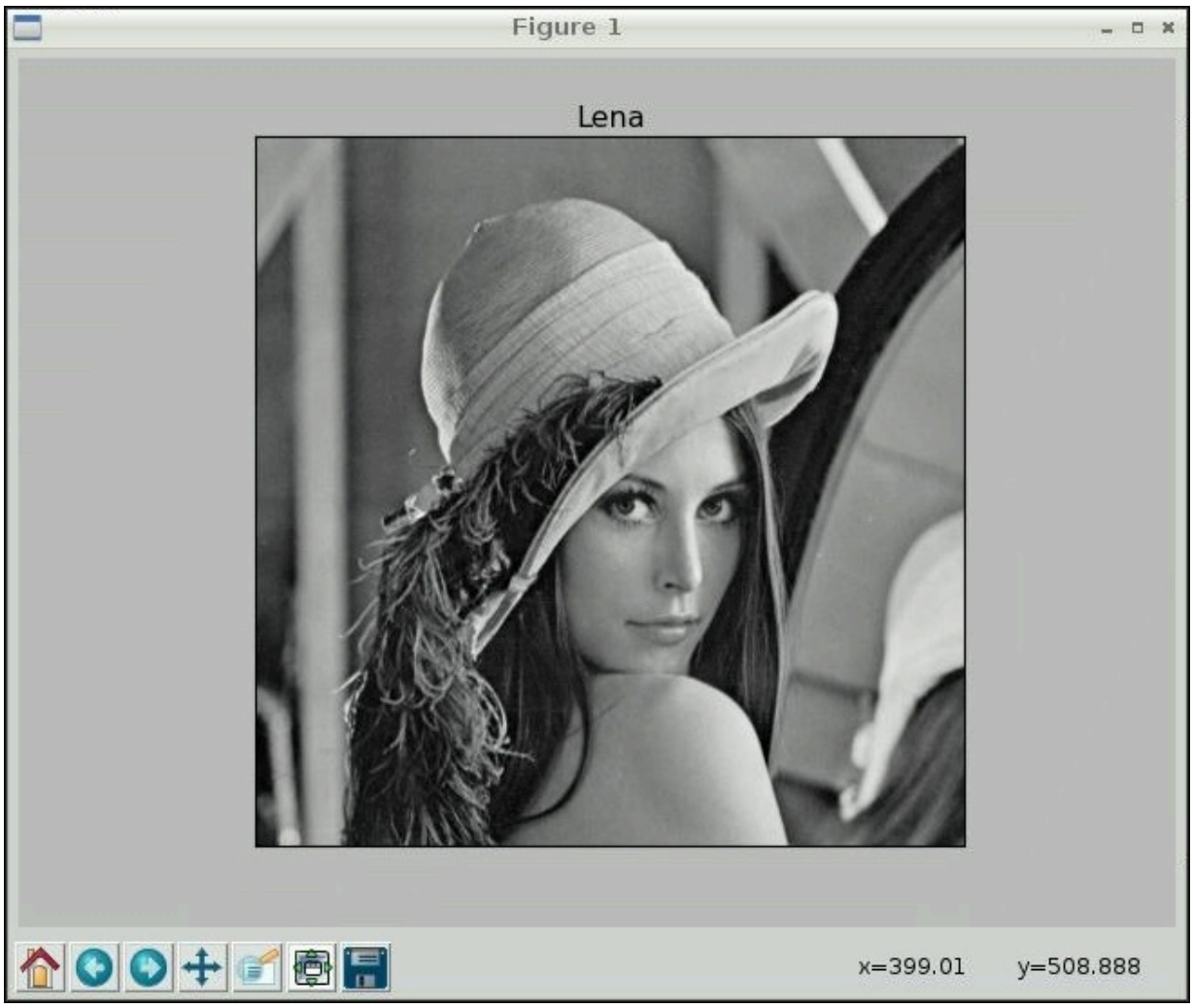
Here, `keyPress = cv2.waitKey(0)` is used to save the value of the keystroke in the `keyPress` variable. Given a string of length one, `ord()` returns an integer representing the Unicode code point of the character when the argument is a Unicode object or the value of the byte when the argument is an 8-bit string. Based on `keyPress`, we either exit or exit after saving the image. For example, if the *Esc* key is pressed, the `cv2.waitKey()`function will return *27*.

Using matplotlib

We can also use `matplotlib` to display images. `matplotlib` is a 2D plotting library for Python. It provides a wide range of plotting options, which we will be using in the next chapter. Let's look at a basic example of `matplotlib`:

```
import cv2
import matplotlib.pyplot as plt
#Program to load a color image in gray scale and
to display using matplotlib
img = cv2.imread('lena_color_512.tif',0)
plt.imshow(img,cmap='gray')
plt.title('Lena')
plt.xticks([])
plt.yticks([])
plt.show()
```

In this example, we are reading an image in grayscale and displaying it using `matplotlib`. The following screenshot shows the plot of the image:



The `plt.xticks([])` and `plt.yticks([])` functions can be used to disable *x* and *y* axis. Run the preceding code again, and this time, comment out the two lines with the `plt.xticks([])` and `plt.yticks([])` functions.

The `cv2.imread()` OpenCV function reads images and saves them as NumPy arrays of Blue, Green, and Red (BGR) pixels.

However, `plt.imshow()` displays images in the RGB format. So, if we read an image as it is with `cv2.imread()` and display it using `plt.imshow()`, then the value for blue will be treated as the value for red and vice versa by `plt.imshow()`, and it will display an image with distorted colors. Try out the preceding code with the following alterations in the respective lines to experience the concept:

```
img = cv2.imread('lena_color_512.tif',1)
plt.imshow(img)
```

To remedy this issue, we need to convert an image read in the BGR format into an RGB array format by `cv2.imread()`so that `plt.imshow()` will be able to render it in a way that makes sense to us. We will be using the `cv2.cvtColor()` function for this, which we will learn soon.

Note

Explore this URL to get more information on `matplotlib`: http://matplotlib.org/.

Working with Webcam using OpenCV

OpenCV has a functionality to work with standard USB webcams. Let's take a look at an example to capture an image from a webcam:

```
import cv2

# initialize the camera
cam = cv2.VideoCapture(0)
ret, image = cam.read()

if ret:
    cv2.imshow('SnapshotTest',image)
    cv2.waitKey(0)
    cv2.destroyWindow('SnapshotTest')

cv2.imwrite('/home/pi/book/output/SnapshotTest.jpg
',image)
cam.release()
```

In the preceding code, `cv2.VideoCapture()` creates a video capture object. The argument for it can either be a video device or a file. In this case, we are passing a device index, which is 0. If we have more cameras, then we can pass the appropriate device index based on what camera to choose. If you have one camera, just pass 0.

You can find out the number of cameras and associated device indexes using the following command:

```
ls -l /dev/video*
```

Once `cam.read()` returns a Boolean value `ret` and the `frame` which is the image it captured. If the image capture is successful, then `ret` will be *True*; otherwise, it will be False. The previously listed code will capture an image with the camera device, `/dev/video0`, display it, and then save it. `cam.release()` will release the device.

This code can be used with slight modifications to display live video stream from the webcam:

```
import cv2

cam = cv2.VideoCapture(0)
print 'Default Resolution is ' +
str(int(cam.get(3))) + 'x' + str(int(cam.get(4)))
w=1024
h=768
cam.set(3,w)
cam.set(4,h)
print 'Now resolution is set to ' + str(w) + 'x' +
str(h)

while(True):
    # Capture frame-by-frame
    ret, frame = cam.read()

    # Display the resulting frame
    cv2.imshow('Video Test',frame)

    # Wait for Escape Key
    if cv2.waitKey(1) == 27 :
        break

# When everything done, release the capture
cam.release()
cv2.destroyAllWindows()
```

You can access the features of the video device with `cam.get(propertyID)`. `3` stands for the width and `4` stands for the height. These properties can be set with `cam.set(propertyID, value)`.

The preceding code first displays the default resolution and then sets it to 1024 x 768 and displays the live video stream till the *Esc* key is pressed. This is the basic skeleton logic for all the live video processing with OpenCV. We will make use of this in future.

Saving a video using OpenCV

We need to use the `cv2.VideoWriter()` function to write a video to a file. Take a look at the following code:

```
import cv2
cam = cv2.VideoCapture(0)
output = cv2.VideoWriter('VideoStream.avi',
cv2.cv.CV_FOURCC(*'WMV2'),40.0,(640,480))

while (cam.isOpened()):
    ret, frame = cam.read()
    if ret == True:
        output.write(frame)
        cv2.imshow('VideoStream', frame )
        if cv2.waitKey(1) == 27 :
            break
    else:
        break

cam.release()
output.release()
cv2.destroyAllWindows()
```

In the preceding code, `cv2.VideoWriter()` accepts the following parameters:

- **Filename**: This is the name of the video file.
- **FourCC**: This stands for **Four Character Code**. We have to use the `cv2.cv.CV_FOURCC()`function for this. This function accepts FourCC in the *'code' format. This means that for DIVX, we need to pass *'DIVX', and so on. Some supported formats are DIVX, XVID, H264, MJPG, WMV1, and WMV2.

> ## Note
>
> You can read more about FourCC at www.fourcc.org.

- **Framerate**: This is the rate of the frames to be captured per second.

- **Resolution**: This is the resolution of the video to be captured.

The preceding code records the video till the *Esc* key is pressed and saves it in the specified file.

Pi Camera and OpenCV

The following code demonstrates the use of Picamera with OpenCV. It shows a preview for 3 seconds, captures an image, and displays it on screen using `cv2.imshow()`:

```
import picamera
import picamera.array
import time
import cv2

with picamera.PiCamera() as camera:
    rawCap=picamera.array.PiRGBArray(camera)
    camera.start_preview()
    time.sleep(3)
    camera.capture(rawCap,format="bgr")
    image=rawCap.array
cv2.imshow("Test",image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Retrieving image properties

We can retrieve and use many image properties with OpenCV functions. Take a look at the following code:

```
import cv2
img = cv2.imread('lena_color_512.tif',1)
print img.shape
print img.size
print img.dtype
```

The `img.shape` operation returns the shape of the image, that is, its dimensions and the number of color channels. The output of the previously listed code will be as follows:

```
(512, 512, 3)
786432
uint8
```

If the image is colored, then `img.shape` returns a triplet containing the number of rows, the number of columns, and the number of channels in the image. Usually, the number of channels is three, representing the red, green, and blue channels. If the image is grayscale, then `img.shape` only returns the number of rows and the number of columns. Try to modify the preceding code to read the image in the grayscale mode and observe the output of `img.shape`.

The `img.size` operation returns the total number of pixels, and `img.dtype` returns the image datatype.

Arithmetic operations on images

In this section, we will take a look at the various arithmetic operations that can be performed on images. Images are represented as matrices in OpenCV. So, arithmetic operations on images are the same as arithmetic operations on matrices. Images must be of the same size in order to perform arithmetic operations with images, and these operations are performed on individual pixels .`cv2.add()` method is used to add two images, where images are passed as parameters.

The `cv2.subtract()` method is used to subtract one image from another.

Note

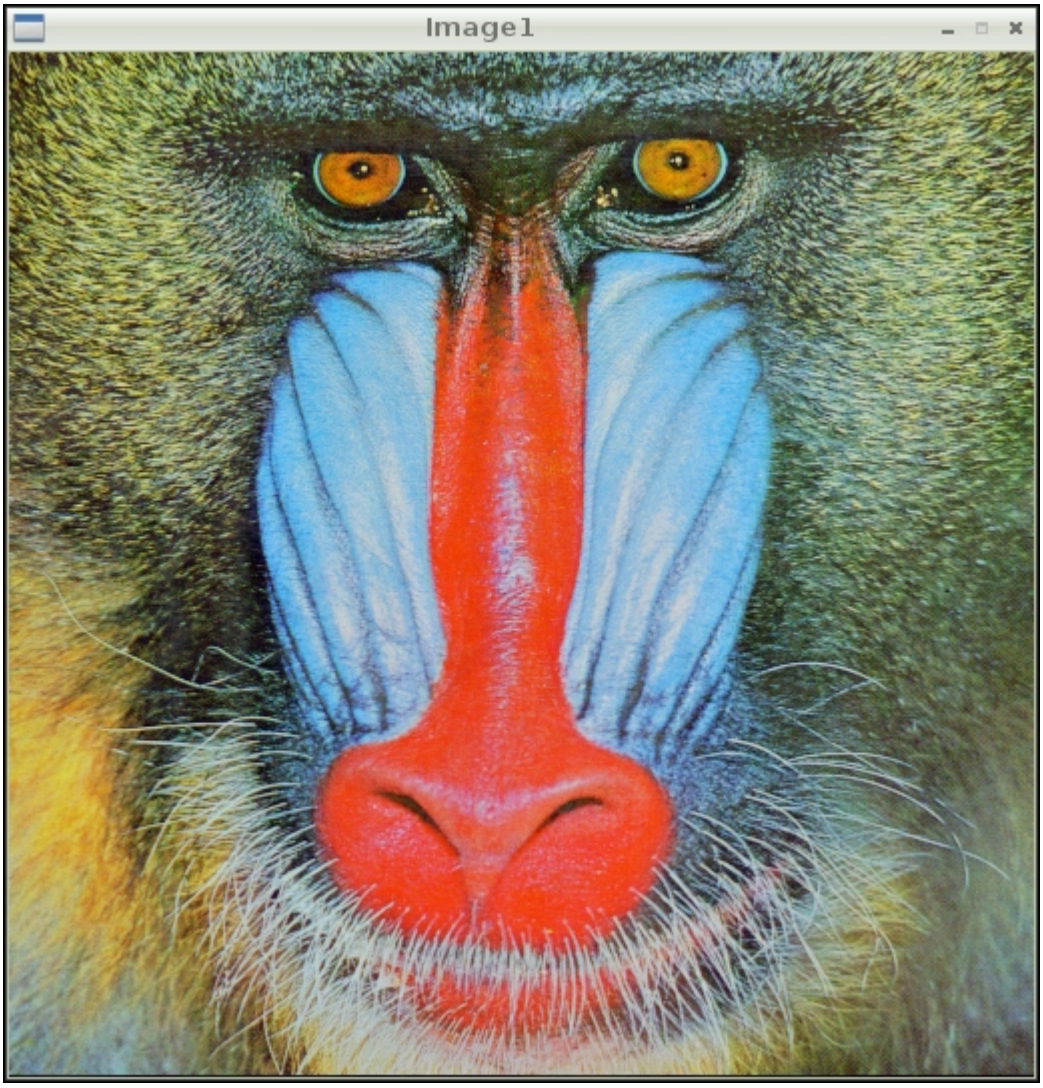
We know that subtraction operation is not commutative; so, `cv2.subtract(img1,img2)` and `cv2.(img2,img1)` will yield different results, whereas `cv2.add(img1,img2)` and `cv2.add(img2,img1)` will yield the same result as the addition operation is commutative. Both the images have to be of the same size and type as that explained earlier.

Check out the following code:

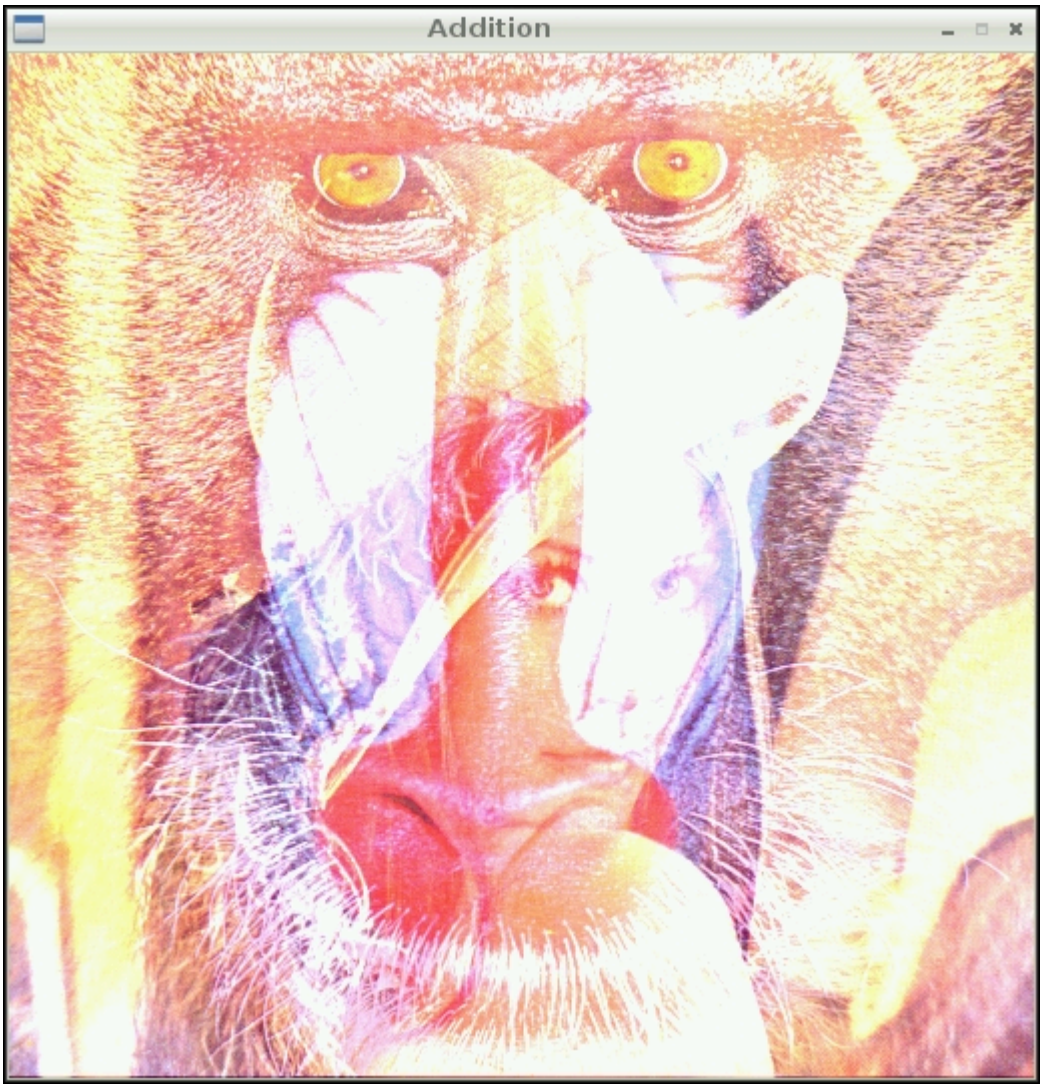
```
import cv2
img1 = cv2.imread('4.2.03.tiff',1)
img2 = cv2.imread('4.2.04.tiff',1)
cv2.imshow('Image1',img1)
cv2.waitKey(0)
cv2.imshow('Image2',img2)
cv2.waitKey(0)
cv2.imshow('Addition',cv2.add(img1,img2))
cv2.waitKey(0)
```

```
cv2.imshow('Image1-
Image2',cv2.subtract(img1,img2))
cv2.waitKey(0)
cv2.imshow('Image2-
Image1',cv2.subtract(img2,img1))
cv2.waitKey(0)
cv2.destroyAllWindows()
```

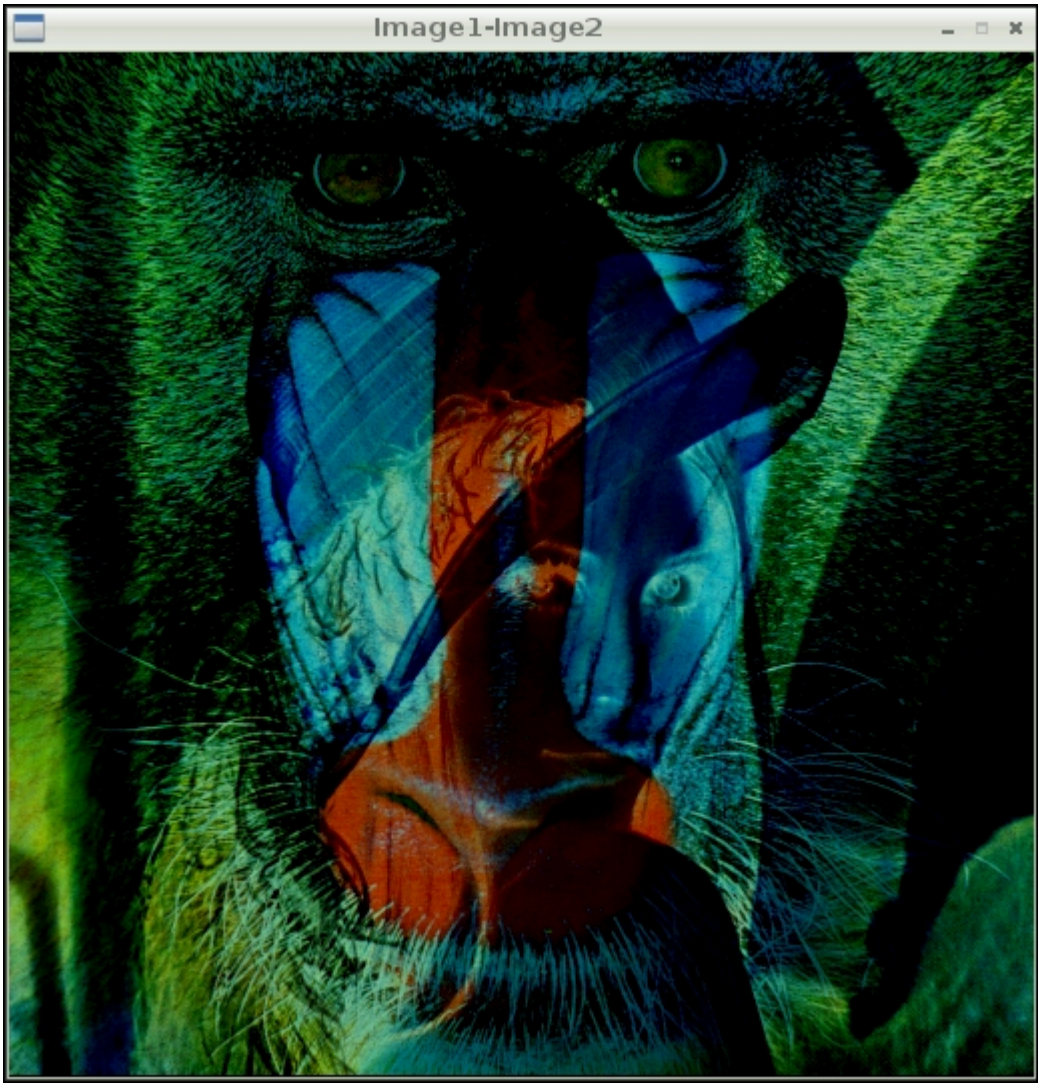
The preceding code demonstrates the usage of arithmetic functions on images. Image2 is the same *Lena* image that we experimented with in the previous chapter, so I am not including its output window. The following is the output window of **Image1**:



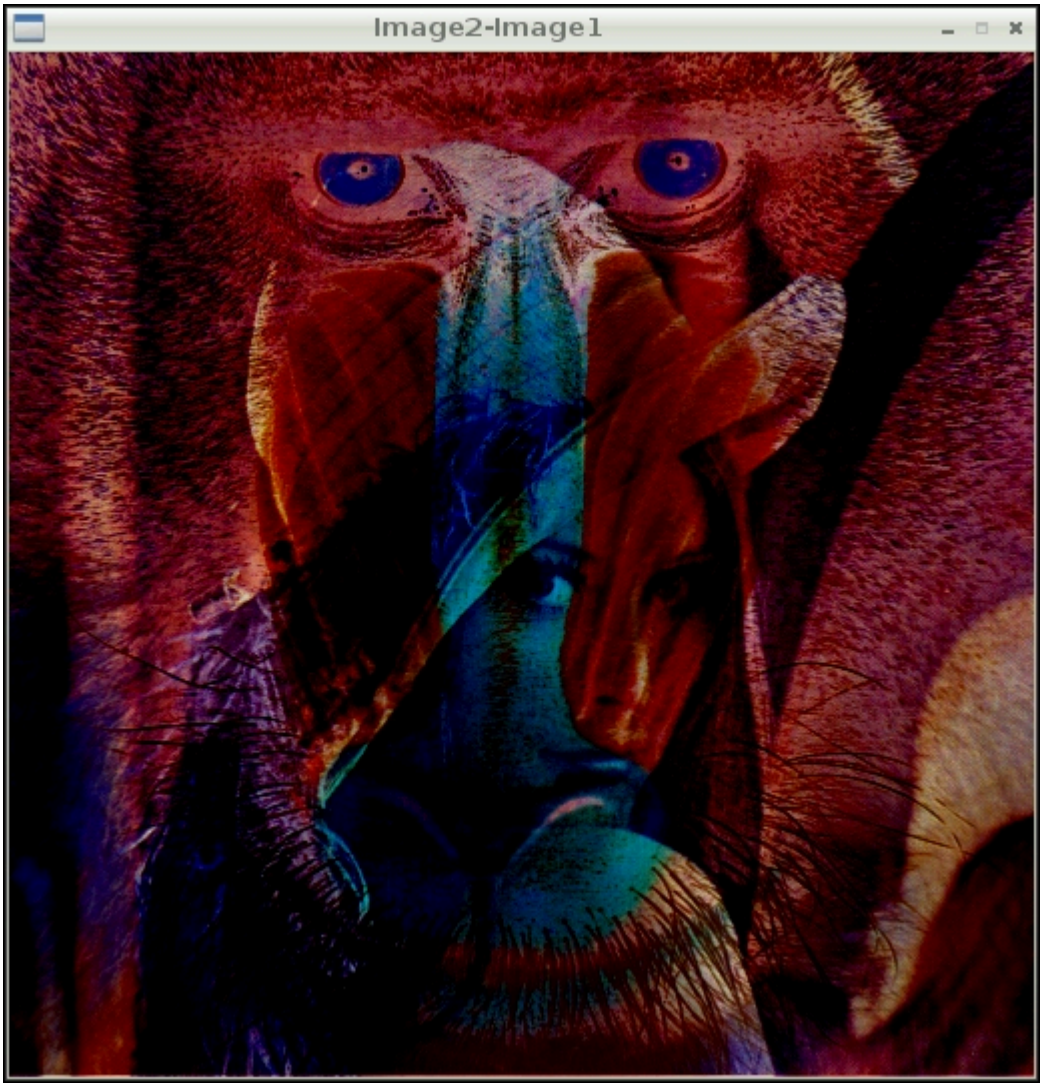
The following is the output of the **Addition**:



The following is the output window of **Image1**-**Image2**:



The following is the output window if **Image2**-**Image1**:


Image2-Image1

Splitting and merging image color channels

On several occasions, we might be interested in working separately with the red, green, and blue channels. For example, we might want to build a histogram for each channel of an image.

The `cv2.split()` method is used to split an image into three different intensity arrays for each color channel, whereas `cv2.merge()` is used to merge different arrays into a single multichannel array, that is, a color image. Let's take a look at an example:

```
import cv2
img = cv2.imread('4.2.03.tiff',1)
b,g,r = cv2.split (img)
cv2.imshow('Blue Channel',b)
cv2.imshow('Green Channel',g)
cv2.imshow('Red Channel',r)
img=cv2.merge((b,g,r))
cv2.imshow('Merged Output',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The preceding program first splits the image into three channels (blue, green, and red) and then displays each one of them. The separate channels will only hold the intensity values of that color, and they will be essentially displayed as grayscale intensity images. Then, the program will merge all the channels back into an image and display it.

Negating an image

In mathematical terms, the negative of an image is the inversion of colors. For a grayscale image, it is even simpler! The negative of a grayscale image is just the intensity inversion, which can be achieved by finding the complement of the intensity from 255. A pixel

value ranges from 0 to 255 and, therefore, negation is the subtraction of the pixel value from the maximum value, that is, 255. The code for this is as follows:

```
import cv2
img = cv2.imread('4.2.07.tiff')
grayscale = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
negative = abs(255-grayscale)
cv2.imshow('Original',img)
cv2.imshow('Grayscale',grayscale)
cv2.imshow('Negative',negative)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Tip

The negative of a negative will be the original grayscale image. Try this on your own by taking the image negative of a negative again.

Logical operations on images

OpenCV provides bitwise logical operation functions on images. We will take a look at functions that provide bitwise logical AND, OR, XOR (exclusive OR), and NOT (inversion) functionalities. These functions can be better demonstrated visually with grayscale images. I am going to use barcode images in horizontal and vertical orientations for demonstration. Look at the following code:

```
import cv2
import matplotlib.pyplot as plt

img1 = cv2.imread('Barcode_Hor.png',0)
img2 = cv2.imread('Barcode_Ver.png',0)
not_out=cv2.bitwise_not(img1)
and_out=cv2.bitwise_and(img1,img2)
or_out=cv2.bitwise_or(img1,img2)
xor_out=cv2.bitwise_xor(img1,img2)

titles = ['Image 1','Image 2','Image 1
NOT','AND','OR','XOR']
images =
[img1,img2,not_out,and_out,or_out,xor_out]

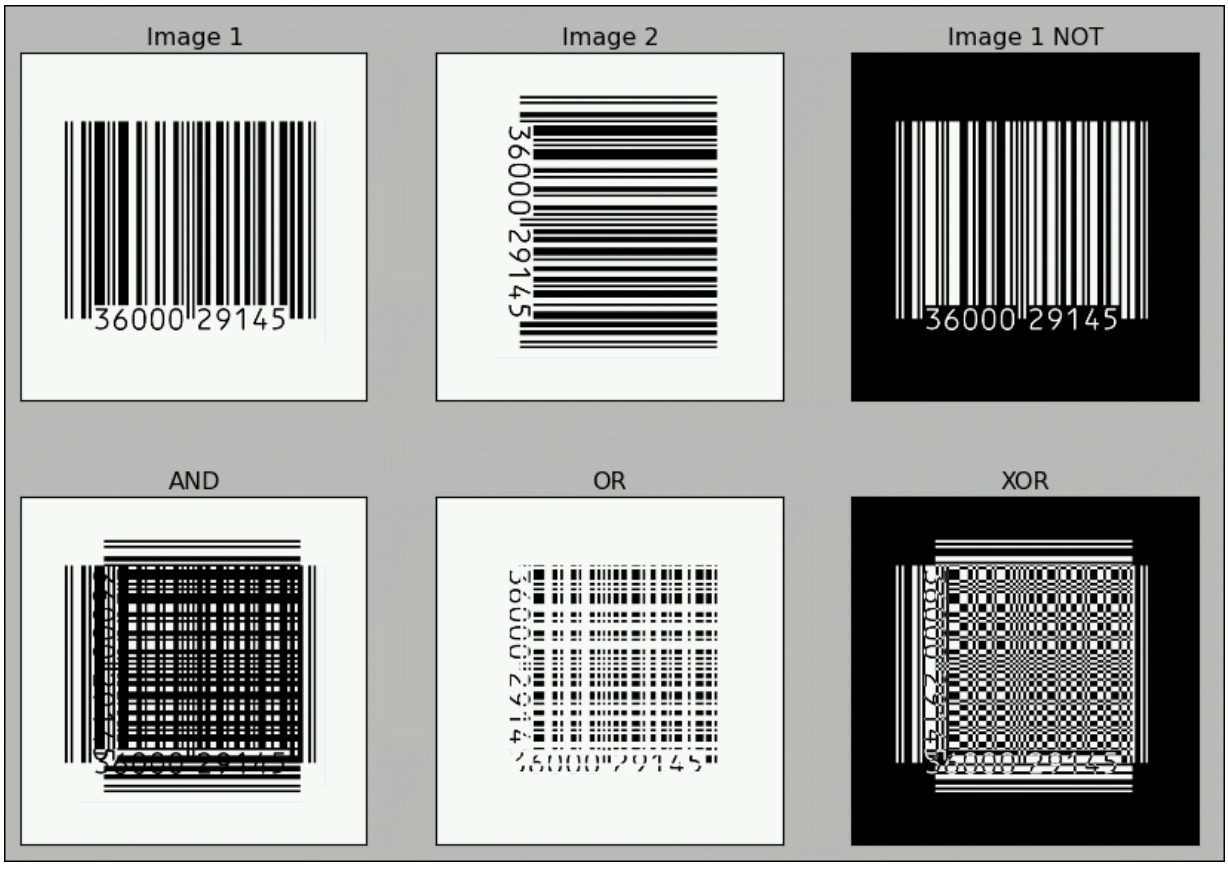
for i in xrange(6):
    plt.subplot(2,3,i+1)
    plt.imshow(images[i],cmap='gray')
    plt.title(titles[i])
    plt.xticks([]),plt.yticks([])
plt.show()
```

We first read images in the grayscale mode and calculate the NOT, AND, OR, and XOR, and then, with `matplotlib`, we display them in a neat way. We are leveraging the `plt.subplot()` function here to display multiple images. In this example, we are creating a two row and three column grid for our images and displaying each image in every part of the grid. You can modify this line and make it `plt.subplot(3,2,i+1)` in order to create a three row and two column grid.

We can do this without a loop in the following way. For each image, you have to write the following statements. I am writing this for the first image here only. Go ahead and write it for the rest of the five images:

```
plt.subplot(2,3,1) , plt.imshow(img1,cmap='gray')
, plt.title('Image 1') ,
plt.xticks([]),plt.yticks([])
```

Finally, use `plt.show()` to display. This technique is to avoid the loop where there is very small number of images to be displayed: usually 2 or 3. The output of this will be exactly the same, as follows:



Note

You might want to make a note of the fact that a logical NOT operation is the negative of the image.

You can check out the Python OpenCV API documentation at http://docs.opencv.org/modules/refman.html.

Colorspaces and conversions

A **colorspace** is a mathematical model used to represent colors. Usually, colorspaces are used to represent colors in a numerical form and perform mathematical and logical operations with them. In this book, the colorspaces we mostly use are BGR (OpenCV's default colorspace), RGB, HSV, and grayscale. BGR stand for Blue, Green, and Red. HSV represents colors in the Hue, Saturation, and Value format. OpenCV has a `cv2.cvtColor(img,conv_flag)` function that allows us to change the colorspace of an `img` image, while the source and target colorspaces are indicated in the `conv_flag` parameter. We have learned that OpenCV loads images in the BGR format, and `matplotlib` uses the RGB format for images. So, before displaying images with `matplotlib`, we need to convert images from BGR to the RGB colorspace. Take a look at the following code. The programs read image in the color mode using `cv2.imread()`, which imports the image in the BGR colorspace. Then, it converts it into RGB using `cv2.cvtColor()`, and finally, it uses `matplotlib` to display the image:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('4.2.07.tiff',1)
img = cv2.cvtColor( img , cv2.COLOR_BGR2RGB )
plt.imshow( img ), plt.title('COLOR IMAGE'),
plt.xticks([]), plt.yticks([])
plt.show()
```

Another way to convert an image from BGR to RGB is to first split the image into three separate channels (B, G, and R channels) and merge them in the BGR order. However, this takes more time as split and merge operations are inherently computationally costly, making them slower and inefficient. The following code shows this method:

```
import cv2
import matplotlib.pyplot as plt
img = cv2.imread('4.2.07.tiff',1)
```

```
b,g,r = cv2.split( img )
img=cv2.merge((r,g,b))
plt.imshow( img ), plt.title('COLOR IMAGE'),
plt.xticks([]), plt.yticks([])
plt.show()
```

The output of both the programs is the same as that shown in the following screenshot:



If you need to know the colorspace conversion flags, then the following snippet of code will assist you in finding the list of available flags for your current OpenCV installation:

```
import cv2
j=0
for filename in dir(cv2):
    if filename.startswith('COLOR_'):
        print filename
        j=j+1

print 'There are ' + str(j) + ' Colorspace
Conversion flags in OpenCV'
```

The last few lines of the output will be as follows (I am not including the complete output due to space limitation):

```
.
.
.
COLOR_YUV420P2BGRA
COLOR_YUV420P2GRAY
COLOR_YUV420P2RGB
COLOR_YUV420P2RGBA
COLOR_YUV420SP2BGR
COLOR_YUV420SP2BGRA
COLOR_YUV420SP2GRAY
COLOR_YUV420SP2RGB
COLOR_YUV420SP2RGBA
There are 176 Colorspace Conversion flags in
OpenCV
```

The following code converts a color from BGR to HSV and prints it:

```
>>> import cv2
>>> import numpy as np
>>> c =
cv2.cvtColor(np.uint8[[[255,0,0]]]),cv2.COLOR_BGR2
HSV)
>>> print c
[[[120 255 255]]]
```

The preceding snippet of code prints an HSV value of Blue represented in BGR.

Hue, Saturation, Value (HSV) is a color model that describes colors (hue or tint) in terms of their shade (the saturation or the amount of gray) and their brightness (the value or luminance). Hue is expressed as a number representing hues of red, yellow, green, cyan , blue, and magenta. Saturation is the amount of gray in the color. Value works in conjunction with saturation and describes the brightness or intensity of the color.

Tracking in real time based on color

Let's study a real-life application of this concept. In the HSV format, it's much easier to recognize the color range. If we need to track a specific color object, we will need to define a color range in HSV and then convert the captured image in the HSV format and check whether the part of that image falls within the HSV color range of our interest. We can use the `cv2.inRange()` function to achieve this. This function takes an image, the upper and lower bounds of the colors, and then it checks the range criteria for each pixel. If the pixel value falls in the given color range, then the corresponding pixel in the output image is 0; otherwise, it is 255, thus creating a binary mask. We can use `bitwise_and()` to extract the color range we're interested in using this binary mask thereafter. Take a look at the following code to understand this concept:

```
import numpy as np
import cv2

cam = cv2.VideoCapture(0)

while (True):
    ret, frame = cam.read()

hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

image_mask = cv2.inRange(hsv, np.array([40, 50,
50]), np.array([80, 255, 255]))

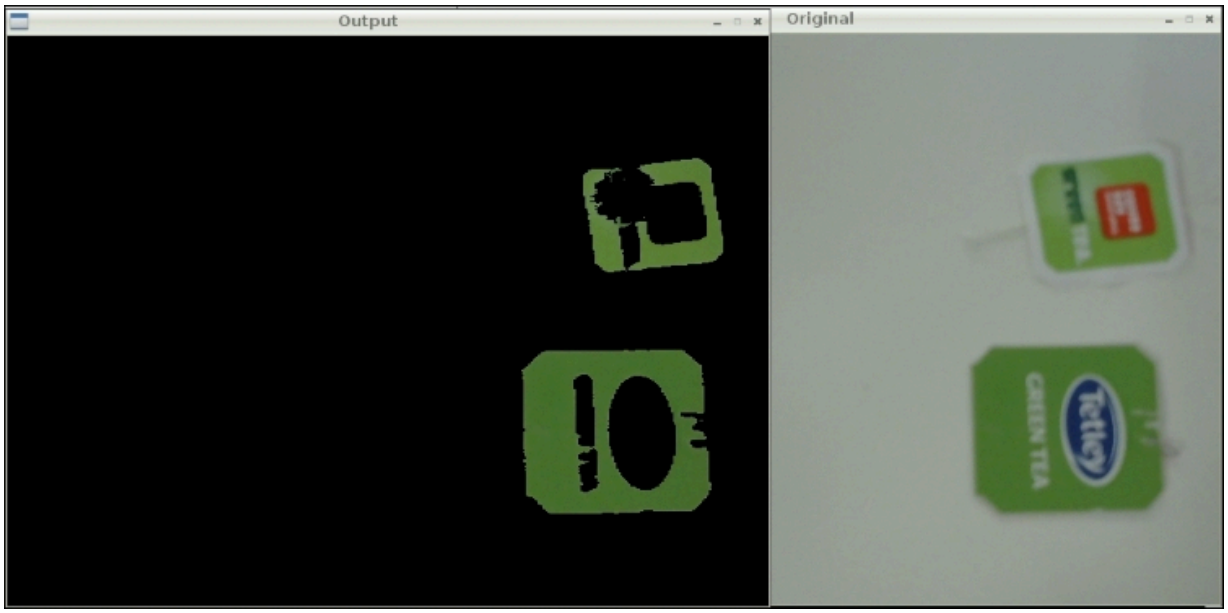
output = cv2.bitwise_and(frame, frame, mask =
image_mask)

cv2.imshow('Original', frame)
cv2.imshow('Output', output)

if cv2.waitKey(1) == 27:
    break
```

```
cv2.destroyAllWindows()
cam.release()
```

We're tracking the green-colored objects in this program. The output should be similar to the following figure. I used green tea bag tags as the test object.



The mask image is not included in the preceding figure. You can see it yourself by adding `cv2.imshow('Image Mask',image_mask)` to the code. It will be a binary (pure black and white) image.

We can also track multiple colors by tweaking this code a bit. We need to modify the preceding code by creating a mask for another color range. Then, we can use `cv2.add()` to get the combined mask for two distinct color ranges, as follows:

```
blue=cv2.inRange(hsv, np.array([100,50,50]),
np.array([140,255,255]))
green=cv2.inRange(hsv,np.array([40,50,50]),np.arra
y([80,255,255]))
image_mask=cv2.add(blue,green)
output=cv2.bitwise_and(frame,frame,mask=image_mask
)
```

Try this code and check the output by yourself.

Summary

In this chapter, we learned the basics of computer vision with OpenCV and Pi. We also went through the basic image processing operations and implemented a real-life project to track objects in a live video stream based on the color.

In the next chapter, we will learn some more advanced concepts in computer vision and implement a fully fledged motion detection system with Pi and a webcam with the use of these concepts.

Chapter 8. Creating Your Own Motion Detection and Tracking System

In the previous chapter, we studied the basics of the OpenCV library. We set up our Pi for OpenCV programming with Python and implemented a simple project to track an object based on the color in OpenCV with a live webcam feed. In this chapter, we will learn about some more advanced concepts and implement one more project based on OpenCV. In this chapter, we will learn about the following topics:

- Thresholding
- Noise reduction
- Morphological operations on images
- Contours in OpenCV
- Real-time motion detection and tracking

Thresholding images

Thresholding is a way to segment images. Although thresholding methods and algorithms are available for colored images, it works best on grayscale images. Thresholding usually (but not always) converts grayscale images into binary images (in a binary image, each pixel can have only one of the two possible values: white or black). Thresholding the image is usually the first step in many image processing applications.

The way thresholding works is very simple. We define a threshold value. For a pixel in a grayscale image, if the value of grayscale intensity is greater than the threshold, then we assign a value to the pixel (for example, white); otherwise, we assign a black value to the

pixel. This is the simplest form of thresholding. Also, there are many other variations of this method, which we will look at now.

In OpenCV, the `cv2.threshold()` function is used to threshold images. Its input includes grayscale image, threshold values, `maxVal`, and threshold methods as parameters and returns the thresholded image as the output. `maxVal` is the value assigned to the pixel if the pixel intensity is greater (or lesser in some methods) than the threshold. There are many threshold methods available in OpenCV; in the beginning, the simplest form of thresholding we saw was `cv2.THRESH_BINARY`. Let's look at the mathematical representation of some of the threshold methods.

Say (x,y) is the input pixel; then, operations for threshold methods will be as follows:

- `cv2.THRESH_BINARY`

  If `intensity(x,y)> threshold`, then set `intensity(x,y)=maxVal`; else, set `intensity(x,y) = 0`

- `cv2.THRESH_BINARY_INV`

  If `intensity(x,y)> threshold`, then set `intensity(x,y)=0`; else, set `intensity(x,y) = maxVal`

- `cv2.THRESH_TRUNC`

  If `intensity(x,y)> threshold`, then set `intensity(x,y)=threshold`; else, `leave intensity(x,y)` as it is

- `cv2.THRESH_TOZERO`

  If `intensity(x,y)> threshold`, then leave `intensity(x,y)` as it is; else, set `intensity(x,y) = 0`

- `cv2.THRESH_TOZERO_INV`

If `intensity(x,y)> threshold`, then set `intensity(x,y)=0`; else, leave `intensity(x,y)` as it is

The demonstration of the threshold functionality usually works best on grayscale images with a gradually increasing gradient. In the following example, we are setting the value of the threshold as `127`, so the image is segmened in two sets of pixels depending on the value of their intensity:

```
import cv2
import matplotlib.pyplot as plt

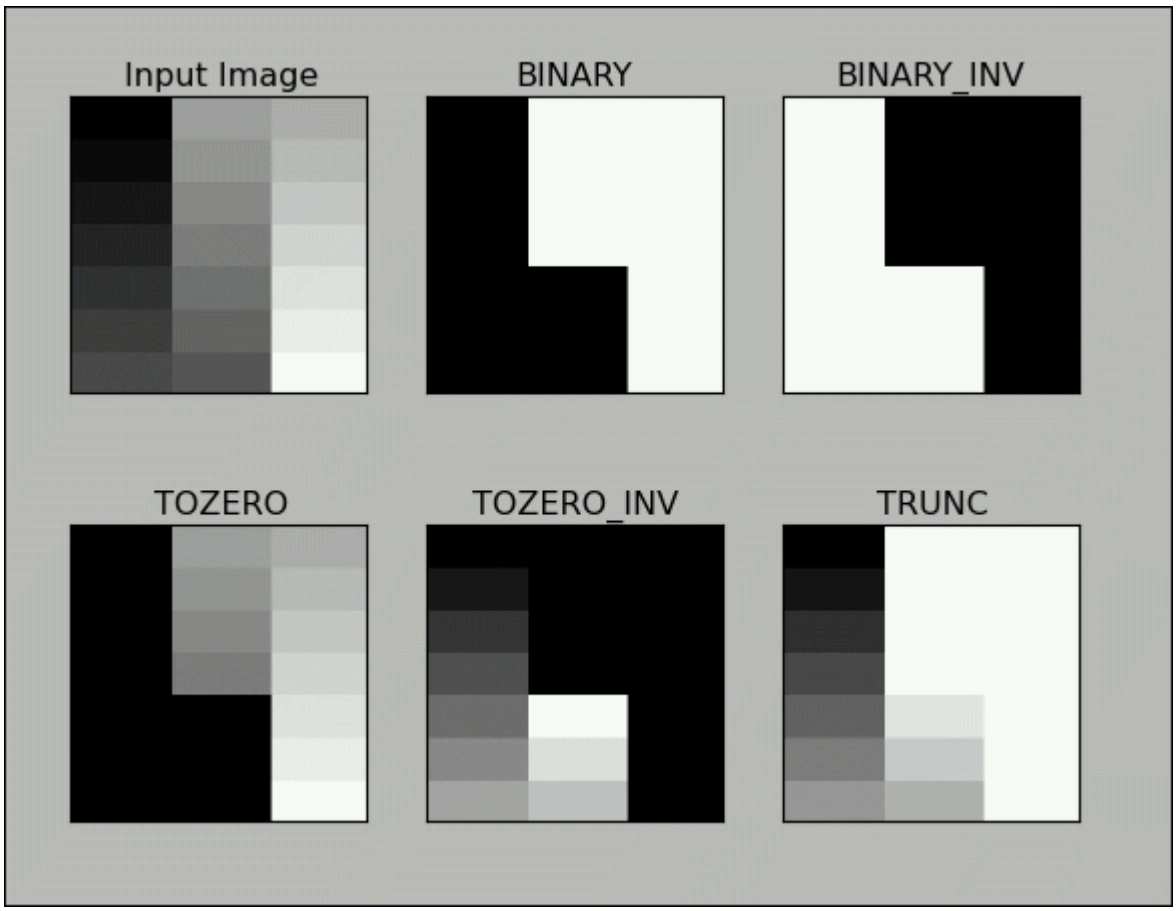
img = cv2.imread('gray21.512.tiff',0)
th=127
max_val=255
ret,o1 =
cv2.threshold(img,th,max_val,cv2.THRESH_BINARY)
ret,o2 =
cv2.threshold(img,th,max_val,cv2.THRESH_BINARY_INV
)
ret,o3 =
cv2.threshold(img,th,max_val,cv2.THRESH_TOZERO)
ret,o4 =
cv2.threshold(img,th,max_val,cv2.THRESH_TOZERO_INV
)
ret,o5 =
cv2.threshold(img,th,max_val,cv2.THRESH_TRUNC)

titles = ['Input
Image','BINARY','BINARY_INV','TOZERO','TOZERO_INV'
,'TRUNC']
output = [img, o1, o2, o3, o4, o5]

for i in xrange(6):

plt.subplot(2,3,i+1),plt.imshow(output[i],cmap='gr
ay')
plt.title(titles[i])
plt.xticks([]),plt.yticks([])
plt.show()
```

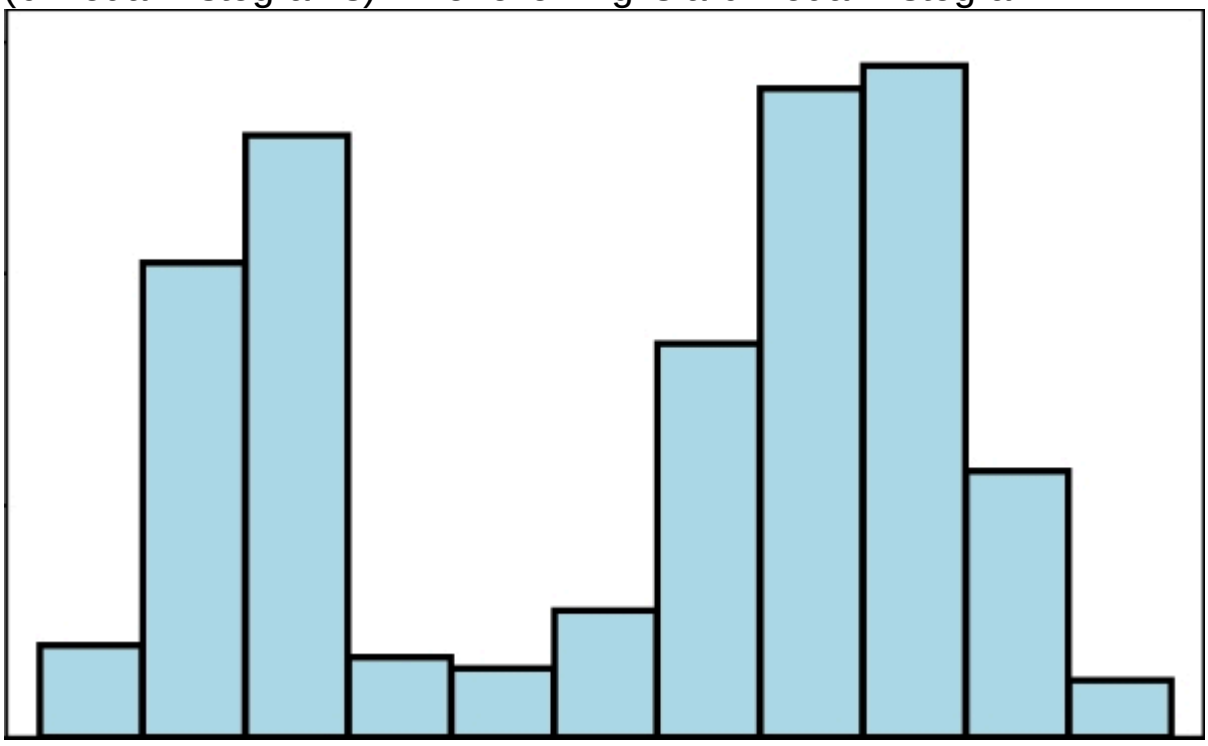
The output of the preceding code will be as follows:



Otsu's method

Otsu's method for thresholding automatically determines the value of the threshold for images that have two peaks in their histogram

(bimodal histograms). The following is a bimodal histogram:



This usually means that the image has background and foreground pixels and Otsu's method is the best way to separate these two sets of pixels automatically without specifying a threshold value.

Otsu's method is not the best way for those images that are not in the background and foreground model and may provide improper output if applied.

This method is applied in addition to other methods, and the threshold is passed as `0`. Try out the following code:

```
ret,output=cv2.threshold(image,0,255,cv2.THRESH_BI
NARY+cv2.THRESH_OTSU)
```

The output of this will be as follows. This is a screenshot of a tank in a desert:


Threshold

Noise

Noise means an unwanted signal. Image/video noise means unwanted variations in intensity (for grayscale image) or colors (for color images) that are not present in the real object being photographed or recorded. Image noise is a form of electronic disruption and could come from many sources, such as camera sensors and circuitry in digital or analog cameras. Noise in digital cameras is equivalent to the film grain of analog cameras. Though some noise is always present in any output of electronic devices, a high amount of image noise considerably degrades the overall image quality, making it useless for the intended purpose. To represent the quality of the electronic output (in our case, digital images), the mathematical term **signal-to-noise ratio** (**SNR**) is a very useful term. Mathematically, it's defined as follows:

$$SNR = \frac{SignalPower}{NoisePower}$$

> ## Note
>
> More signal-to-noise ratio translates into better quality image.

Kernels for noise removal

In the following concepts and their implementations, we are going to use kernels. **Kernels** are square matrices used in image processing. We can apply a kernel to an image to get different results, such as the blurring, smoothing, edge detection, and sharpening of an image. One of the main uses of kernels is to apply a low pass filter to an image. Low pass filters average out rapid changes in the intensity of the image pixels. This basically smoothens or blurs the image. A

simple averaging kernel can be mathematically represented as follows:

$$K = \frac{All\ Ones\ Matrix}{Rows * Cols}$$

For row = cols = 3, kernel will be as follows:

$$K = \frac{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}{9}$$

The value of the rows and columns in the kernel is always odd.

We can use the following NumPy code to create this kernel:

```
K=np.ones((3,3),np.uint32)/9
```

2D convolution filtering

`cv2.filter2D() function` convolves the previously mentioned kernel with the image, thus applying a linear filter to the image. This function accepts the source image and depth of the destination image (`-1` in our case, where `-1` means the same depth as the source image) and a kernel. Take a look at the following code. It applies a 7 x 7 averaging filter to an image:

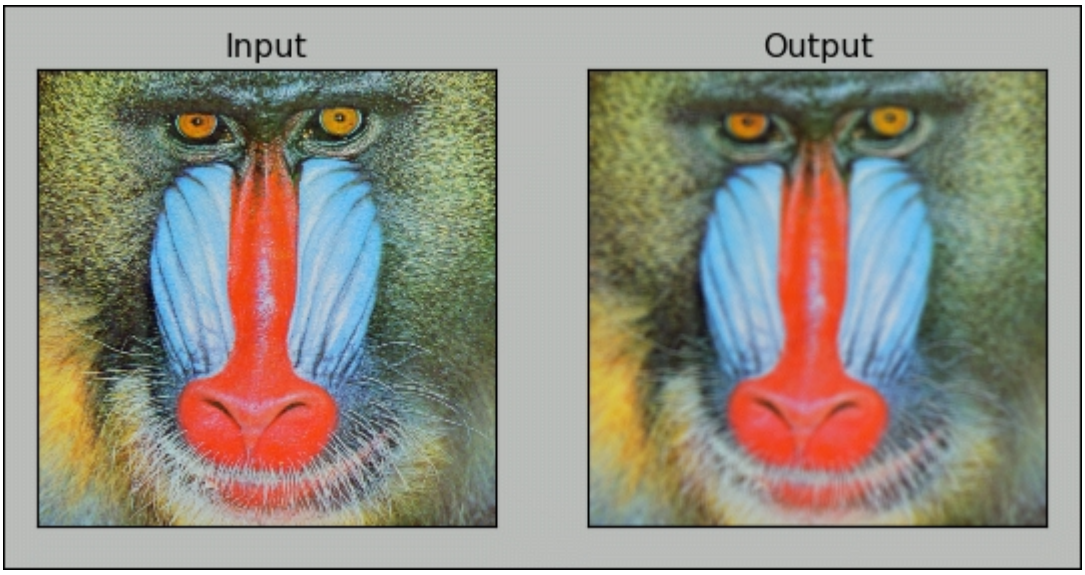
```
import cv2
importnumpy as np
frommatplotlib import pyplot as plt

img = cv2.imread('4.2.03.tiff',1)

input = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
output =
cv2.filter2D(input,-1,np.ones((7,7),np.float32)/49
)
```

```
plt.subplot(121),plt.imshow(input),plt.title('Inpu
t')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(output),plt.title('Out
put')
plt.xticks([]), plt.yticks([])
plt.show()
```

The output will be a filtered image, as follows:



Low pass filtering

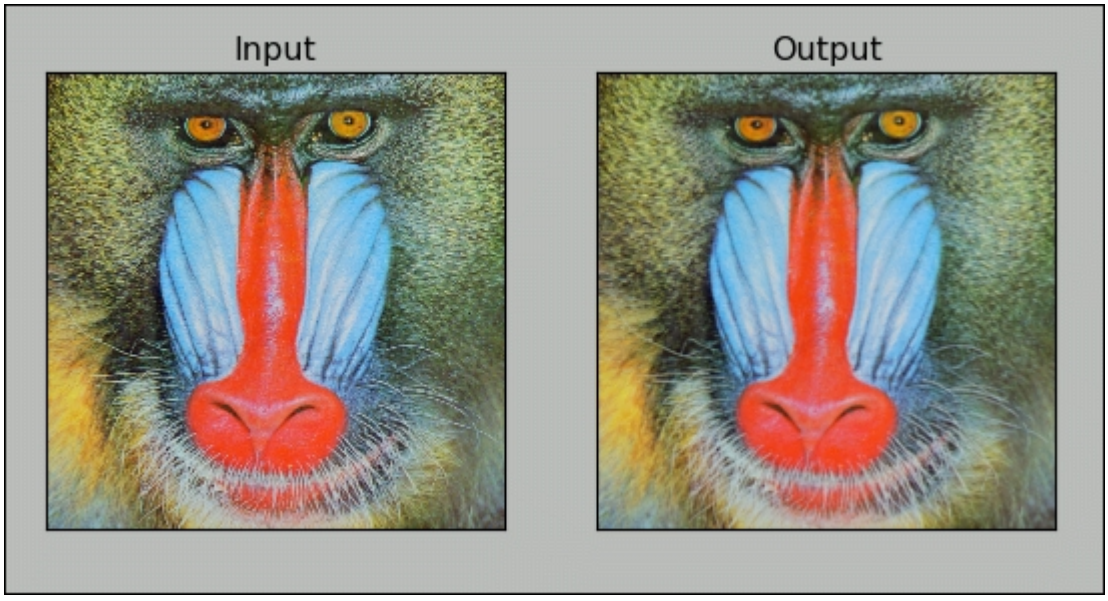
As discussed in the kernels section, low pass filters are excellent when it comes to removing sharp components (high frequency information) such as edges and noise and retaining low frequency information (so-called low pass filters), thus blurring or smoothening them.

Let's explore the low pass filtering functions available in OpenCV. We do not have to create and pass the kernel as an argument to these functions; instead, these functions create the kernel based on the size of the kernel we pass as the parameter.

`cv2.boxFilter()` function takes the image, depth, and size of the kernel as inputs and blurs the image. We can specify `normalize` as either `True` or `False`. If it's `True`, the matrix in the kernel would have $\frac{1}{rows * cols}$ as its coefficient; thus, the matrix is called a normalized box filter. If `normalize` is `False`, then the coefficient will be 1, and it will be an unnormalized box filter. An unnormalized box filter is useful for the computing of various integral characteristics over each pixel neighborhood, such as covariance matrices of image derivatives (used in dense optical flow algorithms, and so on). The following code demonstrates a normalized box filter:

```
output=cv2.boxFilter(input,-1,
(3,3),normalize=True)
```

The output of the code will be as follows, and it will have less smoothing than the previous one due to the size of the kernel matrix:



The `cv2.blur()` function directly provides the normalized box filter by accepting the input image and the kernel size as parameters without the need to specify the `normalize` parameter. The output for the following code will be exactly the same as the preceding output:

```
output = cv2.blur(input,(3,3))
```

As an exercise, try passing `normalize` as `False` for an unnormalised box filter to `cv2.boxFilter()` and view the output.

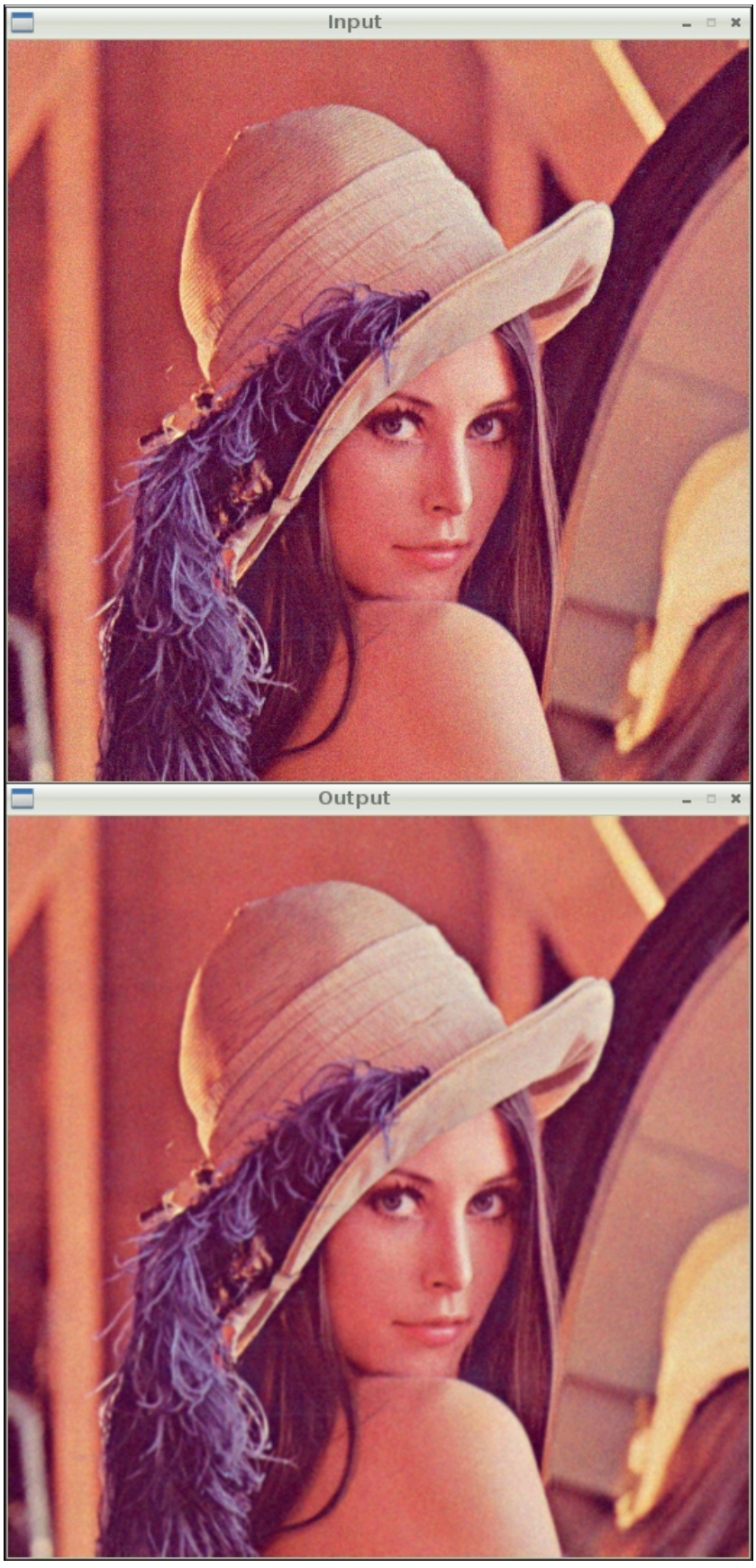
The `cv2.GaussianBlur()` function uses the Gaussian kernel in place of the box filter to be applied. This filter is highly effective against Gaussian noise. The following is the code that can be used for this function:

```
output = cv2.GaussianBlur(input,(3,3),0)
```

Note

You might want to read more about Gaussian noise at http://homepages.inf.ed.ac.uk/rbf/HIPR2/noise.htm.

The following is the output of the earlier code where the input is the image with Gaussian noise and the output is the image with removed Gaussian noise.


Input
Output

`cv2.medianBlur()` is used for the median blurring of the image using the median filter. It calculates the median of all the values under the kernel, and the center pixel in the kernel is replaced with the calculated median. In this filter, a window slides along the image, and the median intensity value of the pixels within the window becomes the output intensity of the pixel being processed. This is highly effective against salt and pepper noise. We need to pass an input image and an odd positive integer (not the rows, columns tuple like the previous two functions) to this function. The following code introduces salt and pepper noise in the image and then applies `cv2.medianBlur()` to that in order to remove the noise:

```
import cv2
importnumpy as np
import random
frommatplotlib
import pyplot as plt

img = cv2.imread('lena_color_512.tif', 1)

input = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

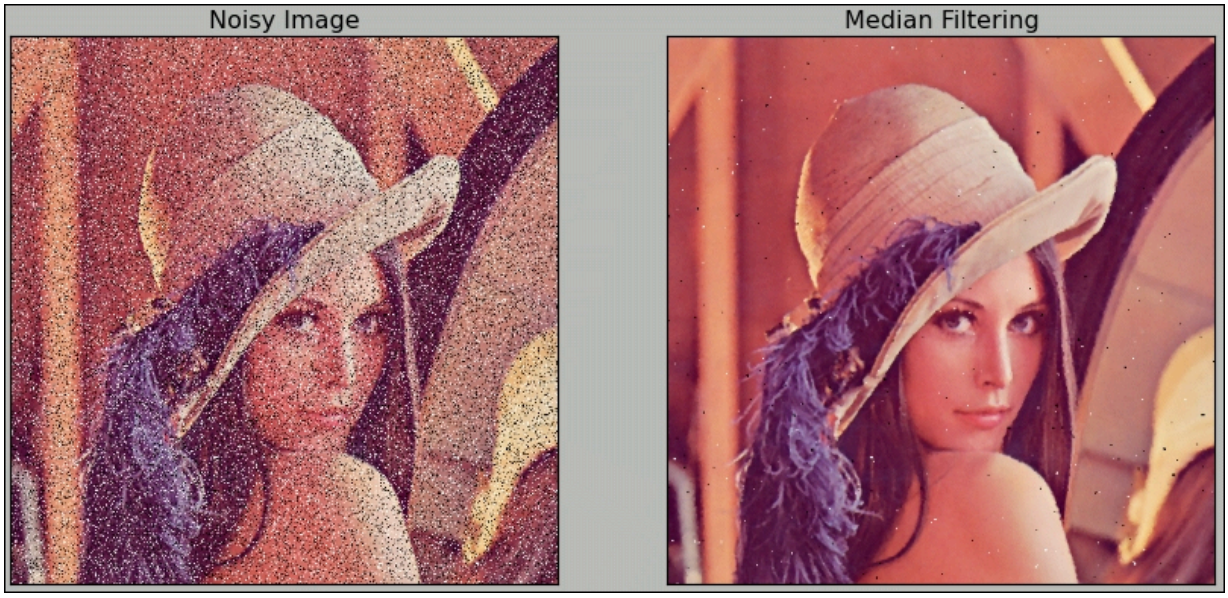
output = np.zeros(input.shape, np.uint8)
p = 0.2# probablity of noise
fori in range(input.shape[0]):
    for j in range(input.shape[1]):
    r = random.random()
if r < p / 2:
    output[i][j] = 0, 0, 0
elif r < p:
    output[i][j] = 255, 255, 255
else :
    output[i][j] = input[i][j]

noise_removed = cv2.medianBlur(output, 3)

plt.subplot(121), plt.imshow(output),
plt.title('Noisy Image')
plt.xticks([]), plt.yticks([])
plt.subplot(122), plt.imshow(noise_removed),
plt.title('Median Filtering')
```

```
plt.xticks([]), plt.yticks([])
plt.show()
```

You will find that the salt and pepper noise is drastically reduced and the image is much more comprehensible to the human eye.



Morphological transformations on images

Morphological operations are based on image shapes, and they work best on binary images. We can use these to get away with a lot of unwanted information, such as noise in an image. Any morphological operation requires two inputs: image and kernel. In this section, we will explore the erosion, dilation, and gradient of an image. Since binary images are most suitable for explaining this concept, we will use a binary image (black and white) to study the concepts.

Erosion removes the boundaries in the image and slims it. In a binary image, white is the foreground and black is the background. All the pixels at the boundary of the white foreground image are made zero, thus slimming the image and eroding away the boundary. Dilation is exactly opposite of erosion; it expands the foreground image boundary and flattens it. The extent of to erosion and dilation depends on the kernel and the number of iterations. The morphological gradient of an image is the difference between dilation and erosion. It will return the outline of an image. Check out the following code for the basic usage of these operations in OpenCV. We will be using these in our next chapter to refine our image for better output:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('morphological.tif',0)
kernel = np.ones((5,5),np.uint8)
erosion = cv2.erode(img,kernel,iterations = 2)
dilation = cv2.dilate(img,kernel,iterations = 2)
gradient = cv2.morphologyEx(img,
cv2.MORPH_GRADIENT, kernel)

titles=
```

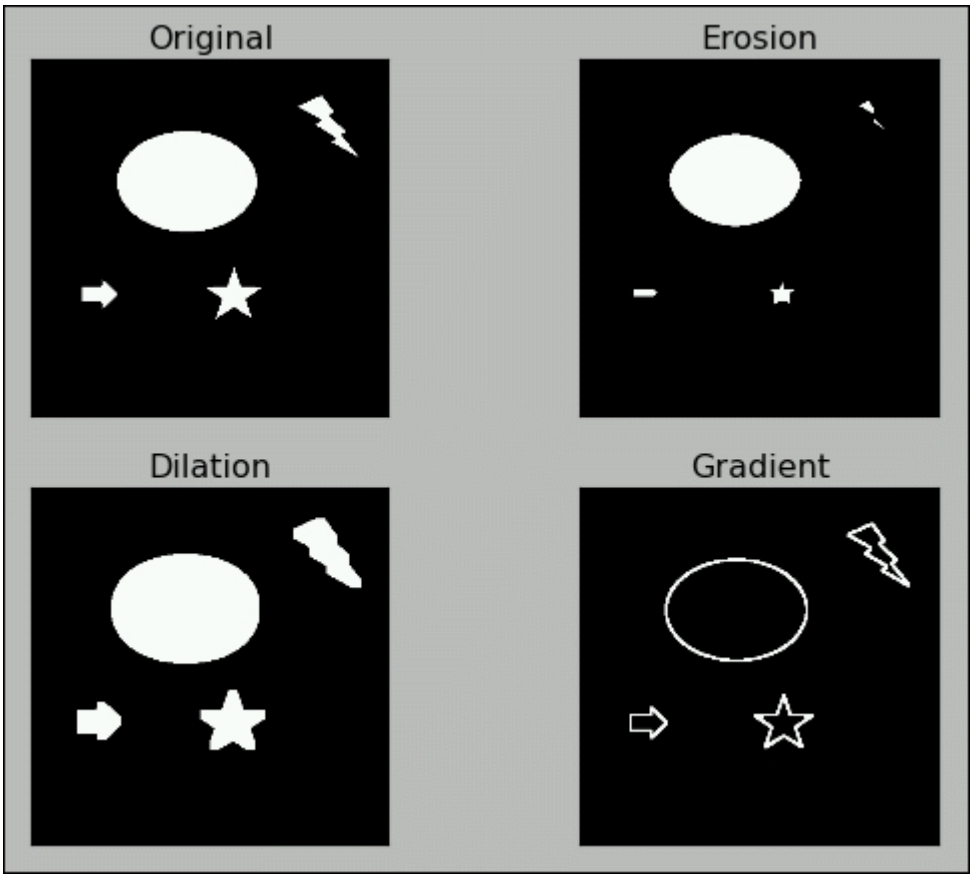
```
['Original','Erosion','Dilation','Gradient']
output=[img,erosion,dilation,gradient]

for i in xrange(4):

plt.subplot(2,2,i+1),plt.imshow(output[i],cmap='gr
ay')

plt.title(titles[i]),plt.xticks([]),plt.yticks([])
plt.show()
```

The output will be as follows:



Motion detection and tracking

We will now build a sophisticated motion detection and tracking system with simple logic to find the difference between subsequent frames from a video feed, such as a webcam stream and plotting contours around the area where the difference is detected.

Let's import the required libraries and initialize the webcam:

```
import cv2
import numpy as np

cap = cv2.VideoCapture(0)
```

We will need a kernel for the dilation operation that we will create in advance rather than creating it every time in the loop:

```
k=np.ones((3,3),np.uint8)
```

The following code will capture and store the subsequent frames:

```
t0 = cap.read()[1]
t1 = cap.read()[1]
```

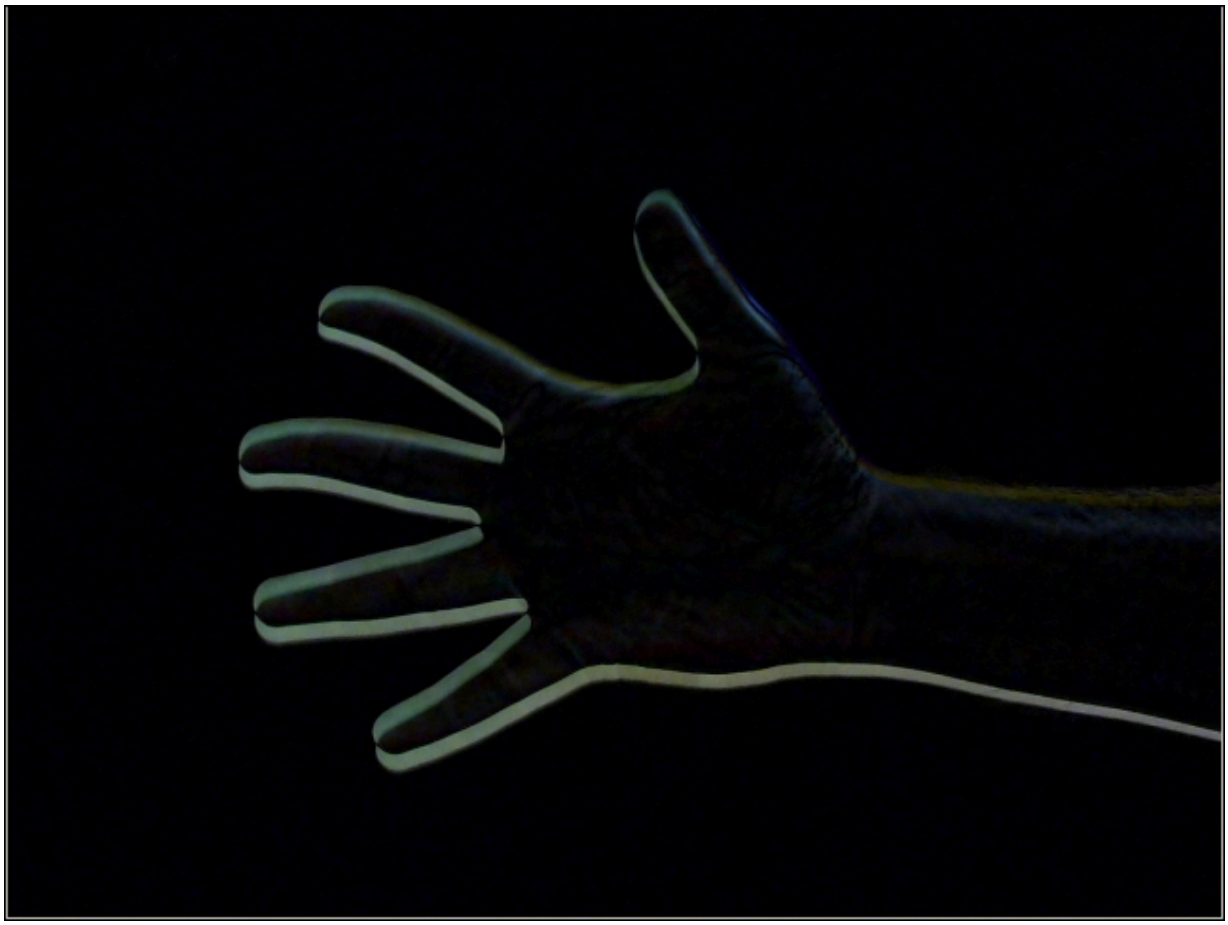
Now we initiate the `while` loop and calculate the difference between the frames and convert the output to grayscale for further processing:

```
while(True):

    d=cv2.absdiff(t1,t0)

    grey = cv2.cvtColor(d, cv2.COLOR_BGR2GRAY)
```

The output will be as follows, and it shows difference of pixels between the frames:



This image might contain some noise, so we will blur it first:

```
blur = cv2.GaussianBlur(grey,(3,3),0)
```

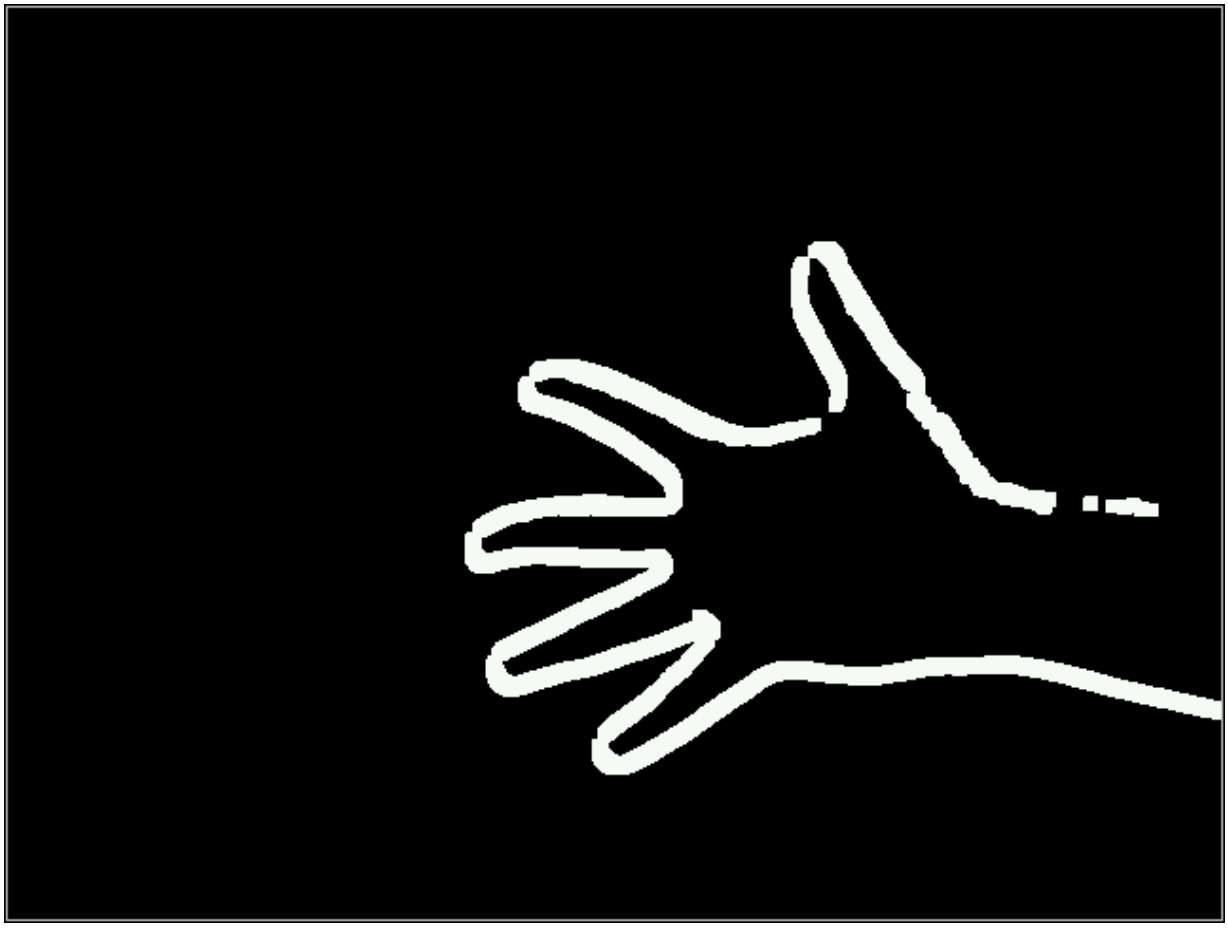
We use binary threshold to convert this noise-removed output into a binary image with the following code:

```
ret, th = cv2.threshold( blur, 15, 255,
cv2.THRESH_BINARY )
```

The final operation will be to dilate the image so that it will be easier for us to find the boundary clearly:

```
dilated=cv2.dilate(th,k,iterations=2)
```

The output of the preceding step will be the following:



Then, we find and draw the contours for the preceding image with the following code:

```
contours, hierarchy =
cv2.findContours(dilated,cv2.RETR_TREE,cv2.CHAIN_A
PPROX_SIMPLE)

    t2=t0
    cv2.drawContours(t2, contours, -1, (0,255,0),
2 )

    cv2.imshow('Output', t2 )
```

Finally, we assign the latest frame to the older frame and capture the next frame with the webcam with the following code:

```
t0=t1
```

```
        t1=cap.read()[1]
```

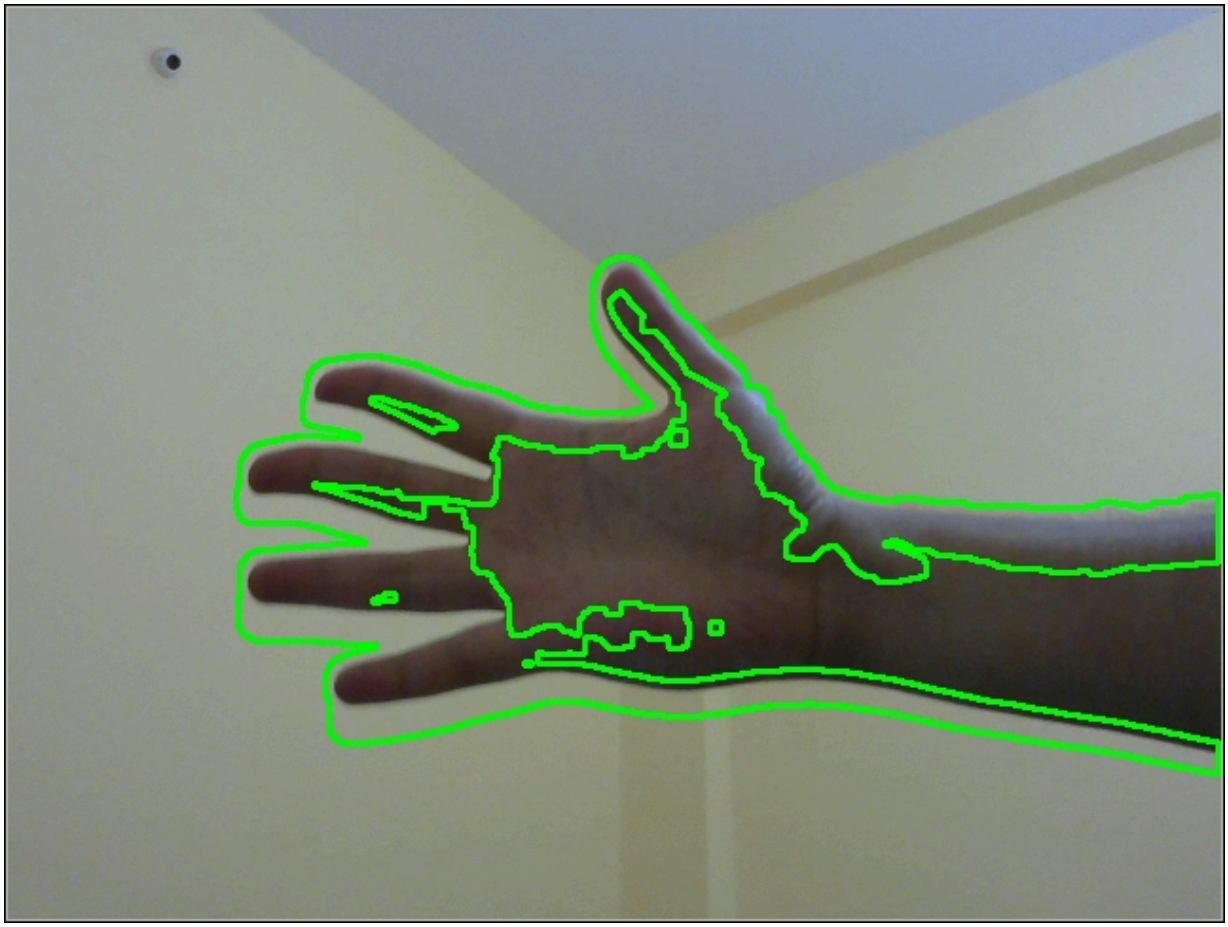
We terminate the loop once we detect the *Esc* keypress, as usual:

```
    if cv2.waitKey(5) == 27 :
            break
```

Once the loop is terminated, we release the camera and destroy the display window:

```
    cap.release()
    cv2.destroyAllWindows()
```

This will draw the contour roughly around the area where the movement is detected, as shown in the following screenshot:



This code works very well for slow movements. You can make the output more interesting by drawing contours with different colors. Also, you can find out the centroid of the contours and draw crosshairs or circles corresponding to the centroids.

Note

If you wish to explore OpenCV with Raspberry Pi in more depth, go through *Raspberry Pi Computer Vision Programming*. Here is the link: https://www.packtpub.com/hardware-and-creative/raspberry-pi-computer-vision-programming.

Summary

In this chapter, we learned about the advanced topics of computer vision with OpenCV and Pi.

We learned about advanced image processing techniques, such as thresholding, noise reduction, contours, and morphological operations. Finally, we implemented all these techniques to build a real-life application for image processing.

In the next chapter, we will learn about the basics of interfacing Pi with Grove Shield and Grove Sensors.

Chapter 9. Grove Sensors and the Raspberry Pi

In the previous chapter, we discussed object detection and tracking and also learned to detect motion in scenes. Along the way, we also learned various concepts of image processing that may also be used for different applications.

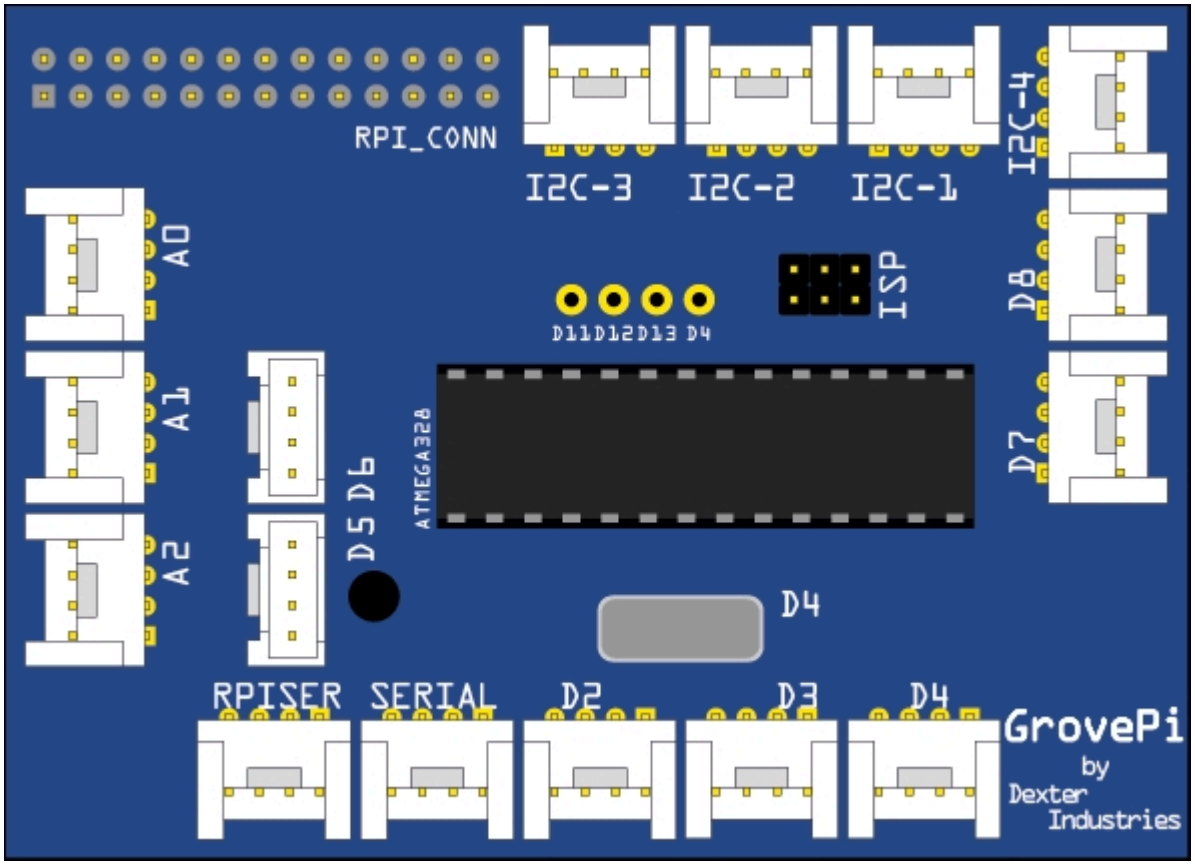
In this chapter, we will be moving forward from the software-based approach of the previous chapter and explore the hardware capabilities of the Raspberry Pi. We will also learn about an add-on called the GrovePi Shield, which will extend the GPIO capabilities of the Pi platform and make it even easier to connect new sensors to it. In this chapter, we will learn about the following:

- Introducing the GrovePi Shield and sensors
- Setting up the Shield with the Raspberry Pi
- Building a weather station
- Building an intruder detection system

Introducing the GrovePi

The GrovePi Shield is an open source platform to connect a range of sensors called the Grove sensors to the Raspberry Pi. We can create our own Internet of Things applications without any need for soldering using the GrovePi! Grove is an easy-to-use collection of more than a hundred Plug and Play modules that we can use to sense and interact with the physical world. The array of sensors include relays, temperature sensors, OLED displays, ultrasonic ranger, joystick, accelerometer, humidity sensor, GPS, and so on. They are divided on the basis of the following six categories: Environment Monitoring, Motion Sensing, User Interface, Physical Monitoring, Logic Gate, and Power.

You can interact and monitor the world using the sensors and then store the data on your Raspberry Pi bridging the gap to the real world!



As you can see from the preceding image, the GrovePi slips over the Raspberry Pi and has a variety of sockets to hold the different sensors that you might want to add to your Raspberry Pi. It works with the Raspberry Pi models A, A+, B, B+, 2, and uses an open source library made available by its creators to allow users to program in Python, C, C++, Go, and NodeJS.

For this chapter, we will need the following components:

- Humidity and temperature sensor (http://www.seeedstudio.com/depot/Grove-TempHumi-Sensor-p-745.html?cPath=25_125)
- LED (http://www.seeedstudio.com/depot/Grove-Red-LED-p-1142.html?cPath=81_35)

- Ultrasonic ranger (http://www.seeedstudio.com/depot/Grove-Ultrasonic-Ranger-p-960.html?cPath=25_31)
- OLED display (http://www.seeedstudio.com/depot/Grove-OLED-Display-096-p-824.html?cPath=34_36)
- Buzzer (http://www.seeedstudio.com/depot/Grove-Buzzer-p-768.html?cPath=38)
- GrovePi+ (http://www.seeedstudio.com/depot/GrovePi-p-2241.html)

All of the preceding are available on the SeeedStudio website.

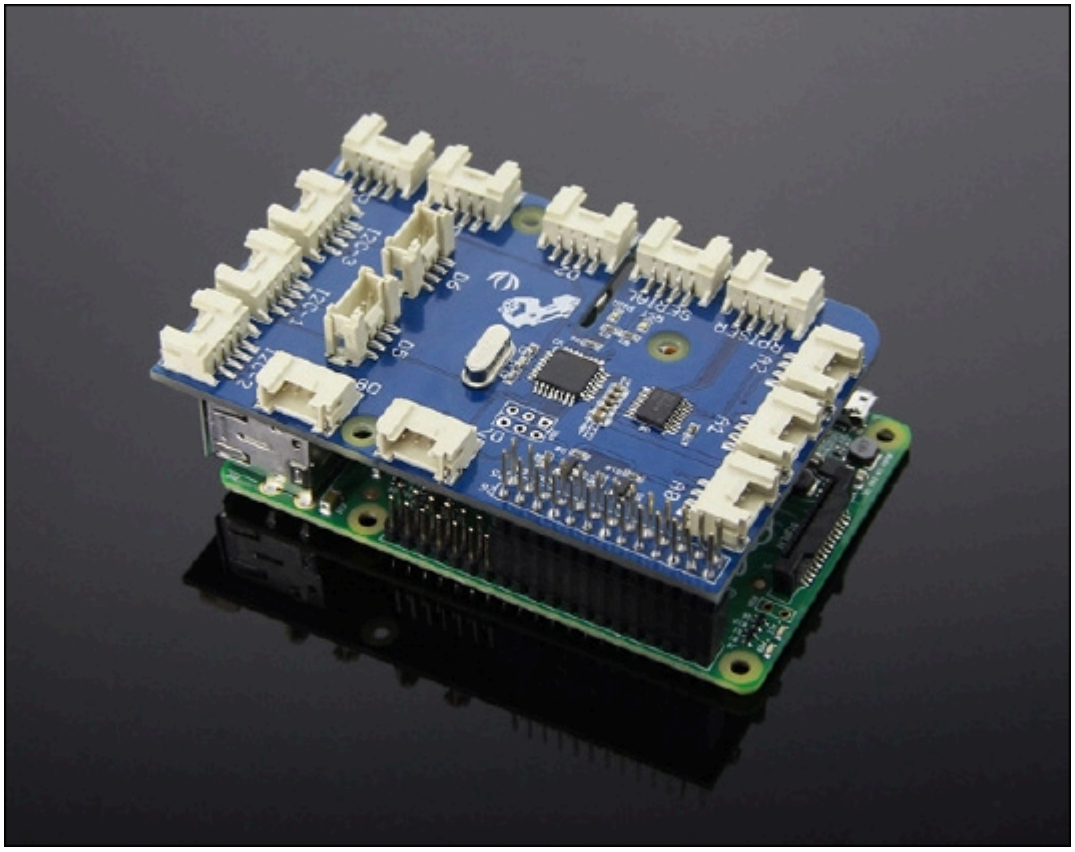
Note

You can find out more about the GrovePi and the sensors on their creator's website:

http://www.dexterindustries.com/

Setting up the GrovePi

Let us have a look at an image of the GrovePi:



As you can see from the preceding image, GrovePi has a socket that allows it to fit directly over the Raspberry Pi. Once we do that, we can begin to set up the software on the Raspberry Pi to use the shield. The steps are given next.

1. Turn on the Raspberry Pi without the GrovePi attached and open a terminal window either by using a monitor or through ssh:

```
The authenticity of host '192.168.1.5 (192.168.1.5)' can't be established.
ECDSA key fingerprint is 09:e5:6a:c3:01:5f:79:84:78:49:43:f4:43:55:1d:31.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.5' (ECDSA) to the list of known hosts.
pi@192.168.1.5's password:
Linux raspberrypi 3.18.11+ #781 PREEMPT Tue Apr 21 18:02:18 BST 2015 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Oct 24 08:17:37 2015
pi@raspberrypi ~ $
```

2. It is recommended that we install GrovePi on the desktop so we first cd into the desktop directory with the following command:

```
$ cd /home/pi/Desktop
```

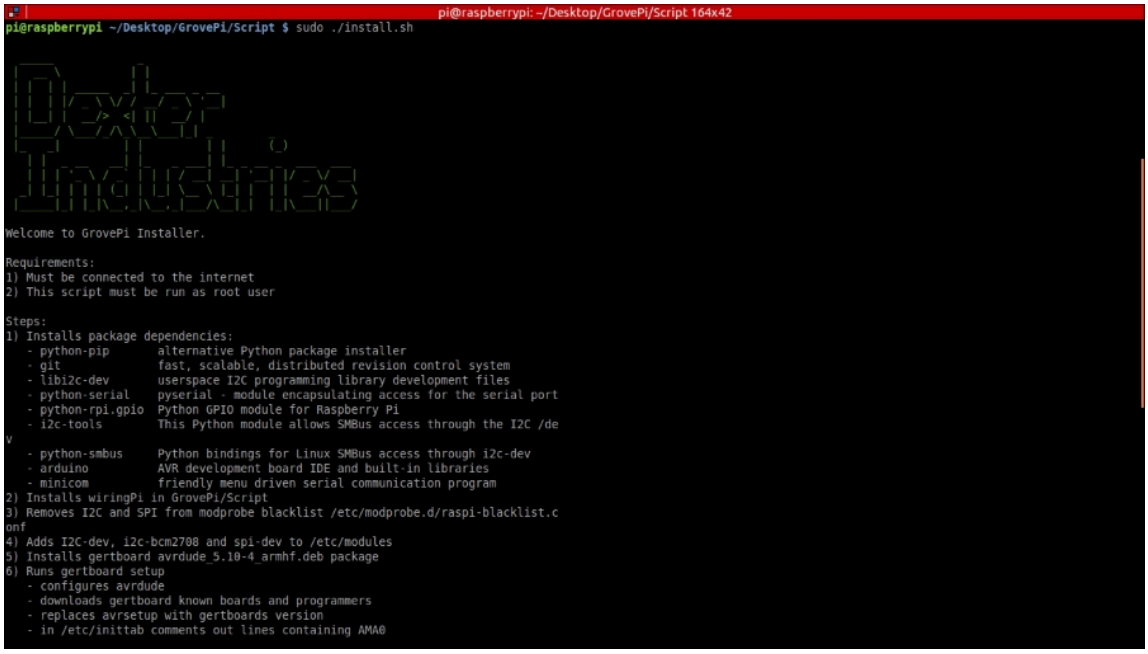
3. Now clone the repository from GitHub into the desktop with the following command:

```
$ sudo git clone
https://github.com/DexterInd/GrovePi
```

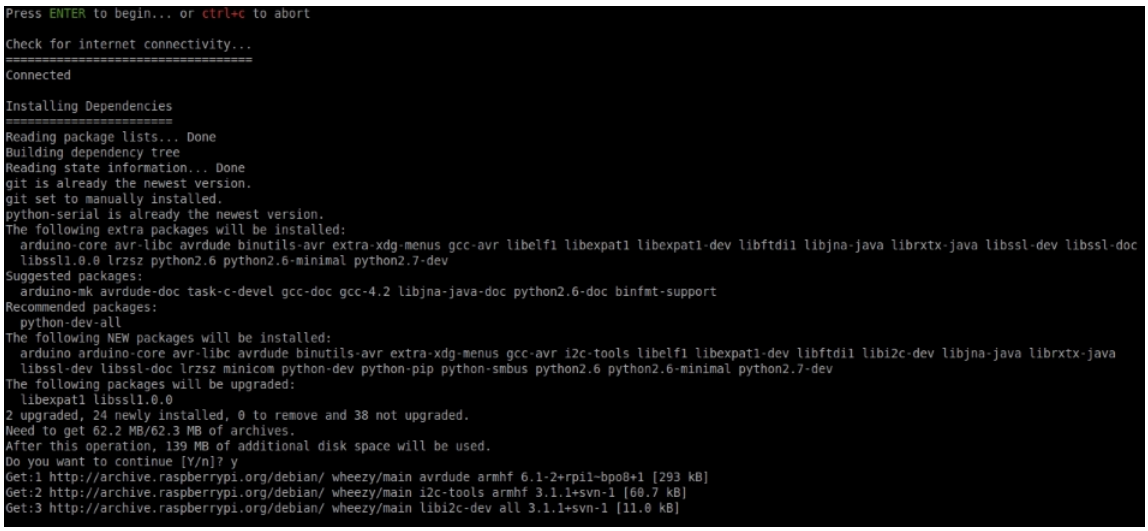
```
pi@raspberrypi ~/Desktop $ sudo git clone https://github.com/DexterInd/GrovePi
Cloning into 'GrovePi'...
remote: Counting objects: 2653, done.
remote: Total 2653 (delta 0), reused 0 (delta 0), pack-reused 2653
Receiving objects: 100% (2653/2653), 1.80 MiB | 29 KiB/s, done.
Resolving deltas: 100% (1424/1424), done.
pi@raspberrypi ~/Desktop $
```

4. We are now ready to execute the install script for the GrovePi. But we must first make the script executable and then run it:

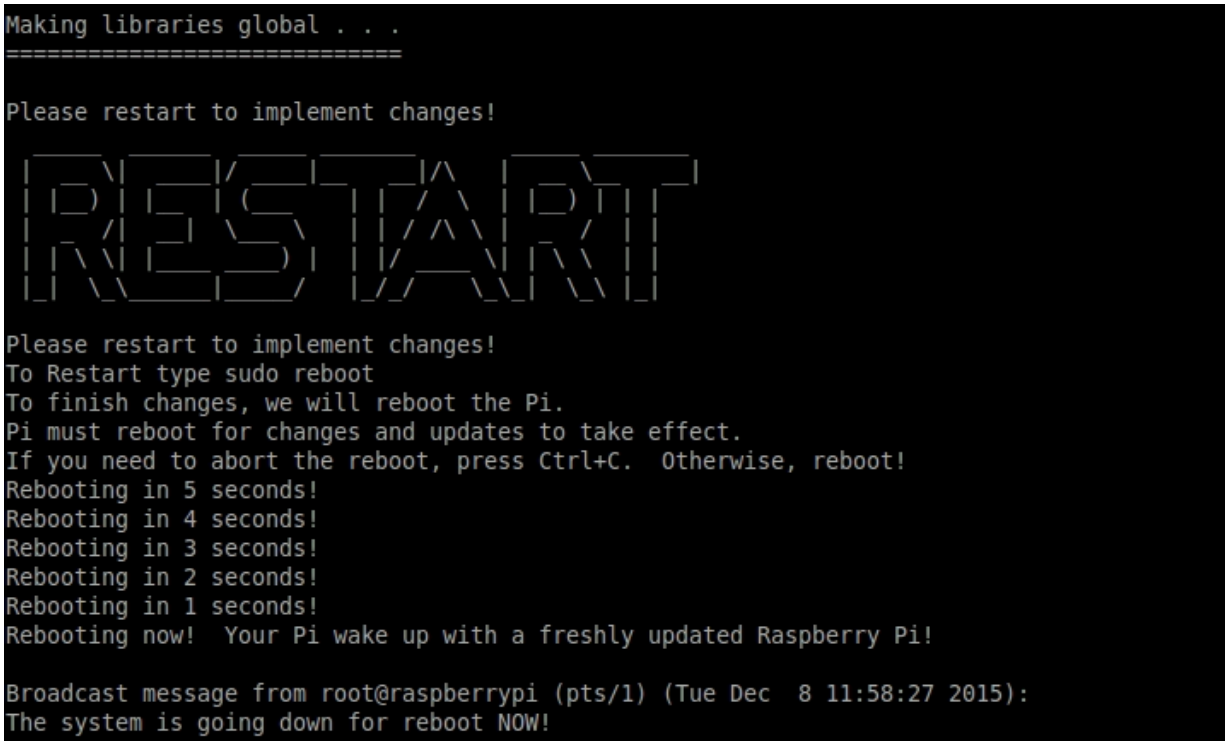
```
$ cd /home/pi/Desktop/GrovePi/Script
$ sudo chmod +x install.sh
$ sudo ./install.sh
```



5. The terminal will ask you to press *ENTER* and subsequently *Y* to continue, so please do press *Enter* to install all the required packages.



The setup will now be completed and your Raspberry Pi will be restarted automatically. With the Raspberry Pi powered off, attach the GrovePi and then turn it on.



Now we will check if the Raspberry Pi detected the GrovePi upon reboot. Run:

```
$ sudo i2cdetect -y 1
```

The command `i2cdetect` detects any peripherals that are connected to the I2C Pins of the Raspberry Pi, which are GPIO02 and GPIO03 in the case of Pi B+ and Pi 2. These are the pins that the GrovePi Shield uses to establish a connection to the Raspberry Pi.

In case you have a Raspberry Pi model that was produced before October 2012, you can run the following:

```
$ sudo i2cdetect -y 0
```

```
^Cpi@raspberrypi ~/Desktop/GrovePi/Software/Python $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- 04 -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

If you see a '04' in the output matrix, that means the GrovePi is detected by the Raspberry Pi I2C port. We can now test to see if it works properly. To test, connect an LED module to the D4 port and run the LED Blink Python example that comes bundled with the Github repository we cloned in an earlier step. To do that, run the following command:

```
$ cd /home/pi/Desktop/GrovePi/Software/Python
$ sudo python grove_led_blink.py
```

If everything was installed correctly in the previous steps, the LED on port D4 should now start blinking.

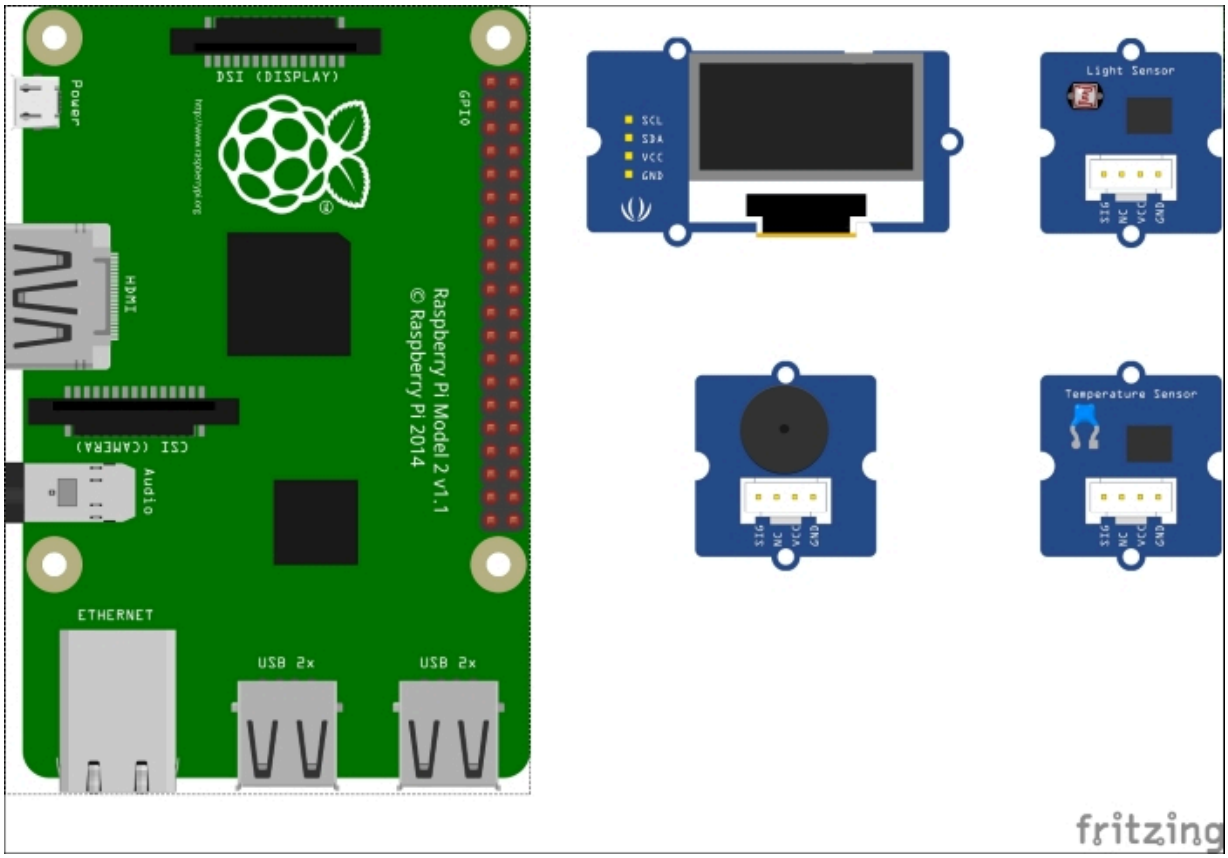
```
pi@raspberrypi ~/Desktop/GrovePi/Software/Python $ sudo python grove_led_blink.py
This example will blink a Grove LED connected to the GrovePi+ on the port labeled
D4.  If you're having trouble seeing the LED blink, be sure to check the LED conne
ction and the port number.  You may also try reversing the direction of the LED or
 the sensor.

Connect the LED to the port labele D4!
LED ON!
LED OFF!
LED ON!
LED OFF!
LED ON!
LED OFF!
```

Congratulations, we have successfully set up the GrovePi to work with the Raspberry Pi and we will now move on to some complex and interesting examples to explore the functionality of the GrovePi.

Displaying the weather

For our first project, we will be using the humidity and temperature sensor of the gross array and use its measurements to be displayed on an OLED display. You will need the two modules and some connection cables to set up the hardware for this project.



Connect the only display module to any of the three i2c ports on the GrovePi, and connect the humidity and temperature sensor to port D7. Now we will have a look at the following code and then learn what each statement does:

```
from grovepi import *
from grove_oled import *
dht_sensor_port = 7
oled_init()
oled_clearDisplay()
oled_setNormalDisplay()
```

```
oled_setVerticalMode()
time.sleep(.1)
while True:
       try:
        [ temp,hum ] = dht(dht_sensor_port,1)
        print "temp =", temp, "C\thumidity =",
hum,"%"
        t = str(temp)
        h = str(hum)
        oled_setTextXY(0,1)
        oled_putString("WEATHER")
        oled_setTextXY(2,0)
        oled_putString("Temp:")
        oled_putString(t+'C')
        oled_setTextXY(3,0)
        oled_putString("Hum :")
        oled_putString(h+"%")
        except (IOError,TypeError) as e:
        print "Error"
```

Save the preceding code as `prog1.py` and run it. You should see the temperature and humidity values being reported on the terminal, as well as on the OLED screen:

```
pi@raspberrypi ~ $ sudo python prog1.py
temp = 24.5 C   humidity = 53.3%
temp = 24.5 C   humidity = 53.3%
temp = 24.5 C   humidity = 53.4%
temp = 24.6 C   humidity = 53.3%
temp = 24.4 C   humidity = 53.3%
temp = 24.5 C   humidity = 53.3%
temp = 24.5 C   humidity = 53.2%
temp = 24.6 C   humidity = 53.3%
```

The first two statements are responsible for importing the dependencies required for the GrovePi Shield and OLED module to work. The asterisk tells Python to import all the classes and functions from the given modules. The second statement creates a variable where the port number for the DHT module is saved. The next statements set up the OLED module for writing data to it. The functions initialize the module, clear the display, and set the vertical mode so that if we write anything subsequently it appears natural. It

is recommended that these four statements be executed at the start of any Python program that uses the OLED module.

Next, we add an infinite while loop so that the program keeps on running until the user terminates it. Inside the while loop, we run the code that is responsible for getting the data from the sensor and displaying it on the OLED module. One of the common mistakes when getting data from any source is that the data might not be available. So we always check the validity and availability of data before executing any code to manipulate or use that data. This is why we must add a try catch statement so that if the data is not available, the program does not crash but gives us an error statement.

Inside the try block, our first statement is to get the data from the DHT module using the `dht()` function. This function takes two arguments: the sensor port number and the type of board connected. We'll enter 1 if the board is DHT Pro and 0 if the board is DHT. It also gives us the output in the form of a list containing two elements: the temperature and humidity level.

In the next statement, we print the humidity level and temperature to the terminal so that the user can look at it. The values that we get from the `dht()` command are integers, but to display them on the OLED screen we need to convert them to strings. This is what the `str()` method does. It takes an input like integer, float, list, and so on and gives a string output.

The subsequent commands that begin with `oled_` are all to manipulate the display of the data on the OLED screen. With the `oled_setTextXY()` method, we can set the cursor to display the data. It takes the `X` and `Y` coordinates as arguments and sets the cursor there. Next, with the `oled_putString()` method, we actually push the string data to be displayed on the cursor we just set. We do this three times: first to set the title as 'WEATHER', second to set the temperature and its units, and finally to set the humidity and its percentage. Since this method only takes the string as input, it was

necessary to convert the integer values of temperature and humidity to the string before using it here.

Finally, we read the last statement that does nothing but handle any error that might be encountered in the `try` block. If it does encounter an error of any sort, it prints **Error** instead of crashing.

We will now move on to our next project, which uses an ultrasonic sensor and a Pi Camera (or a normal webcam if you so choose). Any idea about what we're going to make?

Intruder detection system

Do you have any data that would cause damage if it fell into the wrong hands? Or do you wish to know who enters a specific place when you're not present? Well, you can now easily answer these questions by building your own intruder detection system!

As mentioned before, this project uses an ultrasonic range sensor for the GrovePi and a PiCam to take pictures if any movement is detected by the ultrasonic ranger. Connect the ultrasonic ranger to port D4 of the Grove Pi, the buzzer to D5, the status LED to D3, and the Pi Camera to the port provided on the Raspberry Pi as learned in Chapter 4, *Working with Webcam and Pi Camera*. Following the theme of the book, we will first look at the whole code, and then learn what each statement does, line by line:

```
import picamera
import grovepi
from time import sleep

camera = picamera.PiCamera()
counter = 0
led = 3
buzzer = 5
ultrasonic = 4

while True:
    try:
        if grovepi.ultrasonicRead(ultrasonic) <
100:
        print 'Intruder Detected'
            grovepi.analogWrite(buzzer, 100)
            grovepi.digitalWrite(led, 1)
            sleep(.5)
            grovepi.analogWrite(buzzer, 0)
        grovepi.digitalWrite(led, 0)
            camera.capture('image' + counter +
'.jpg')
            print 'Image Captured'
        sleep(2)
```

```
except IOError:
    print "Error"
```

Save the preceding code as `prog2.py` and run it. Now, try to come in front of the ultrasonic sensor and you will see the output on the terminal, and have your picture taken as an intruder!

The code itself is very simple to understand, as it is just an amalgamation of parts from Chapter 4, *Working with Webcam and Pi Camera*, which we have already learned, and parts from the previous example. We start by importing the dependencies for the Pi Camera and GrovePi sensors.

As a setting-up step, we set the camera variable to be an object for our Pi Camera and it can be used to take pictures. We set a counter for the number of pictures and set the LED and buzzer to be used on ports D3 and D5 respectively. We also import the sleep method from the 'time' module to add delays. Like before, we include a 'try, except' block in the infinite while loop, so that when errors are encountered, it does not crash.

Basic algorithm for the intruder detection can be described as: Whenever anything comes in front of the ultrasonic ranger whose range is less than 100 cms, it means that there is an intruder. If the preceding condition is true, then sound the buzzer, light the LED, and take a picture. The `ultrasonicRead()` method from the GrovePi module reads the output from the ultrasonic ranger that is connected to port D7 of the GrovePi. If the distance is less than 100 cms, then we proceed to execute the intruder detection alarm procedure. First, we output `Intruder detected` to the terminal, then we turn on the buzzer by the `analogWrite()` method and the LED with the `digitalWrite()` method. The analogue write method takes the port number of the peripheral and a value from 0 to 255. The higher the value, the higher the intensity of the signal transmitted to the peripheral. The `digitalWrite()` method also takes the port number as an input and a binary number, to set the port high or low. We wait for half a second and then turn both the buzzer and LED off. Finally,

we capture a photo with the camera pointed toward the direction of the ultrasonic ranger. Congratulations! We now have a photo of the intruder. Now, if we do not add some delay after taking a photo, the code will repeat itself and since the intruder is likely to still be there, the Raspberry Pi may take an unnecessarily large amount of photos. This is likely to burden the CPU and take resources away from other threads that might be running. So we add a two-second delay.

We can also experiment by varying the delay values to see what works best. I leave the reader with an interesting exercise of adding more peripherals to expand the functionality of the system, such as adding a keypad or a switch to turn off the system when it is not required.

Summary

In this chapter, we learned how to connect and interface the GrovePi Shield with the Raspberry Pi, and how to connect the various sensors to it, and use their data to display on external peripherals. We also looked at and understood two examples that can be used easily in real life for useful tasks.

In the next chapter, we will learn about the Internet of Things and how to connect sensors to get the data online and use it effectively. For example, we can extend the intruder detection example and make it such that it e-mails us the photo of the intruder it just took. In this way, we can receive real-time updates from our chosen monitored location. The Internet of Things is a very useful concept and due to the inexpensiveness of the components it takes to get a working system, it is becoming very popular across the globe to make 'dumb' devices 'intelligent'.

Chapter 10. Internet of Things with the Raspberry Pi

In the previous chapter, we learned to interface different kinds of sensors to the Raspberry Pi to get data from the real world and output some text so that the user can infer the state of his surroundings, such as temperature, humidity, and so on. We also built a surveillance system that takes pictures of any intruder. Now, these are all standalone systems and cannot communicate with any other systems.

In this chapter, we will concentrate on the concept of the *Internet of Things*. This means that we can connect standalone systems, just like the ones we built in the previous chapters, to the Internet. We will connect devices to the Internet and learn how to interact with different *web services*. The following topics will be covered in this chapter:

- Installing the Twitter API for Python
- Using Tweepy
- Setting up a SQLite database
- Building a tweeting weather station

Introducing the Internet of Things

The Internet of Things is all about connecting the devices that were previously not connected to the Internet, in order to make them smarter and more controllable. Since the size and cost of electronic devices are dropping exponentially, we can now afford to incorporate them in our daily use, are garden watering systems, home automation/monitoring systems, and so on. The Internet of Things in this way has opened up a whole host of opportunities, the

advantages of which are being taken up by hobbyists, startups, and major companies alike. At the heart of the Internet of Things revolution are small micro computers and sensors that allow these devices to connect to the Internet and exchange data between them.

As mentioned before, in this particular chapter we will extend our previous projects by connecting the Raspberry Pi to the Internet, and make additional projects to demonstrate the capability of the Internet of Things.

We will now learn to exchange data from the Twitter API and post our sensor readings to Twitter.

Installing the Twitter API for Python

API (**Application Programming Interface**) and it allows developers to communicate with various web services. Since Python is, by default, installed on the Raspberry Pi, we will use a Python script to communicate with the Twitter API over the Web. For this, we will use an open source library called **Tweepy**. It works with Python versions as old as 2.6 and is still under active development. So bugs are likely to be fewer and, if any are found, they are likely to be fixed quickly.

> ## Note
>
> The GitHub repository for Tweepy can be found at the following link, so that new issues can be submitted and the code can be understood much more clearly:
>
> https://github.com/tweepy/tweepy

We will now proceed to install it and learn how to use it to do things such as sending tweets.

This library can be installed using `pip`, with the following command:

```
sudo pip install python-twitter
```

We can also install bleeding-edge versions of the library by cloning it from GitHub. Run the following command:

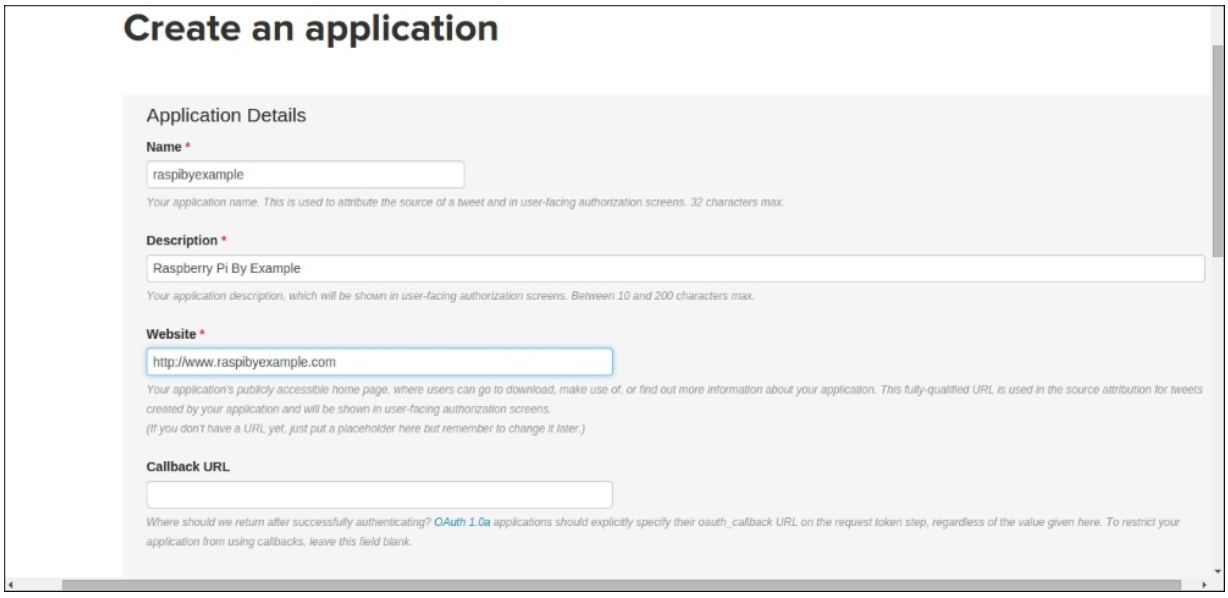
```
git clone https://github.com/tweepy/tweepy.git
cd tweepy
sudo python setup.py install
```

Note

You can also find more information about using Tweepy and the full extent of its capabilities on its official documentation website: http://tweepy.readthedocs.org/en/v3.5.0/.

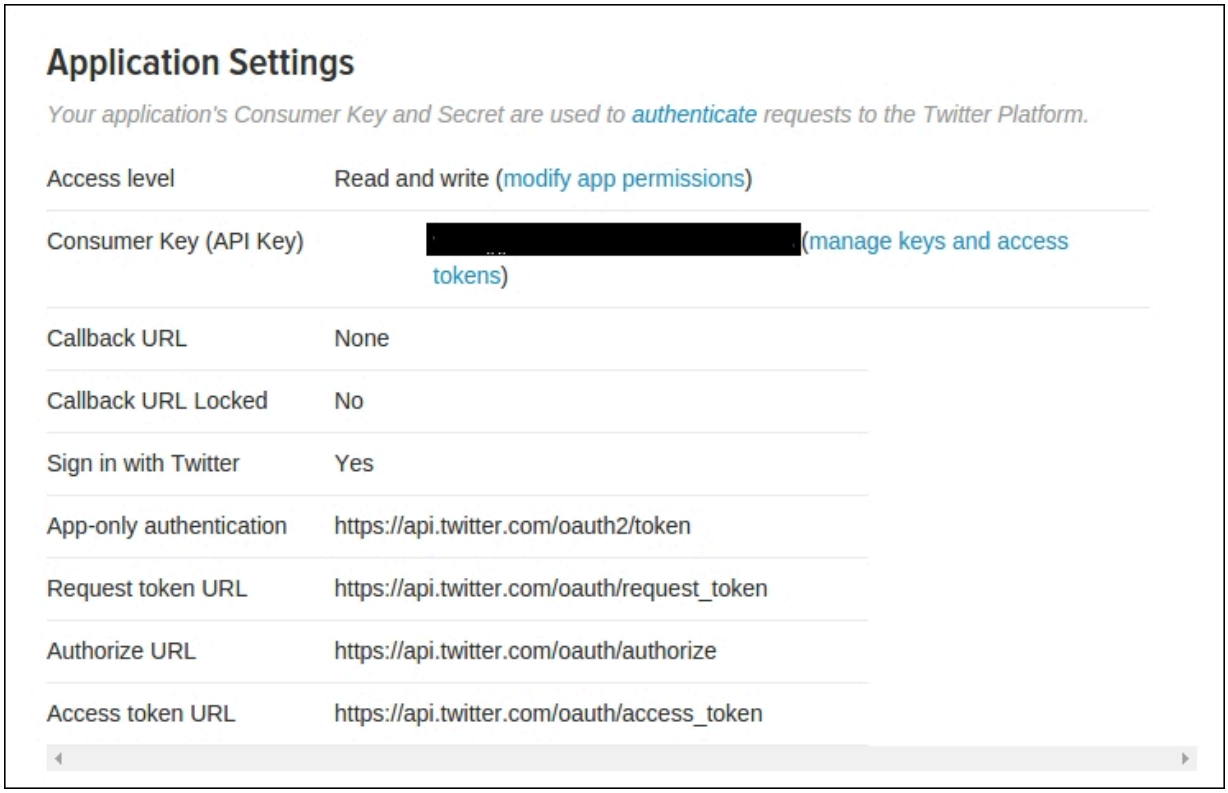
Using Tweepy

Tweepy was built to be easy to set up and use, and in that regard it is of one of the most useable libraries for communicating with the Twitter API. It takes no more than four lines of code to set it up for use. However, before we can use it to communicate with our account on Twitter, we need to set up a developer app on Twitter. For this, we log on to https://apps.twitter.com, sign in to our Twitter account and hit **Create New App**, after which we will be greeted by a window that allows us to create a new application, as shown in the following window:



Fill in all the required details such as **Name**, **Description**, and **Website**. Remember, if you don't plan on using this app on a website, you can also put in a placeholder instead of an actual

website. Finally, sign the developer agreement and you will have successfully created a Twitter app to use with your Python script. From this app, we will need a Consumer Secret (API Secret) and a Consumer Key (API Key) to use the Tweepy library, which is required for obtaining oauth authentication from the Twitter app. Next, we need to create an access code for our app that is required for communication with the Twitter API.

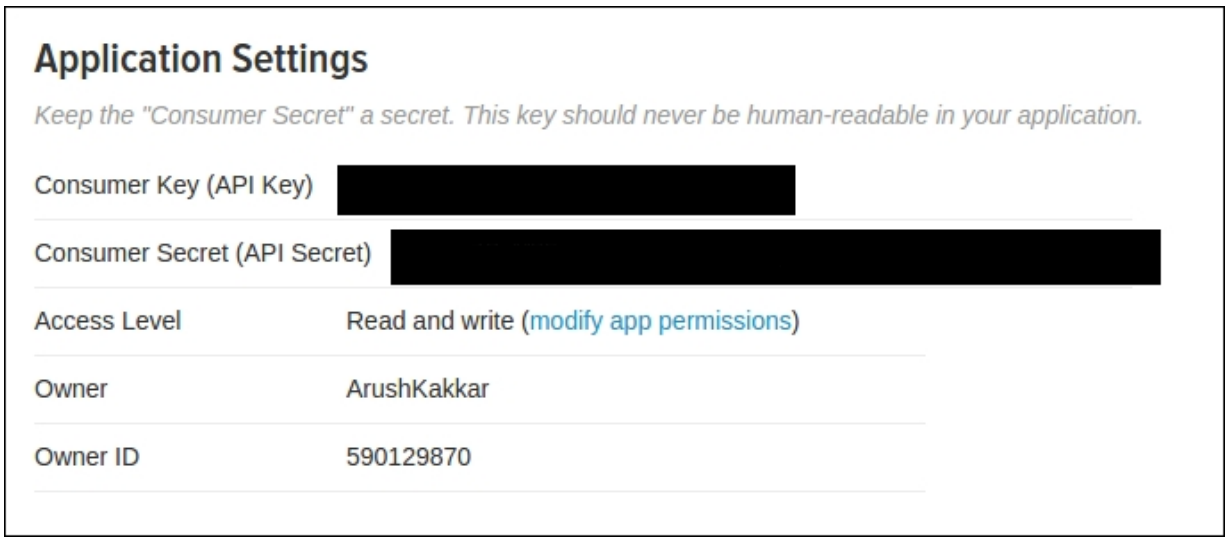


Application Settings

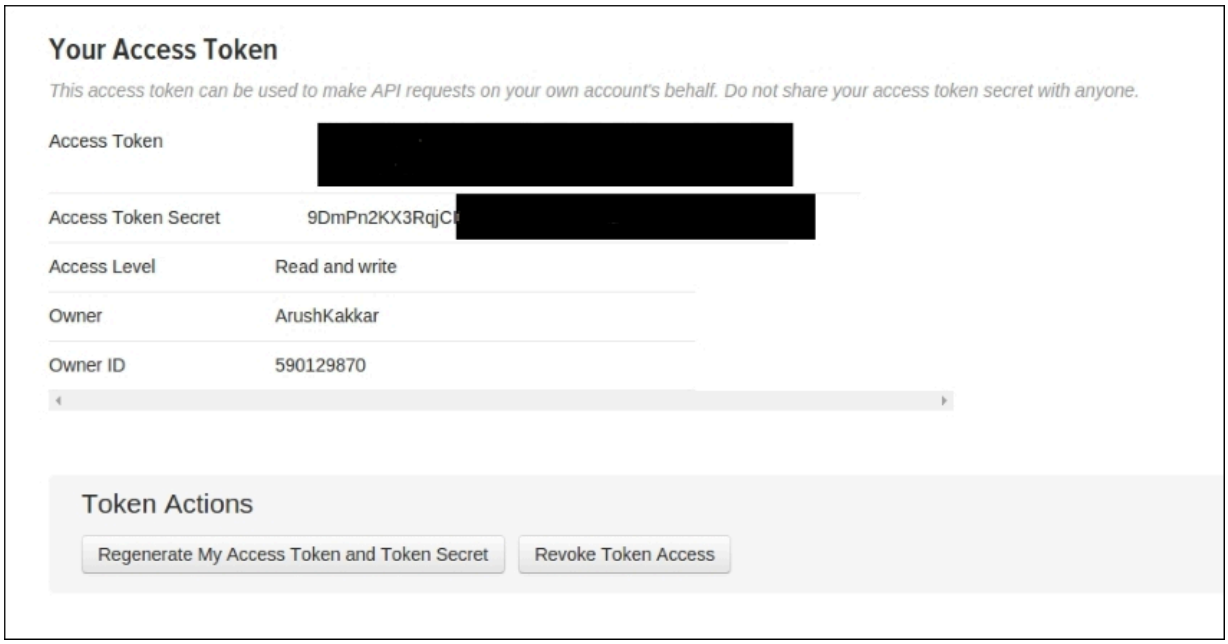
Your application's Consumer Key and Secret are used to authenticate requests to the Twitter Platform.

Access level	Read and write (modify app permissions)
Consumer Key (API Key)	[redacted] (manage keys and access tokens)
Callback URL	None
Callback URL Locked	No
Sign in with Twitter	Yes
App-only authentication	https://api.twitter.com/oauth2/token
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token

From the preceding page, click the **manage keys and access tokens** to get to the page where you can get your consumer codes and access tokens. The following menu will give the consumer keys:



And the following screen will show us our application codes and also give an option to generate new access codes for our application:



We can now proceed to learn how to use Tweepy to get a list of followers. The code is given next, and the explanation follows:

```
import tweepy

auth = tweepy.OAuthHandler(consumer_key,
consumer_secret)
auth.set_access_token(access_token,
```

```
access_token_secret)

api = tweepy.API(auth)

user = api.get_user('twitter')

print user.screen_name
print user.followers_count
for friend in user.friends():
    print friend.screen_name

api.update_status(' Tweepy says : Hello, world!')
```

In this example, we first import the Tweepy library into Python to use its functionality. We first need to register our Python script (client) to be able to communicate with the Twitter API. For this, we will enter the Consumer Key and the Consumer Secret in the argument of the `OAuthHandler` method of the Tweepy API. Next, we set the access codes for opening up the Twitter API for our client's use. This is important because we need to have an access token for the app to be able to do things like send tweets, reading tweets, or viewing followers, and so on. Access codes for the Twitter app can be found in its settings, just below the Consumer Codes.

We have so far set the authentication details, but we still need to do the authentication to send tweets. This is accomplished by the `API()` function in the `Tweepy` module. We first create an `api` variable that contains all the methods for interacting with the Twitter API.

The next statement gets the user associated with our unique Consumer Key. Every registered user will have this key and it is unique for every user so they can correctly identify themselves. The `get_user()` method takes as an argument, the name of the service (Twitter in this case) that we need from the user, and passes the authentication codes that we have got from the previous statement to get the user details. This is stored in the user variable. This variable will have members such as the name, followers, following, and so on.

To check that we really get the correct user from this method, we only need to print the correct members of the user variable. So we first print to the command line, the screen name, and the follower count. The next while loop loops over the `friends()` method of the user variable and stores each follower detail in the friend variable. Then we print the name of each friend to the command line. This way, we can check if we have our user details correct or not.

Finally, we can send a tweet using the `update_status()` method of the Tweepy API. This method takes a string as input and sends out a tweet. However, if a string is more than 140 characters in length, the statement will silently fail that is, it will fail without giving an error.

Now enter your Consumer and Access codes! Save this example as `prog1.py` and run it with the following command:

```
python prog1.py
```

You should see the user details on the terminal, and a tweet from your user:

```
'Tweepy says : Hello, World!'
```

In addition to the preceding functionality, we can also use Tweepy to do other things such as reading statuses. Some of the more important functionality is detailed next:

- `api.home_timeline([since_id], [max_id], [count], [page])`: Returns the 20 most recent tweets on the user's timeline. `count` specifies the number of tweets to retrieve
- `Api.user_timeline([id/user_id/screen_name], [since_id], [max_id], [count], [page])`: Returns the 20 most recent tweets from the specified user or, if none is given, from the authenticating user
- `Api.get_status(id)`: Returns the status by ID
- `Api.update_with_media(filename, [status], [in_reply_to_status_id], [lat], [long], [source],`

`[place_id], [file])`: Updates the authenticated user's status with media

Congratulations! We have successfully set up a Twitter app using Python and communicated with the API. We will now move on to the next example where we learn how to set up a SQLite database to store some data on the Raspberry Pi.

Setting up a SQLite database in Python

SQLite (pronounced **Sequel Lite**) is a lightweight disk-based database that doesn't require a separate server process and uses a modified version of the SQL query language to access the data. The entire database is saved in a single file and can be used for internal data storage. It is also possible to make a prototype of an application using SQLite and then port the code to work with a larger database with minimal modifications. The Python standard library has a module called `sqlite3` included, which is intended to work with this database. This module is SQL interface-compliant with DB-API 2.0.

> ## Note
>
> You can find more information and the full API reference for the SQLite3 module on its official website: https://docs.python.org/2/library/sqlite3.html.

Next is an example on how to use the `sqlite3` module to create a SQLite database to store and retrieve some data. We will go through the code, statement by statement, and analyze what each statement does and also discuss any variations that might be possible.

```
import sqlite3

conn = sqlite3.connect('raspi.db')
cursor = conn.cursor()

#Inserting Data
cursor.execute('''
    CREATE TABLE users(id INTEGER PRIMARY KEY,
name TEXT,
                       email TEXT unique,
password TEXT)
```

```
''')
cursor.execute('''INSERT INTO users(name, email,
password)
                            VALUES(?,?,?)''',
('Luke', 'luke@skywalker.com', 'iamurfather'))

cursor.execute('''INSERT INTO users(name, email,
password)
                            VALUES(?,?,?)''',
('Darth Vader', 'darth@vader.com', 'darkside'))

conn.commit()


#Retrieving Data

user_id = 1
cursor.execute('''SELECT name, email, password
FROM users WHERE id=?''', (user_id,))
user1 = cursor.fetchone()
print(user1[0])
allusers = cursor.fetchall()
for row in allusers:
    print('{0} : {1}, {2}'.format(row[0], row[1],
row[2]))



#Deleting or Updating Data

new_email = 'vader@deathstar.com'
userid = 2
cursor.execute('''UPDATE users SET email = ? WHERE
id = ? ''',
(new_email, userid))

del_userid = 1
cursor.execute('''DELETE FROM users WHERE id = ?
''', (del_userid,))

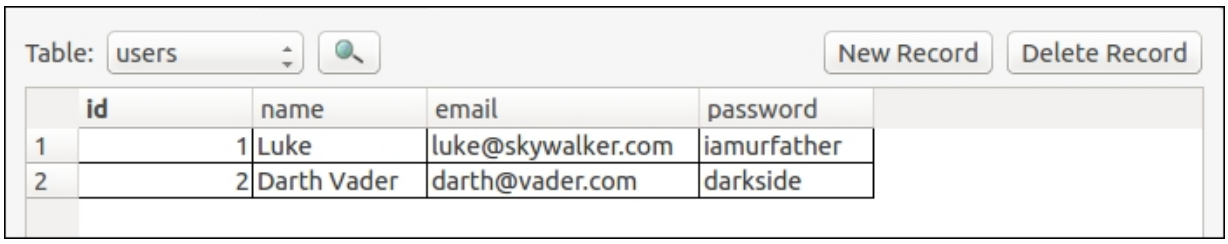
conn.commit()
conn.close()
```

We will now study the data exchanges that took place in the database in the preceding given Python script. Since `sqlite3` is a

default Python module, we don't need to install any additional dependencies. So we import the module into our Python program, and create a new file to save the database with the `sqlite.connect()` method. This method creates a new file for the database if none exists, and if it does the method connects to it and loads the existing database inside the file. Next, we create a cursor that acts as a pointer to the data inside the database. As such, we can only manipulate the data that the cursor is pointing to. We do this with the `cursor()` method that operates on the connection variable, which we created in the last statement.

Currently, our file does not have a database, so with the next lines of code we create a database and insert some data in it. The `execute()` method takes SQL commands as one of the arguments to manipulate data. Hence, to create the database, we execute the `CREATE TABLE` query. In this case, we create a table named `users` that will have a minimum of one data column, which is the key number of the data. This ensures that the data has a unique serial ID by which it can be accessed. We save three fields for each user: the name, the e-mail ID, and the password. So this database can be used for logging in to an application where different users can log in. The `TEXT` type in SQL is the same as `str` in Python, `INTEGER` is `int`, `REAL` is `float`. To make sure that no two users enter the same e-mail ID, we add a "unique" marker to the e-mail column. Now that we have created a database, we need to add some data to it. We use the `INSERT INTO` command to accomplish this. After inserting into vi, mention the name of the database into which we want to insert the data with the names of the arguments. Then the values function will take the actual values of the data. The `?` means that the data is given as a value in a tuple. We repeat the same process with different data to add rows to the database. Notice that we didn't need to set the serial ID of the data for it to be saved properly. That is one advantage of having a cursor in the database. It automatically places new data in the next available row in the database. To actually write the data into the database file and save it, we need to commit our changes. This is done by the `commit()` function that operates on the connection variable. You can visualize the database with a free tool

called SQLite Database Browser. This is what the database looks like after we have made the changes just specified:



	id	name	email	password
1	1	Luke	luke@skywalker.com	iamurfather
2	2	Darth Vader	darth@vader.com	darkside

Now that we have learned how to insert new data into the database, we will see how to read existing data from the database.

First, we need to know the serial-id, name, e-mail, or password of the user that we need to retrieve. Let's assume in this case we want to retrieve the user details by the serial ID. So we execute the `SELECT` command, which selects a row based on the condition that is given. If no condition is given, it will automatically select the first row in the database. The whole statement is close to plain English, so should be easy to understand. Basically, it selects the columns specified from the database name that is given according to a condition given after `WHERE`.

So, after the preceding statement our cursor is pointing toward the row whose ID is 1. The `fetchone()` method fetches the data from the row that the cursor is pointing to, and saves it in the `user1` variable as a list. Then, we simply print the first element of the list, which is the user's name in this case. However, if we want to fetch the data from all the rows in the database, we execute the `fetchall()` method. This points the cursor at all the rows in the database starting from the first one, and saves the result in the `allusers` variable. Then, we simply print the data in a `for` loop.

We have now learned how to save and retrieve data from the database. But what if we want to update user details or delete data? The next block of code (in the preceding code snippet) teaches us exactly that.

Assume that we want to update the e-mail ID of the second user in the database. We first set the new e-mail ID in a string variable, and then the user that we want to select in an integer variable. Then, to update the data, we use the `UPDATE` SQL query. It takes two or more arguments in addition to the database that we want to update: the data that we want to update and a condition, in this case the serial ID of the row. Both of which we give by using the `?` technique we described earlier. The condition operator `WHERE` operates in exactly the same way in `UPDATE` as it does in `SELECT`, pointing the cursor according to the condition given. After the update, this is what our table looks like:



	id	name	email	password
1	1	Luke	luke@skywalker.com	iamurfather
2	2	Darth Vader	vader@deathstar.com	darkside

To delete some data, we use the `DELETE` query and select the database with the `FROM` query. We also employ the condition by using the `WHERE` query and giving it the serial ID we want to delete.

Since we have changed the data in the database, to write the changes into the file we use the `commit()` method, and finally release the file from the Python script with the `close()` method. This ensures that no new changes can be made to the file unless it is connected again. If we close the file without committing the changes, no changes will be reflected in the database and we will have lost any changes that we previously made. This is what the table looks like after we delete the first row:



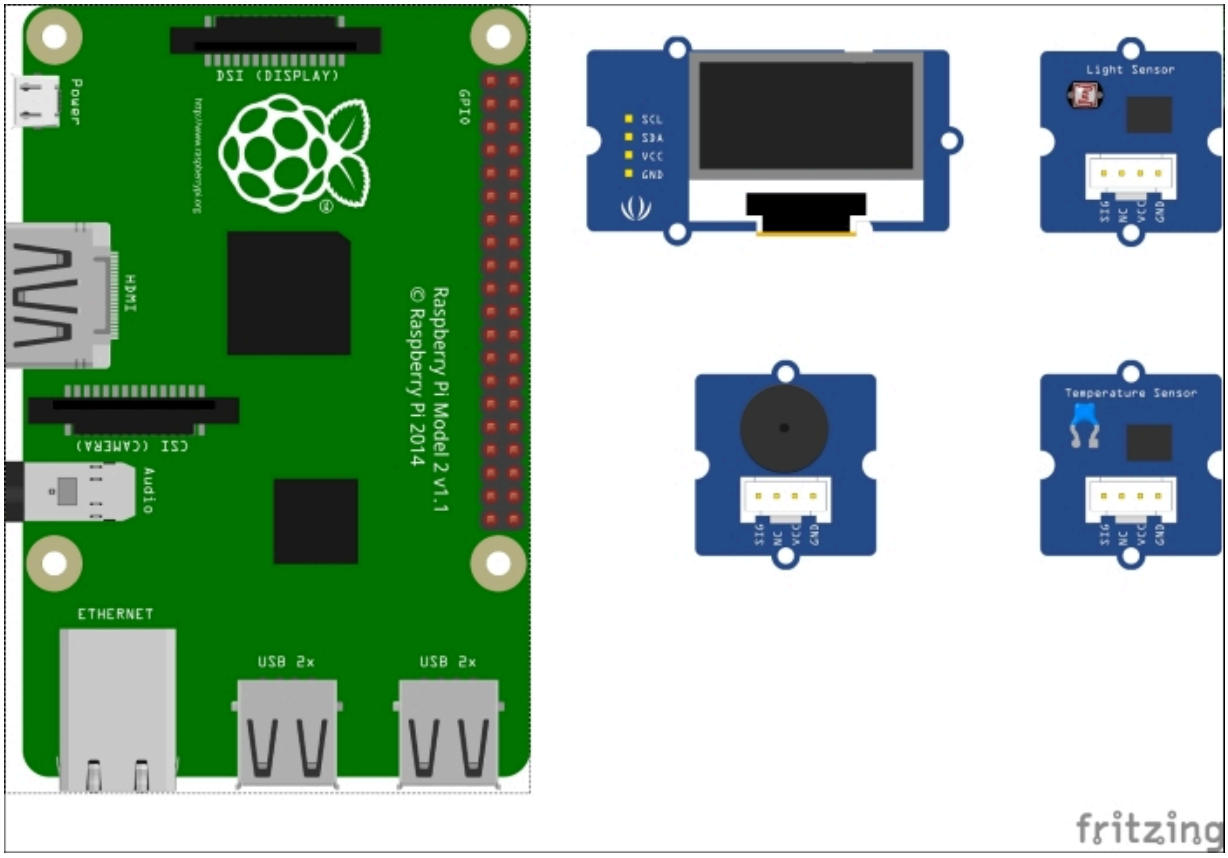
	id	name	email	password
1	2	Darth Vader	vader@deathstar.com	darkside

Finally, we have learned how to save and update data into a database, and retrieve it on demand for our use. We also know, from our previous Twitter tutorial, how to send updates to Twitter. So for our next project, we will build a tweeting weather station that will also save the data into a database. Sounds interesting? Let's get into it.

Building a tweeting weather station

This is a continuation of the project that we built in the previous chapter and that used the temperature and humidity Grove sensor with the Raspberry Pi to display the weather conditions. We will be using the same hardware.

The only difference now is that instead of displaying the weather on the terminal, we will be live-tweeting it, and also saving the information into a database for later retrieval.



To revisit the project, we should go through it once again by turning to Chapter 9, *Grove Sensors and the Raspberry Pi*. Much of the code that was used previously will be used again, with the addition of

the blocks where we tweet the information and save it in a database. For this project, it is recommended that you make a new Twitter account and create a new app on it to get the Consumer and Access codes.

The full Python script is presented next. We know what most of the code does because of the previous projects that we have built, including the weather station, Twitter bot, and SQLite database, but there is a brief summary at the end of the code.

```
from grovepi import *
from grove_oled import *
import tweepy
import sqlite3
import datetime

dht_sensor_port = 7

oled_init()
oled_clearDisplay()
oled_setNormalDisplay()
oled_setVerticalMode()
time.sleep(.1)

auth = tweepy.OAuthHandler(consumer_key,
consumer_secret)
auth.set_access_token(access_token,
access_token_secret)
api = tweepy.API(auth)

conn = sqlite3.connect('raspi_weather.db')
cursor = conn.cursor()

cursor.execute('''
    CREATE TABLE weather(id INTEGER PRIMARY KEY,
time TEXT,
                        temp INTEGER, hum INTEGER)
''')


while True:
       try:
        [ temp,hum ] = dht(dht_sensor_port,1)
        print "temp =", temp, "C\thumidity =",
hum,"%"
```

```
        t = str(temp)
        h = str(hum)
                time =
str(datetime.datetime.time(datetime.datetime.now()
))

        oled_setTextXY(0,1)
        oled_putString("WEATHER")

        oled_setTextXY(2,0)
        oled_putString("Temp:")
        oled_putString(t+'C')

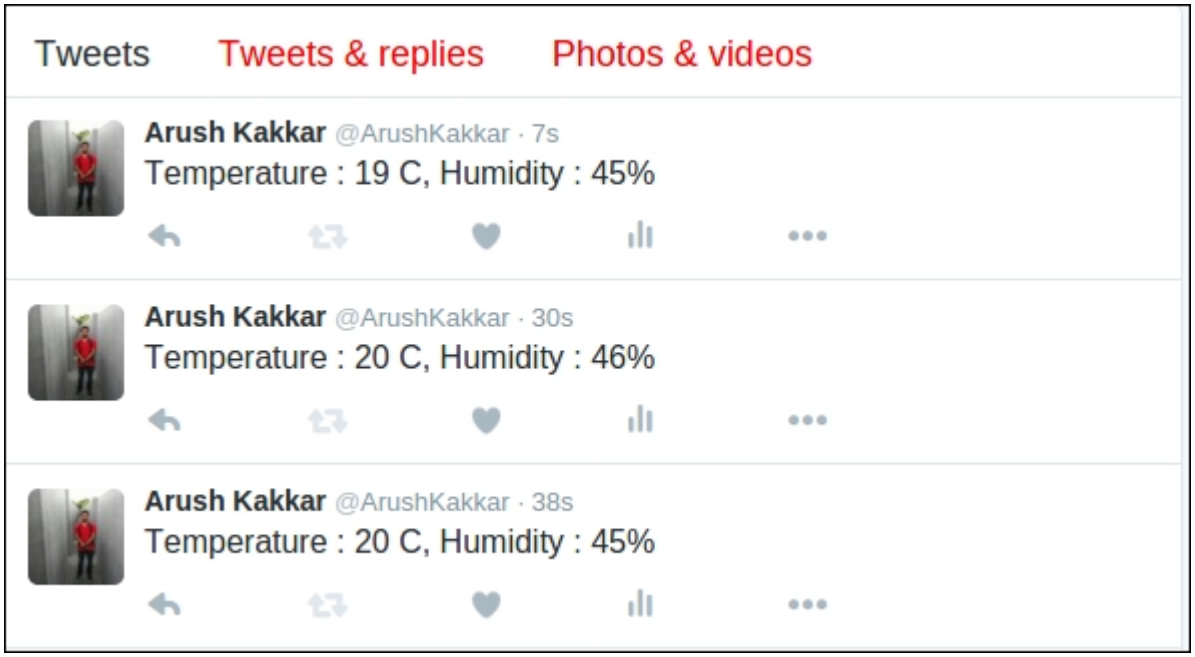
        oled_setTextXY(3,0)
        oled_putString("Hum :")
        oled_putString(h+"%")
                api.update_status("Temperature : "
+ t + " C, " + "Humidity : " + h + "% ")
                cursor.execute('''INSERT INTO
weather(time, temp, hum)

VALUES(?,?,?)''', (time, t, h))

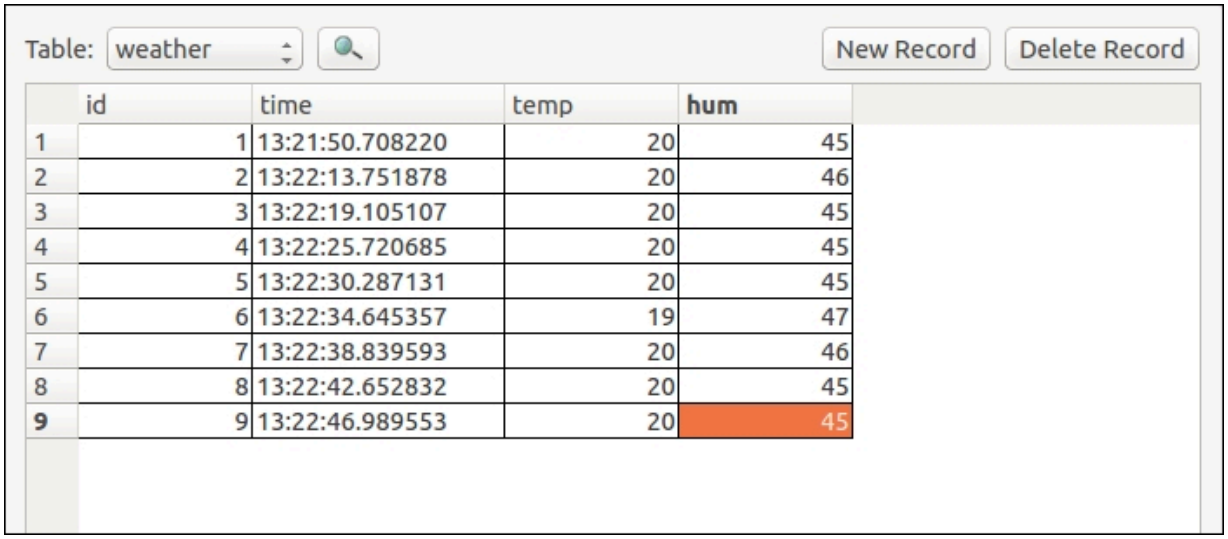
        except (IOError,TypeError) as e:
        print "Error"
```

As is evident from the preceding code, it is an updated version of the weather station we built in the previous chapter. We add the consumer and access codes for our Twitter application so that we can tweet the weather information directly from this script. We import the `Tweepy` library and set up the access. We also import the `sqlite3` module and create a new database called `weather`, which has three data columns: `time`, `temperature`, and `humidity`, in addition to the serial number column. Finally, in addition to displaying the temperature and humidity on the OLED screen, we also update it in our tweet and save it in the database we just created along with the time information. The time information is given by the `datetime` module that is included with the default Python installation. `datetime.datetime.now()` gives the current date and time. `datetime.datetime.time()` gives the current time provided we are only given `datetime.datetime.now()` as an argument.

The tweets from our script will look like this:



And if we browse the database that we just created, it looks something like this:



	id	time	temp	hum
1	1	13:21:50.708220	20	45
2	2	13:22:13.751878	20	46
3	3	13:22:19.105107	20	45
4	4	13:22:25.720685	20	45
5	5	13:22:30.287131	20	45
6	6	13:22:34.645357	19	47
7	7	13:22:38.839593	20	46
8	8	13:22:42.652832	20	45
9	9	13:22:46.989553	20	45

We have now built a weather station that not only monitors and saves weather information for you but also broadcasts it to the world! An interesting extension to this would be to add speech capabilities so that the Raspberry Pi speaks out the weather information via a speaker. The `espeak` library is an easy-to-use library that

accomplishes this easily and effectively. Moreover, you can also hook up to a weather service that has a web API and get your Pi to speak out weather forecasts too!

Adding speech capabilities to our weather station

Speech is one of the many ways in which humans communicate. And it's fascinating to make machines do the things that humans do effortlessly. For precisely that reason, we will add speech capabilities to our tweeting weather station so that it can speak out the weather, and we no longer have to stare at our OLED screen or scroll our Twitter feed to learn about the weather. For this task, we will be using an open source tool called `espeak`, which is a text-to-speech engine and also has an API for Python.

First, install the `espeak` library with the following commands:

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

The good thing about this library is that we can use it from the terminal or from the Python console. To try it out, make sure that you have a speaker connected to the Pi and execute the following command from the Linux terminal:

```
espeak I will tell you the weather'
```

This will produce the speech output and we will be able to hear it from the speaker. Cool, right?

Next, we will learn how to use it with Python and then also add it to our weather station project. Open a Python console by typing `python` in a Linux terminal. To import the dependencies, execute the following statement:

```
>>> from espeak import espeak
```

This imports the dependencies required for the speech output. Using `espeak` from Python is also as easy as:

```
>>> espeak.synth("I am the weatherman")
```

The `synth` method in the `espeak` module accepts a string and produces speech based on that. It's not the best-sounding voice, but it works!

To add this to the weather station, we can add the following statement inside our while loop:

```
espeak.synth("The current temperature is : " +
str(temp))
espeak.synth("Humidity level is : " + str(hum) +
"percent")
```

This works because Python supports string concatenation. This means that we can combine two strings just by adding a `+` sign between them. However, the `temp` and `hum` variables are not string variables. So we need to convert them to strings by the `str()` method. The `Str()` method takes any type of variable and converts it into a string.

We can also use the `espeak` library to tell us many different things about our environment by getting the state from the Grove Sensors.

Summary

In this chapter, we learned how to use Python to communicate with Twitter by using an open source library, Tweepy. We learned how to use Tweepy to send tweets. We also learned how to create a database, save and update data in it, and get some data from an existing database. Toward the end, we extended the capabilities of the weather station we built in Chapter 9, *Grove Sensors and the Raspberry Pi* and added tweeting and data-saving functionalities to it. We also added speech capabilities to it to make it easier to use.

In the next chapter, we will be exploring the interesting topic of parallel computing. We will be using multiple Raspberry Pis to create a cluster of computers that perform computations faster by working in tandem towards a single task. A master Pi will control other slave Pis and outsource its computations to them, so that the task is performed faster. We will use this setup to perform an n-body simulation, which is by itself a very computationally challenging task.

Chapter 11. Build Your Own Supercomputer with Raspberry Pi

In the previous chapter, we learned how to get the Raspberry Pi to communicate with the Internet, along with some more topics. We learned how to set up the Twitter API and how to send out tweets using it. We also learned how to save data in a database and retrieve it later. Finally, we combined the two concepts and the weather station we built in Chapter 9, *Grove Sensors and the Raspberry Pi* to create a tweeting weather station that also saves the temperature and humidity data in a database.

In all the chapters prior to this one, we explored the Raspberry Pi functionalities. But this chapter will be different in the sense that we will learn how to combine two or more Raspberry Pi devices to create a more powerful computer. Indeed, if you combine enough Raspberry Pi devices, you can make your own supercomputer! Sounds interesting? We will cover the following in this chapter:

- Using a network switch to create a static network for Pi devices
- Installing and setting up MPICH2 and MPI4PY and running on a single node
- Using the static network of multiple Pi devices is to create an **MPI** (short for **Message Parsing Interface**) cluster
- Performance the benchmarking of cluster
- Running GalaxSee (N Body simulation)

Introducing a Pi-based supercomputer

The basic requirements for the building of a supercomputer are as follows:

- Multiple processors
- A mechanism to interconnect them

We already have multiple processors in the form of multiple Raspberry Pi devices. We connect them via a networking hub, which is connected to each Pi via an Ethernet cable. A networking hub is nothing but a router that can accept Ethernet cables. There are dedicated networking hubs that can support many connections but for the purposes of this chapter, we can use a normal Wi-Fi router that accepts four LAN connections in addition to wireless connections. This is because we will not build a cluster of more than four Pi devices. You are welcome to add more devices to this cluster in order to increase your computing power. So, the hardware requirements for this chapter are as follows:

- Three Raspberry Pi devices
- Three Raspbian-installed SD cards
- A networking hub
- Three Ethernet cables
- Power supply for every device

Installing and configuring MPICH2 and MPI4PY

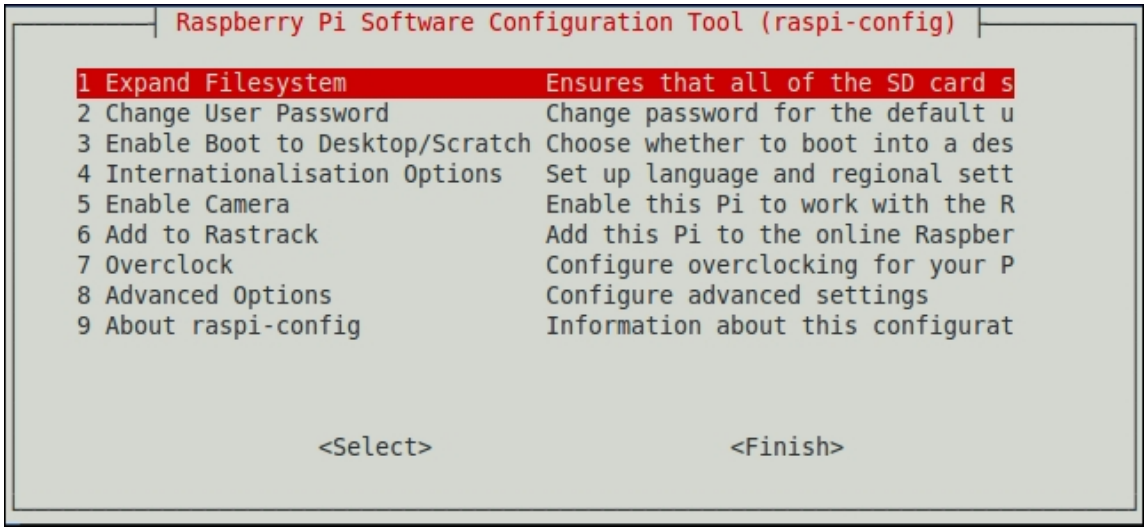
Before we can begin installing the libraries to a network of multiple Pi devices, we need to configure our Raspbian installation to make things a bit easier.

Boot up a Raspberry Pi and in the terminal, enter the following command:

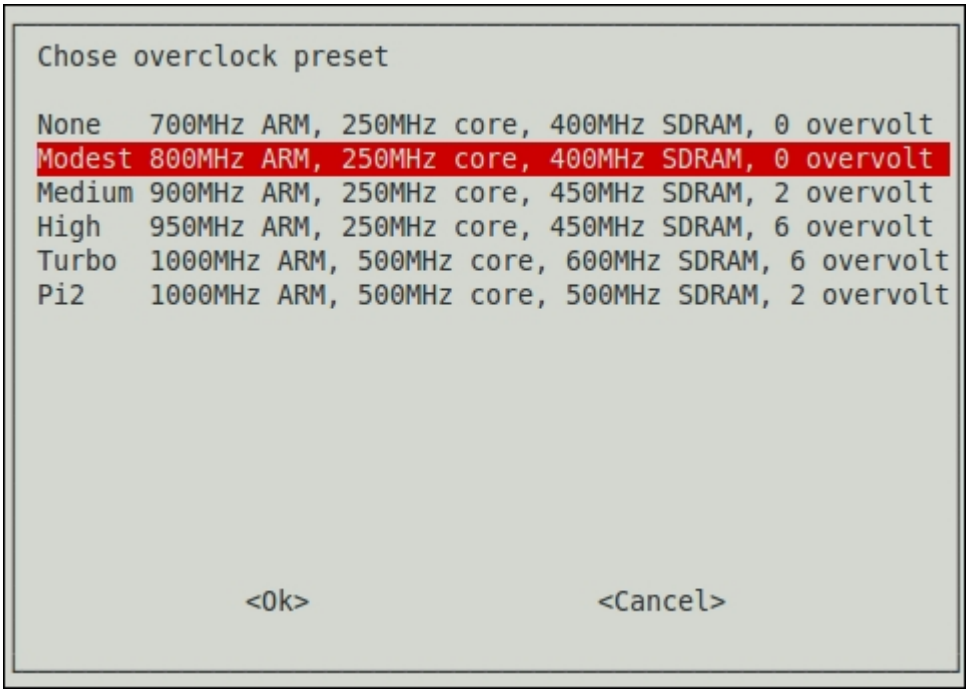
```
sudo raspi-config
```

It is assumed that you have followed all the setup steps that are preferred on first starting up the Raspberry Pi. These are as follows:

1. Expand the filesystem.

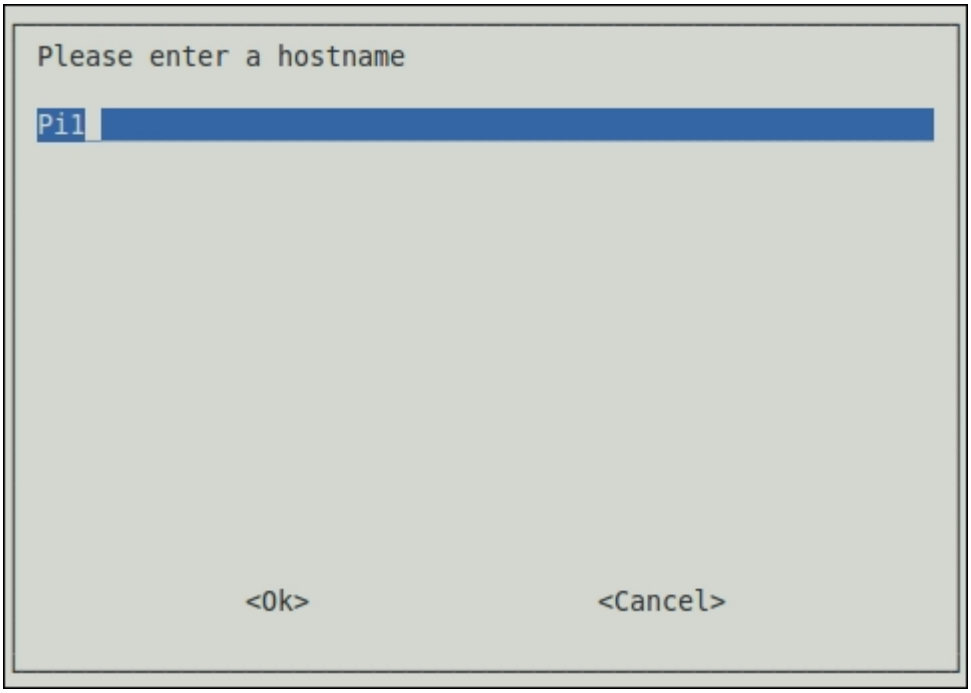


2. Overclock the system (this is optional). We will be using a Pi overclocked to 800MHz as shown below:

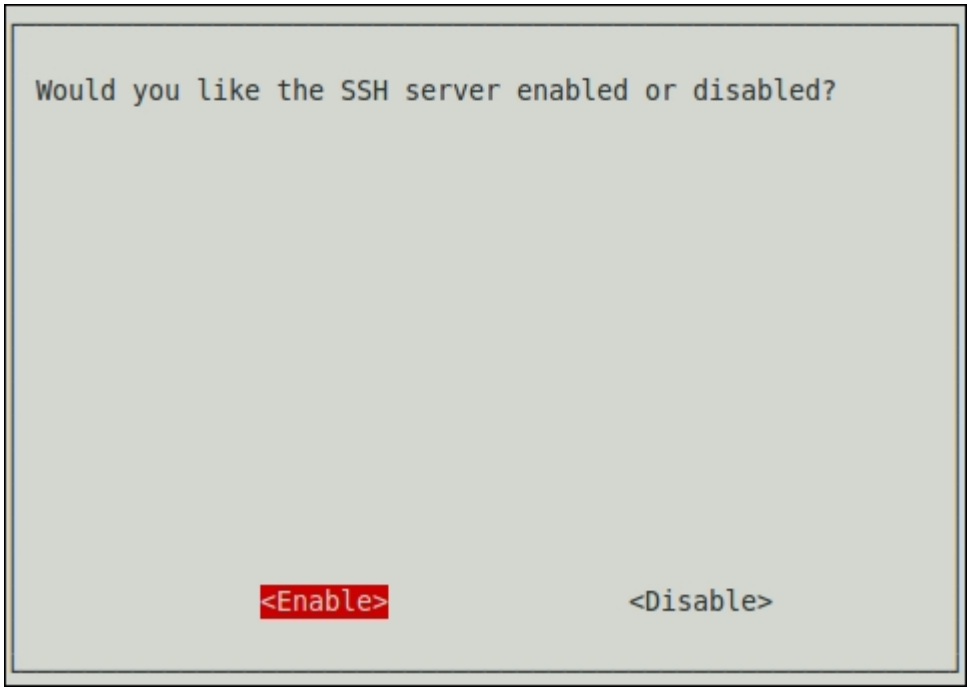


3. Now, enter the advanced menu by selecting **Advanced Options**. We need to configure the following:

    1. Set the hostname to `Pi1`.



    2. Enable SSH.



We need to enable auto login so that we do not need to manually log in every Pi once we fire it up. Auto login is enabled by default in the latest versions of the Raspbian operating system, so we don't need to perform the following procedure on the latest version. To enable auto login, exit the configuration menu, but don't reboot yet. To set the auto login, open the `inittab` file with the following command:

```
sudo nano /etc/inittab
```

Comment out the following line:

```
1:2345:respawn:/sbin/getty --noclear 38400 tty1
```

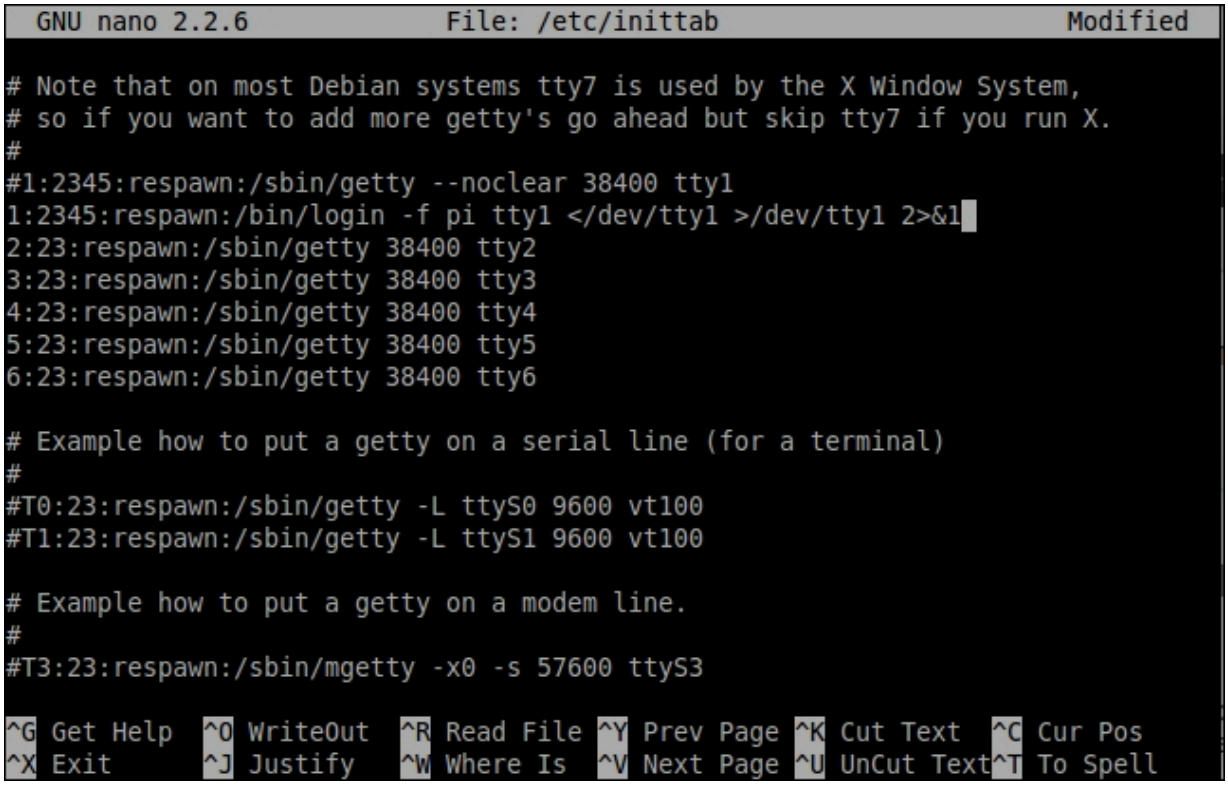
This is done so that it looks like this:

```
#1:2345:respawn:/sbin/getty --noclear 38400 tty1
```

Then, add the following line:

```
1:2345:respawn:/bin/login -f pi tty1 </dev/tty1
```

```
>/dev/tty1 2>&1
```



Save the file and reboot the Pi with `sudo reboot`. When it boots up, you should be auto-logged in.

We are now ready to proceed and begin installing the software that is required to run a cluster of computers.

Installing the MPICH library

MPICH is a freely available, easily portable, and widely used implementation of the MPI standard, which is a message parsing protocol for distributed memory applications used in parallel computing. Work on the first version of MPICH was started around 1992 and is currently in its third version. MPICH is one of the most popular implementations of the MPI standard due to the fact that it is used as a foundation for a vast majority of MPI implementations, including ones by IBM, Cray, Microsoft, and so on.

Note

To know more about the MPICH library, visit http://www.mpich.org/.

At the time of writing, the latest version of MPICH is 3.2. The installation instructions for this are presented. However, these should work on all the sub-versions of MPICH3 and most of the later versions of MPICH2.

To begin the installation, ensure that the Raspberry Pi has a valid Internet connection. Run the following commands in order to get a working installation of MPICH:

1. Update your Raspbian distribution:

```
sudo apt-get update
```

2. Create a directory to store the MPICH files:

```
mkdir mpich
cd ~/mpich
```

3. Get the MPICH tar file:

```
wget
http://www.mpich.org/static/downloads/3.2/mpic
h-3.2.tar.gz
```

4. Extract the tar archive:

```
tar xfz mpich-3.2.tar.gz
```

5. This will contain the MPICH installation and build the following:

```
sudo mkdir /home/rpimpi/
sudo mkdir /home/rpimpi/mpi-install
mkdir /home/pi/mpi-build
cd /home/pi/mpi-build
```

6. Fortran is a dependency for MPI:

```
sudo apt-get install gfortran
```

Now, since we created a build folder for the MPICH installation earlier, we need to configure the installation script and point it to the folder where we want the interface to be installed. We do this by running the `configure` script and setting the build prefix to the installation directory:

```
sudo /home/pi/mpich/mpich-3.2/configure -
prefix=/home/rpimpi/mpi-install
```

Finally, we carry out the actual installation.

```
sudo make
sudo make install
```

Note

Keep in mind that this build might take about an hour to complete. Make sure that the Raspberry Pi stays on for the entire period of the build.

Edit the `bashrc` file so that the path to the MPICH installation is loaded to the `PATH` variable every time we open a new terminal window. To do that, open the file in the terminal with the following command:

```
nano ~/.bashrc
```

Then, add the following line at the end:

```
PATH=$PATH:/home/rpimpi/mpi-install/bin
```

Save and exit the `bashrc` file by pressing *Ctrl* + *X* and then click on **Return**. We will now test the installation. For that, reboot the Pi:

```
sudo reboot
```

Then, run the following command:

```
mpiexec -n 1 hostname
```

If the preceding command returns `Pi1`, or whatever you set your hostname to be, then our MPICH installation is successful and we can proceed to the next step.

We will now look at a short C program that uses MPICH. This is given as follows:

```
#include "mpi.h"
#include <stdio.h>

void main(int argc, char *argv[]) {
    int rank, size
    MPI_Init(argc, argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("I have %d rank and %d size \n", rank,
size);
    MPI_Finalize();
}
```

This program runs on a single node and discovers the total number of nodes in a network. It then communicates with the MPI processes on other nodes and finds out its own rank.

The `MPI_Init()`function initiates the MPICH process on our computer according to the given information about the IP addresses in a network. The `MPI_Comm_rank()` function and the `MPI_Comm_size()` function get the rank of the current process and the total size of the network and save them in `rank` and `size` variables, respectively. The `MPI_Finalize()` function is nothing but an exit routine that ends the MPI process cleanly.

On its own, MPICH can be run using C and Fortran languages. But that might be a hindrance for a lot of users. But fret not! We can use MPICH in Python by installing the Python API called MPI4PY.

Installing MPI4PY

The conventional way of installing the `mpi4py` module from a package manager such as apt-get or aptitude does not work because it tries to install the `openMPI` library along with it, which conflicts with the existing installation of MPICH.

So, we have to install it manually. We do this by first cloning the repository from BitBucket:

```
git clone https://bitbucket.org/mpi4py/mpi4py
```

```
pi@Pi1 ~ $ git clone https://bitbucket.org/mpi4py/mpi4py
Cloning into 'mpi4py'...
remote: Counting objects: 9993, done.
remote: Compressing objects: 100% (3831/3831), done.
remote: Total 9993 (delta 6960), reused 8820 (delta 5965)
Receiving objects: 100% (9993/9993), 6.01 MiB | 100 KiB/s, done.
Resolving deltas: 100% (6960/6960), done.
pi@Pi1 ~ $
```

Make sure that all the dependencies for the library are met. These are as follows:

- Python 2.6 or higher
- A functional MPICH installation
- Cython

The first two dependencies are already met. We can install Cython with the following command:

```
sudo apt-get install cython
```

The following commands will install mpi4py with the `setup.py` installation script:

```
cd mpi4py
python setup.py build
python setup.py install
```

Add the path to the installation of mpi4py to `PYTHONPATH` so that the Python environment knows where the installation files are located. This ensures that we are easily able to import mpi4py into our Python applications:

```
export PYTHONPATH=/home/pi/mpi4py
```

Finally, run a test script to test the installation:

```
python demo/helloworld.py
```

Now that we have completely set up the system to use MPICH, we need to do this for every Pi we wish to use. We can run all of the earlier commands in freshly installed SD cards, but the better option would be to copy the card we prepared earlier into blank SD cards. This saves us the hassle of completely setting up the system from scratch. But remember to change the hostname to our standard scheme (such as Pi01 and Pi02) after the fresh SD card has been set up in a new Raspberry Pi. For Windows, there is an easy-to-use tool called Win32 Disk Imager, which we can use to clone the card into an `.img` file and then copy it to another blank SD card. If you are using OS X or Linux, then the following procedure should be followed:

1. Copy the OS from an existing card:

```
sudo dd bs=4M if=/dev/disk2
of=~/Desktop/raspi.img
```

Note

Replace `disk2` with your own SD card

2. Remove the card and insert a new card.
3. Unmount the card.
4. Format the SD card.
5. Copy the OS to the SD card:

```
sudo dd bs=4M if=~/Desktop/raspi.img
of=/dev/disk2
```

Note

To know more about the mpi4py library and to have a look at the various demos, you can visit the official Git repository on BitBucket at https://bitbucket.org/mpi4py/mpi4py.

Setting up the Raspberry Pi cluster

Now that we have a running Raspbian distribution in all the Raspberry Pi devices that we wish to use, we will now connect them to create a cluster. We would need a networking hub that has at least the same number of LAN ports as the size of your cluster should be. For the purposes of this chapter, we will use three Raspberry Pis, so we would need a hub that has at least three LAN ports.

If you have a Wi-Fi router, then it should already have some ports at the back, as shown in the following image:



We also need three networking cables, such as those with an Ethernet connector, where one port goes into the Raspberry Pi and the other goes into the networking hub. They will look like the ones shown in the following image:



Connect one end of the Ethernet cable to the Raspberry Pi and the other end to the router for all the devices. Once we complete this networking setup, we can move on to connecting the Raspberry Pi devices with software.

Setting up SSH access from the host to the client

In the setup described here, one master node will control the other slave nodes. The master is called the host and the slaves are called clients. To access the client from the host, we use something called SSH. It is used to get the terminal access to the client from the host, and the command used is as follows:

```
ssh pi@192.168.1.5
```

Here, the preceding IP address can be replaced with the IP address of our Raspberry Pi. The IP address of a Pi can be found by simply opening up a new terminal and entering the following command:

```
ifconfig
```

This will give you the IP address associated with your connected network and the interface, which is `eth0` in our case. A small problem with using this is that every time we try to SSH into a client, we need a password. To remove this restriction, we need to authorize the master to log in to the client. How we do that is by generating an RSA key from the master and then transferring it to the client. Each device has a unique RSA key, and hence, whenever a client receives an SSH request from an authorized RSA device, it will not ask for the password. To do that, we must first generate a public private key pair from the master, which is done using a command in the following format:

```
ssh-keygen -t rsa -C "your_email@youremail.com"
```

It will look something like this:

```
pi@Pi1 ~ $ ssh-keygen -t rsa -C "arushk1@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/pi/.ssh/id_rsa):
Created directory '/home/pi/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/pi/.ssh/id_rsa.
Your public key has been saved in /home/pi/.ssh/id_rsa.pub.
The key fingerprint is:
41:68:30:de:fb:78:ec:df:5e:ed:1c:0a:ec:86:de:ba arushk1@gmail.com
The key's randomart image is:
+--[ RSA 2048]----+
|    o. ..        |
|   . oo.         |
|    ... .        |
|       . .       |
|      . S        |
|       +  .    . |
|      . + .o  ...|
|       o .oo...o.|
|        oE=+o.  o|
+-----------------+
pi@Pi1 ~ $
```

To transfer this key into the slave Raspberry Pis, we need to know each of their IP addresses. This is easily done by logging into their Pi and entering `ifconfig`.

Next, we need to copy our keys to the slave Raspberry Pi. This is done by the following command, replacing the given IP address with the IP address of your own Raspberry Pi:

```
cat ~/.ssh/id_rsa.pub | ssh pi@192.168.1.5 "mkdir
.ssh;cat >> .ssh/authorized_keys"
```

Now that we have copied our keys to the slave Raspberry Pi, we should be able to log in without a password with the standard SSH command:

```
ssh pi@192.168.1.5
```

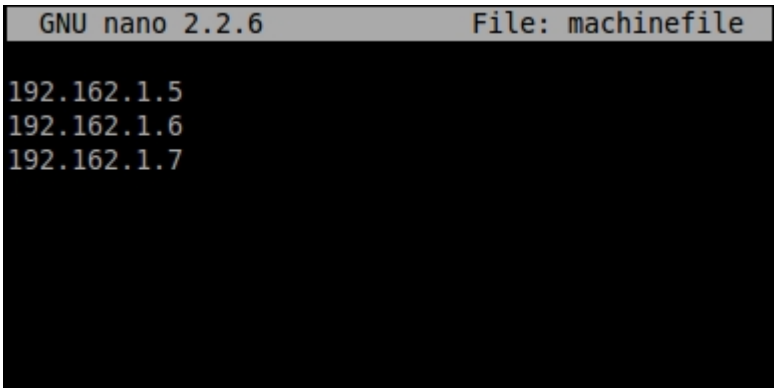
Running code in parallel

We have now successfully completed the prerequisites for the running of code on multiple machines. We will now learn how to run our code on the Raspberry Pi cluster using the MPICH library. We will first run a demo included with the `mpi4py` module, and then we will run an N-Body simulation on our cluster. Sounds fun? Let's get into it.

First, we need to tell the MPICH library the IP address of each of the Pis in our network. This is done by simply creating a new file called `machinefile` in the home folder, which contains a list of the IP addresses of all the Raspberry Pis connected to our network. Execute the following command to create the file:

```
nano machinefile
```

Then, add the IP address of each Raspberry Pi in a new line so that the final result looks like this:



Note that we have added three IP addresses because two Raspberry Pis are slaves and one is a master. So, the MPICH library knows that it needs to run the code on these three devices. Also, note that MPICH does not differentiate between devices. It just needs a valid installation of the library to run properly. This means that we can add different versions of Raspberry Pi to our cluster or

even other computers such as a BeagleBone, which has a valid installation of MPICH.

To test our setup, we navigate to the `mpi4py/demo` directory with the following command:

```
cd ~/mpi4py/demo
```

Then, we run the following command:

```
mpiexec -f ~/machinefile -n 3 python helloworld.py
```

It will give an output similar to this:

```
Hello, World! I am process 1 of 3 on Pi1.
Hello, World! I am process 2 of 3 on Pi2.
Hello, World! I am process 3 of 3 on Pi3.
```

Performance benchmarking of the cluster

Since we have successfully created a cluster of Raspberry Pis, we now need to test their performance. One way of doing that is to measure the latency between different nodes. To this end, Ohio State University has created some benchmarking tests that are included with the `mpi4py` library. We will run a few of these to measure the performance of our cluster. Simply put, latency measures the amount of time a packet takes to reach its destination and get a response. The lower the latency, the better a cluster will perform. The tests are given in the demo folder of the `mpi4py` directory. These run in this fashion: the `osu_bcast` benchmark measures the latency of the `MPI_Bcast` collective operation across N processes. It measures the minimum, maximum, and average latency for various message lengths and over a large number of iterations.

To run the `osu_allgather.py` test, we execute the following command:

```
mpiexec -f machinefile -n 3 python
osu_allgather.py
```

As an interesting experiment, try to gather the latency variation with different sizes of data. It should give you a unique insight into the functioning of the network.

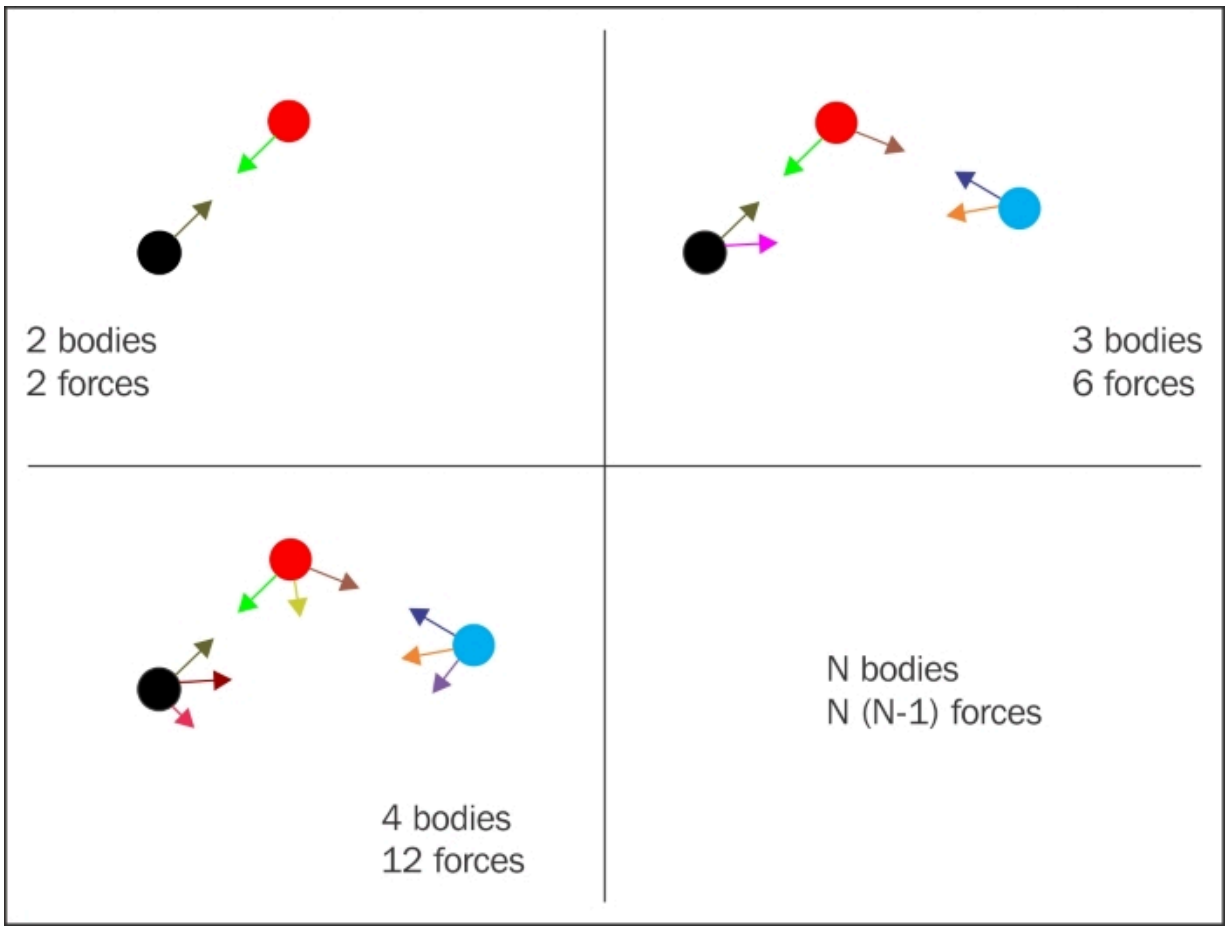
Introducing N-Body simulations

An N-Body simulation is a simulation of a dynamic system of particles that are under the influence of physical forces, such as gravity or magnetism. It is mostly used in astrophysics to study the processes involving nonlinear structure formation, such as the formation of galaxies, planets, and suns. They are also used to study the evolution of the large-scale structure of the universe, including estimating the dynamics of a few body systems such as the earth, moon, and sun.

Now, N-body simulations require a lot of computational resources. For example, a 10-body system has 10*9 forces that need to be computed at the same time. You can see that, if we increase the number of particles, the number of forces increases exponentially. So a 100-body system will require 9,900 forces to be calculated simultaneously. That is why N-Body simulations are generally run on powerful computers. However, we are going to use our Raspberry Pi cluster to accomplish this task because it has three times the computational power of a single Raspberry Pi. For this task, we will use an open source library called **GalaxSee**.

GalaxSee is modeled as discrete bodies interacting through gravity. The acceleration of an object is given by the sum of the forces acting on that object, divided by its mass. If you know the acceleration of a body, you can calculate the change in velocity, and if you know the velocity, you can calculate the change in the position. Consequently, you will have the position of each object at every point of time. The more objects you have, the more forces you will need to calculate, and every object must know about every other object. To reduce the computational load on a single processor, the Galaxy code is an implementation of simple parallelism. The most computationally heavy calculations involve calculating the forces on a single object, and hence, each client is involved with that calculation given the object's information and the information about each object that

exerts force on it. The master then collects the computed forces from the clients.



In this way, we can simulate an N-Body simulation with code running in parallel. Next, we will see how to install and get GalaxSee running.

Note

To learn more about the GalaxSee library, visit https://www.shodor.org/master/galaxsee/.

Installing and running GalaxSee

Now that we have a basic idea of what an N-Body simulation is and how different bodies interact with each other, we will proceed to the installation of the GalaxSee library.

First, we need to download the archive of the source code. We can do this with the `wget` command:

```
wget
http://www.shodor.org/refdesk/Resources/Tutorials/
MPIExamples/Gal.tgz
```

Next, we extract the `.tgz` archive, assuming it was downloaded to the home folder. If the path is different, we need to enter the correct path:

```
tar -xvzf ~/Gal.tgz
```

Navigate to the folder:

```
cd Gal
```

We need to use `Makefile` to successfully build GalaxSee on our Raspberry Pi. To do that, open the file in the terminal:

```
nano Makefile
```

Then, change the line:

```
cc = mpicc
```

to:

```
cc = mpic++
```

This is done so that the final result looks like this:

```
 GNU nano 2.2.6                                          File: Makefile


CC        = mpic++

LIBS      = -lX11 -lm

CFLAGS    = -I/opt/mpich/include
LDFLAGS   = -L/usr/X11R6/lib -L/opt/mpich/lib $(LIBS)

PROGRAM   = GalaxSee                    # name of the binary
SRCS      = Gal.cpp derivs.cpp diffeq.cpp modeldata.cpp point.cpp derivs_client.cpp
OBJS      = $(SRCS:.cpp=.o)             # object file

.SUFFIXES: .c .o

.cpp.o:
        $(CC) -c $(CFLAGS) $<

default: all

all: $(PROGRAM)

$(PROGRAM): $(OBJS)
        $(CC) -o $(PROGRAM) $(SRCS) $(CFLAGS) $(LDFLAGS)

clean:
        /bin/rm -f $(OBJS) $(PROGRAM)
```

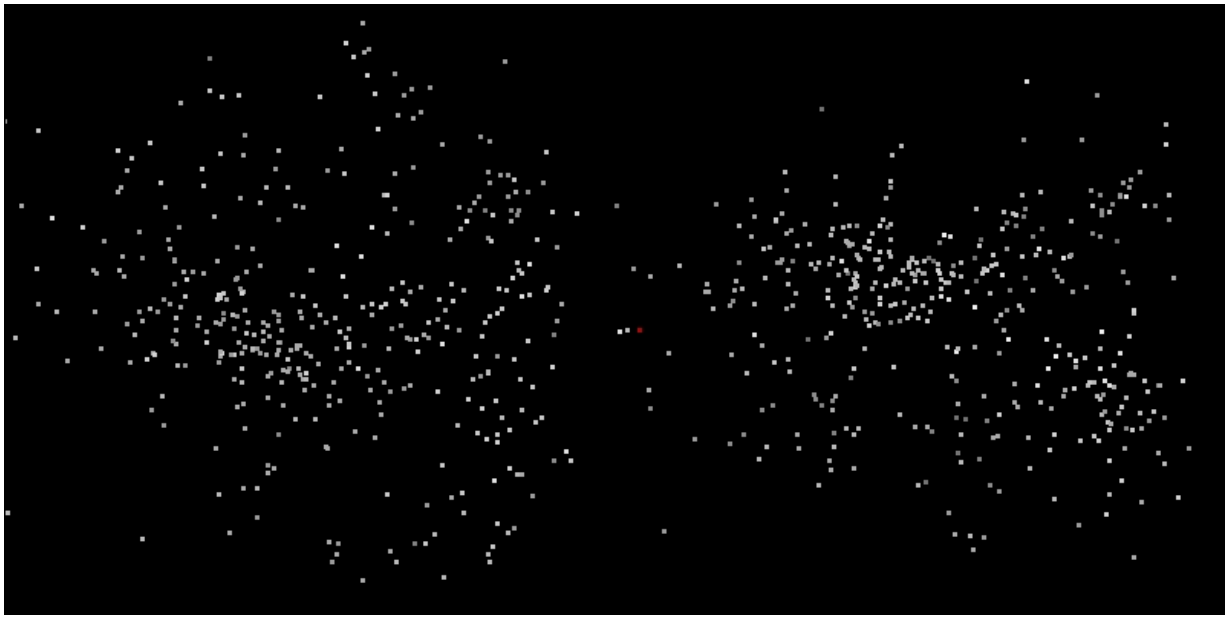
To build the program, simply run this:

```
make
```

Now if you enter `ls` inside the `Gal` folder, you will see a newly created executable file named `GalaxSee`. Run the file with the following command:

```
./GalaxSee
```

This will execute the program and will open a new window that looks like the following:



Here, the white dots are the bodies that are interacting with each other with the force of gravity. There are a few options that you can configure in GalaxSee to change the number of bodies or their interaction. For example, run the following command:

```
./GalaxSee 200 400 10000
```

Here, the first argument to the program, `200`, gives the number of bodies to be simulated. The second argument gives the speed of the simulation. The higher the number, the higher the speed. The third argument gives the amount of time to run the simulation in milliseconds. So, in the preceding command, we will simulate 200 bodies for 10 seconds. Obviously, we can experiment with different variables and see how these variables affect the simulation.

The preceding example was running on a single Raspberry Pi. But as promised, we will now see how to run it on a cluster. It will probably be surprising if we learn that the program can run on a cluster of Raspberry Pi devices with a single command and one that we've seen before. The following command does the trick:

```
mpiexec -f ~/machinefile -n 3 ./GalaxSee
```

In the preceding command, `mpiexec` initiates the MPI process on our host computer. Here, we will give it three points of data on which to execute the GalaxSee program. These are as follows:

- The path to the file containing the IP addresses for each device in the network
- The number of devices to be used
- The path to the executable file

The `-f` option specifies the machinefile. The `-n` option specifies the number of devices to be used, and finally, we give the `GalaxSee` executable file to it.

Try it for yourself! If you try to increase the number of bodies in the simulation on a single Raspberry Pi, it would probably run slower. But if you try the same on a cluster, it will definitely run faster because now, we have a lot more computational resources at our disposal.

Summary

This chapter was unique in the sense that we learned how to execute software not on a single Raspberry Pi but on multiple ones. We learned the advantages of using parallel computation and learned how to build our own network of clustered Raspberry Pi devices. We also learned how to install the libraries required for parallel computations so that we can run our own software in the cluster. We learned how to configure the Raspberry Pis in our network to communicate with each other and make the communication hassle-free. Once we set up a system, we tested it to to measure the response in the form of latency.

Finally, we learned about the concept of N-body simulation and configured an open source program to simulate it. W also saw how increasing the number of bodies in the simulation could affect the speed of the program's execution.

In the next chapter, we will expand upon this knowledge about clusters and learn about advanced networking concepts, such as DNS and DHCP. We will learn how to add functionalities such as a domain name service and auto detection of nodes and how to initiate a remote shutdown of the cluster.

Chapter 12. Advanced Networking with Raspberry Pi

In the previous chapter, we built a cluster of Raspberry Pis that all interact with each other on a common network to accomplish a single task at a much greater speed than a single Raspberry Pi would perform. In fact, the MPI library is basically a networking library that delegates some computations to other processors connected in a network so that the master node doesn't have to deal with them. Now, in this chapter, we will learn about special networking concepts that power them kinds of cluster computers and in fact, the whole Internet. In this chapter, we will learn about:

- Adding DHCP capabilities to our cluster
- Adding a domain name system capability to our cluster
- Writing a script for the autodetection of nodes
- Writing scripts for the remote shutdown of cluster nodes

Introducing DHCP

DHCP (**Dynamic Host Configuration Protocol**): The host is simply a server, which means that the host computer is responsible for providing a service. The service could be of any kind. For example, a web server is responsible for hosting the files required for viewing a website. Again, the server implements the necessary communication protocols to allow many players to play against each other. A computer or application that uses the services offered by a server is called a client. Now you might know that every computer on the Internet is identified by a unique address known as the IP address.

A DHCP server is responsible for allocating such IP addresses to all the nodes in a network, and keeping track of the allocated addresses. In DHCP, dynamic means constantly changing, host means server, configuration refers to configuring your network

settings, and protocol means a set of rules on how to do things. Described next is the procedure for the allocation of IP addresses by a DHCP server.

Suppose a node wants to join a network. It first sends a message to the server asking to be allocated an IP address. The server sends a message with an IP address and a time duration for which the node might keep that IP address. Then the server notes down the IP address given to the node in a table, along with the time it was given at and the validity duration.

Now, when the node joins a network, it has no way of knowing if there is a DHCP server. So it sends out a broadcast signal to the whole network intended just for the DHCP server. If there is a server, it sends out a reply offering an address.

Now, consider the situation when a node is shutting down or leaving the network cleanly. It sends out a signal to the server identifying itself and offers the IP address back to the DHCP server. The server then modifies the table, strikes off the host's name from the IP address, and then marks it available for allocation to other nodes in the network. But what if the node doesn't exit cleanly? For example, when the machine loses power or the network cable is just yanked out of it. This is where the time duration of allocation comes in handy. Once the duration of allocation of that IP address has passed, the node is automatically struck off from the table and that address becomes available again. Every time the server receives a packet of data from the host, its duration of allocation is automatically refreshed.

A few networking concepts

Since we have learned how a DHCP server operates, we will now work through an introduction to some important concepts related to networking that might be useful in understanding the operation of the network and also for debugging any errors we may encounter.

Though it might not look like it on first read, an IP address follows a predefined set of conventions and it is not handed out randomly. The IP address that is visible, for example `192.168.1.10`, is actually composed of binary numbers. Each number separated by a decimal is actually a binary octet. So the preceding IP address actually looks like the following:

```
11000000 10101000 00000001 00001010
```

Now, an IP address is classified into multiple classes according to the address allocated to it by the DHCP server. They serve as a hierarchical system and also help to segment the IP address into classes such as private address and external address. They also help in distributing the IP address ranges to entities such as countries, service providers, and so on. There are the following classes of IP addresses:

- Class A: In this class, the first bit of the first octet is always set to zero. So, the first octet ranges from `00000000` – `01111111`, or `0` – `127`.

  So Class A can only have an IP address from `1.X.X.X` – `126.X.X.X` as `127.0.0.1` is reserved for loopback IP addresses, that is, for the machine to refer to itself. Hence, this class can only have 126 networks (2^7 – 2) and 16777214 (2^24 -2) hosts.

- Class B: Here, the first two bits of the first are set to `1` and `0` respectively. So, the first octet ranges from `10000000` – `10111111` or, `128` – `191`.

So, Class B can only have IP addresses ranging from `128.0.X.X` – `191.255.X.X`. Hence, this class can have 16,384 (2^14) networks and 65,534 (2^16 – 2) hosts.

- Class C: The first three bits of the first octet are set to 110. The range of the first octet is from `11000000` – `11011111`, or `192` – `223`. So, Class C can have IP addresses ranging from `192.0.0.X` – `234.255.255.X`. This means that Class C has a total of 20,97,152 (2^21) networks and 254 (2^8 -2) host addresses.
- Class D: This class of network has the first four bits of the first octet set as `1110`. That gives it a range of `11100000` – `11101111` or `235` – `239`. However, unlike the preceding three classes, Class D is used exclusively for multicasting. So, there is no need to get individual IP addresses, because individual host data is not required for this network to operate.
- Class E: This class is reserved exclusively use for experiments, or R&D, or study. The IP addresses in this class can range from `240.0.0.0` – `255.255.255.254`.

Now, we move on to the concept of a subnet mask. A subnet mask is basically a 32-bit address that divides an IP address into the network and host addresses. A subnet mask sets all the bits for the network as 1 and all the bits for the host as 0. So, the subnet masks for the different classes of networks are given as follows:

- Class A: `255.0.0.0`
- Class B: `255.255.0.0`
- Class C: `255.255.255.0`

Class D and Class E do not have subnet addresses because Class D has multicast addresses and Class E only consists of reserved addresses.

Configuring a Raspberry Pi to act as a DHCP server

Now, we will perform a hands-on implementation of the preceding concept by setting up a Raspberry Pi from our cluster to act as the DHCP server. The Raspberry Pi selected for the purpose should be properly marked for identification because there can be only one DHCP server in a network. More than one can cause a lot of problems! If we have more than one DHCP server in a single network, the client nodes will get confused as to which server to send the IP allocation request to. We first need to install a program called `dnsmasq`, and we can do that with the following commands:

```
sudo apt-get update
sudo apt-get install dnsmasq
```

Once this is installed, we can go ahead and disconnect the Internet from the LAN and connect the network to a non-DHCP hub. Why, you may ask? Because the conventional routers that we use have a DHCP server built into them. So configuring a Raspberry Pi as a server and connecting it to the router might cause some problems. So we just connect it to a networking hub that interconnects all the nodes. We are now ready to modify the Raspberry Pi to get it running as a server.

By convention, DHCP servers will have the lowest IP address in a network from the range of IP addresses. And for a lot of private networks, this IP address space looks like this: `192.168.0.X`. This means that the DHCP server will generally have `192.168.0.1` as the IP address and the other nodes will have `192.168.0.2`, `3`, `4`, and so on up to 254 IP addresses.

So, logically, we need our Raspberry Pi to have this address and hold on to it so it defaults to the DHCP server every time the network is initiated. So, edit the network interfaces file:

```
sudo nano /etc/network/interfaces
```

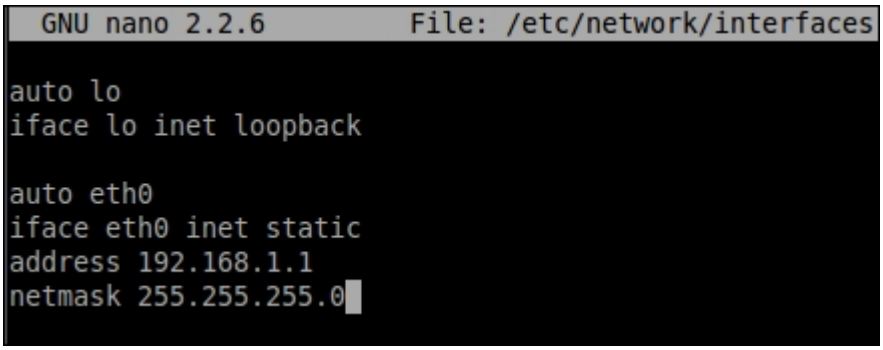
In the interfaces file, `eth0` refers to the connection from the `ethernet` port. If you've connected a Wi-Fi dongle to it, you will also see a `wlan0` interface. This translates to `Wireless LAN 0`. Now, since our Pi is connected via an `ethernet` port, we find the following statement:

```
iface eth0 inet dhcp
```

Basically, this line is the one responsible for the Raspberry Pi acting as a DHCP client because it tells the Pi to try to get an IP address from a DHCP server, from the interface `eth0`. But since we want our Raspberry Pi to act as a DHCP server in the network, we comment this line out. Then, to set a static IP address for our server, we add the following lines:

```
# iface eth0 inet dhcp
auto eth0
iface eth0 inet static
address 192.168.1.1
netmask 255.255.255.0
```

The `interfaces` file might look something like:



The **static** option tells the Raspberry Pi to configure a static IP address and **netmask**, which is given in the statement below. Now, save this file by pressing *Ctrl* + *X* and *Enter*. To restart the networking service, enter the following command:

```
sudo service networking restart
```

Now, the Raspberry Pi's IP address will always be set to **192.168.1.1**. We can also check this, as we did previously, by entering the command:

```
ifconfig
```

Now, we need to set up the software we installed earlier: `dnsmasq`. This is nothing but a DHCP server application for which we need to manually specify a configuration file according to our preferences. But since there already exists a default configuration file, we will make a backup of it. To do that, enter the following commands:

```
cd /etc
sudo mv dnsmasq.conf dnsmasq.default
```

The following command creates a new configuration file for `dnsmasq`:

```
sudo nano dnsmasq.conf
```

Once in the editing screen, enter the following lines into the configuration file:

```
interface=eth0
dhcp-
range=192.168.1.2,192.168.1.254,255.255.255.0,12h
```

The first line in the file specifies the interface on which to listen to incoming DHCP requests. In this case, we have set it up to be the `ethernet` port of the Pi. The `dhcp-range` takes in four parameters:

- Lowest IP address to be allotted
- Highest IP address to be allotted
- Netmask
- Lease duration

We can now save and close the file by pressing *Ctrl* + *X* and *Enter*. For the settings to take effect, we need to ensure that our DHCP serves the Raspberry Pi and is the only device connected to the networking hub. Once all the other devices are removed, we can restart the DHCP service by:

```
sudo service dnsmasq restart
```

The DHCP server is now running and can listen to requests from clients connected to the same networking hub.

Finally, ensure that none of the other Raspberry Pis have a static IP address that is in conflict with our DHCP server. Preferably, there should not be any Pi with a static IP configured. The `/etc/network/interfaces` file should be as follows:

```
iface eth0 inet dhcp
# auto eth0
# iface eth0 inet static
# address 192.168.0.1
# netmask 255.255.255.0
```

Exit nano and restart the networking service with the following command:

```
sudo service networking restart
```

Now, we can connect all the Client Raspberry Pis to the networking hub. Once they are all switched on, they should be able to acquire their respective IP addresses from the DHCP server immediately. We can also check the allocated IP addresses by running the `ifconfig` command on the clients. The IP addresses are allocated randomly from the range given to the DHCP server.

Once the IP addresses are acquired, everything should work as expected. We can check the connections by pinging different nodes from another node. Do this with the following command:

```
ping 192.168.1.X
```

where `X` is the IP address of the other node. We can also check the connection to the DHCP server with:

```
ping 192.168.1.1
```

Another interesting experiment that can be performed with the setup is actually observing the connections go up and down. Firstly, we can shut down a client and observe it give back the IP address to the server. Run the following on a client of your selection:

```
sudo ifdown eth0
```

Once we do that, we will see an output similar to the following:

```
Listening on LPF/eth0/b8:27:eb:a8:6b:cc
Sending on   LPF/eth0/b8:27:eb:a8:6b:cc
Sending on   Socket/fallback
DHCPRELEASE on eth0 to 192.168.1.1 port 67
```

`DHCPRELEASE` is the actual statement where the IP address is handed back to the server by the client. Similarly, we restart the `eth0` and observe the DHCP address being acquired. Run the following command:

```
sudo ifup eth0
```

We will see an output similar to the following:

```
Listening on LPF/eth0/b8:27:eb:a8:6b:cc
Sending on   LPF/eth0/b8:27:eb:a8:6b:cc
Sending on   Socket/fallback
DHCPDISCOVER on eth0 to 255.255.255.255 port 67
interval 7
DHCPREQUEST on eth0 to 255.255.255.255 port 67
DHCPOFFER from 192.168.1.1
```

```
DHCPACK from 192.168.1.1 bound to 192.168.0.X --
renewal in 40000 seconds
```

In this exchange, `DCHPDISCOVER`, `DHCPREQUEST`, `DHCPOFFER`, and `DHCPACK` are of importance. They correlate with the acquisition procedure in the IP address through a DHCP server, as described before. It is left as an exercise to the reader to figure out their respective functions.

Introducing Domain Naming System (DNS)

In the previous section, we learned to add the DHCP server to our Raspberry Pi cluster. This was done mainly so that each Raspberry Pi in our cluster could be identified by a unique IP address. Now, as you can imagine, the bigger our cluster grows, the harder it is going to be to remember the IP address of each Raspberry Pi. Humans find it hard to remember long sequences of numbers, which is what the IP addresses are. But there is a way by which we can easily connect to different nodes without having to remember their IP addresses.

This method is called a **Domain Name System** (**DNS**). Its basic function is to translate IP addresses into words so that it becomes easier for us to remember. For example, the DNS server will translate the IP address `192.168.5.43` into `raspberrypi.org` and vice versa to make it easier to connect to. Indeed, this is what happens on the Internet. Each website has an IP address behind it.

When you type a URL into your browser, your computer first contacts the DNS server and asks for the IP address that it is associated with. The DNS server then searches its database for the name and returns the IP address to your computer. This whole process is called a DNS query.

Now, we can have multiple DNS servers in a single network. This way, when one DNS server doesn't have an address saved in its database, it can pass the query to the second DNS server, and so on. This is known as an iterative DNS query. This is what actually happens on the Internet. There are millions of websites on the Internet and having a single DNS server that holds a large database is not only unfeasible but also unreliable. So, there are millions of DNS servers that are distributed across the world and act together to resolve billions of translation requests from millions of clients.

Note

But here is an interesting problem. In a DHCP network, IP addresses are dynamic, which means that the IP address of a node might change. Think about how a DNS server might handle that situation. You can do that as an exercise.

Setting up a DNS server on the Pi

Now, we will learn about the practical part of setting up a Domain Name System server on our own cluster of Raspberry Pis so as to learn the intricacies of the system. We will set up a DNS server on the same Pi on which we set up the DHCP server earlier. Since we took care to see that it is easily identifiable from the other client Raspberry Pis, we can begin the setting-up procedure.

We use the same software, `dnsmasq`, which we used for the DHCP server, as it also has the capability to run a DNS server. Other than that, we don't need to install anything extra. As a sanity check, before beginning the following tutorial, try to ping the server from one of the clients. Although multiple DNS servers can be configured, for the purposes of this tutorial, we will be using only one Raspberry Pi as the server.

As the DNS server doesn't have a broadcast system like DHCP, the clients can't locate it easily. So they need to be told the IP address of the DNS server, and one way of doing it is to make the DHCP server pass the IP address to the client. This can be done at the same time as the DHCP server is allotting its own IP address. In this case, the IP address of both the DHCP server and the DNS server is the same. Open up the dnsmasq configuration file (`dnsmasq.conf`) for editing with the following command:

```
sudo nano /etc/dnsmasq.conf
```

The first thing we do is specify that the IP address of the DNS server and DHCP server is the same. To do this, add the following line at the end of the file:

```
dhcp-option=6,192.168.1.1
```

Now that the clients know where the DNS server is located, we need to create a lookup database that contains the IP addresses with their corresponding names. Although in actual DNS servers a more sophisticated database platform is used, for example SQL, here we will use a simple text file as a demonstration. Add the following two lines to the configuration file:

```
no-hosts
addn-hosts=/etc/hosts.dnsmasq
```

Here, the `no-hosts` line tells `dnsmasq` to ignore the default `/etc/hosts` file for DNS queries. And the next line specifies that our own file, `/etc/hosts.dnsmasq`, should be used for DNS queries.

The next step in this process is to create our hosts file. To do that, enter the following command:

```
sudo nano /etc/hosts.dnsmasq
```

Since this file that didn't exist before, we should now be editing a blank file. New entries can be added in a specific format, which is the IP Address, followed by a tab, followed by the name. For example:

```
192.168.1.1     server
```

Here, the first part is the IP address, the second part is the tab spacing, and the third part is the name associated with that IP address.

Note

We have to add a tab instead of multiple spaces for it to work.

Now, save and exit this file. Make sure that the server is the only device connected to the networking switch. Unplug all other devices

before restarting the `dnsmasq` service. Restart the service with the following command:

```
sudo service dnsmasq restart
```

But there is one more thing. Since our server is set up with a static IP address, we also need to tell the DHCP server to use its own DNS server. We do that through a file named `/etc/resolv.conf`. Open it for editing by:

```
sudo nano /etc/resolve.conf
```

Change the contents of the file to:

```
nameserver 127.0.0.1
```

Now, `127.0.0.1` is a reserved IP address that is used to refer to one's own system. So, it would be the same if we put `192.168.1.1` as the IP address. However, if the server's IP address changes, we would need to change this file manually. So, we put 127.0.0.1 as the IP address of the nameserver. Exit the file and restart the networking service:

```
sudo service networking restart
```

Finally, connect all the client Raspberry Pis to the networking hub. They should start receiving their allocated IP addresses. Once all the Pis are connected to the network, you can run:

```
ping server
```

And you will start getting a response. If this is successful, it means that the ping request was first passed to the DNS server inside the server Pi and, when the hostname was resolved, a response was obtained from the IP address of the server.

Configuring the setup for a web server

Let's say, for example, one of the Raspberry Pis is configured as a web server. So, the task now will be to set up the cluster in such a way that you can type a name in a browser and have the home page of the web server load up!

By convention, web servers are supposed to have static IP addresses. This is so that DNS servers can store their hostnames without worrying too much about IP addresses changing. Since we earlier put in the entire range of IP addresses possible as the range for the DHCP server, we will modify that range so that we can reserve some static IP addresses for our web servers. For this, again edit the `/etc/dnsmasq.conf` file and modify the `dhcp-range` variable to the following:

```
dhcp-
range=192.168.1.40,192.168.1.254,255.255.255.0,12h
```

So, now we have non-DHCP IP addresses from 192.168.1.2 to 192.168.1.39, which we can allot to the web servers. One of the ways to do this is to modify `/etc/network/interfaces` and `/etc/resolve.conf` on the web server itself and set it so that the web server always gets a static IP address. But, we can also identify the web server by its MAC address, which is the unique hardware address of the network interface, and give it the same IP address every time.

This would require a modification of the `dnsmasq.conf` file. We need to know the MAC address of our web server Pi for this. It can be easily found by executing `ifconfig` in a new terminal and looking for the `HWaddr` field ; it will be in the form `b8:27:eb:a8:6b:cc`. Remember that this address is unique for every device. Once we have this

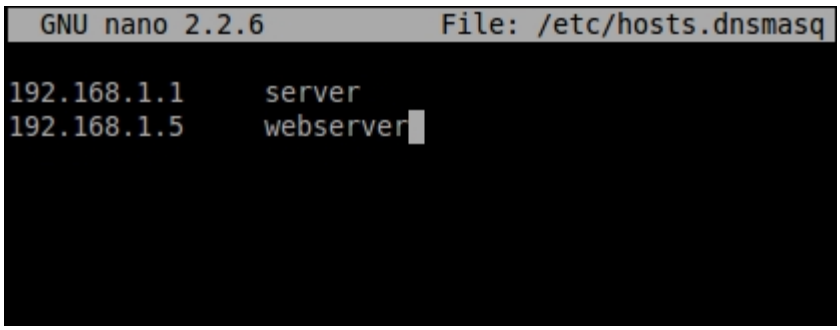
address, we will add the following line in the `dnsmasq.conf` file on the server Pi to assign a specific IP address for this MAC address:

```
dhcp-host=b8:27:eb:a8:6b:cc,192.168.1.5
```

This configures the DHCP server to always allot the `192.168.1.5` IP address for this MAC address. Now, we just need to add a name for our web server in the `hosts.dnsconf` file. Open it for editing with `nano` and just add the following line at the end of the file:

```
192.168.1.5     webserver
```

So, it finally looks like this:



Restart the `dnsmasq` service for the changes to take effect by:

```
sudo service dnsmasq restart
```

Finally, we have to make the web server Pi take up the new IP address so that it can be addressed by name in the DNS server. To do this, execute the following commands:

```
sudo ifdown eth0
sudo ifup eth0
```

This will make the Raspberry Pi disengage with the DHCP server; when it tries to re-acquire the IP address, the DHCP server will allot the static IP address we defined in the configuration file.

To test if the DNS server works with the web server or not, you can run the following command from any of the other client Pis:

```
ping webserver
```

If you get a response, it means a DNS server is set up correctly. If not, then you need to check whether all the steps were followed correctly.

Automating node discovery in a network

Currently, we have very few devices in our network. But, there will be a situation when there will be many nodes and managing them manually will be a difficult task. In a large cluster, there will be many nodes that are inactive or unserviceable. If we try to connect to them using a script that has the IP address of the clients hardcoded, we will meet with a lot of errors. This is why there is a need for automatic node detection that automatically detects the active nodes in a network and saves that information so that other programs can use it. One such program is MPICH, which we learned about in the previous chapter. As you might have learned, whenever we execute a program using MPICH, we have specified a filename that contains a list of active nodes that can be used for computations. However, if one of the nodes is inactive or unresponsive, the script will not work properly and will waste valuable computation resources.

For node discovery in a network, we will use a program called Nmap, which is short for network map. Nmap is a free security scanner, port discovery, and network exploration tool. Install it using the following command:

```
sudo apt-get install nmap
```

> **Note**
>
> Find out more about Nmap here:
>
> https://nmap.org/.

As we learned with the IP addresses in the second section, the following IP address:

`192.168.1.1`

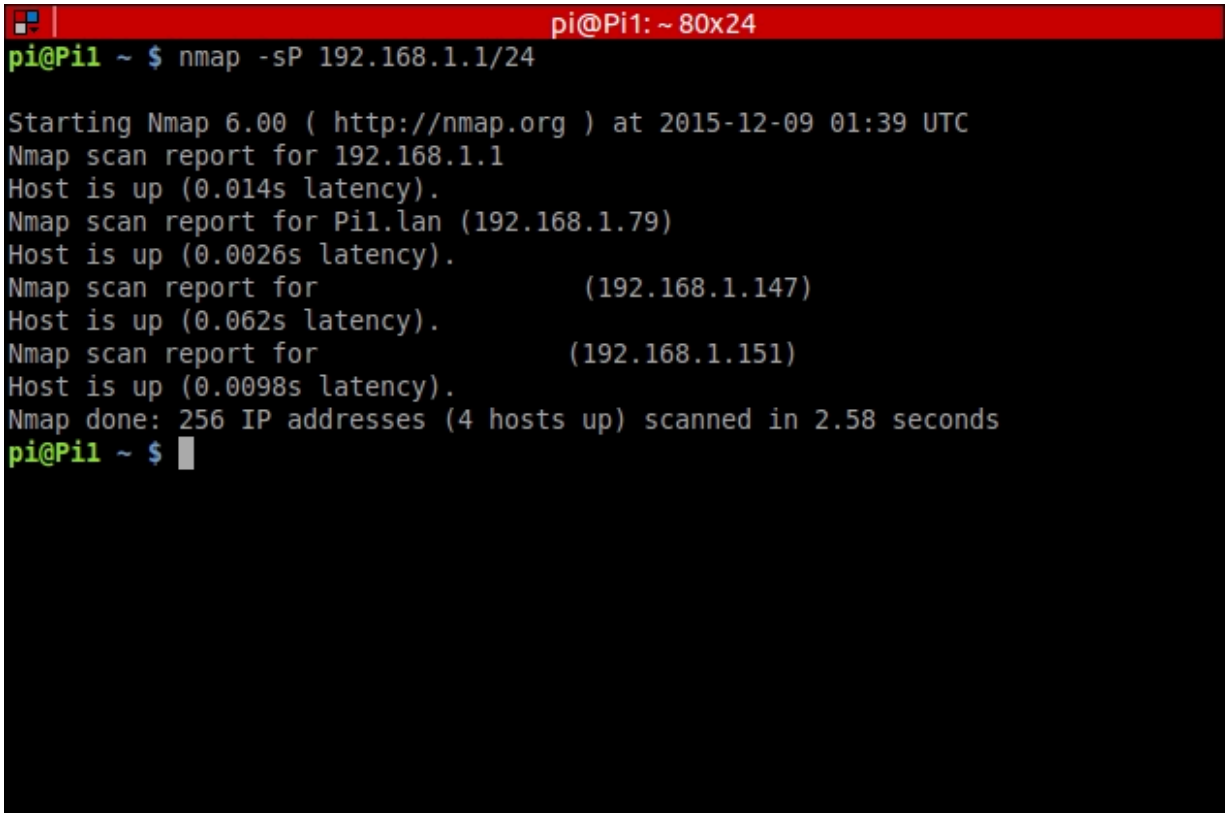
actually looks like the following:

`11000000 10101000 00000001 00000001`

Now that we have this information, we can move on to learn how to use the Nmap tool. The commands for scanning active IP addresses are:

```
nmap -sP 192.168.1.0/24
nmap -sP 192.168.1.5-100
```

This is what the output looks like:



The preceding are two separate commands and we will learn what each of them does. The first command scans the whole range of IP addresses in a network from `192.168.1.0` – `192.168.1.255`. In this command, we specify a mask of 24 bits, which basically tells Nmap

to not change the first 24 bits of the IP address, which, according to the information we learned earlier, means that the first three numbers of the IP address will remain constant and only the last number will be varied and checked for a response. Since our network will only contain IP addresses starting with `192.168.1`, due to the way our DHCP server is set up, this approach is good.

The second command explicitly specifies the range of the IP addresses to scan. This is useful as you want to find active nodes in a specific range. One use case for this is to find out all the client nodes in a network.

Summary

In this chapter, we built upon the foundations laid down in the previous chapter to gain more understanding about the concepts of networking and how they are applied to a cluster of Raspberry Pis.

Specifically, we learned how to set up a DHCP server on a Raspberry Pi and learned the mechanics of how IP addresses are allocated to different clients. We also observed the exact procedure of how the allocation takes place.

As a next step, we set up a DNS server, too, that works with the DHCP server and assigns names to the IP addresses so that the nodes are easily accessible. Using the same methodology, we learned how to assign a name for a Raspberry Pi-based web server so that we can open websites on our own network.

Finally, we learned about a tool that helps us discover active nodes on the network so that we do not have to face the inconvenience of assigning computations to a non-active node that could potentially crash a program or wrong computations.

Chapter 13. Setting Up a Web Server on the Raspberry Pi

In the previous two chapters, we discussed the networking capabilities of the Raspberry Pi. In Chapter 11, *Build Your Own Supercomputer with Raspberry Pi*, we built a network of Raspberry Pi devices that were able to communicate with each other and share the computational burden, thereby significantly reducing the time required for a task to complete. In the next chapter, Chapter 12, *Advanced Networking with Raspberry Pi*, we learned about more advanced networking concepts such as DHCP servers, DNS systems, and so on. In addition to that, we made our own intranet with a few Pi devices.

Continuing the same theme, we will now set up a web server on a single Raspberry Pi and set up a WordPress installation on it. The web server that we will be using is Apache, as it is one of the most popular servers, and has high usage within the open source community. In this chapter, we will be covering the following topics:

- Introducing and installing Apache
- Learning to serve HTML websites
- Installing PHP and MySQL
- Installing WordPress
- Configuring WordPress

Introducing and installing Apache on Raspbian

The Apache HTTP server is extremely popular and is the world's most used web server software currently. It enjoys the support of the open source community because it is developed and maintained by an open community of developers under the umbrella of the Apache

Software Foundation. Although most commonly used on a Unix-like system, it is available for a variety of other operating systems including Microsoft Windows. In 2009, it became the first web server to be used by more than a hundred million websites and currently it is estimated to serve over 50 percent of all websites and 37 percent of the top servers across domains.

One of the reasons it is so popular is that it supports a wide variety of features and many more features can be implemented as compiled modules that can extend the core functionality. For example, on its own, Apache can serve HTML files over HTTP, but with additional modules installed, it can also serve web pages using languages such as Python. It also has a variety of language interfaces that include Python, PHP, and Perl.

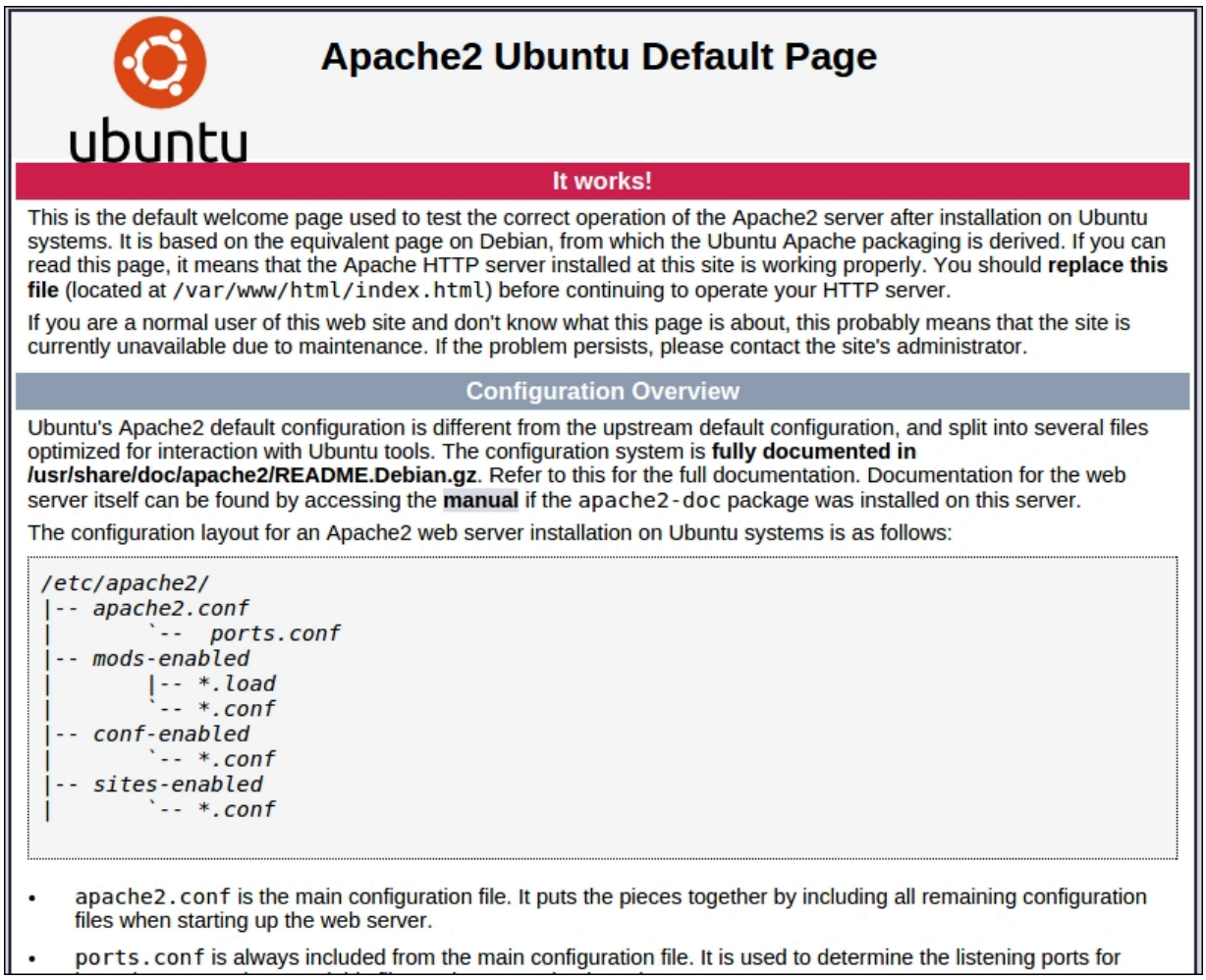
Now, installing the web server on a Raspbian distribution is extremely easy. We need to install the `apache2` package with the following command:

```
$ sudo apt-get install apache2
```

During installation, the package automatically creates our demo HTML file that you can see if you type `127.0.0.1` or `http://localhost/` on your URL bar. We can also see this if we navigate to the Pi's IP address from another machine connected to the same network such as `192.168.1.5`. Replace this IP address with your own Pi's IP address in the network. You can find the IP address by typing the following on the Raspberry Pi's terminal:

```
$ hostname -I
```

The default page looks like this:



If we see this page, it means that our installation has been successful. This is just a single HTML file that can be found at `/var/www/html/index.html`. Let's look at the details of this folder. Navigate to the folder by:

```
$ cd /var/www/html/
```

And then execute the following command to see the properties of the files in the folder:

```
$ ls -al
```

This command lists the files and a few of their properties. You will see the following output:

```
total 20
drwxr-xr-x 2 root root  4096 Dec 11 12:13 .
drwxr-xr-x 4 root root  4096 Dec 11 12:21 ..
-rw-r--r-- 1 root root 11510 Dec 11 12:13 index.html
```

The meaning of the columns respectively is:

- Permissions of the file
- Number of files in the directory
- User that owns the file/directory
- Group that owns the file/directory
- File size
- Last modified date

Since the owner of the `index.html` file is root, we need to take ownership of the file before editing it. To do that, enter the following command:

```
$ sudo chown pi: index.html
```

We can also replace the `index.html` file with our own HTML file and serve a fully fledged HTML website complete with CSS classes.

However, a simple HTML website sounds too basic, right? Since we are on the final leg of this book, we should do something more advanced. That's right, in the following sections we will be learning how to set up a WordPress website. For that, however, we first need to install a few tools that WordPress uses.

Installing PHP and MySQL

PHP is one of the most popular programming languages in use today for the web look at. It is the code that runs on the server when it receives a request for a website. Unlike HTML, PHP can generate dynamic web pages that can change the look and content on the fly. We chose PHP to work with because WordPress uses PHP. Now, install PHP and the accompanying Apache packages with the following command:

```
$ sudo apt-get install php5 libapache2-mod-php5
```

We now have PHP installed with its accompanying Apache packages, and we will proceed to test it.

Navigate to the HTML folder and create a new `index.php` file:

```
$ cd /var/www/html/
$ sudo nano index.php
```

Add a test line to it:

```
<?php echo "hello world"; ?>
```

Save the file and exit by pressing *Ctrl* + *X* and then *Y*. We also need to remove the `index.html` file because HTML takes precedence over PHP. Do that with the following command:

```
sudo rm index.html
```

Navigate to `127.0.0.1` or `localhost` in the browser and you will be greeted by the hello world message. Let's now set up something static.

Replace the hello world line with the following:

```
<?php echo date('Y-m-d H:i:s'); ?>
```

Now that we have installed PHP to process the web pages that we want, the next step is to install a database to store some data. Our database of choice will be MySQL simply because WordPress uses it to store IDs of things such as Post, Pages, Comments, Users, and so on.

To install MySQL and its PHP packages, run the following command:

```
sudo apt-get install mysql-server php5-mysql
```

Once the installation proceeds, you will be asked for a root password for creating the databases. Enter the password and remember it for future use.

Installing WordPress

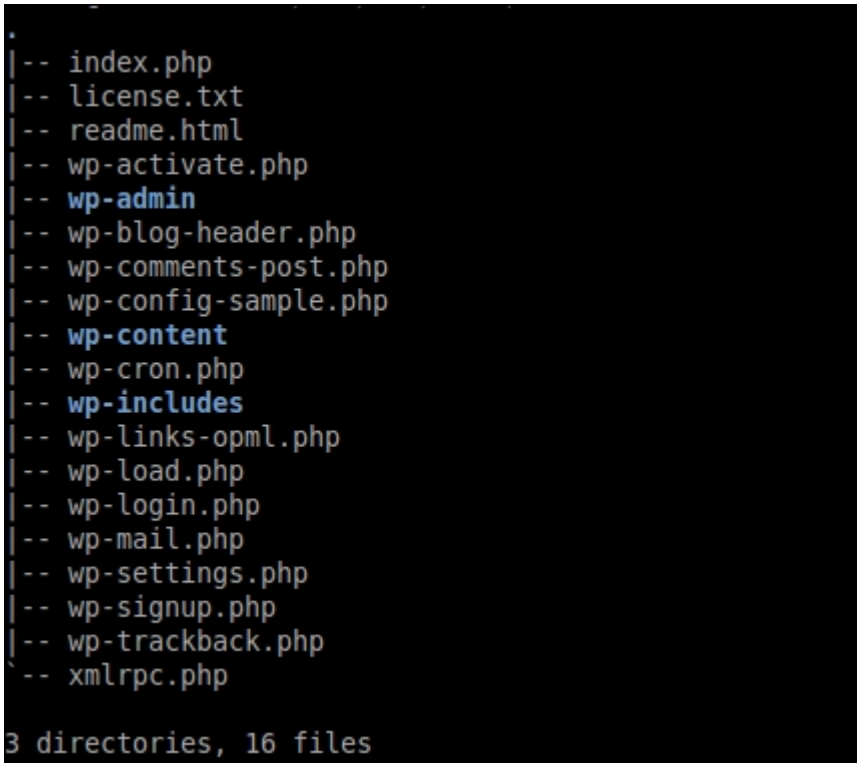
Now that we are done installing the dependencies, the next step is getting the WordPress files in the HTML folder. Fortunately for us, WordPress provides a tar archive of the latest version of WordPress on this link: http://wordpress.org/latest.tar.gz. We now need to place the files contained in this folder in our web server folder. To do that, we first need to take ownership of the folder. To carry out that entire process, we execute the following commands:

```
$ cd /var/www/html/
$ sudo chown pi:
$ sudo rm *
$ wget http://wordpress.org/latest.tar.gz
```

In a nutshell, we first navigate to the HTML folder, take ownership of the folder, remove all the pre-existing files, and then fetch the WordPress archive. Next, we extract the contents of the archive and place them in the HTML folder. We do that with the following commands:

```
$ tar xzf latest.tar.gz
$ mv wordpress/*
$ rm -rf wordpress latest.tar.gz
```

Now, if you run the `ls` command, you will see the following output:

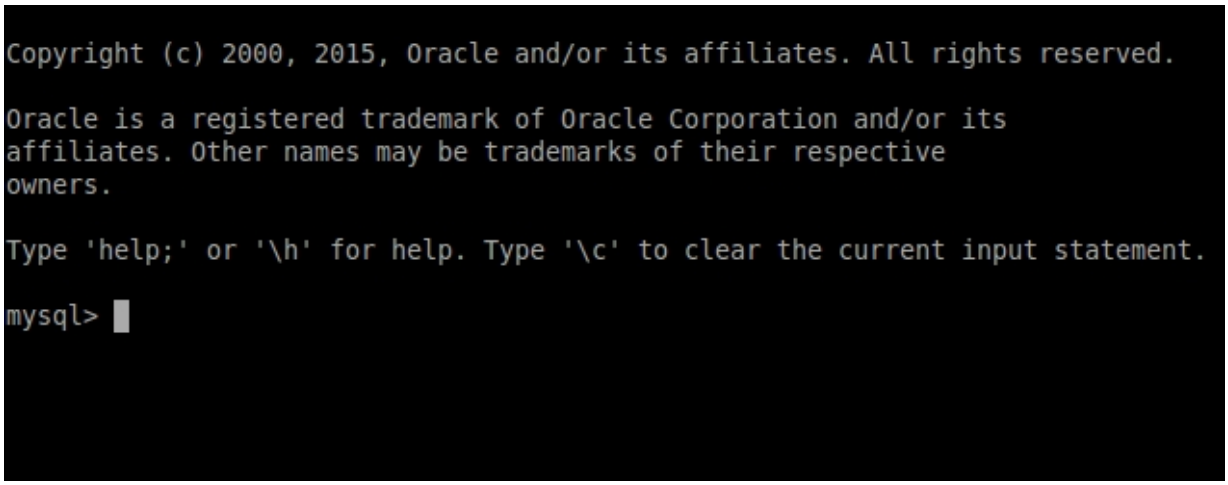


Configuring the WordPress installation

Once the files are properly placed in the right locations, to activate the installation and start using the WordPress website, we first need to set up a MySQL database that WordPress can use. To set up the database, execute the following command:

```
$ mysql -uroot -p
```

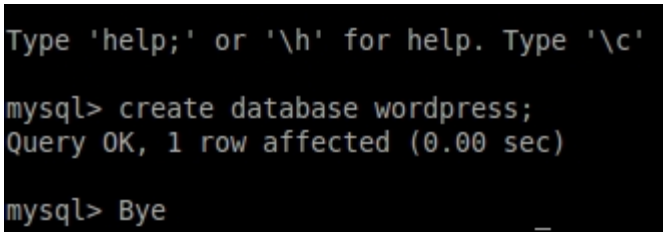
This sets up the database so that the user is root and the `-p` option prompts the terminal to ask for the password. Enter the same password that you entered while installing the MySQL database. We are then greeted with a MySQL terminal that looks like this:



Once inside the terminal, execute the following statement:

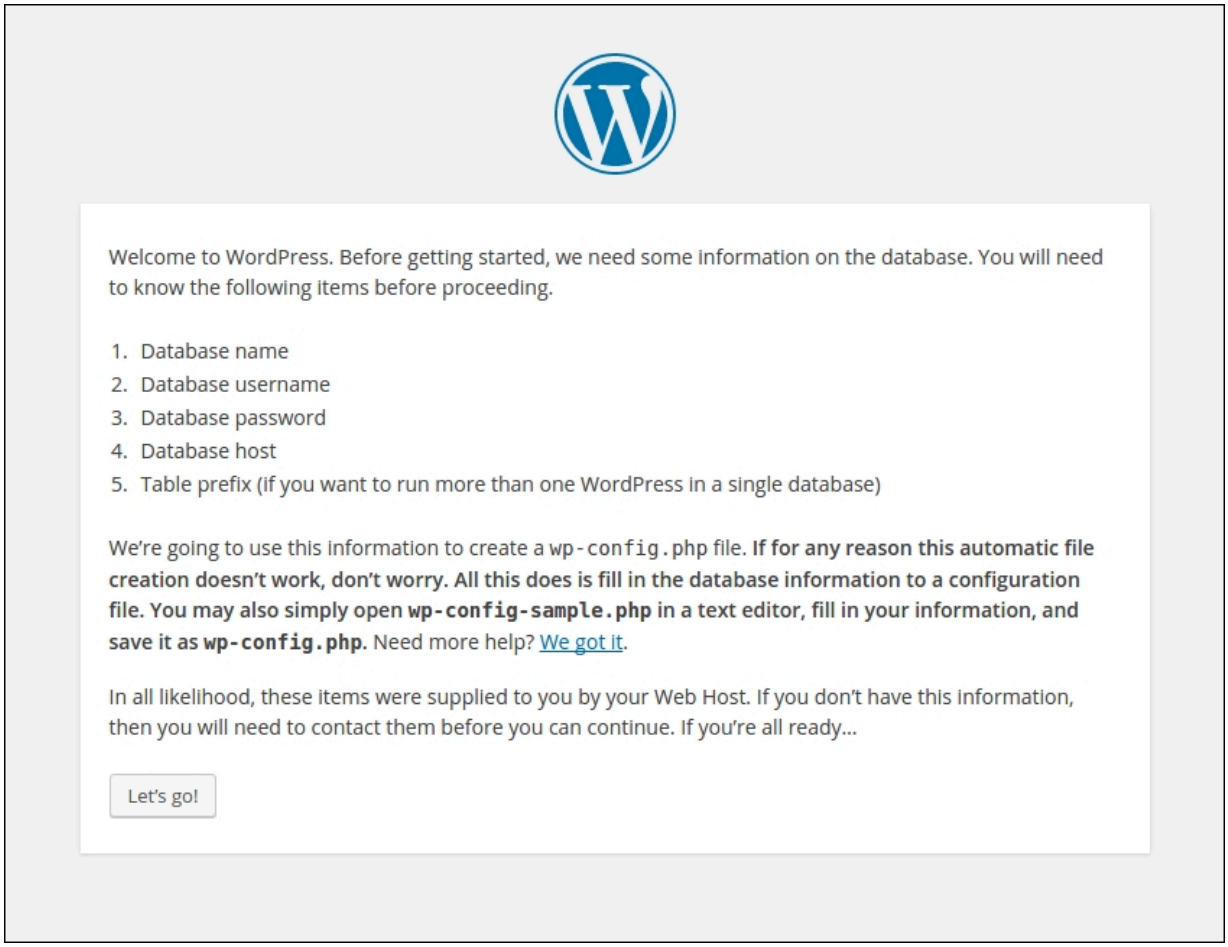
```
mysql> create database wordpress;
```

You will be greeted by the following message:

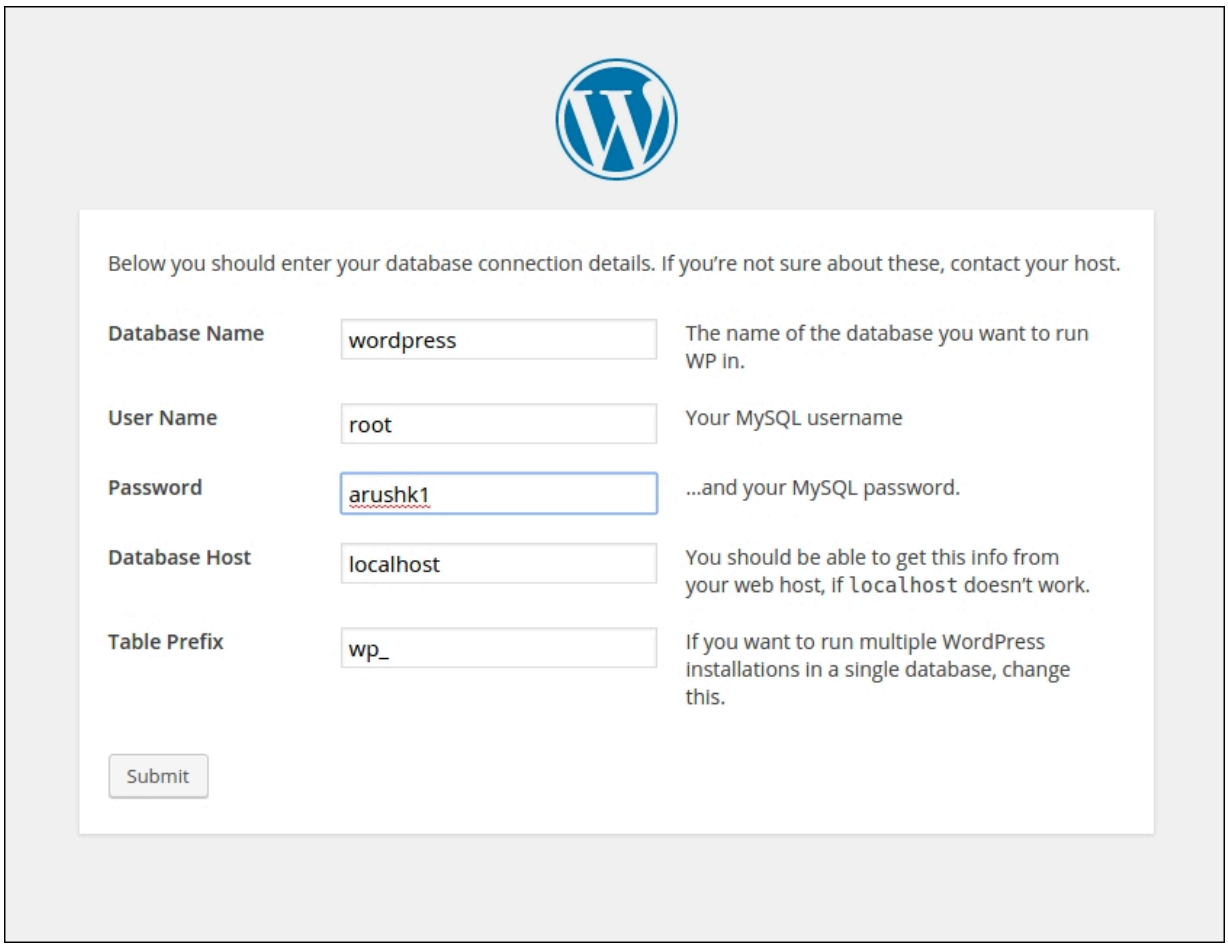


Finally, exit the terminal by pressing *Ctrl* + *D*.

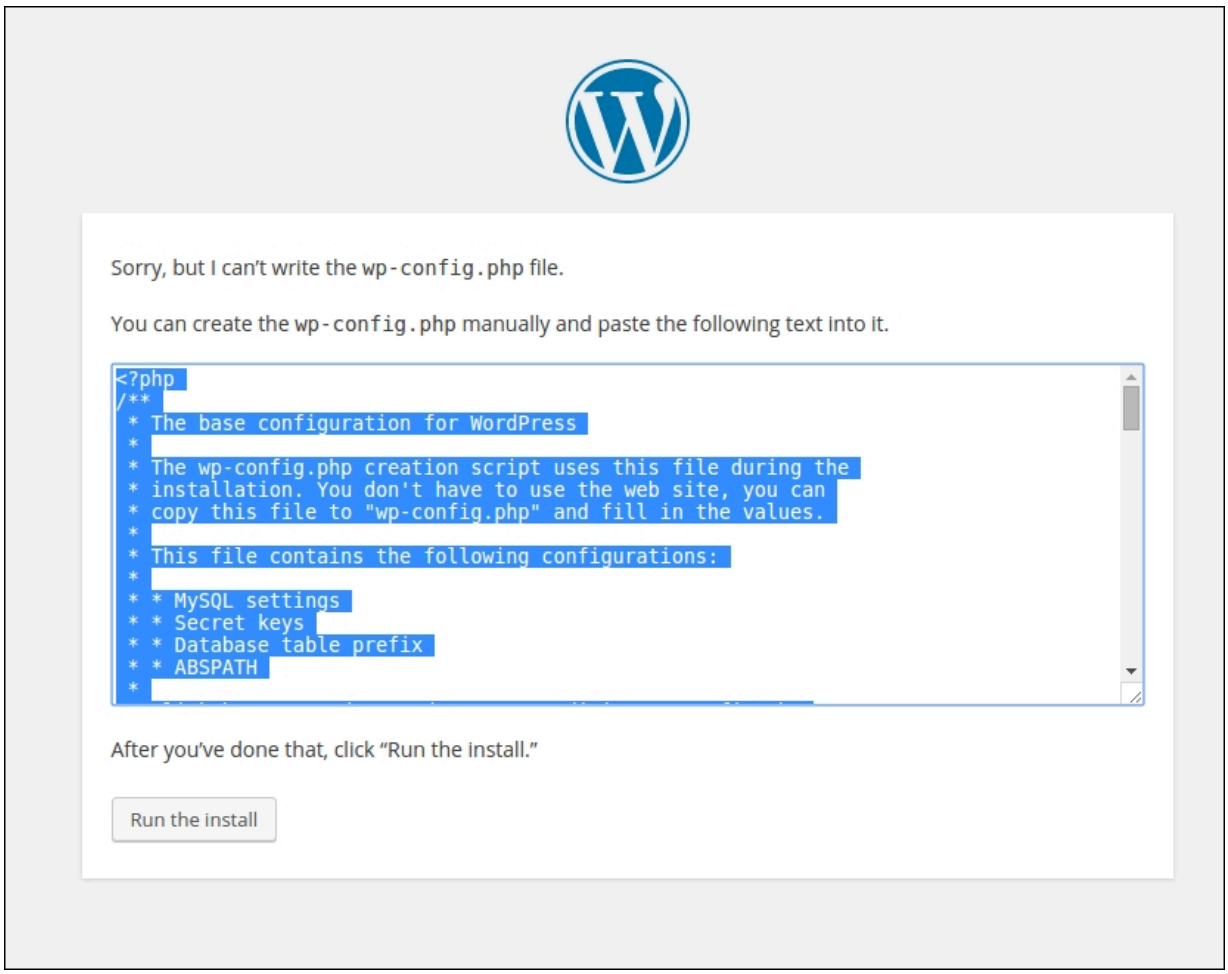
To complete the installation, we need to access the WordPress installation page, located on our local host. So go ahead and enter `127.0.0.1` in your browser and you will be greeted with the following page:



This means that WordPress is properly configured. We now hit **Let's go!** to proceed to the next step. This takes us to the following page:



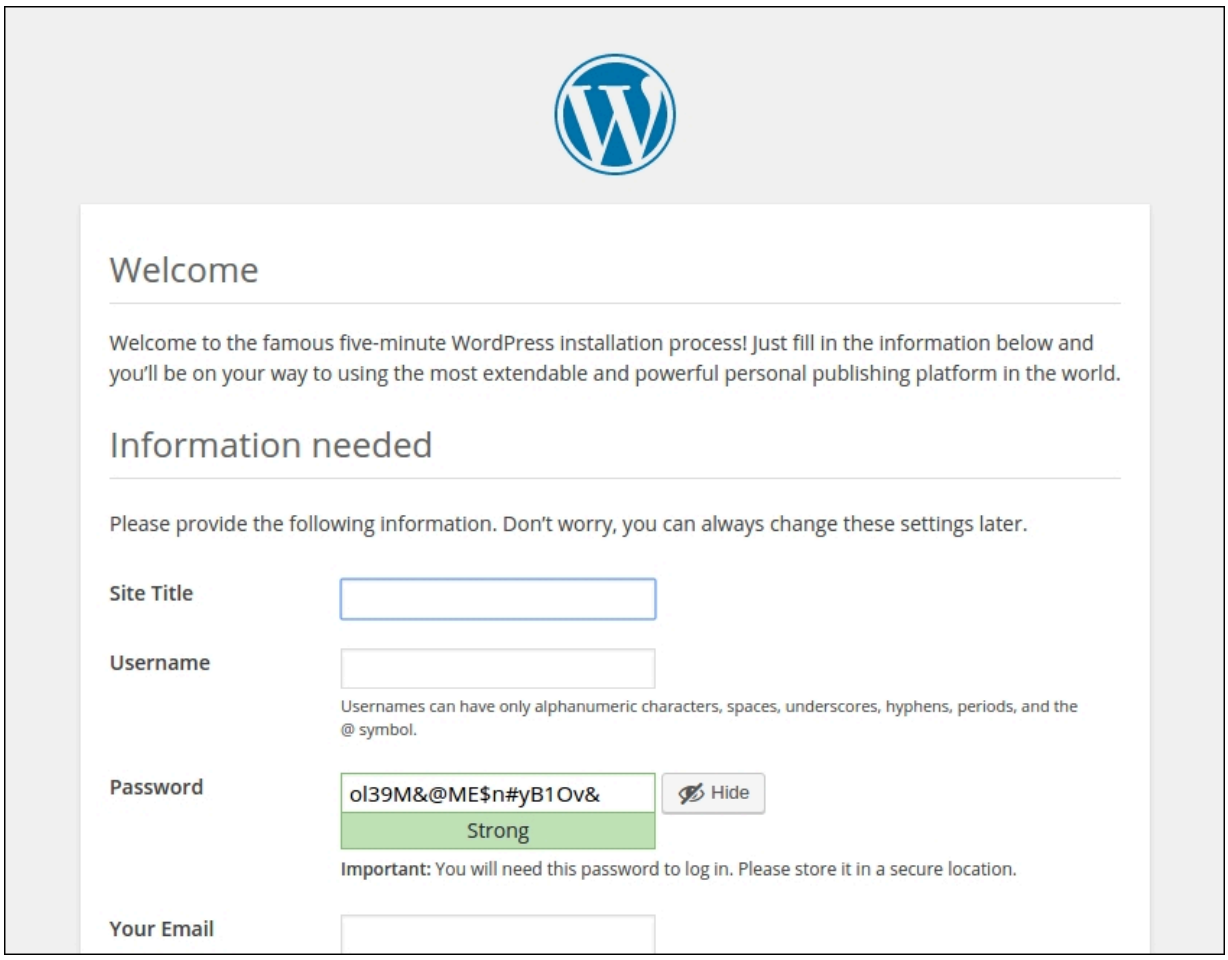
Enter all the details for the database including the password that you set earlier. Enter the details on the page as given in the preceding screenshot, and then press **Submit**. The installation wizard will give an error message that it was unable to write the `wp-config.php` file to the filesystem. There is no reason to panic here, because it also gives us the content of the `wp-config.php` file that we can use to create the file manually. The error looks like this:



So, now copy the text given in the dialog box, and open a new terminal. Create a new `wp-config.php` file in the HTML folder with the following commands:

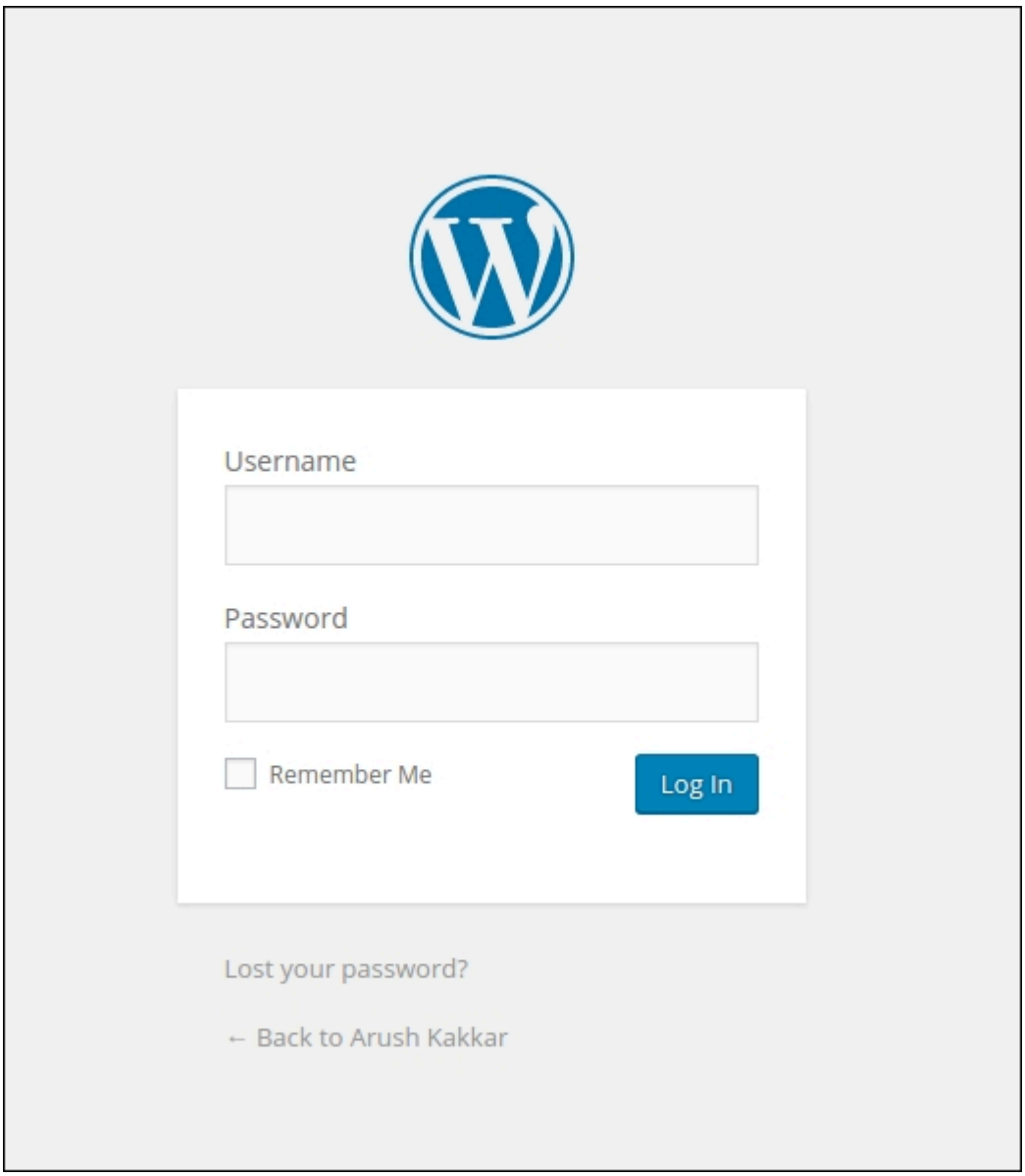
```
$ cd /var/www/html
$ sudo nano wp-config.php
```

Go ahead and paste the content that was copied. We can right-click to paste or simply press *Ctrl* + *Shift* + *V*. Save and exit the file by pressing *Ctrl* + *X* and then *Y*. Once the file is created, go back to the browser and hit the **Run the Install** button. After processing for a few seconds, the page returns the following screen:



Now, we are ready to enter our details to create a user and begin using the WordPress website. Enter the details, and you've successfully created it.

You can log in to your website by appending `/wp-admin.php` to your website URL. For example, in our case the admin page will be given by `127.0.0.1/wp-admin.php`, and it looks like this:



Note

To find out more about WordPress, log on to their website:

https://wordpress.org/

It would be an interesting exercise for you to continue with the process and try to run a fully fledged website on the Raspberry Pi

and see how it handles the loads of running a heavy website complete with custom theme, pictures, and so on.

Summary

This chapter was a general extension of the previous two chapters, expanding on the concepts of networking built previously and using them to set up a web server on our intranet.

In this chapter, we installed the Apache web server on our Pi. To set up the WordPress website, we installed all the dependencies including PHP and MySQL. Then we downloaded the WordPress files and set up a MySQL database to work with it.

Finally, we configured the installation and resolved the errors that we generated to arrive at a complete installation.

In next chapter, we will learn network programming with Raspberry Pi using sockets in Python.

Chapter 14. Network Programming in Python with the Pi

A network socket is an endpoint of a connection across computer networks. Nowadays, almost all communication between computers and distinct networks is based on the Internet Protocol, which uses sockets as a basis of communication. A socket is an abstract reference that a local program can pass to the network API to make use of a connection. Sockets are internally often represented in network programming APIs simply as integers, which identify which connection to use. In brief, we will be covering the following topics in this chapter:

- The basics of sockets
- The difference between TCP and UDP sockets
- UDP sockets
- TCP sockets
- The Telnet program
- A chat application

The basics of sockets

A socket **API** is an **application programming interface**, provided by the **operating system** (**OS**), that allows application programs to initiate, control, and use network sockets programmatically for communication. Internet socket APIs are usually based on the Berkeley sockets standard. In the Berkeley sockets standard, sockets are a form of file descriptor, adhering to the Unix philosophy that *everything is a file*. Thus, we can read, write, open, and close sockets in the same way as we do files. In inter-process communications, each end will have its own socket, but these may

use different socket-programming APIs. However, they are abstracted by the network protocol.

A socket address is a combination of an **Internet Protocol** (**IP**) address and a port number. Internet sockets deliver and receive data packets to and from the appropriate application process or thread. On a computer with an IP address, every network-related program or utility will have its own unique socket or set of sockets. This ensures that the incoming data is redirected to the correct application.

The difference between TCP and UDP

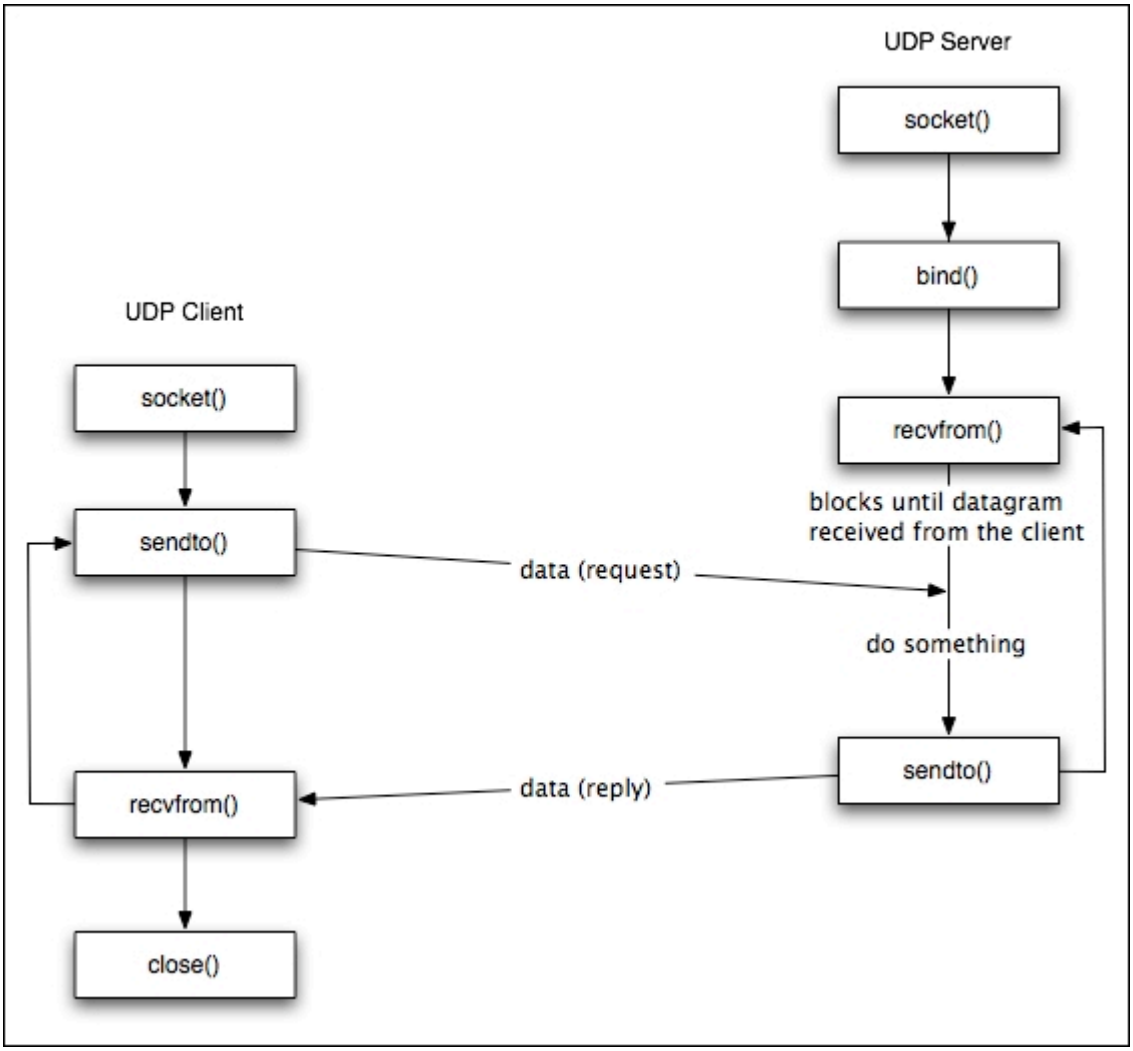
There are two types of protocols in the IP suite. They are **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**). TCP is a connection-oriented IP which means that once a connection is established, data can be sent in a bidirectional manner. UDP is a much simpler, connectionless Internet protocol. Multiple messages are sent as packets in chunks using UDP. Let's distinguish between the two with clear points, as follows:

TCP	**UDP**
TCP is a connection-oriented protocol.	UDP is a connectionless protocol.
Using this mode, a message makes its way across the Internet from one computer and network to another. This is connection based.	UDP is also a protocol used in message transport or transfer. It is not a connection-based protocol. A program using UDP can send a lot of packets to another, and that would be the end of the relationship.
TCP is suited to applications that require high reliability, and transmission time is relatively less critical.	UDP is suitable for applications that need fast, efficient transmission, such as games and live-streaming videos.

TCP	UDP
Protocols utilizing TCP include HTTP, HTTPS, FTP, SMTP, and Telnet.	Protocols utilizing UDP include DNS, DHCP, TFTP, SNMP, RIP, and VoIP.
Data is read as a stream of bytes; no distinguishing indications are transmitted to signal message segment boundaries.	Packets are sent individually and are checked for integrity only on arrival. Packets have headers and endpoints.
TCP performs error checking.	UDP performs error checking. However, it has no recovery options for missed or corrupt packets.
The data transferred remains intact and arrives in the same order (known as in-order) in which it was sent.	There is no guarantee that the messages or packets sent will reach at all or will be in the same order as transmitted.

The architecture and programming of UDP sockets

Here is the architecture of a UDP client-server system:



UDP is an Internet protocol. Just like its counterpart TCP (which we will discuss soon), UDP is a protocol for the transfer of packets from one host to another. However, as seen in the diagram, it has some important differences from TCP. Unlike TCP, UDP is connectionless and is not a stream-oriented protocol. This means a UDP server just catches incoming packets from any and many hosts without establishing a reliable and dedicated connection for the transfer of data between processes.

A UDP socket is created as follows in Python:

```
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
```

`SOCK_DGRAM` specifies a UDP socket.

Sending and receiving data with UDP

As UDP sockets are connectionless sockets, communication is done with the socket functions `sendto()` and `recvfrom()`. These functions do not require a socket to be connected to another peer explicitly. They just send and receive directly to and from a given IP address.

UDP servers and NCAT

The simplest form of a UDP server in Python is as follows:

```
import socket
port = 5000
s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
s.bind(("", port))
print "waiting on port:", port
while 1:
    data, addr = s.recvfrom(1024)
    print data
```

Instead of having a `listen()` function, a UDP server has to open a socket and wait to receive incoming packets. As is evident from the code snippet, there is no `listen()` or `accept()` function. Save the above code and execute it from a terminal and then connect to it using the **NCAT** utility. NCAT is an alternative to Telnet, and it is more powerful than Telnet and packed with more features. Run the program and, in another terminal window, use NCAT. Here is the output of the execution of the program and NCAT:

```
pi@raspberrypi ~/book/chapter14 $ nc localhost
5000 -u -v
Connection to localhost 5000 port [udp/*]
succeeded!
```

```
Hello
Ok
```

The `-u` flag in the command indicates the UDP protocol. The message we send should be displayed on the server terminal.

An echo server using Python UDP sockets

Here is the code for an echo server in Python:

```
import socket
import sys

HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 8888 # Arbitrary non-privileged port

# Datagram (udp) socket
try :
    s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
    print 'Socket created'
except socket.error, msg :
    print 'Failed to create socket. Error Code : '
+ str(msg[0]) + ' Message ' + msg[1]
    sys.exit()


# Bind socket to local host and port
try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'

#now keep talking with the client
while 1:
    # receive data from client (data, addr)
    d = s.recvfrom(1024)
    data = d[0]
    addr = d[1]
```

```
        if not data:
            break

        reply = 'OK...' + data

        s.sendto(reply , addr)
        print 'Message[' + addr[0] + ':' +
    str(addr[1]) + '] - ' + data.strip()

    s.close()
```

This program will start a UDP server process on the mentioned port (in our case, it's `8888`). Save the program and run it in a terminal. To test the program, open another terminal and use the NCAT utility to connect to this server, as follows:

```
pi@raspberrypi ~/book/chapter14 $ nc -vv localhost
8888 -u
Connection to localhost 8888 port [udp/*]
succeeded!
OK...XOK...XOK...XOK...XOK...X
OK...
Hello
OK...Hello
How are you?
OK...How are you?
```

Use NCAT again to send messages to the UDP server, and the UDP server will reply back with `OK...` prefixed to the message.

The server process terminal also displays the details about the client connected, as follows:

```
$ python prog2.py
Socket created
Socket bind complete
Message[127.0.0.1:46622] - Hello
Message[127.0.0.1:46622] - How are you?
```

It is important to note that unlike a TCP server, a UDP server can handle multiple clients directly as there is no explicit connection with a client (hence connectionless). It can receive from any client and send a reply to it. No threads are required as we do in TCP servers.

A UDP client

The code for a UDP client is as follows:

```
import socket   #for sockets
import sys  #for exit

# create dgram udp socket
try:
    s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
except socket.error:
    print 'Failed to create socket'
    sys.exit()

host = 'localhost';
port = 8888;

while(1) :
    msg = raw_input('Enter message to send : ')

    try :
        #Set the whole string
        s.sendto(msg, (host, port))

        # receive data from client (data, addr)
        d = s.recvfrom(1024)
        reply = d[0]
        addr = d[1]

        print 'Server reply : ' + reply

    except socket.error, msg:
        print 'Error Code : ' + str(msg[0]) + '
Message ' + msg[1]
        sys.exit()
```

The client will connect to the UDP server and exchange messages as follows:

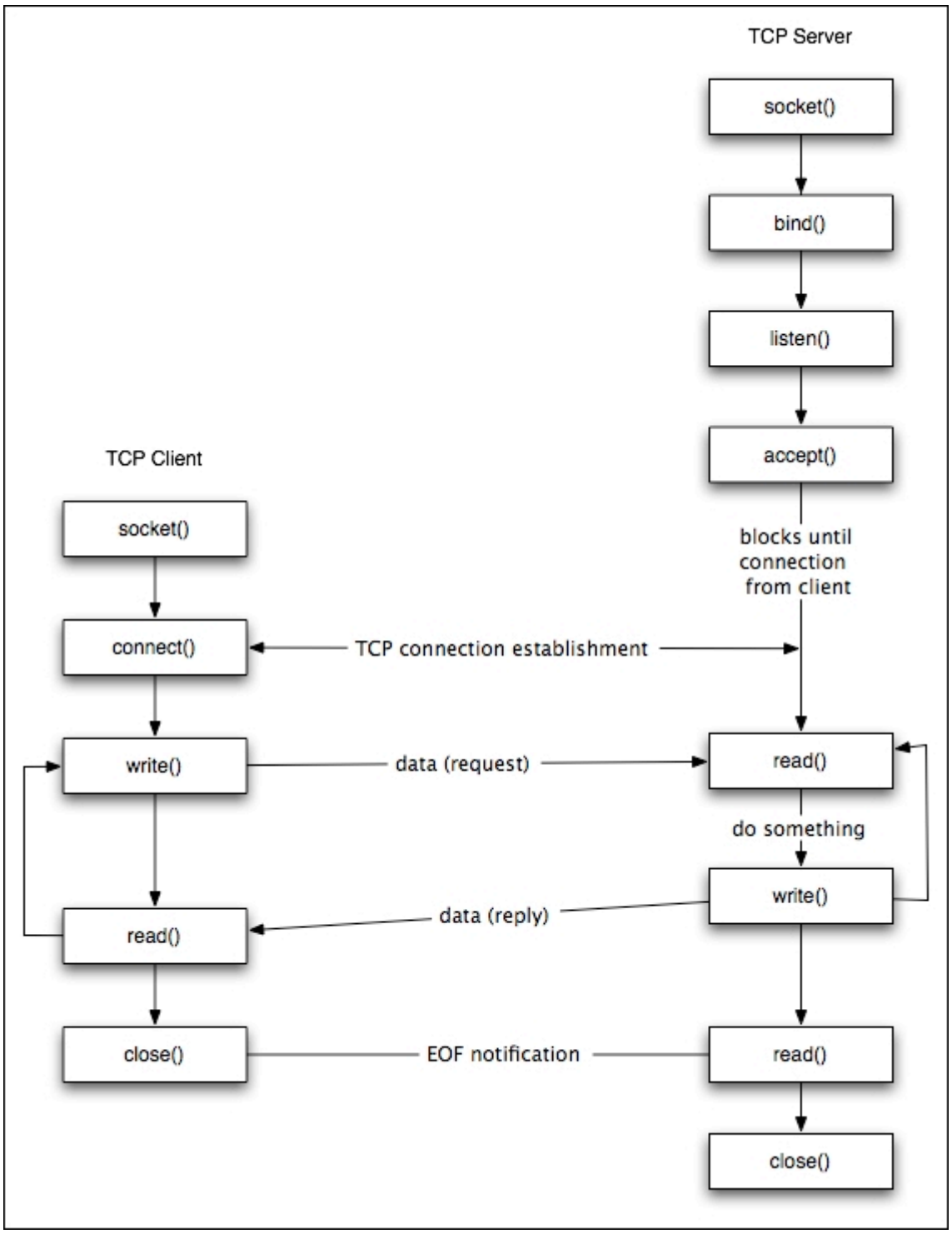
```
pi@raspberrypi ~/book/chapter14 $ python prog3.py
Enter message to send : Hello
Server reply : OK...Hello
Enter message to send : How are you
Server reply : OK...How are you
Enter message to send : Ok
Server reply : OK...Ok
Enter message to send :
```

The programs for the UDP protocol are simple to code as there are no explicit connections from the UDP clients to the UDP server.

In the next subsection, we will learn about TCP sockets.

The architecture of TCP sockets

The following is a diagram of the TCP client-server architecture:



Creating a TCP socket

Here is an example of creating a TCP socket:

```
import socket-- #for sockets
-
create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
-
print 'Socket Created'
```

The `socket.socket()`function creates a socket and returns a socket descriptor, which can be used in other socket-related functions.

This code will create a socket with the following properties:

- Address family: `AF_INET` (this is for IP version 4, or IPv4)
- Type: `SOCK_STREAM` (this specifies a connection-oriented protocol, that is, TCP)

If any of the socket functions fail, then Python throws an exception called `socket.error`, which must be caught as follows:

```
import socket-- #for sockets
import sys- for exit
-
try:
----create an AF_INET, STREAM socket (TCP)
----s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
except socket.error, msg:
----print 'Failed to create socket. Error code: '
+ str(msg[0]) + ' , Error message : ' + msg[1]
----sys.exit();
-
print 'Socket Created'
```

In this way, we have created a TCP socket successfully. We can connect to a server, for example, http://www.google.com, using this socket.

Connecting to a server with a TCP socket

We can connect to a remote server on a certain port number. We need two things for this: the IP address of the remote server we are connecting to and a port number to connect to. We will use the IP address of https://www.google.com as a sample in the following code.

First, we need to get the IP address of the remote host or URL, since before connecting to a remote host, its IP address is required. In Python, obtaining the IP address is quite simple:

```
import socket   #for sockets
import sys  #for exit

try:
    #create an AF_INET, STREAM socket (TCP)
    s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
except socket.error, msg:
    print 'Failed to create socket. Error code: '
+ str(msg[0]) + ' , Error message : ' + msg[1]
    sys.exit();

print 'Socket Created'

host = 'www.google.com'

try:
    remote_ip = socket.gethostbyname( host )

except socket.gaierror:
    #could not resolve
    print 'Hostname could not be resolved.
Exiting'
    sys.exit()

print 'Ip address of ' + host + ' is ' + remote_ip
```

Now that we have the IP address of the remote host or URL, we can connect to it on a certain port using the `connect()` function:

```
import socket-- #for sockets
import sys- #for exit
-
```

```
try:
----create an AF_INET, STREAM socket (TCP)
----s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
except socket.error, msg:
----print 'Failed to create socket. Error code: '
+ str(msg[0]) + ' , Error message : ' + msg[1]
----sys.exit();
-
print 'Socket Created'
-
host = 'www.google.com'
port = 80
-
try:
----remote_ip = socket.gethostbyname( host )
-
except socket.gaierror:
----could not resolve
----print 'Hostname could not be resolved.
Exiting'
----sys.exit()
-----
print 'Ip address of ' + host + ' is ' + remote_ip
-
Connect to remote server
s.connect((remote_ip , port))
-
print 'Socket Connected to ' + host + ' on ip ' +
remote_ip
```

Run the program and notice that its output in the terminal is as follows:

```
pi@raspberrypi ~/book/chapter14 $ python prog4.py
Socket Created
Ip address of www.google.com is 74.125.236.83
Socket Connected to www.google.com on ip
74.125.236.83
```

It creates a TCP socket and then connects to a remote host. If we try connecting to a port different from port `80`, then we should not be

able to connect, which indicates that the port is not open for any connections. This logic can be used to build a port scanner.

Note

The concept of *connections* applies to `SOCK_STREAM/TCP` type of sockets. A connection means an explicit or reliable stream or pipeline of data such that there can be multiple such streams and each can have communication of its own. Think of this as a pipe that is not interfered with by data from other pipes. Another important property of stream connections is that the packets have an order or sequence; hence, they are always sent, arrive, and are processed in order.

Other sockets, such as UDP, ICMP, and ARP, don't have the concept of an explicit *connection* or pipeline. These are connectionless communications, as we have seen with an example in the case of UDP. This means that we keep sending or receiving packets from anybody and everybody.

The `sendall()`function will send all the data. Let's send some data to https://www.google.com. The code for it is as follows:

```
import socket   #for sockets
import sys  #for exit

try:
    #create an AF_INET, STREAM socket (TCP)
    s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
except socket.error, msg:
    print 'Failed to create socket. Error code: '
+ str(msg[0]) + ' , Error message : ' + msg[1]
    sys.exit();

print 'Socket Created'

host = 'www.google.com'
port = 80
```

```
try:
    remote_ip = socket.gethostbyname( host )

except socket.gaierror:
    #could not resolve
    print 'Hostname could not be resolved.
Exiting'
    sys.exit()

print 'Ip address of ' + host + ' is ' + remote_ip

#Connect to remote server
s.connect((remote_ip , port))

print 'Socket Connected to ' + host + ' on ip ' +
remote_ip

#Send some data to remote server
message = "GET / HTTP/1.1\r\n\r\n"

try :
    #Set the whole string
    s.sendall(message)
except socket.error:
    #Send failed
    print 'Send failed'
    sys.exit()

print 'Message send successfully'
```

In this example, we first connect to an IP address and then send the string message `GET / HTTP/1.1\r\n\r\n` to it. The message is actually an HTTP command to fetch the main page of the website.

Now that we have sent some data, it's time to receive a reply from the server. So let's do it.

Receiving data from the server

The `recv()`function is used to receive data on a socket. In the following example, we will send a message and receive a reply from the server with Python:

```
import socket   #for sockets
import sys  #for exit

#create an INET, STREAMing socket
try:
    s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
except socket.error:
    print 'Failed to create socket'
    sys.exit()

print 'Socket Created'

host = 'www.google.com';
port = 80;

try:
    remote_ip = socket.gethostbyname( host )

except socket.gaierror:
    #could not resolve
    print 'Hostname could not be resolved.
Exiting'
    sys.exit()

#Connect to remote server
s.connect((remote_ip , port))

print 'Socket Connected to ' + host + ' on ip ' +
remote_ip

#Send some data to remote server
message = "GET / HTTP/1.1\r\n\r\n"

try :
    #Set the whole string
    s.sendall(message)
except socket.error:
    #Send failed
    print 'Send failed'
    sys.exit()

print 'Message send successfully'

#Now receive data
reply = s.recv(4096)
```

```
print reply
```

Your web browser also does the same thing when you use it to open www.google.com.

This socket activity represents a client socket. A client is a system that connects to a remote host to fetch the required data.

The other type of socket activity is called a server system. A server is a system that uses sockets to receive incoming connections and provide them with the required data. It is just the opposite of the client system. So, https://www.google.com is a server system and a web browser is a client system. To be more specific, https://www.google.com is an HTTP server and a web browser is an HTTP client.

Programming socket servers

Now, we move on to learning how to program socket servers. Servers basically perform the following sequence of tasks:

1. Open a socket
2. Bind to an address (and port)
3. Listen for incoming client connection requests
4. Accept connections
5. Receive/send data from/to clients

We already know to open a socket. So, the next thing to learn will be how to bind it.

Binding a socket

The `bind()`function can be used to bind a socket to a particular IP address and port. It needs a `sockaddr_in` structure similar to the `connect()` function:

```
import socket
import sys

HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 8888 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
print 'Socket created'

try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'
```

Now that the binding is done, it's time to make the socket listen to incoming connection requests. We bind a socket to a particular IP address and a certain port number. By doing this, we ensure that all incoming data packets directed towards this port number are received by the server application.

Also, there cannot be more than one socket bound to the same port.

Listening for incoming connections

After binding a socket to a particular port, the next thing we need to do is listen for incoming connections. For this, we need to switch the socket to listening mode. The `socket_listen()`function is used to put the socket in listening mode. To accomplish this, we need to add the following line after the bind code:

```
s.listen(10)
print 'Socket now listening'
```

The parameter of the `listen()`function is called the **backlog**. It is used to control the number of incoming connections that are kept

waiting if the program using that port is already busy. So, by mentioning `10`, we mean that if 10 connections are already waiting to be processed, then the eleventh new connection request shall be rejected. This will be clearer after checking `socket_accept()`.

Here is the code to do that:

```
import socket
import sys

HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 8888 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
print 'Socket created'

try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'

s.listen(10)
print 'Socket now listening'

#wait to accept a connection - blocking call
conn, addr = s.accept()

#display client information
print 'Connected with ' + addr[0] + ':' +
str(addr[1])
```

Run the program. It should show you the following:

```
pi@raspberrypi ~/book/chapter14 $ python prog7.py
Socket created
Socket bind complete
Socket now listening
```

So now, this program is waiting for incoming client connections on port `8888`. Don't close this program; ensure that you keep it running.

Now, a client can connect to the server on this port. We will use the Telnet client for testing this. Open a terminal and type this:

```
$ telnet localhost 8888
```

It will immediately show the following:

```
pi@raspberrypi ~/book/chapter14 $ telnet localhost
8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
pi@raspberrypi ~/book/chapter14 $
```

And this is what the server will show:

```
pi@raspberrypi ~/book/chapter14 $ python prog7.py
Socket created
Socket bind complete
Socket now listening
Connected with 127.0.0.1:59954
```

So now, we can see that the client is connected to the server. We accepted an incoming connection but closed it immediately. There are lots of tasks that can be accomplished after an incoming connection is established. The connection was established between two hosts for the purpose of communication. So let's reply to the client.

The `sendall()`function can be used to send something to the socket of the incoming connection, and the client should be able to receive it. Here is the code for that:

```
import socket
import sys
```

```
HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 8888 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
print 'Socket created'

try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'

s.listen(10)
print 'Socket now listening'

#wait to accept a connection - blocking call
conn, addr = s.accept()

print 'Connected with ' + addr[0] + ':' +
str(addr[1])

#now keep talking with the client
data = conn.recv(1024)
conn.sendall(data)

conn.close()
s.close()
```

Run this code in a terminal window and connect to this server using Telnet from another terminal; you should be able to see this:

```
pi@raspberrypi ~/book/chapter14 $ telnet localhost
8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
happy
```

```
happy
Connection closed by foreign host.
```

So, the client (Telnet) received a reply from the server.

We can see that the connection is closed immediately after this, simply because the server program terminates immediately after accepting the request and responding with the reply. A server process is supposed to be running all the time; the simplest way to accomplish this is to iterate the `accept` method in a loop so that it can receive incoming connections all the time.

So, a live server will always be up and running. The code for this is as follows:

```
import socket
import sys

HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 5000 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
print 'Socket created'

try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'

s.listen(10)
print 'Socket now listening'

#now keep talking with the client
while 1:
    #wait to accept a connection - blocking call
    conn, addr = s.accept()
    print 'Connected with ' + addr[0] + ':' +
```

```
str(addr[1])

    data = conn.recv(1024)
    reply = 'OK...' + data
    if not data:
        break

    conn.sendall(reply)

conn.close()
s.close()
```

Now run this server program in a terminal, and open three other terminals.

From each of the three terminals, perform a Telnet to the server port.

Each of the Telnet terminals will show output as follows:

```
pi@raspberrypi ~/book/chapter14 $ telnet localhost
5000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
happy
OK .. happy
Connection closed by foreign host.
```

The server terminal will show the following:

```
pi@raspberrypi ~/book/chapter14 $ python prog9.py
Socket created
Socket bind complete
Socket now listening
Connected with 127.0.0.1:60225
Connected with 127.0.0.1:60237
Connected with 127.0.0.1:60239
```

So now, the server is up and the Telnet terminals are also connected to it. Now, terminate the server program. All Telnet terminals will show `Connection closed by foreign host`.

Handling multiple connections

To handle every connection, we need separate handling code to run along with the main server thread, which accepts incoming connection requests. One way to achieve this is to use threads. The main server program accepts an incoming connection request and provisions a new thread to handle communication for the connection, and then, the server goes back to accept more incoming connection requests.

This is the required code:

```
import socket
import sys
from thread import *

HOST = ''   # Symbolic name meaning all available
interfaces
PORT = 8888 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
print 'Socket created'

#Bind socket to local host and port
try:
    s.bind((HOST, PORT))
except socket.error , msg:
    print 'Bind failed. Error Code : ' +
str(msg[0]) + ' Message ' + msg[1]
    sys.exit()

print 'Socket bind complete'

#Start listening on socket
s.listen(10)
print 'Socket now listening'

#Function for handling connections. This will be
used to create threads
def clientthread(conn):
    #Sending message to connected client
    conn.send('Welcome to the server. Type
something and hit enter\n') #send only takes
```

```
string

    #infinite loop so that function do not
terminate and thread do not end.
    while True:

        #Receiving from client
        data = conn.recv(1024)
        reply = 'OK...' + data
        if not data:
            break

        conn.sendall(reply)

    #came out of loop
    conn.close()

#now keep talking with the client
while 1:
    #wait to accept a connection - blocking call
    conn, addr = s.accept()
    print 'Connected with ' + addr[0] + ':' +
str(addr[1])

    #start new thread takes 1st argument as a
function name to be run, second is the tuple of
arguments to the function.
    start_new_thread(clientthread ,(conn,))

s.close()
```

Run this server code and open three terminals like before. Now, the server will create a thread for each client connecting to it.

The Telnet terminals will show output as follows:

```
pi@raspberrypi ~/book/chapter14 $ telnet localhost
8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome to the server. Type something and hit
enter
hi
OK...hi
```

```
asd
OK...asd
cv
OK...cv
```

The server terminal will look like this:

```
pi@raspberrypi ~/book/chapter14 $ python prog10.py
Socket created
Socket bind complete
Socket now listening
Connected with 127.0.0.1:60730
Connected with 127.0.0.1:60731
```

This connection handler takes the input from the client and replies with the same.

Looking back

By now, we have learned the basics of socket programming in Python. When testing these programs, if you face the following error, simply change the port number, and the server will run fine:

```
Bind failed. Error Code : 98 Message Address
already in use
```

A Telnet client in Python

The Telnet client is a simple command-line program that is used to connect to socket servers and exchange messages. The following is an example of how to use Telnet to connect to https://www.google.com and fetch the homepage:

```
$ telnet www.google.com 80
```

This command will connect to `www.google.com` on port `80`.

```
$ telnet www.google.com 80
Trying 74.125.236.69...
Connected to www.google.com.
Escape character is '^]'
```

Now that it is connected, the Telnet command can take user input and send it to the respective server, and whatever the server replies will be displayed in the terminal. For example, send the `HTTP GET` command in the following format and hit *Enter* twice:

```
GET / HTTP/1.1
```

Sending this will generate a response from the server. Now we will make a similar Telnet program. The program is simple: we will implement a program that takes user input and fetches results from the remote server in parallel using the threads. One thread will keep receiving messages from the server and another will keep taking in user input. But there is another way to do this apart from threads: the `select()` function. This function allows you to monitor multiple sockets or streams for readability and will generate an event if any of the sockets are ready.

The code for it is as follows:

```
import socket, select, string, sys

#main function
if __name__ == "__main__":

    if(len(sys.argv) < 3) :
        sys.exit()

    host = sys.argv[1]
    port = int(sys.argv[2])

    s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    s.settimeout(2)

    # connect to remote host
    try :
        s.connect((host, port))
    except :
        print 'Unable to connect'
        sys.exit()

    print 'Connected to remote host'

    while 1:
        socket_list = [sys.stdin, s]

        # Get the list sockets which are readable
        read_sockets, write_sockets, error_sockets
= select.select(socket_list , [], [])

        for sock in read_sockets:
            #incoming message from remote server
            if sock == s:
                data = sock.recv(4096)
                if not data :
                    print 'Connection closed'
                    sys.exit()
                else :
                    #print data
                    sys.stdout.write(data)

            #user entered a message
            else :
                msg = sys.stdin.readline()
                s.send(msg)
```

The execution of this program is shown here. It connects to the remote host `google.com`.

```
$ python telnet.py google.com 80
Connected to remote host
```

Once connected, it shows the appropriate connected message. Once the message is displayed, type in some message to send to the remote server. Type the same GET message and send it by hitting *Enter* twice. A response will be generated.

A chat program

In the previous section, we went through the basics of creating a socket server and client in Python. In this section, we will write a chat application in Python that is powered by Python sockets.

The chat application we are going to make will be a common chat room rather than a peer-to-peer chat. So this means that multiple chat users can connect to the chat server and exchange messages. Every message is broadcast to every connected chat user.

The chat server

The chat server performs the following tasks:

- It accepts multiple incoming connections for the client.
- It reads incoming messages from each client and broadcasts them to all the other connected clients.

The following is the code for the chat server:

```
import socket, select

#Function to broadcast chat messages to all
connected clients
def broadcast_data (sock, message):
    #Do not send the message to master socket and
the client who has send us the message
    for socket in CONNECTION_LIST:
        if socket != server_socket and socket !=
sock :
            try :
                socket.send(message)
            except :
                # broken socket connection may be,
chat client pressed ctrl+c for example
                socket.close()
                CONNECTION_LIST.remove(socket)

if __name__ == "__main__":
```

```
    # List to keep track of socket descriptors
    CONNECTION_LIST = []
    RECV_BUFFER = 4096 # Advisable to keep it as
an exponent of 2
    PORT = 5000

    server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    # this has no effect, why ?
    server_socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1)
    server_socket.bind(("0.0.0.0", PORT))
    server_socket.listen(10)

    # Add server socket to the list of readable
connections
    CONNECTION_LIST.append(server_socket)

    print "Chat server started on port " +
str(PORT)

    while 1:
        # Get the list sockets which are ready to
be read through select
        read_sockets,write_sockets,error_sockets =
select.select(CONNECTION_LIST,[],[])

        for sock in read_sockets:
            #New connection
            if sock == server_socket:
                # Handle the case in which there
is a new connection recieved through server_socket
                sockfd, addr =
server_socket.accept()
                CONNECTION_LIST.append(sockfd)
                print "Client (%s, %s) connected"
% addr

                broadcast_data(sockfd, "[%s:%s]
entered room\n" % addr)

            #Some incoming message from a client
            else:
                # Data received from client,
process it
                try:
                    #In Windows, sometimes when a
```

```
TCP program closes abruptly,
                    # a "Connection reset by peer"
exception will be thrown
                    data = sock.recv(RECV_BUFFER)
                    if data:
                        broadcast_data(sock, "\r"
+ '<' + str(sock.getpeername()) + '> ' + data)

                except:
                    broadcast_data(sock, "Client
(%s, %s) is offline" % addr)
                    print "Client (%s, %s) is
offline" % addr
                    sock.close()
                    CONNECTION_LIST.remove(sock)
                    continue

    server_socket.close()
```

In this code, the server program opens up port `5000` to listen for incoming connections from the clients. The chat client must connect to the same port. We can change the port number if we want by specifying another number that is not used by any other program or process.

The server handles multiple chat clients with multiplexing based on the `select()` function. The `select()` function monitors all the client sockets and the master socket for any readable activity. If any of the client sockets are readable, then it means that one of the chat clients has sent a message to the server.

When the `select()` function returns, `read_sockets` will be an array consisting of all socket descriptors that are readable. When the master socket is readable, the server will accept the new connection. If any of the client sockets is readable, the server will read the message and broadcast it back to all the other clients except the one who sent the message. If the broadcast function fails to send the message to any of the clients, the client is presumed to be disconnected, the connection is closed, and the corresponding socket is removed from the connections list.

The chat client

Now, let's code the chat client that will connect to the chat server and exchange the messages. The chat client is based on the Telnet program in Python, which we have already worked on. It connects to a remote server and exchanges messages.

The chat client performs the following tasks:

- It listens for incoming messages from the server.
- It checks for user input, in case the user types in a message and then sends it to the chat server to which it is connected.

The client has to actually listen for server messages and user input simultaneously. To accomplish this, we can use the `select()` function. The `select()` function can monitor multiple sockets or file descriptors simultaneously for activity. When a message from the server arrives on the connected socket, it is readable, and when the user types a message from the keyboard and hits *Enter*, the `stdin` stream is readable.

So, the `select()` function has to monitor two streams: the first is the socket that is connected to the remote web server, and the second is `stdin` or terminal input stream from the keyboard. The `select()` function waits until some activity happens. So, after calling the `select()` function, the function itself will return only when either the server socket receives a message or the user enters a message using the keyboard. If nothing happens, it keeps on waiting.

We can create an array of the `stdin` file descriptor that is available from the `sys` module and the server sockets. Then, we call the `select()` function, passing it the list. The `select()` function returns a list of arrays that are readable, writable, or have an error.

Here is the Python code that implements the previous logic using the `select()` function as follows:

```
import socket, select, string, sys

def prompt() :
    sys.stdout.write('<You> ')
    sys.stdout.flush()

#main function
if __name__ == "__main__":

    if(len(sys.argv) < 3) :
        print 'Usage : python telnet.py hostname
port'
        sys.exit()

    host = sys.argv[1]
    port = int(sys.argv[2])

    s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
    s.settimeout(2)

    # connect to remote host
    try :
        s.connect((host, port))
    except :
        print 'Unable to connect'
        sys.exit()

    print 'Connected to remote host. Start sending
messages'
    prompt()

    while 1:
        socket_list = [sys.stdin, s]

        # Get the list sockets which are readable
        read_sockets, write_sockets, error_sockets
= select.select(socket_list , [], [])

        for sock in read_sockets:
            #incoming message from remote server
            if sock == s:
                data = sock.recv(4096)

                if not data :
                    print '\nDisconnected from
chat server'
```

```
                sys.exit()
            else :
                #print data
                sys.stdout.write(data)
                prompt()

        #user entered a message
        else :
            msg = sys.stdin.readline()
            s.send(msg)
            prompt()
```

Execute the chat client from multiple consoles as follows:

```
$ python telnet.py localhost 5000
Connected to remote host. Start sending messages
<You> hello
<You> I am fine
<('127.0.0.1', 38378)> ok good
<You>
on another console
<You> [127.0.0.1:39339] entered room
<('127.0.0.1', 39339)> hello
<('127.0.0.1', 39339)> I am fine
<You> ok good
```

In this way, the messages sent by one client can be seen on the terminal of the other client.

References

The code and definitions in this chapter have been referred to from a few online sources, as follows:

- http://www.binarytides.com/python-socket-programming-tutorial
- http://www.binarytides.com/python-socket-server-code-example
- http://www.binarytides.com/code-chat-application-server-client-sockets-python
- http://www.binarytides.com/code-telnet-client-sockets-python
- http://www.binarytides.com/programming-udp-sockets-in-python
- http://www.binarytides.com/socket-programming-c-linux-tutorial/
- http://www.diffen.com/difference/TCP_vs_UDP
- http://eprints.uthm.edu.my/7531/1/FAYAD_MOHAMMED_MOHAMMED_GHAWBAR_24.pdf
- http://www.cs.dartmouth.edu/~campbell/cs60/UDPsockets.jpg
- http://www.cs.dartmouth.edu/~campbell/cs60/TCPsockets.jpg

Exercise

Additionally, you can learn more about raw sockets in Python from the following links:

- http://www.binarytides.com/raw-socket-programming-in-python-linux
- http://www.binarytides.com/python-syn-flood-program-raw-sockets-linux

Summary

In this chapter, we learned about the concepts required to understand the transfer of data between two or more machines, in which the sender is termed as a server and the receiver is termed as the client. We also learned the basics of establishing a connection between a server and a client.

We built both UDP and TCP socket programs using Python and learned how to perform tasks such as connecting to a server, sending messages, receiving messages, and so on. Finally, we combined all that we learned to create a simple chat program that would allow two users to communicate with each other. We can now build our own instant messaging service!

Now that we are at the end of the book, take some time to go over the projects that you have successfully built. We have completed some projects, ranging from the beginner level to the advanced level, and learned a lot about the intricacies of working with the Raspberry Pi and all its parallel concepts firsthand. Starting from the introduction of the Raspberry Pi to projects such as motion detection from images and intruder detection using the concepts of the Internet of Things, we have covered almost everything that you would require to build some interesting projects of your own.

We hope this inspires you to keep on building fun projects that are not only interesting to set up and cool to watch but also useful to the world in general. Keep on hacking!

Chapter 15. Newer Raspberry Pi Models

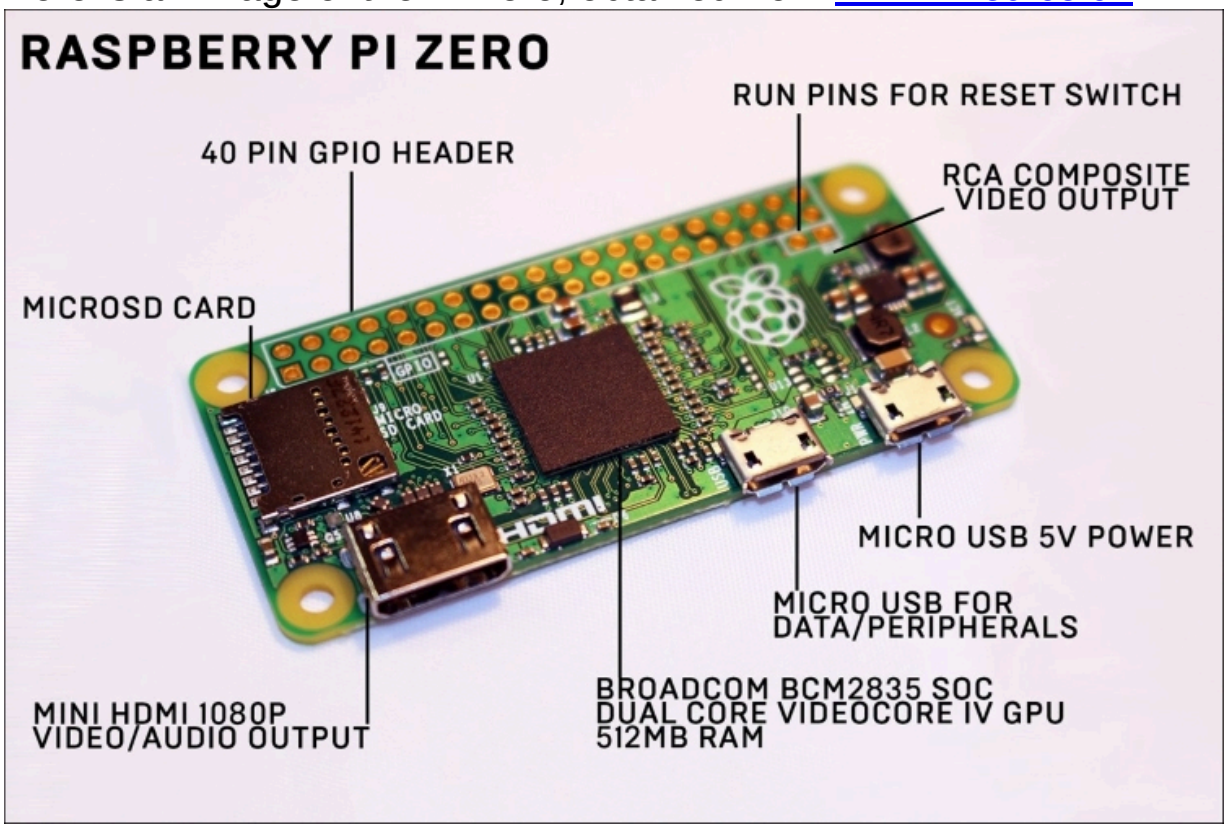
The latest additions in the Raspberry Pi family are the Raspberry Pi Zero and Raspberry Pi 3. We did not discuss these new additions in the earlier chapters. So, let's focus on a few details of both these new members of the Raspberry Pi family.

The Raspberry Pi Zero

The size of a Raspberry Pi Zero is half the size of a Model A+. This is ideal for embedded projects where size and power requirements are stringent. Here are the specifications for the Raspberry Pi Zero:

- A 1-GHz, single-core CPU
- 512 MB of RAM
- Mini HDMI and USB on-the-go ports
- Micro USB power
- A HAT-compatible 40-pin header
- Composite video and reset headers

Here is an image of the Pi Zero, obtained from www.wired.co.uk:



There are two ways in which we can connect the Pi Zero to a display:

- Using mini-HDMI-to-HDMI converter, shown here:



Note

You can purchase one from Adafruit, using these links:

https://www.adafruit.com/products/2775

https://www.adafruit.com/products/2819

- Also, we can use VGA for display. For this, we need to use a mini-HDMI-to-VGA adapter.

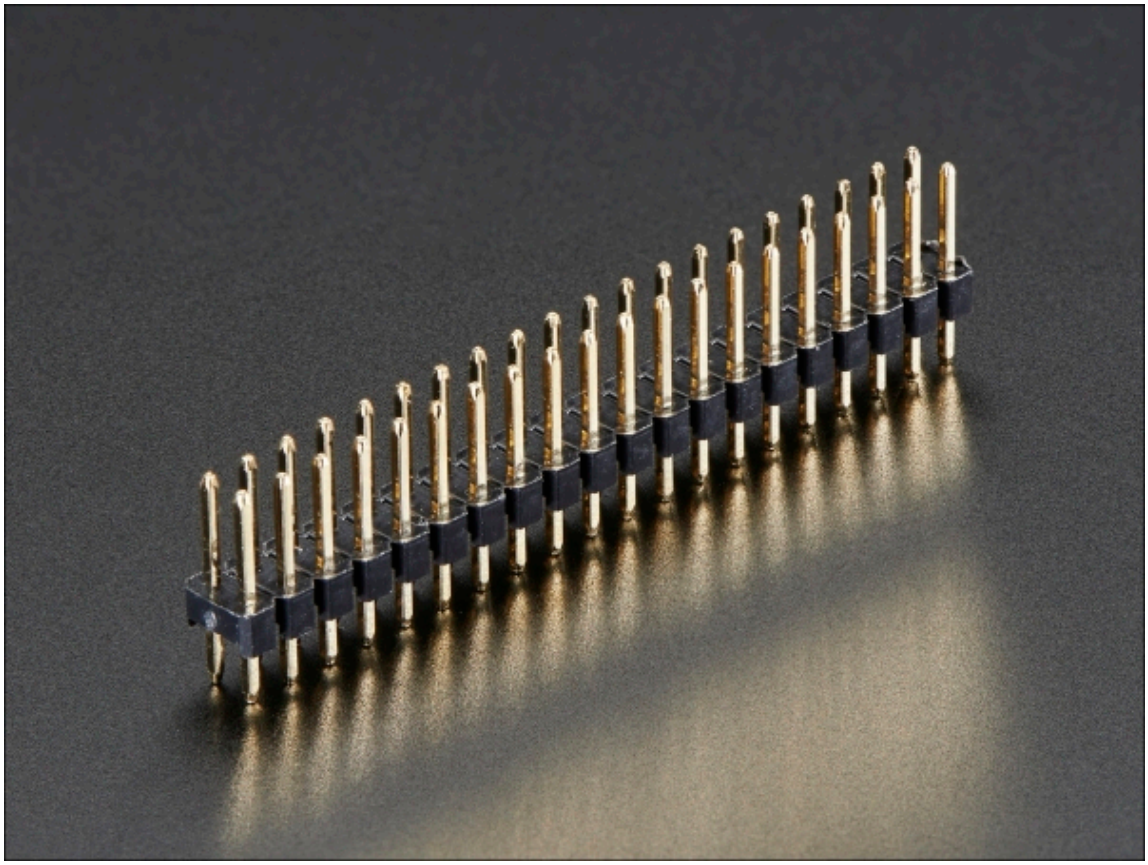


Note

This can be purchased at https://www.adafruit.com/products/3048.

For conveniently using the GPIO pins of the Pi Zero, you can use any one of the following two-pin strip GPIO headers:

- https://www.adafruit.com/products/2822



- https://www.adafruit.com/products/2823



Also, for connecting various devices, a mini-USB-to-USB converter can be used.

Note

This can be obtained from https://www.adafruit.com/products/2910.



The Raspberry Pi 3

The Raspberry Pi 3 is the latest addition to the Raspberry Pi family. It is a third-generation Raspberry Pi. It replaced the Raspberry Pi 2 Model B in February 2016. Compared to the Pi 2, it has the following features:

- A 1.2-GHz 64-bit quad-core ARMv8 CPU
- 802.11n onboard wireless LAN
- Bluetooth 4.1
- **Bluetooth Low Energy** (**BLE**)

Also, like the Raspberry Pi 2, also has the following features:

- Four USB ports
- 40 GPIO pins
- A full-HDMI port
- An Ethernet port
- A combined 3.5mm audio jack and composite video
- A camera interface (CSI)
- A display interface (DSI)
- A micro-SD card slot (now push-pull rather than push-push)
- A VideoCore IV 3D graphics core

The form factor of the Pi 3 is identical to that of the Pi 2 and Pi 1 Model B+. The Pi 3 has complete backward compatibility with the Pi 1 and 2.

Note

The product description for the Pi Zero and Pi 3 can be found on Raspberry Pi's official product pages:
https://www.raspberrypi.org/products/pi-zero/

https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

Part 2. Module 2

Building a Home Security System with Raspberry Pi

Build your own sophisticated modular home security system using the popular Raspberry Pi board

Chapter 1. Setting Up Your Raspberry Pi

Before we can get into the realms of building our home security system, there's a bit of preparation work needed to get our Raspberry Pi up and running. So, we're going to go through the initial steps needed to get our Pi ready to be worked on.

In this chapter, we will cover the following topics:

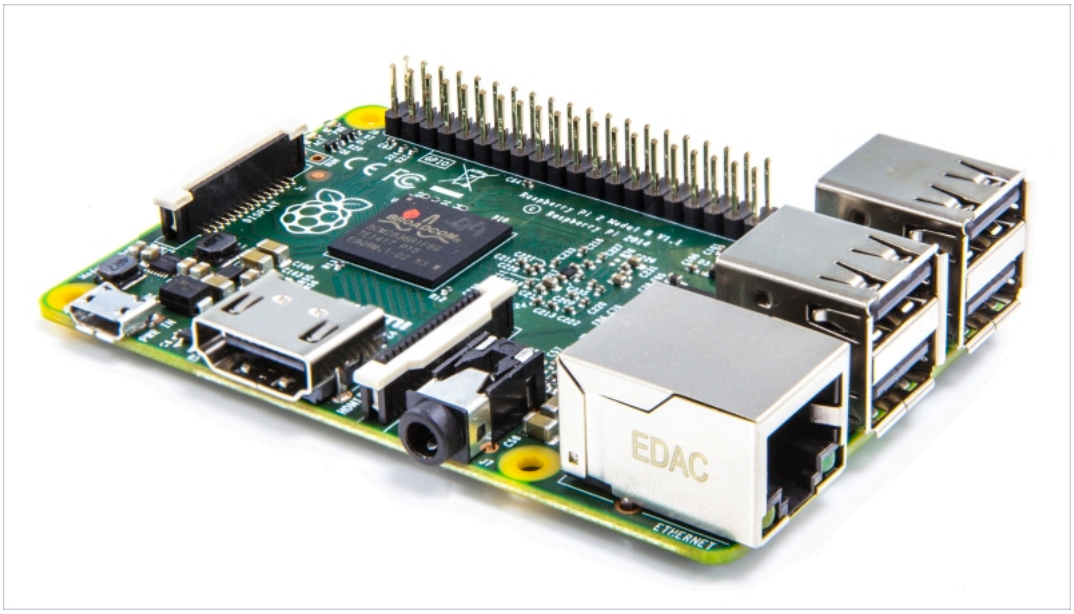
- Exploring the different versions of the Raspberry Pi that are available
- Choosing the right Raspberry Pi version for your system
- Preparing the SD Card with the Raspbian Operating System
- Learning how to remotely access the Raspberry Pi over your home network
- Updating our operating system with the latest packages
- Exploring the time-keeping options on the Raspberry Pi
- Setting the user and root passwords to secure our Raspberry Pi

Which flavor of Pi?

Since the Raspberry Pi was released in 2012, there have been several versions of the mini-PC board released. I'll go though each of the versions released with their respective features so that you can choose which one is suitable for your particular project.

The good news is that it doesn't really matter which version you use in terms of power, as our home security system doesn't necessarily need loads of processing power, depending on what you want your system to do, of course). You might have an older board kicking about that will work for you.

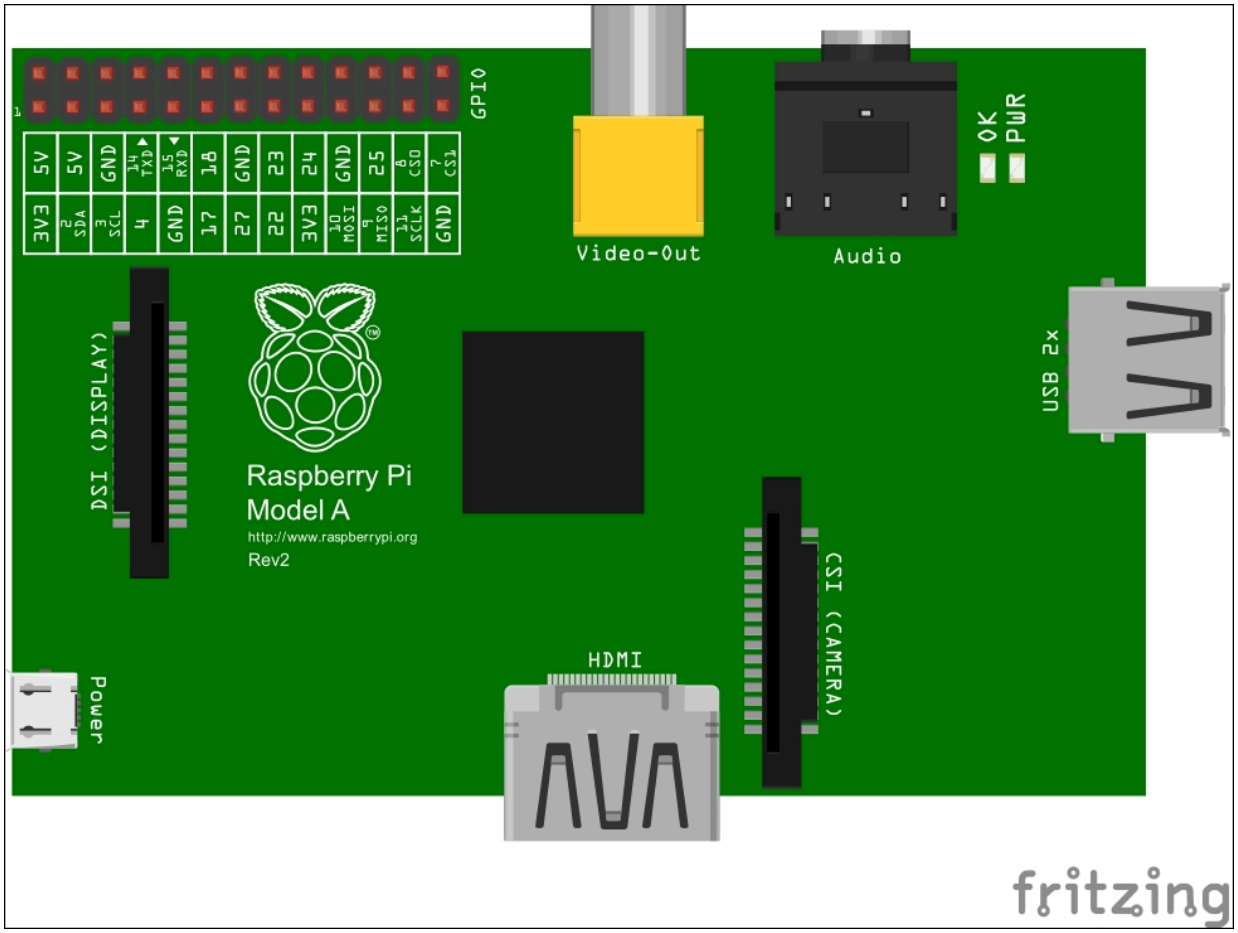
The other piece of good news is that the GPIO interface pin layouts are the same across all the versions. The later versions have more pins, but the original 26 pins still remain in the same place.



The latest Raspberry Pi Version 2

Raspberry Pi Model A

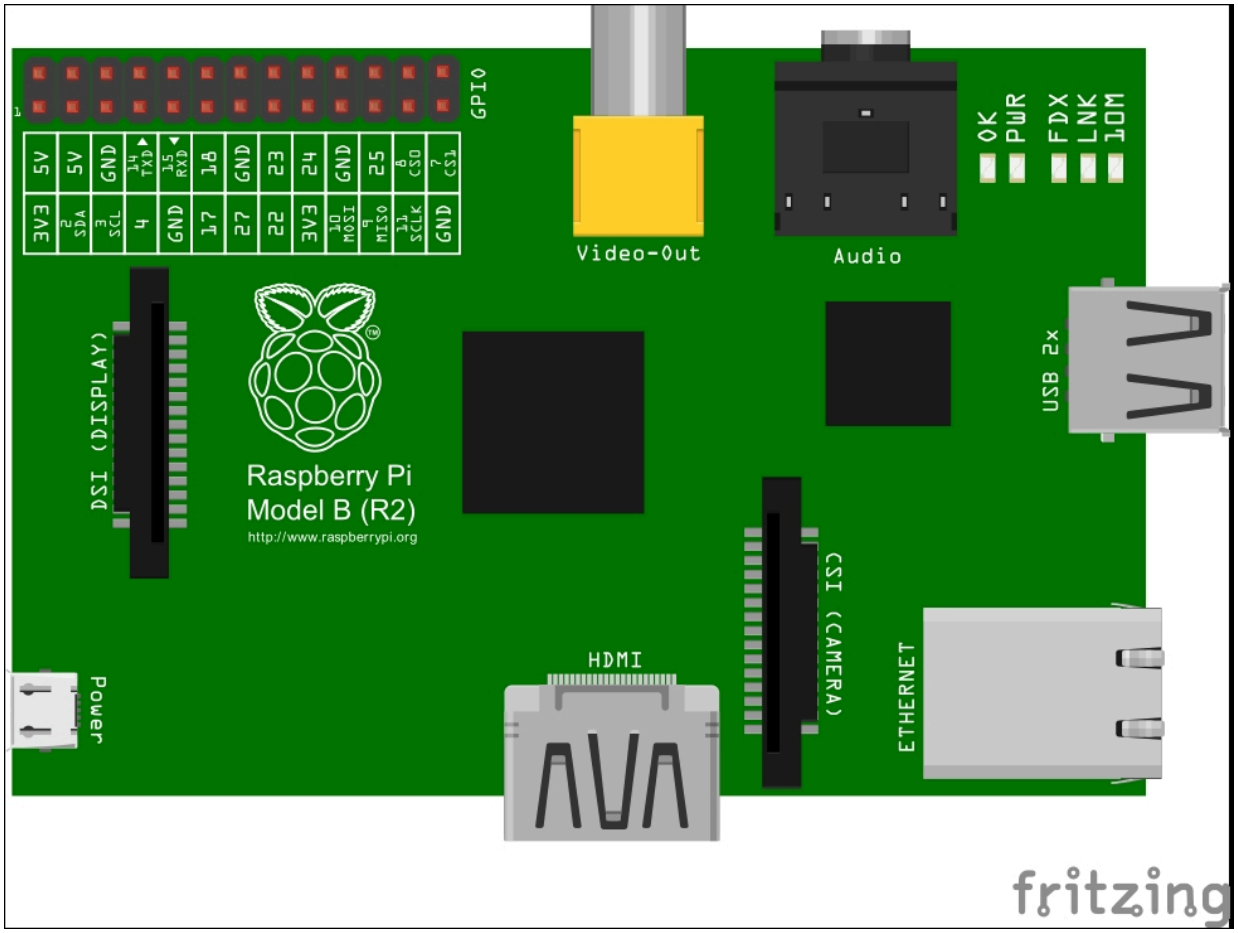
The baby of the family is the Model A; it was released as a lower-cost version of the Model B, shown in the following section. Its main differences from the Model B are that it features just 256Mb of memory and has no Ethernet port; so if you want to connect this board to a network, you are limited to using a USB Wi-Fi dongle.



The Raspberry Pi Model A Board Layout

Raspberry Pi Model B

This was the first version of Raspberry Pi to be released; an updated revision, which improved the power system and USB port protection, came later. It features 512Mb of memory and has an Ethernet port for connecting to your network. This is probably the most common version used, and having the Ethernet port is incredibly useful, especially to get up and run quickly in order to set up and configure your Pi without the need for a keyboard and monitor.



The Raspberry Pi Model B Layout

Raspberry Pi Model B+ and Model 2

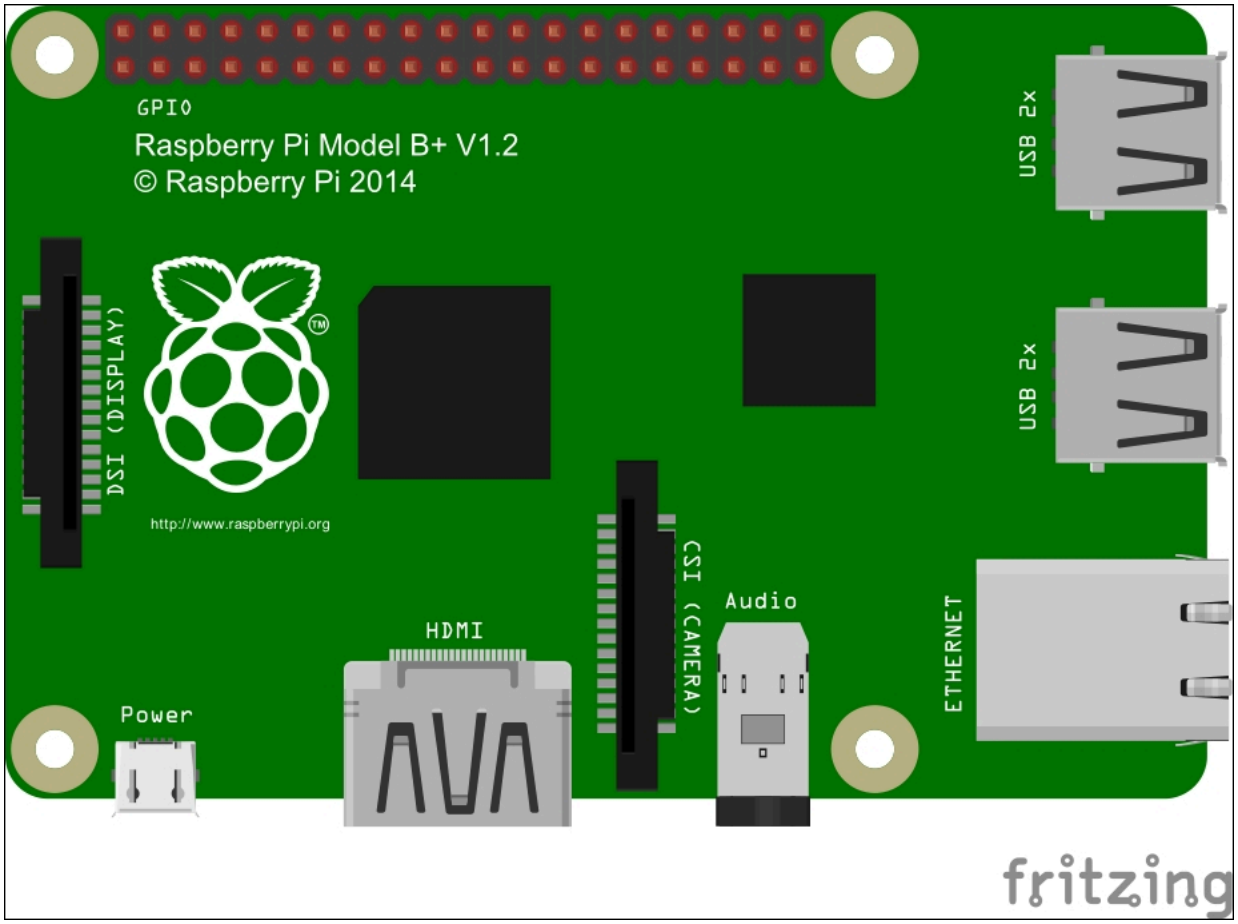
In 2014, the Raspberry Pi Foundation released a new version of the board that had some fundamental changes as compared to the previous version. The most fundamental changes were the board layout, form factor, and mounting points—much to the dismay of the many enclosure and accessory manufacturers out there.

In fact, we were in the middle of prototyping an enclosure for a commercial product that we were developing based on the Raspberry Pi—fortunately we caught wind of the board change in the nick of time and were able to change our enclosure to support the upcoming model B+.

The main electronic changes to this board are the addition of 2 more USB ports that can deliver more power to peripherals, an expanded GPIO interface, and the removal of the composite video port that is now consolidated into the audio jack. It also now uses a micro SD card with a better card slot.

In February 2015, a more powerful Raspberry Pi was released: the Raspberry Pi Model 2. It's similar to the Model B+ in terms of form-factor and interfaces, but is now reportedly 6-times faster than the Model B/B+ with its upgraded ARM processor and 1Gb of memory.

At the same low cost of less than –30, it's a fantastic little board and a great power-house for embedded systems.



The Raspberry Pi Model B+ and Model 2 Layout

Model comparison table

	Model A	Model B	Model B+	Version 2
Processor	ARM1176JZF-S 700 MHz processor, VideoCore IV GPU			Quad-core ARM Cortex-A7 CPU and a VideoCore IV dual-core GPU
Memory	256Kb	512Kb	512Kb	1Gb
USB Ports	2	2	4	4
Ethernet	No	Yes	Yes	Yes
No.GPIO Pins	26	26	40	40

	Model A	Model B	Model B+	Version 2
Storage	SD Card	SD Card	Micro SD Card	Micro SD Card

So which one?

Essentially, any version of the Raspberry Pi will work with the modules presented in this book, but if you want to exploit features such as the camera, which may require more processing power and memory, or want to have an Ethernet connection, you'll need to use the Model B.

If you want to start plugging additional stuff into the USB port, such as a GSM modem, then I recommend that you use the Model B+ as it delivers more power to those kinds of devices without the need for additional USB hubs.

If you want to do more processing with video and images from an attached camera, or want to connect lots of things, then go for the latest **Model 2** board. I'm going to assume that's the one you have chosen for this project, and that's the one I'll be using throughout this book; just be aware of any limitations if you choose to use an earlier model.

Note

The Raspberry Pi Foundation site has more detailed information about each model. You can visit it at https://www.raspberrypi.org/products.

Preparing the SD card

The Raspberry Pi only boots from an SD card (or micro SD card for the B+ and v2 models), and cannot boot from an external drive or USB stick (well that's not strictly true, but is beyond the scope of this book).

It's recommended that you use a Class 10 SD card for performance, but a Class 4 or 6 card will be fine for this project. You'll need to have a minimum card size of 4Gb.

Now that we have our Raspberry Pi board and SD card to hand, we need to prepare the SD Card specifically for our home security system. We're going to use the standard Raspbian operating system as there really is no reason to use any other distribution; it's the de facto choice for the Raspberry Pi.

Downloading the Raspbian image

You'll need to grab the latest **Raspbian OS** image from the Raspberry Pi site at https://www.raspberrypi.org/downloads.

Download the Raspbian OS ZIP file containing the image to your PC.

> ### Note
>
> At the time of writing, the latest version was Raspbian Jessie version 4.1 (`2015-09-24-raspbian-jessie.zip)`.

Once downloaded, unzip the file and you'll have the file, `2015-09-24-raspbian-jessie.img`.

The next thing to do is burn this image to your SD card–

Using Microsoft Windows

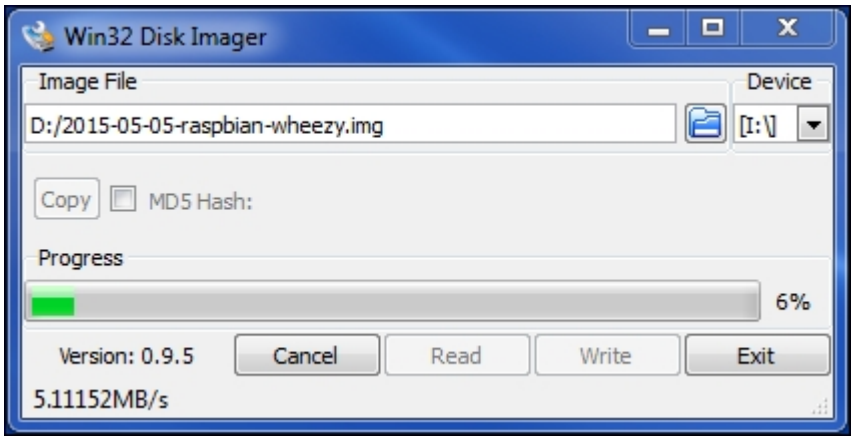
On a Windows PC, the best way to burn the image to your SD card is to use the **Win32 Disk Imager** utility. This can be downloaded from http://sourceforge.net/projects/win32diskimager.

> **Note**
>
> The current version, at the time of writing, is 0.9.5.

It doesn't have an installer, and launches directly from the EXE file.

Now, it's time to create your SD card image:

1. Insert your SD card into the PC and launch the Win32 Disk Imager.
2. Select the SD card device drive letter (make sure it's right!).
3. Choose the Raspbian image file you've just downloaded.
4. Click on the **Write** button to create the SD card image.



Using Linux

On a Linux PC, you'll need to use the **gparted** and **dd** utilities to burn the image on your SD card.

Carry out the following steps to create your SD card image:

1. Extract `2015-09-24-raspbian-jessie.img` to your `Home` folder.
2. Insert your SD card into the PC.
3. If you're not already in a shell terminal window, open one (you can use *Ctrl* + *Alt* + *T* on most graphical-based desktop systems).
4. Type the following command in the shell terminal:

   ```
   $ sudo fdisk -l
   ```

   In the list check, your SD card appears as a drive device (for example, `/dev/sdb`). It's crucial that you ensure you use the right device in the next step. We'll assume that your device is `/sdb`.

5. To burn the image to the SD card, type the following command:

   ```
   $ sudo dd if=2015-09-24-raspbian-jessie.img
   of=/dev/sdb
   ```

6. Hit *Enter* and go make a cup of tea or coffee as this will take a while. You'll know that it's finished when the command (`$`) prompt re-appears.
7. When the command prompt does re-appear, type the following command:

   ```
   $ sudo sync
   ```

8. Once that command has finished, you can remove the SD card from the PC.

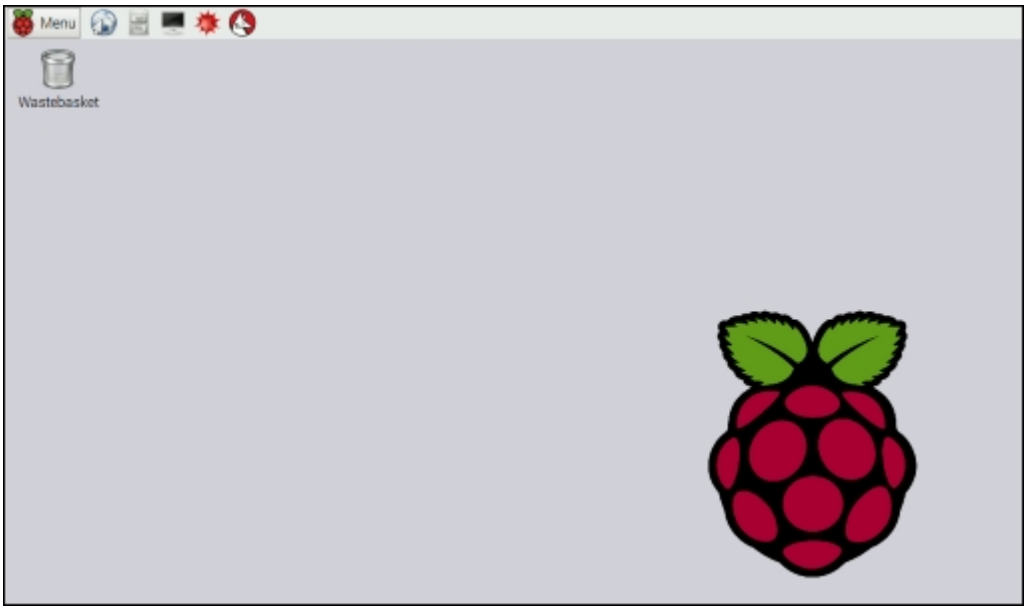
Booting your Pi

You're now ready to boot up your Raspberry Pi. Pop your shiny new SD card into it and plug in the power.

Assuming that you have a monitor attached to your Pi, you should see your system booting up nicely. Although you could wait for it to boot up and connect to it via a terminal session (we'll look at that

later), I recommend that you connect a monitor to it, at least in the first instance, just to make sure everything is working correctly.

In the new **Jessie** version of Raspbian, you'll boot straight into a desktop GUI, which is a major change from previous versions, where you'd be taken to the **raspi-config** utility, the first time the system is run, where you'd set up your Pi, and importantly, expand the file system to use the entire space available on your SD card.



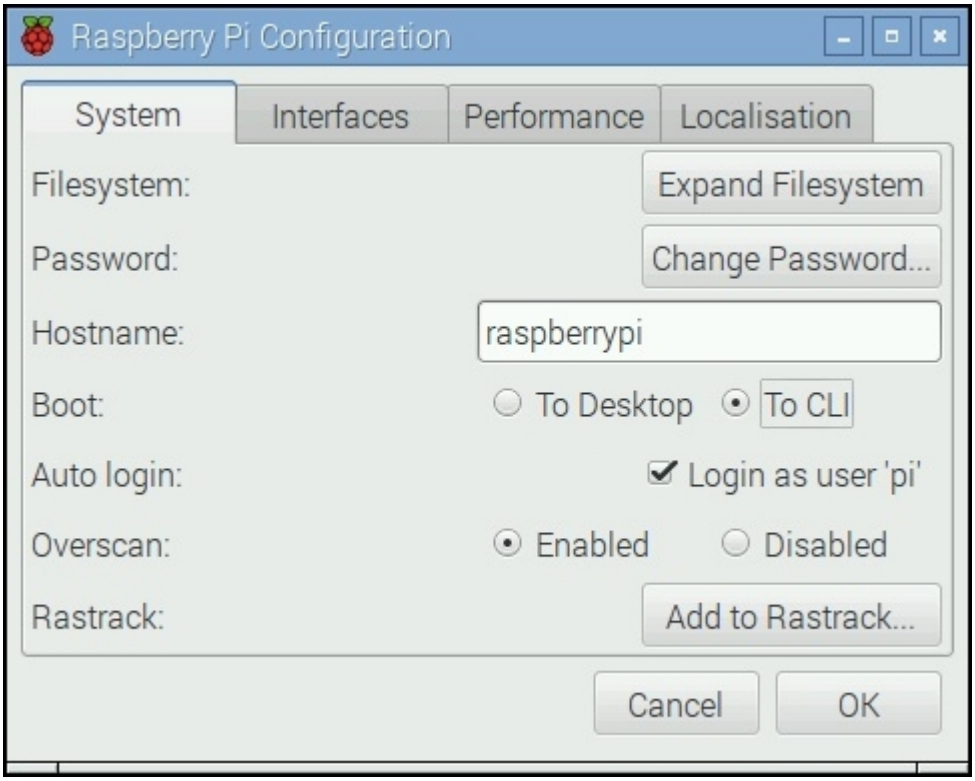
Debian Jessie boots into the GUI by default

Expanding the file system

When you first create your Raspbian SD card, you'll only be left with about 200Mb of space in the file system, regardless of the size of your SD card. This is not much use, so we want to expand the file system so that it uses all of the available space on the card.

Fortunately, this is very easy on the Raspberry Pi now, as this function is available in the Raspberry Pi Configuration Tool on the desktop.

To access the new configuration tool, go to **Menu** and select **Preferences** | **Raspberry Pi Configuration**.



The new Raspberry Pi Configuration Tool

Tip

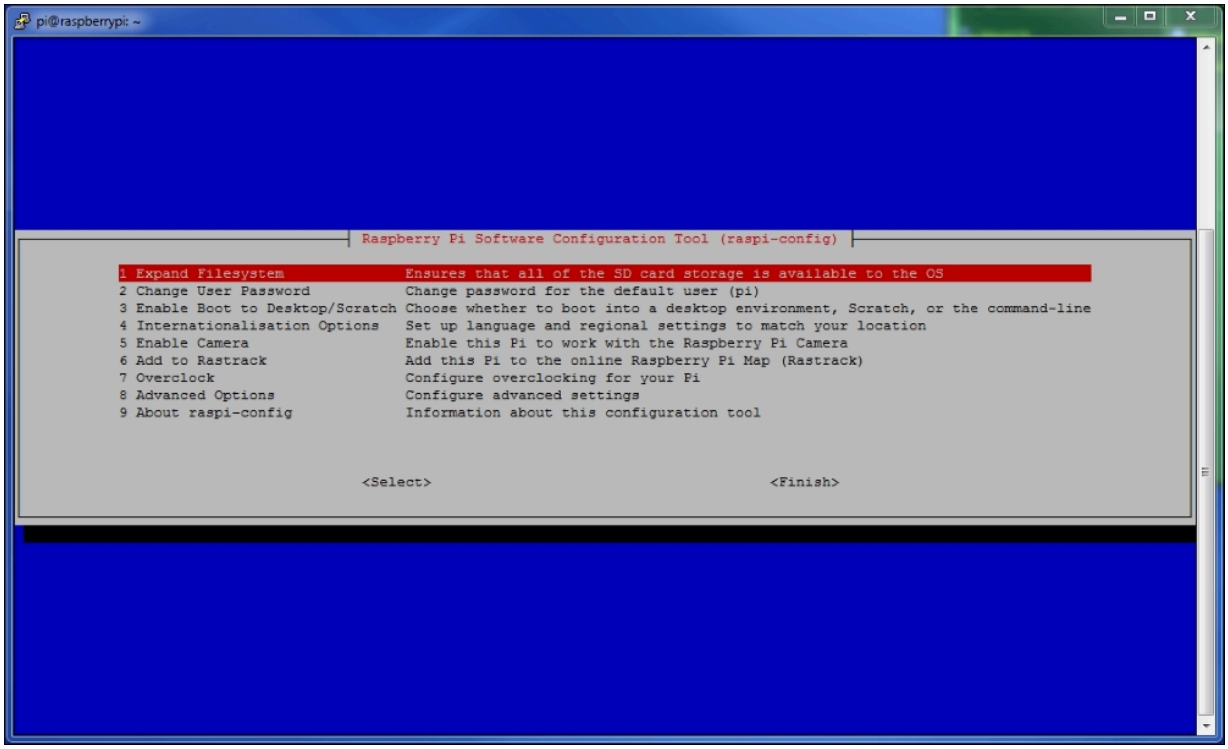
Goodbye GUI

Most of our work is going to be done in the command-line interface (CLI). Therefore, before we reboot the system in a minute, let's change the **Boot** option by selecting **To CLI**, as shown in the previous screenshot, so boot into the command line going forward.

Anyway, now we click on the **Expand Filesystem** button, and in a couple of seconds, you'll see a confirmation message. The filesystem will be expanded when the system next reboots.

Using the raspi-config utility

If you have an older version of Raspbian, or you're not using the desktop GUI, then you'll need to use the raspi-config utility (which is still better than the old days when we had to do this manually in the shell). The first time you boot up, you'll be taken straight to the raspi-config utility.



The first option is the `Expand Filesystem` option; select this and you'll see various commands scrolling up the screen. Once it's finished, you'll see the following message:

```
Root partition has been resized.
The filesystem will be enlarged upon the next
reboot
```

Click on `OK`.

Select `Finish` on the config screen and reboot your Pi when prompted.

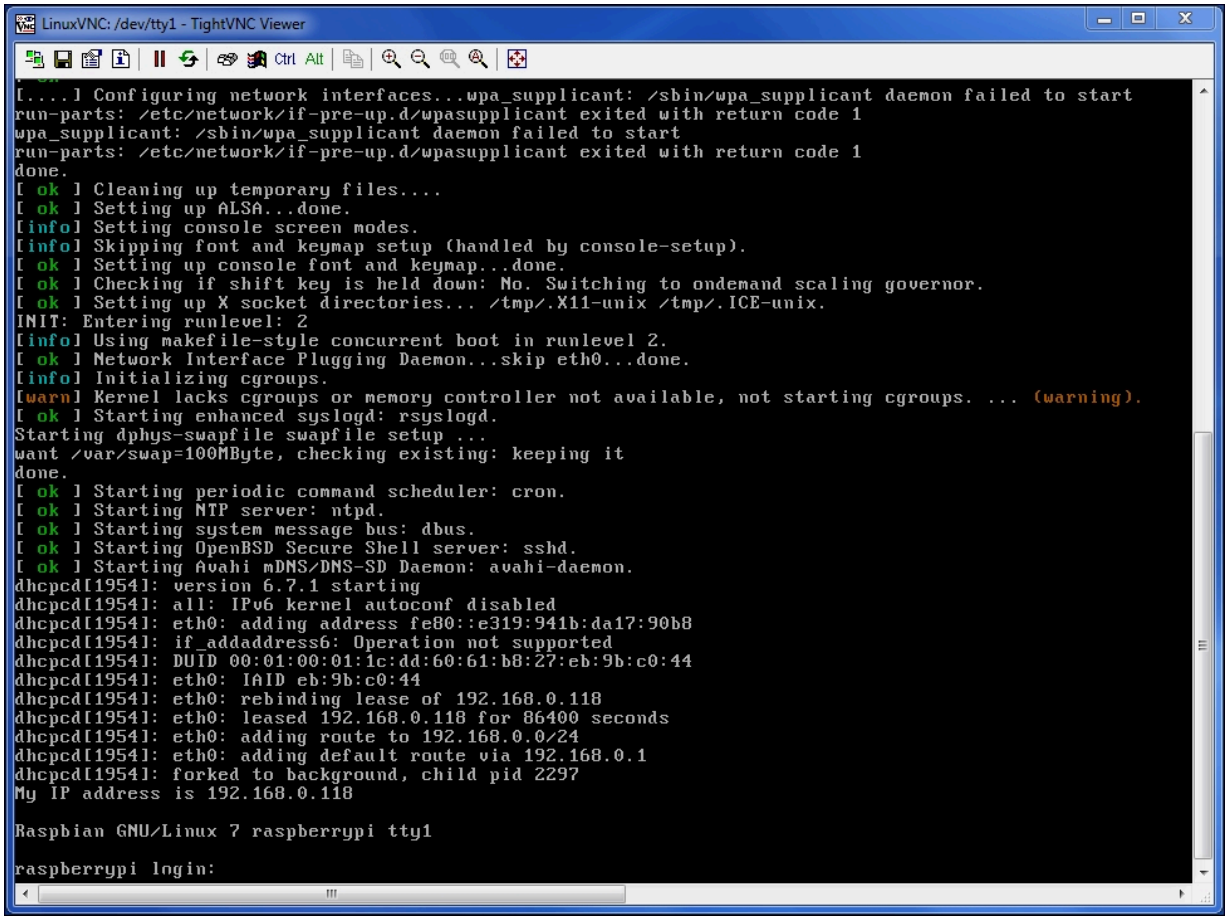
After your Pi reboots with its fuller file system, you'll be taken straight to the shell prompt where you can log in with the default user and password.

Login: `pi`

Password: `raspberry`

Setting up your Pi

When you boot into the shell and have the Ethernet connected, hopefully the Pi will have connected to your home network and acquired an IP address from your router. If this is the case, you should see the **IP address** that has been issued just before the login prompt, as shown in the following screenshot:



As you can see from my screenshot, it's given me the IP address, `192.168.0.118`. This is good because I can now access the Pi remotely, using a secure shell (SSH) client to connect to it from the comfort of my laptop. This is particularly useful when my Pi is in the office and I want to sit on my sofa in front of the telly but still work on it, which I often do when I'm feeling lazy.

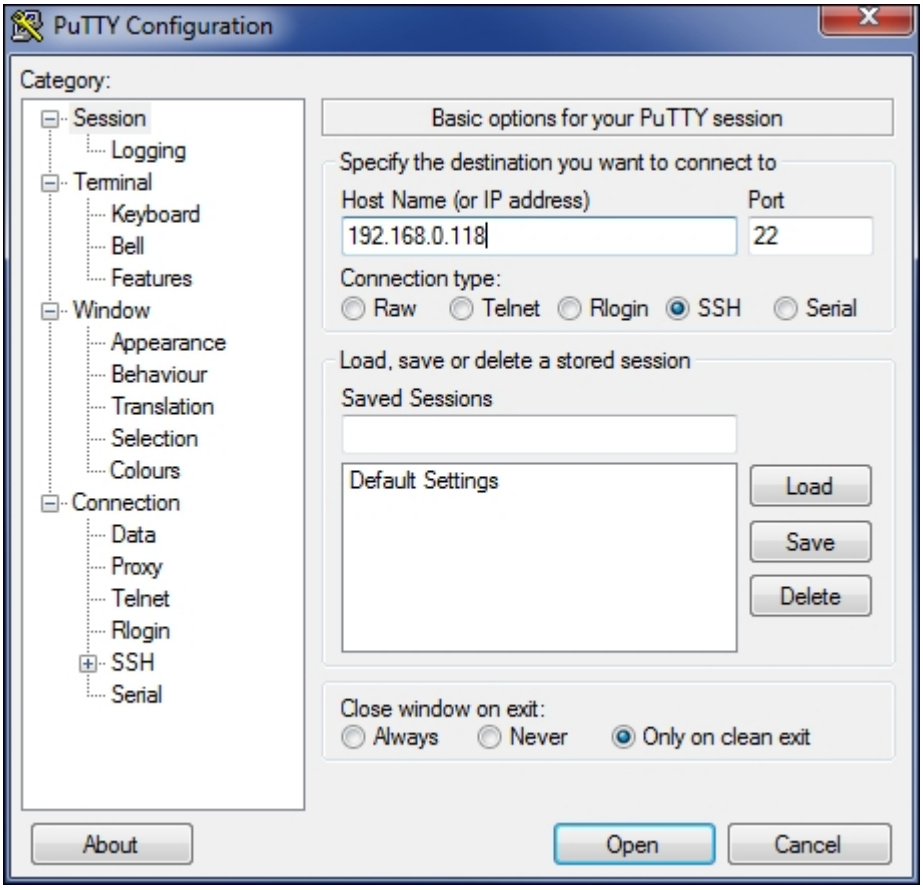
Tip

Downloading the example code

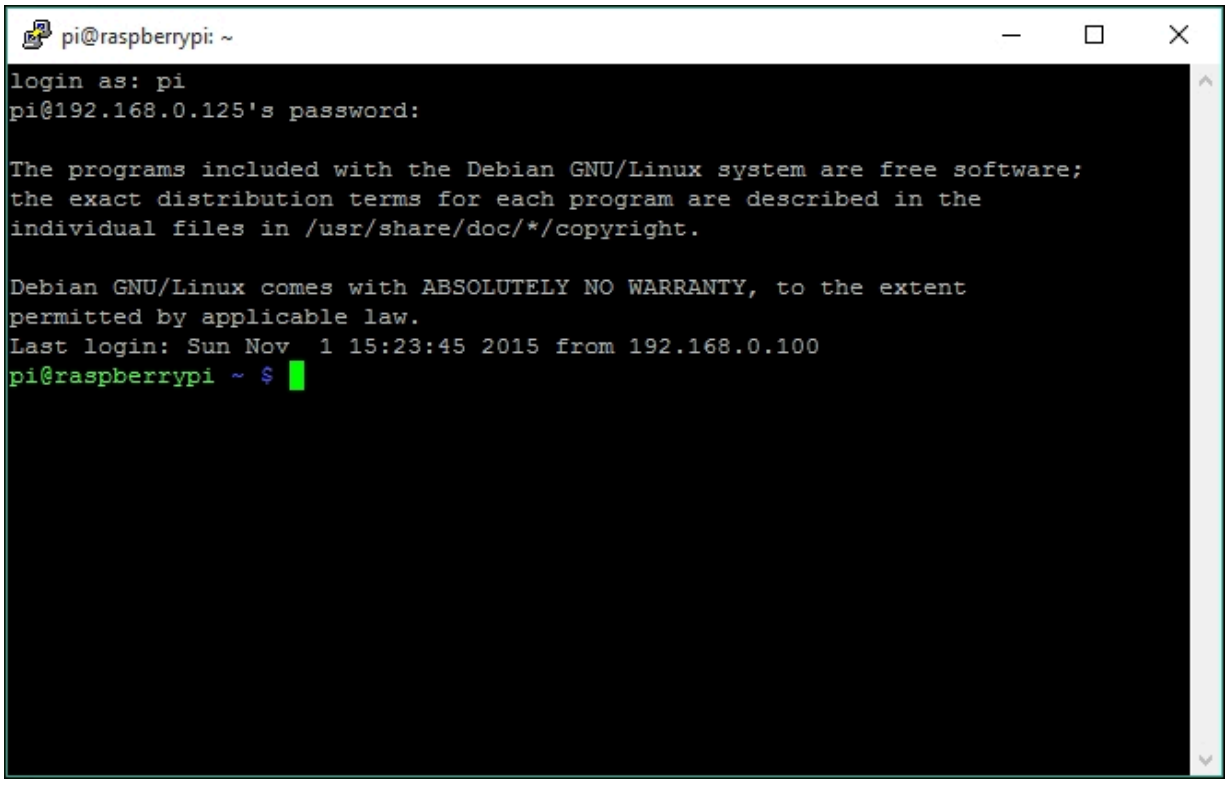
You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

To do this, download **PuTTY**: a utility that allows you to connect to shell terminals remotely over the network. You can download it from http://www.putty.org.

Install and launch **PuTTY** and you're ready to connect to your Pi remotely from the comfort of your sofa.



Type the IP address of the Raspberry Pi into the Host Name box and click on **Open**. You'll be connected to your Pi in a remote terminal window. Once you've logged in, you can do pretty much everything on your Pi, as if you were sitting in front of it.

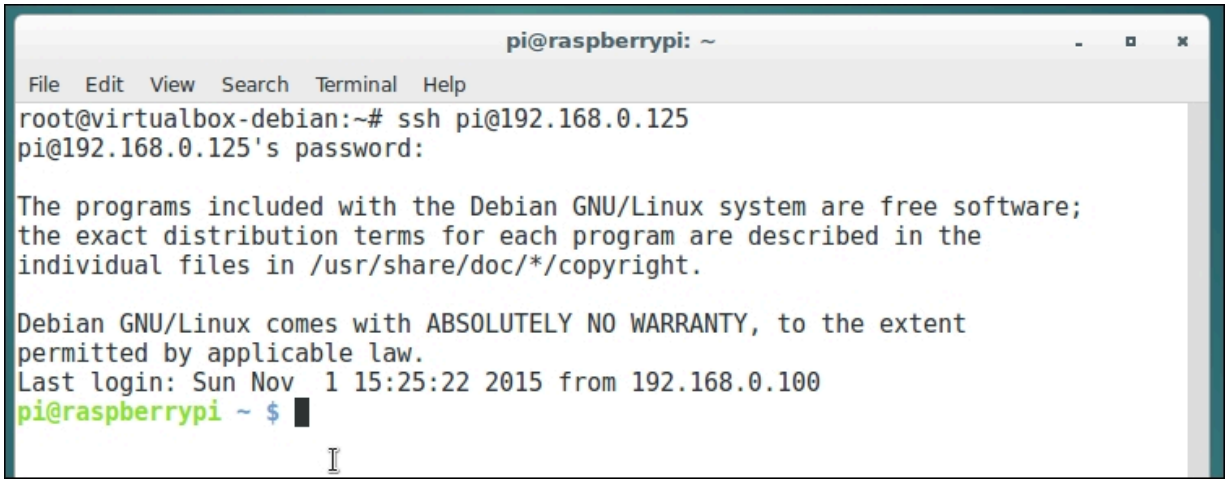


We'll assume from now on that most of the work we do will be through a remote shell session, unless highlighted otherwise.

If you want to use the command line to launch the Raspberry Pi remote shell—for example, from another Linux system—use the following command from your terminal window:

```
# ssh pi@192.168.0.125
```

You'll then be prompted for the Pi's password and taken into a shell session.



The Pi shell session launched from a Debian desktop terminal window

Getting up to date

Something that you should get into the habit of doing is updating the operating system regularly; even though you may have the latest image installed, it's very likely that there are updated packages available. To update your OS, enter the following command:

```
$ sudo apt-get update
```

After this, enter the following one:

```
$ sudo apt-get dist-upgrade
```

This may take a while, depending on the number of updates required.

Getting the right time

The Raspberry Pi doesn't have on-board real-time clock hardware. This is one of the deliberate omissions to keep the cost of the board down. Instead, the Pi gets its time when it boots up from time servers on the Internet using the **Network Time Protocol** (**NTP**). However, if

there is no Internet connection at the time of booting up, then the time will be wrong.

> **Note**
>
> In our security system, it's important that the time is kept accurate so that timestamps on log files and images are correct.

fake-hwclock

The `fake-hwclock` package is included in the latest Raspbian distributions, but in other past versions it wasn't. If you need to install it, use the command that follows:

```
$ sudo apt-get install fake-hwclock
```

`fake-hwclock` is used by the Raspberry Pi to try and keep time when there is no network connection. It will regularly save the current time and restore it at boot-up. The obvious problem with this is that if the Pi has been switched off a few days, then the time will be set to the last time that it was on, using `fake-hwclock`.

If you want to see what time it last logged, type the following command:

```
$ cat /etc/fake-hwclock.data
```

ntp

The **Network Time Protocol** (**NTP**) is used when there is an Internet connection available and it can request the latest most accurate time from one or more time servers on the Internet.

By default, the **ntp service** is enabled on the latest Raspbian distribution, but it will initially get its time at boot-up from `fake-`

`hwclock` if there is no Internet connection. There may be times when it's necessary to force the ntp service to update from the Internet—for example, if the Internet connection is restored sometime after boot-up.

To force the ntp service to update from the Internet, use the following commands:

```
$ service ntp stop
$ ntpd -gq
$ service ntp start
```

Talking of security–

There's no point in having a security system if the system itself is not secure. So, now we'll change the default password for the **pi** user.

From the prompt, type the following command:

```
$ sudo passwd pi
pi@raspberrypi ~ $ sudo passwd pi
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

What is this sudo thing anyway?

You'd have noticed that we've been putting `sudo` at the start of each command that we run in the terminal window. This is so that commands are run as the **root user**—the highest security level. This elevated security is required to perform many operations. `sudo` actually means *super do*.

If you can't be bothered to type `sudo` every time, then you can switch to the super user by typing the following:

```
$ sudo su
```

You'll see that the prompt changed from a `$` to a `#`, which indicates that you are now running as the root user.

So, this might be a good time to change the root user password too! To do this, type the following:

```
# passwd
root@raspberrypi:/home/pi# passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
root@raspberrypi:/home/pi#
```

Connecting via Wi-Fi

You can also connect your Raspberry Pi to your network using Wi-Fi by plugging a USB dongle into it. There are additional configuration steps required to make this work, which are beyond the scope of this chapter, but there are many resources available covering this subject.

> ## Note
>
> You can find recipes for connecting your Raspberry Pi using Wi-Fi in the **Raspberry Pi Networking Cookbook** by Rick Golden, published by Packt Publishing (https://www.packtpub.com/hardware-and-creative/raspberry-pi-networking-cookbook).

Summary

In this chapter, we took our Raspberry Pi out of its box and prepared it to be the centerpiece of our home security system. Along the way, we installed and set up the operating system, connected our Pi to the network, and accessed it remotely. We also secured our Pi and made sure it could keep the right time.

In the next chapter, we're going to explore the GPIO port and the various interfaces it features. We'll look at the various things we can connect to the Raspberry Pi using the GPIO port, including switches and sensors, as we start to build our home security system.

Chapter 2. Connecting Things to Your Pi with GPIO

The Raspberry Pi has lots of ways to connect things to it, such as plugging things into the USB ports, connecting devices to the on-board camera and display ports, and connecting things to the various interfaces that make up the **GPIO connector**. As part of our home security project, we'll be focusing mainly on connecting things to the GPIO connector.

In this chapter, we will cover the following topics:

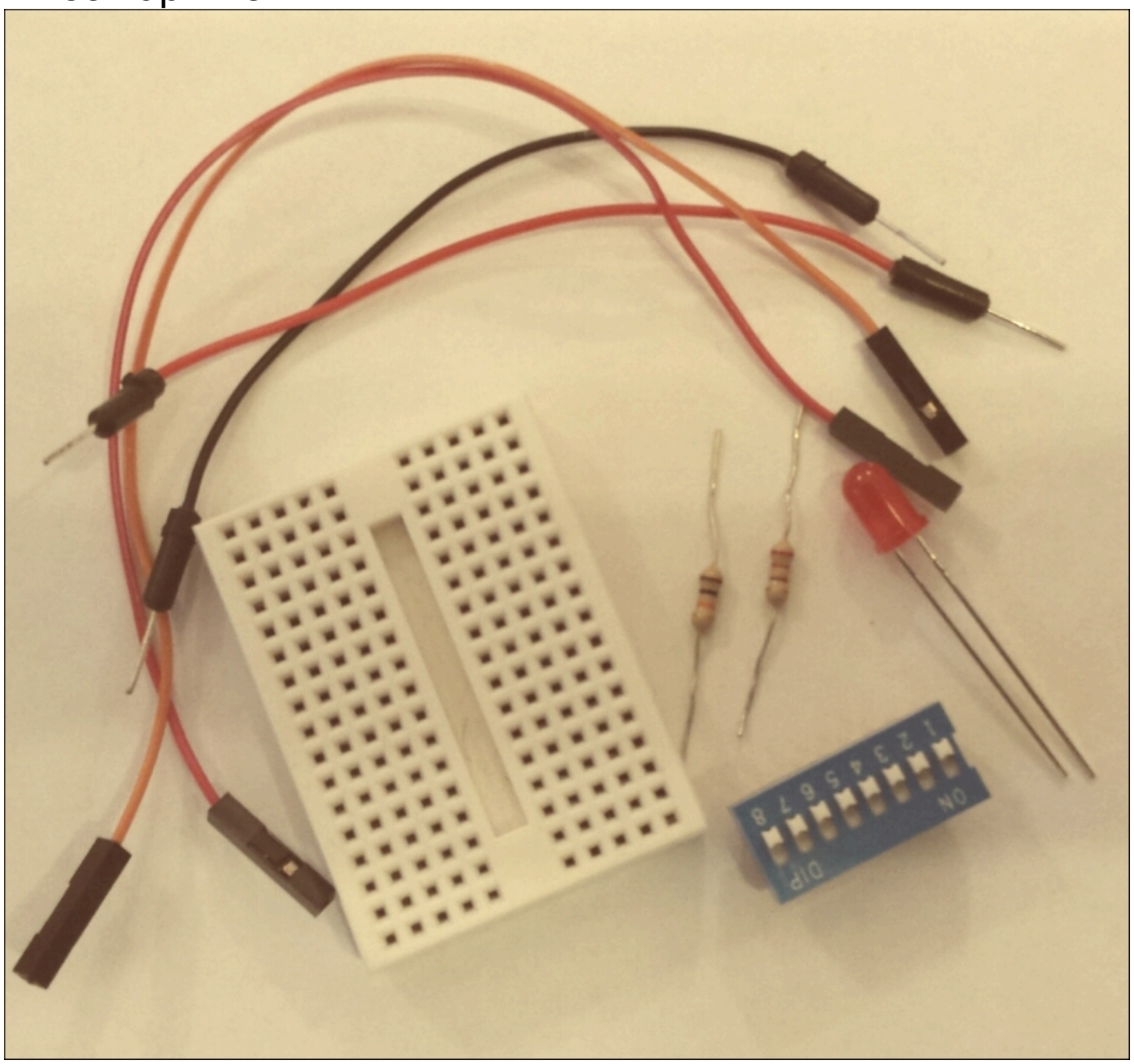
- Examining the GPIO connector and what each of the pins does
- Learning about the I2C and SPI buses that will be used in later chapters
- Connecting an LED and a switch safely to the data pins, and accessing these data pins using simple scripts
- Understanding the USB ports and their limitations

Prerequisites

Along with your Raspberry Pi, you'll need the following parts for the projects in this chapter:

- A breadboard
- An LED
- A 220 ohm resistor (red, red, black)
- A 10K ohm resistor (brown, black, orange)
- A pushbutton or toggle switch

- A hook-up wire:



Our little collection of parts

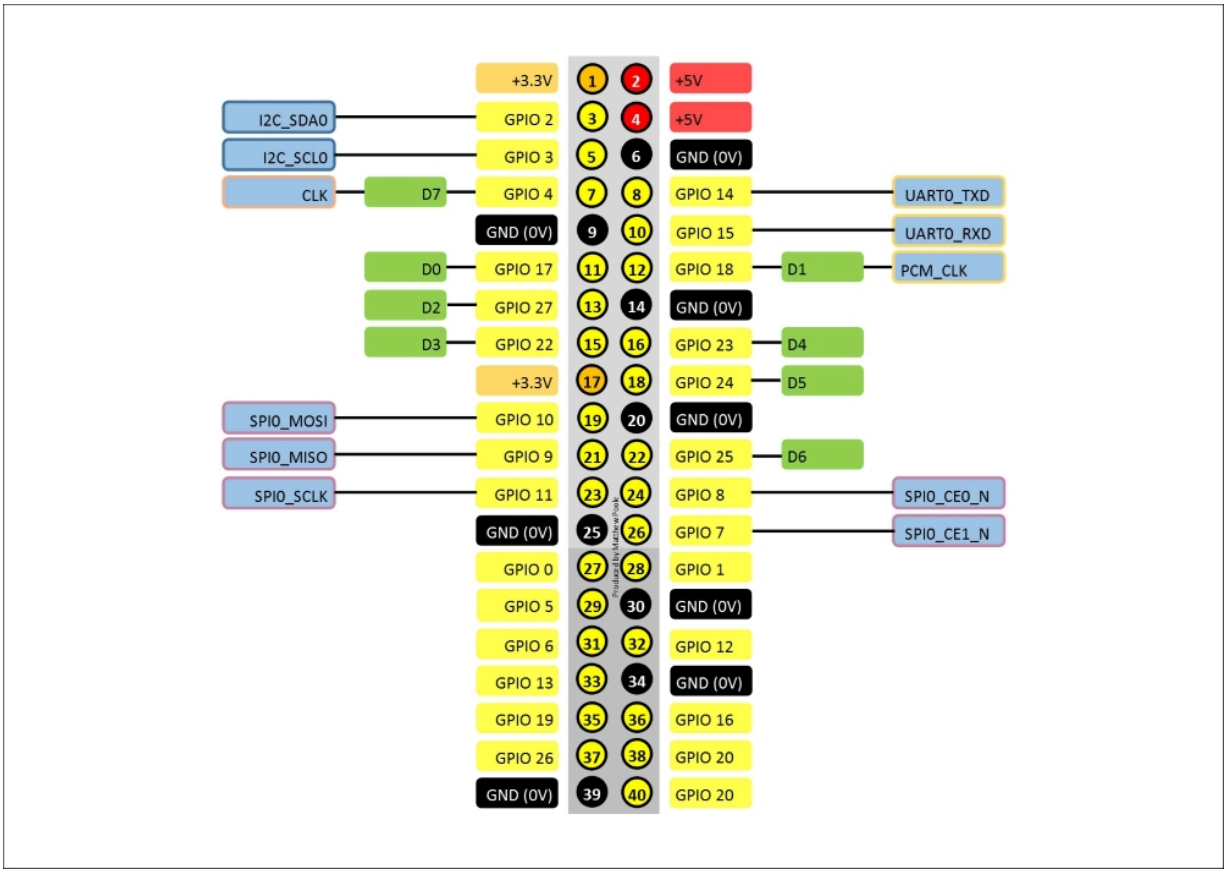
Say hello to the GPIO

The GPIO connector is the large group of pins on the edge of your Raspberry Pi board. On earlier models, there were 26 pins that made up this connector. But, ever since the Model B+, there have been 40 pins, although the first 26 pins are identical to the previous models, and it's these 26 pins we'll be working with. You won't need to worry about the rest of the pins.

Essentially, the GPIO connector provides access to following:

- Power supplies
- Digital I/O pins
- I2C bus
- SPI bus
- UART Serial bus

Some of the pins on the GPIO have more than one purpose, depending on how they are programmed. The following diagram is a reference guide to all of the pins on the GPIO. The GPIO numbers on the yellow labels relate directly to those on the Broadcom chip, and are numbers generally used within the scripts.



Digital I/O pins

The GPIO has 8 digital input/output pins available for use. These can be used to switch things on and off (in output mode), and also to detect when external things are switched on and off (input mode). Each pin can be configured independently for input or output operation, and I have labelled them **D0** to **D7** in the preceding diagram.

Obviously, if we were to use each of these pins to drive or sense an individual device, we would be limited to a maximum of 8 devices that could be connected to our home security system. In many scenarios, this is probably not enough, so in the next chapter we'll learn how to use the GPIO to connect many more things to our Raspberry Pi.

The I2C bus

The **Inter-Integrated Circuit** (**I2C**) bus is a low-speed interface that can connect multiple devices and simple sensors using a 2-wire interface without the need for a separate clock or device select line. Typically, this bus can operate at speeds up to 100kbit/s. We'll be covering this in the next chapter to help us expand our digital I/O and connect more things.

The SPI bus

The **Serial Peripheral Interface** (**SPI**) bus is a synchronous two-way serial connection between a master and a slave device. It can be used to access more complex sensors or drive displays.

The master device provides the synchronization, and each transmission is synchronized by a clock pulse on **SCLK** (GPIO11/pin 23). Data is transmitted on the **MOSI** (master-out-slave-in) and **MISO** (master-in-slave-out) (pins 19 and 21 respectively).

The UART serial bus

The **Universal Asynchronous Receiver and Transmitter** (**UART**) bus is a way to communicate with external devices over a serial data connection, and is a common way for the Raspberry Pi to access data from devices such as GPS modules, which often come with serial connections. It can be a little bit fiddly getting the Pi set up to communicate with UART-connected devices, as it's also tied in with the operating system's serial console.

USB ports

We're probably all familiar with **Universal Serial Bus** (**USB**) ports as we use them to connect all sorts of things to our PCs, such as keyboards, mouses, and hard disks. On the Raspberry Pi, it's just

the same; we can connect keyboards, mouses, and dongles to give us Wi-Fi and Bluetooth connectivity.



Official Raspberry Pi USB Wi-Fi Dongle

On earlier Raspberry Pi models, the amount of current that the ports delivered was pretty low and caused all sorts of problems if too much current was drawn by the connected devices. This was significantly improved from the model B+ onwards, and it's now possible to connect GSM/LTE dongles without any problems.

There are still limitations, however, if you want to connect things such as hard disk drives; these can still draw more current than what can be supplied by the Raspberry Pi USB ports, so it's recommended that a powered USB hub or USB power injector be used when connecting these types of devices to your Pi.

Power connections

The GPIO connector also provides access to the on-board power supplies. The +5V connection (pins 2 and 4) is essentially the +5V input from the external power supply connected to the micro-USB

power port. This can be used to power small external circuits if necessary, although it is recommended that an additional external +5V supply be used if significant current is required.

The +3.3V supply (pins 1 and 17) is the output from the on-board 3.3V regulator and provides a small amount of current up to 50mA. If you need to draw more than 50mA for your external circuits, then you should use an external power supply. I'll show you how to build one later in this book.

Note

The I/O pins on the Raspberry Pi operate at 3.3V levels. Connecting voltages higher than this to the pins could irreversibly damage your Pi. If you follow the instructions in this book, then everything should be fine, but randomly connecting things to your Pi that use lots of power will break it!

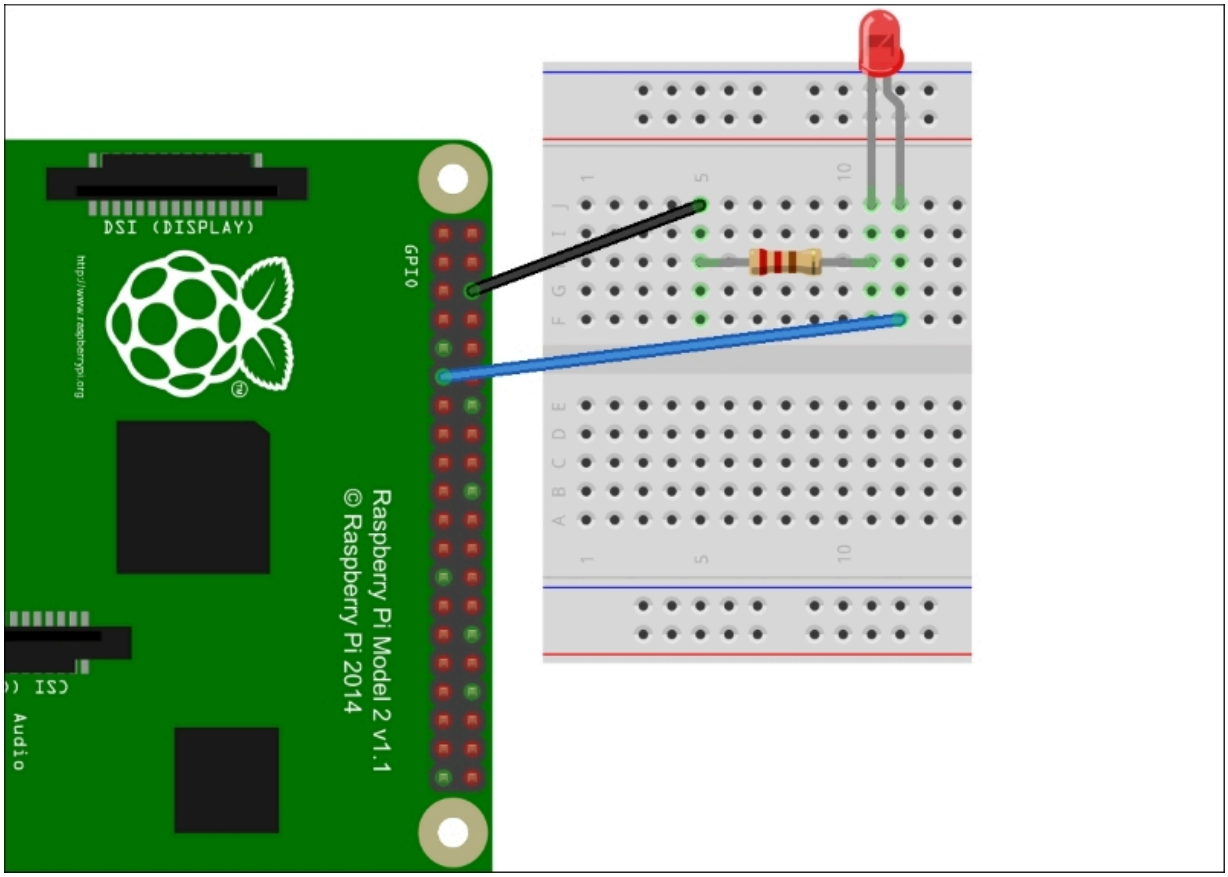
Getting acquainted with the GPIO

Before we embark on connecting lots of things to our Pi board, it might be a good idea to just get acquainted with the GPIO through a couple of simple projects that will help us understand how to interact with the digital I/O pins using shell scripts.

Let there be light

This simple little project shows how to connect a GPIO output to an LED, and switch it on and off using shell commands.

The following diagram shows how to connect up the circuit using a breadboard:



Note

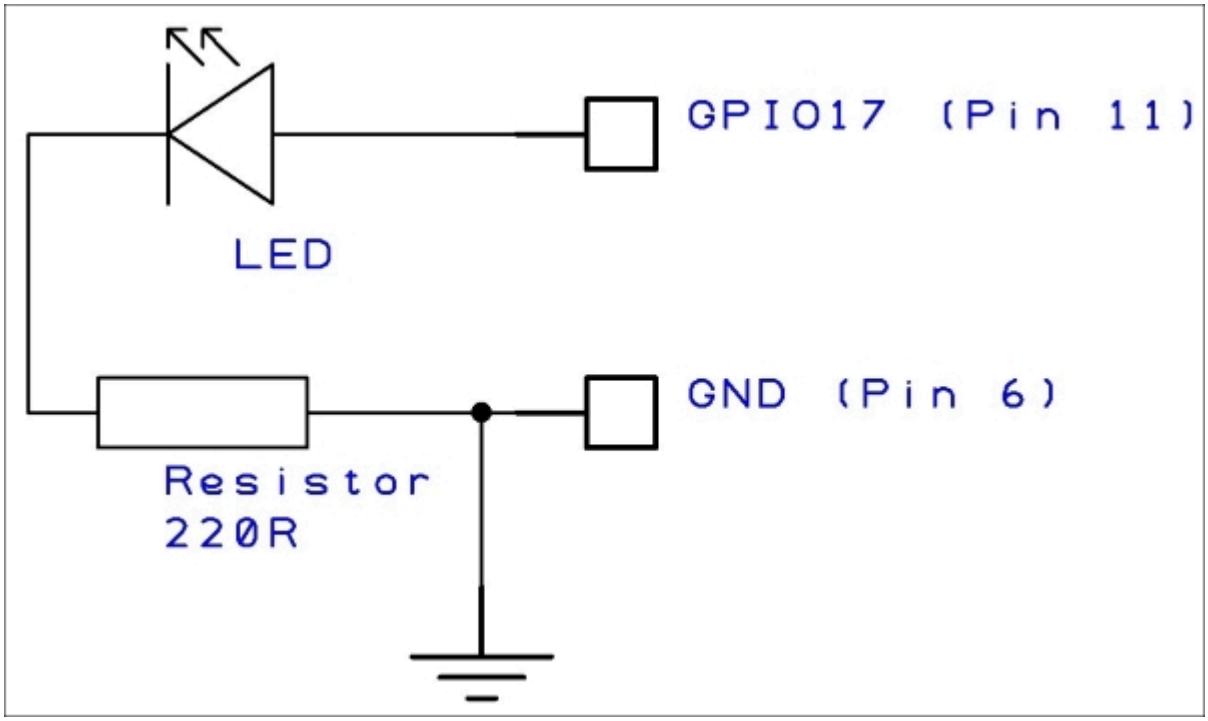
The pretty diagram that you just saw was produced using a free software tool from fritzing, which is an open-source hardware initiative to make electronics accessible as creative material for anyone. Download it from fritzing.org.

The LED anode (the positive side) is connected to the **D0** digital I/O (pin 11 of the connector or GPIO17). When this pin is switched on, it will provide a 3.3V supply to the LED.

The LED is connected to the Ground pin via a 220R resistor on the cathode (negative side). The resistor limits the voltage to the LED and the current through it, otherwise it would burn out, as you can only supply up to about 2V to LEDs. With a current of around 10mA

being drawn by the LED on a 3.3V supply, a 220R resistor works well to protect both it and the GPIO.

Here's the circuit diagram for it:



Note

Calculating LED Resistor Values–

While this book is not really a course on electronics theory, I thought it would be handy to show you how to work out the resistor values for LEDs using Ohms Law, as we'll be covering this again later.

As I mentioned, a typical LED will drop about 2V across it, although this varies according to color and type. This is called the forward voltage of the device or VLED.

The current required by an LED is around 10mA, again depending on its specification. We'll call this current flowing

through the LED, ILED.

Essentially, the voltage across the resistor will be the supply voltage minus the voltage drop across the LED (for example,.2V). So, if we have a 12V supply (VS), the voltage across the resistor will be 10V (VS – VLED).

According to Ohms Law, the resistance R is the voltage across it divided by the current flowing through it: R = V / I. As we require 10mA flowing through it, with a voltage of 10V across it, the resistance required is 10V divided by 0.01A, which is 1,000 ohms or 1K.

In summary, R = (VS-VLED) / ILED.

Now, to turn the LED on and off: the GPIO pins are actually mapped as devices in the Linux file system, so using shell commands is easy, although there are many libraries available out there that allow you to control the GPIO using Python, for example. However, so that you don't have to learn a new language, we're going to do everything using shell commands.

The **D0** pin that we are connected to is actually GPIO17 as far as the Raspberry Pi is concerned (take a look at the previous diagram for reference). The first thing we need to do is create file access to this GPIO pin. We do this with the following command:

```
$ sudo echo 17 > /sys/class/gpio/export
```

We then have to set the pin's direction to out:

```
$ sudo echo out > /sys/class/gpio/gpio17/direction
```

Next we can switch the pin on to turn the LED on:

```
$ sudo echo 1 > /sys/class/gpio/gpio17/value
```

To switch the LED off, we use this command:

```
$ sudo echo 0 > /sys/class/gpio/gpio17/value
```

Once we've finished with a GPIO port we can remove its file access:

```
$ sudo echo 17 > /sys/class/gpio/unexport
```

Getting flashy–

We can put these commands together in a single Bash script to create a flashing LED. To create the flashy script, create a new text file in **nano** or some other text editor. Or, as I usually do (don't forget that I'm quite lazy), create the text file on your laptop, and then copy it to the remote Pi using **WinSCP** (although, read my note in the box that follows if you want to prevent some heartache).

The following is the code listing for `led-flash.sh`:

```
#!/bin/bash
sudo echo 17 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio17/direction
# loop forever
while true
do
  sudo echo 1 > /sys/class/gpio/gpio17/value
  sleep 0.5
  sudo echo 0 > /sys/class/gpio/gpio17/value
  sleep 0.5
done
```

Note

If you use Windows to create your files, remember to save your files with the end-of-line format being Linux (a single `0x0a` or Line Feed character) rather than Windows (`0x0a + 0x0d` or Line Feed + Carriage Return characters), otherwise you might find that your Bash script does not run properly on the Raspberry Pi. Text

editors on Windows, such as the excellent Notepad++, will convert your script line ends for you.

Run the script by calling `led-flash.sh` (assuming that's what you've called it). If you're in the same directory as the script, this can be done by typing the following:

```
$ sudo bash ./led-flash.sh
```

Since this is an endless loop with the LED flashing on and off at half second intervals, you'll need to break out of it by using *CTRL + C* to stop the script.

Don't forget to remove the GPIO pin from file access by using the following command:

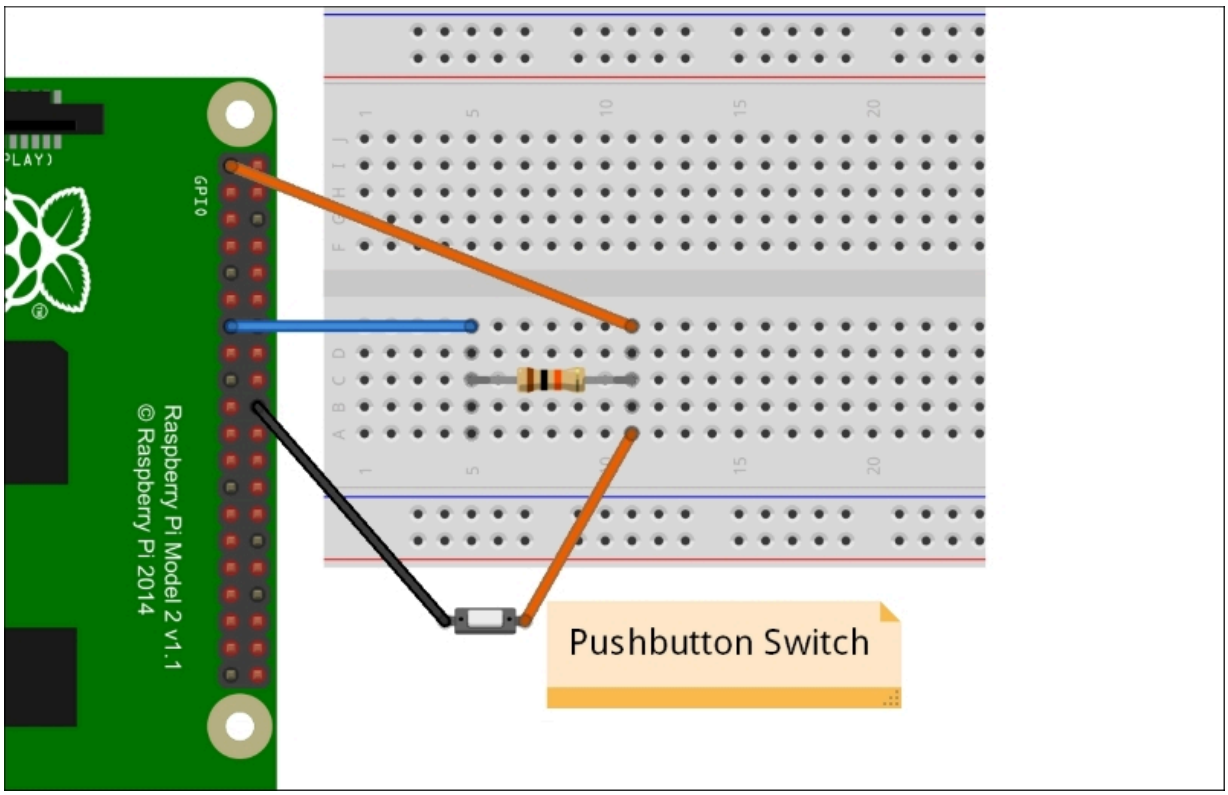
```
$ sudo echo 17 > /sys/class/gpio/unexport
```

Otherwise, you'll see the error, `echo: write error: Device or resource busy`, if you re-run the script, as the first line tries to set GPIO17 for file access again.

Adding a switch

In this project, we'll see how to connect a switch to a GPIO input and write a shell script to read the state of the switch—that is, whether it's switched on or off.

Connect a switch to your Pi's GPIO27 pin, as shown in the following diagram:



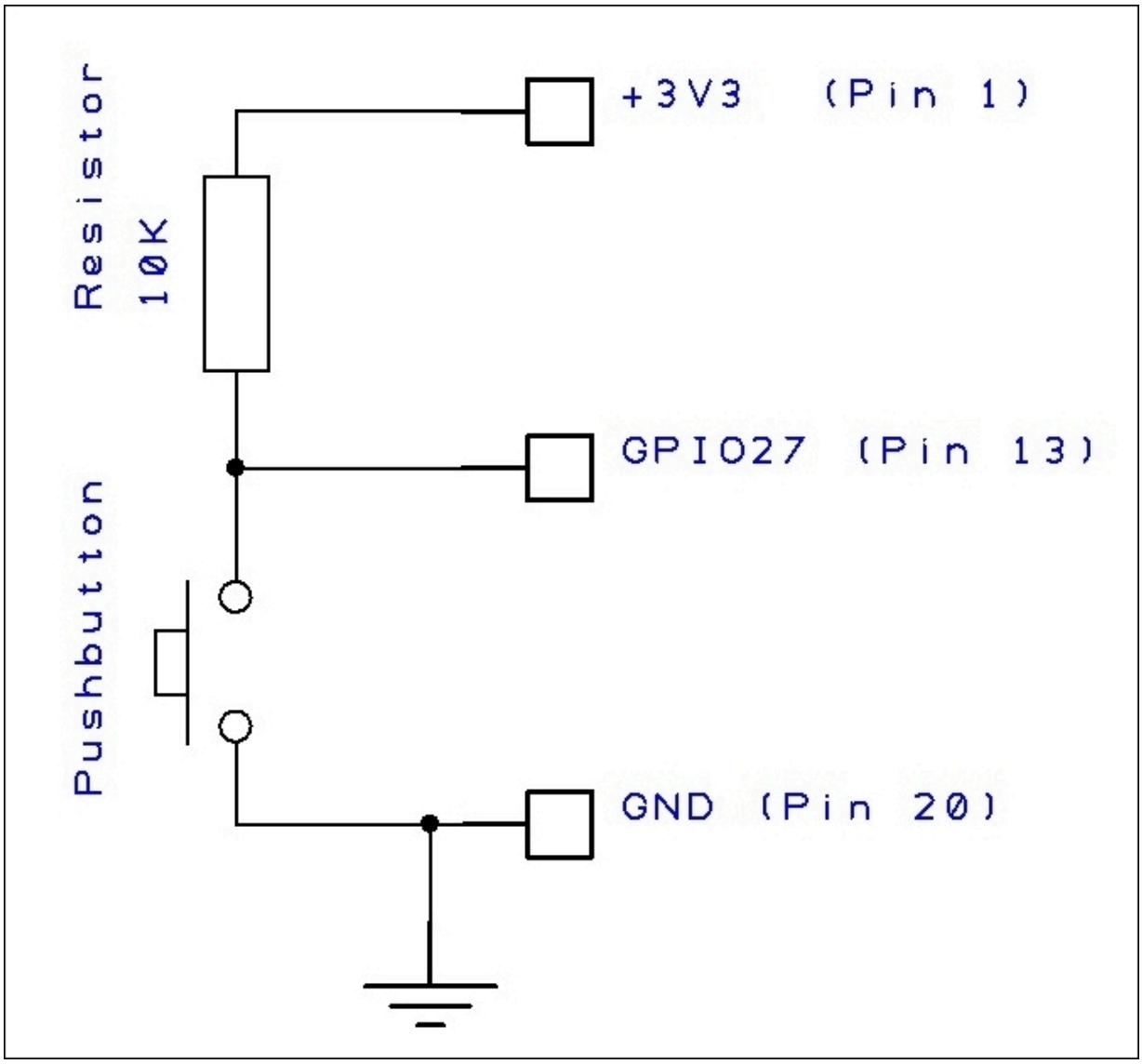
Pulling yourself together

A really important thing to realize about GPIO inputs is that they are in what's called a *floating state*. This means that, as far as the operating system is concerned, it doesn't know what its reference state is unless it is presented with a known voltage.

This is where our resistor comes into play—it pulls up the GPIO pin to a known voltage of 3.3V, which gives it a default state of HIGH (or binary `1`).

When the pushbutton switch is pressed, this takes the GPIO pin to 0V, which is a LOW state (or binary `0`).

Here's the circuit diagram for our GPIO switch:



The detection script

Now that we've connected the switch to our Raspberry Pi, we need to write a little script that will detect when the switch has been pushed.

It's similar to the previous LED script shown, but this time we'll set the GPIO pin as an input and read its logic level.

In this project, we've connected our switch to **D2**, which is **GPIO27** (again, refer to the earlier GPIO pin-out diagram). As before, we

need to create file access for the pin by entering the following command:

```
$ sudo echo 27 > /sys/class/gpio/export
```

And now, set its direction to `in`:

```
$ sudo echo in > /sys/class/gpio/gpio27/direction
```

We're now ready to read its value, and we can do this with the following command:

```
$ sudo cat /sys/class/gpio/gpio17/value
```

You'll notice that it will have returned `1`, or a high state. This is because of the pull-up resistor we were talking about earlier. This means that its default state, when the switch isn't pushed, is high.

When the switch is pushed, the value should be read as `0` or low. If you have more than two hands, you can try this by pushing the button and re-running the command. Or, we can just create a script to poll the switch state.

The code listing for `poll-switch.sh` is as follows:

```
#!/bin/bash
sudo echo 27 > /sys/class/gpio/export
sudo echo in > /sys/class/gpio/gpio27/direction

# loop forever
while true
do
  # read the switch state
  SWITCH=$(sudo cat /sys/class/gpio/gpio27/value)

  if [ $SWITCH == 1 ]; then
    #switch not pushed so wait for a second
    sleep 1
  else
    #switch was pushed
```

```
    echo "You've pushed my button"
  fi
done
```

When you run the script and then push the button, you should see `You've pushed my button` scrolling up the console screen until you stop pressing it.

Don't forget that, once we've finished with the GPIO port, we can remove its file access:

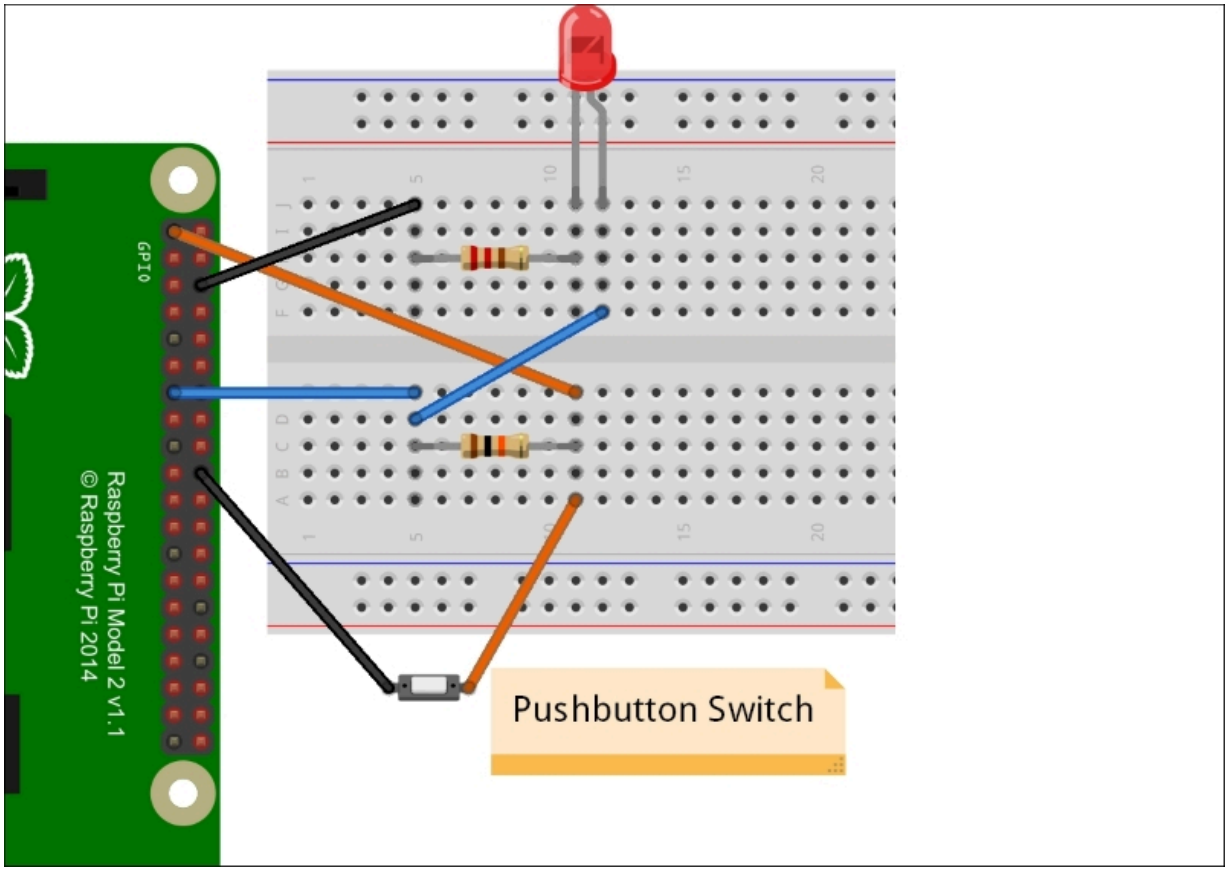
```
$ sudo echo 27 > /sys/class/gpio/unexport
```

We've now seen how to easily read a switch input, and the same circuit and script can be used to read other sensors, such as door contact switches, reed switches, or anything else that has an on and off state.

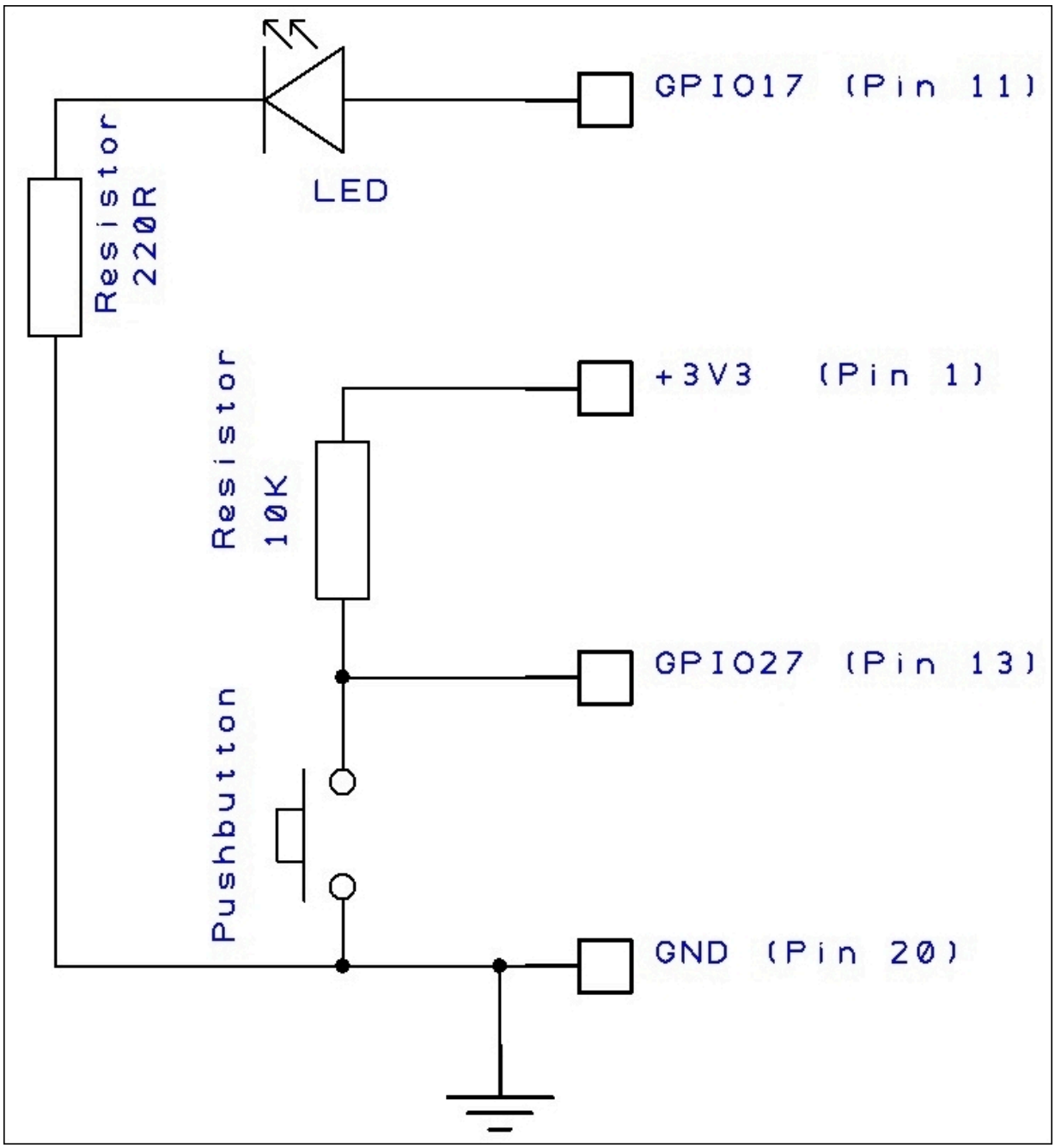
The most elaborate light switch in the world

By combining the two little projects earlier, we can now create a system that will do something useful when the pushbutton switch is pushed—for example, switching on the LED that we also have connected. Granted, we could just connect the LED directly to the switch and a battery, but not only would that be boring, it would defeat the point of what we're trying to do, which is programmatically sensing and controlling things.

Here's the breadboard layout for our elaborate light switch:



And here's the circuit diagram:



The illuminating script

Our full Bash script for our elaborate light switch is demonstrated next. This will loop endlessly, detecting the state of the switch GPIO pin, and will turn on the LED GPIO pin when the switch is pushed.

The code listing for `light-switch.sh` is as follows:

```
#!/bin/bash

#set up the LED GPIO pin
sudo echo 17 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio17/direction

#set up the switch GPIO pin
sudo echo 27 > /sys/class/gpio/export
sudo echo in > /sys/class/gpio/gpio27/direction

# loop forever
while true
do
  # read the switch state
  SWITCH=$(sudo cat /sys/class/gpio/gpio27/value)

  #0=Pushed 1=Not Pushed
  if [ $SWITCH = "1" ]
  then
    #switch not pushed so turn off LED pin
    sudo echo 0 > /sys/class/gpio/gpio17/value
  else
    #switch was pushed so turn on LED pin
    sudo echo 1 > /sys/class/gpio/gpio17/value
  fi
  #short delay
  sleep 0.5
done
```

So, here we are—we have a script that will detect an input state and do something in response; in this case, it will switch on an LED. We're now forming the basis of how we are going to put together our home security system.

Note

Remember, don't connect anything to your Raspberry Pi in place of the LED, such as a buzzer or any other device that consumes lots of current. This is likely to irreversibly render your board dead. We'll look at ways, later on in this book, to control devices with higher power requirements.

Summary

In this chapter, we introduced various ways to connect your Raspberry Pi to the outside world by looking at the various interfaces available on the GPIO. We've understood how to connect things to the digital pins on your Raspberry Pi's GPIO connector, and control and read them using simple Bash scripts. In particular, we've safely and properly connected a switch to a digital input pin, which will form the foundation for our home security detection circuits.

In the next chapter, we'll look at ways to expand the number of things we can connect to our Raspberry Pi, overcoming the limitation of having just the 8 digital pins available to us on the GPIO by tapping into other interfaces on the GPIO and building our own input/output expansion board.

Chapter 3. Extending Your Pi to Connect More Things

We're now going to look at ways to expand the number of things we can connect to our Raspberry Pi, overcoming the limitation of having just the 8 digital pins available. We're going to do this by building our own expansion board to give us what could in theory be an unlimited number of digital inputs and outputs.

We're also going to overcome the limitations of the +3.3V power available to us by building our own +3.3V power supply that taps off the Raspberry Pi's +5V supply.

In this chapter, we will cover the following:

- Looking at the I^2C bus in detail
- Learning about serial-to-parallel and parallel-to-serial conversions
- Building a +3.3V power supply
- Building an I2C-based port expander to give us more inputs and outputs
- Looking at alternative ready-made expansion boards

Prerequisites

Along with your Raspberry Pi, you'll need the following parts for the projects in this chapter:

- A copper strip board (or Veroboard–)
- An LD1117V33 voltage regulator
- A 2 x 100nF, 16V ceramic capacitor
- A 10uF, 16V electrolytic capacitor
- A 1 x MCP23017 16-bit port expander IC
- A 4 x 10K-ohm resistor

- A hook-up wire

The I2C bus

In the previous chapter, we briefly touched on the I2C bus (or Inter-Integrated Circuit bus), which is a way to connect multiple devices together using just two wires. I2C was invented in the early 1980s by Philips as a way to link computer peripherals together using a common protocol. You can think of I2C as a kind of early form of USB.

I2C typically operates at relatively low speeds of up to 100kbit/s, compared to much faster interfaces such as Ethernet, which typically operates at up to 1Gbit/s, or USB, which can operate at up to 480Mbit/s. However, this is fast enough to connect basic sensors, display devices, or other peripherals such as real-time clocks—in fact; there are faster versions of the protocol that some devices will support.

Just 2 wires

I2C is a bi-directional serial communication protocol that operates over two wires:

- The **Serial Data Line** (**SDA**) wire transmits the data to and from the master device. Referring back to the GPIO reference in Chapter 2, *Connecting Things to Your Pi with GPIO*, this is pin 3 of the GPIO connector.
- The **Serial Clock Line** (**SCL**) wire handles all timing and flow control for the data on the bus. This is pin 5 of the GPIO connector.

You'll remember that we spoke about pull-up resistors, in the previous chapter, which ensure that the GPIO digital inputs are pulled to a known state. Well, this is required for the two lines on the I2C bus, and by default the lines should be pulled high with resistors. However, on the Raspberry Pi, this has already been done for us, so we don't need to worry about it in our case.

What's your address?

So, if we can use just two wires to communicate with multiple devices, how does our Raspberry Pi know which device to talk to? This is where the I2C protocol comes into its own. Each device connected to the bus has its own unique ID, or address, made up of 7-bits or 10-bits. Some devices will allow you to set the address to ensure that it's unique within your system, but other devices have their addresses hardcoded by the manufacturer.

The two addressing methods (7- and 10-bit) are interoperable and you can have devices on the same bus that use either method, since the Raspberry Pi itself supports both methods. So, with a 10-bit addressing scheme, you can see that we can connect a lot of things to our Raspberry Pi using the I2C bus, as compared to the limited number of digital pins on the GPIO!

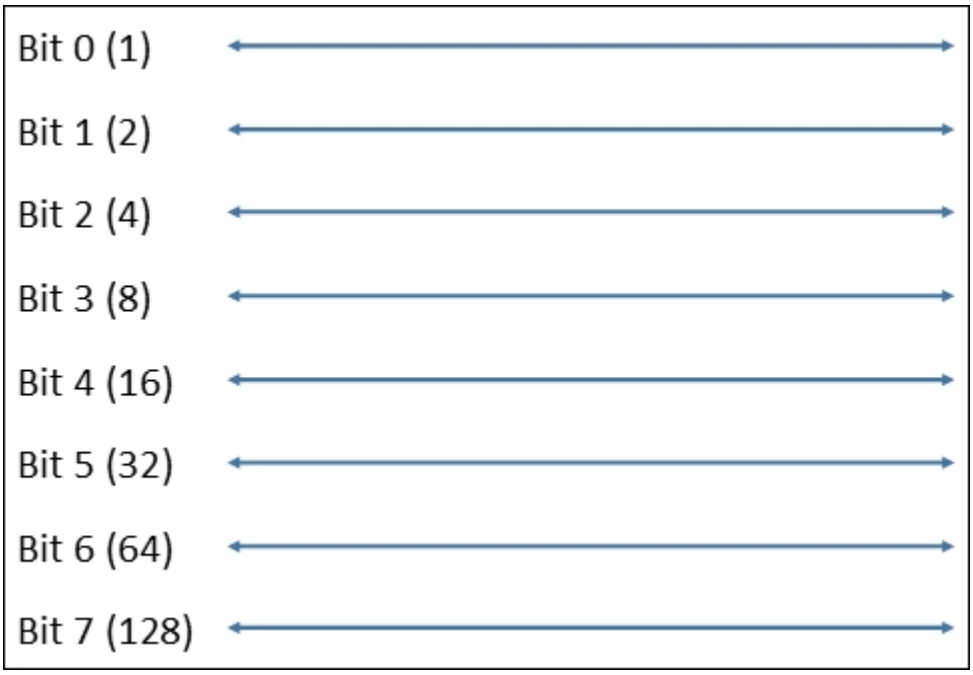
There is a parallel universe

Data is normally transmitted in serial mode or parallel mode, depending on things such as the required data speed, cable distance, and functionality. Most data communication *between* systems is transmitted in serial mode over a couple of wires, such as the I2C bus mentioned earlier, but this also includes things such as the Ethernet, RS232/422, and USB.

Within a computer system, data is transmitted in parallel mode using *buses* whose width matches the word size of the digital system communicating between chips. In parallel mode, all bits of the data word are transmitted simultaneously over their respective data lines within the bus, rather than as sequential bits along a single line.

The digital I/O pins we've been talking about (including the ones on the Raspberry Pi's GPIO connector) are usually grouped together as a parallel bus. On our system, we'll be using parallel buses (groups

of digital I/O pins) that are 8-bits wide. That is, the bus has 8 wires that can be set or read using 8-bit binary values (our word size).



A representation of an 8-bit data bus

So, in the preceding diagram we have the 8 digital I/O wires on our bus. If we wanted to make the bits (or wires) 0, 1, and 4 *high* or *on*, with the rest *low* or *off*, then we'd address the bus and set it to the following values:

- In binary, this would be 00010011
- In hex, this would be *0x13*
- In decimal, this would be 19 (represented by *16+2+1*)

So, in other words, to switch on data lines 0, 1, and 4, we send the byte value, 19, to the bus's address.

Serial-to-parallel conversion

So, now that we know what numbers to send to our bus to switch on or switch off certain digital outputs, or read certain digital inputs, how do we do this using our I2C bus, which is a serial interface?

Fortunately, there are many **integrated circuits** (**ICs**) available that allow us to do this simply and easily. These ICs are called **shift registers** and perform **serial-to-parallel conversions**, taking the data from the serial I2C bus and converting the incoming bits to a parallel representation by setting each of the parallel bus outputs.

When reading the parallel bus data lines as inputs, the reverse happens, converting the bits into a serial form on the I2C bus; this is known as **parallel-to-serial** conversion.

This is quite a simplistic overview and there are many resources available that explain these operations; we'll see this in action later in the chapter, but first–

Give me power

You'll remember from the previous chapter that most things to do with the GPIO operate on a +3.3V level, rather than the +5V level that is often associated with digital circuits. This is the same with our I2C-based shift registers—they need to operate on +3.3V levels as well, in order to work with the Raspberry Pi.

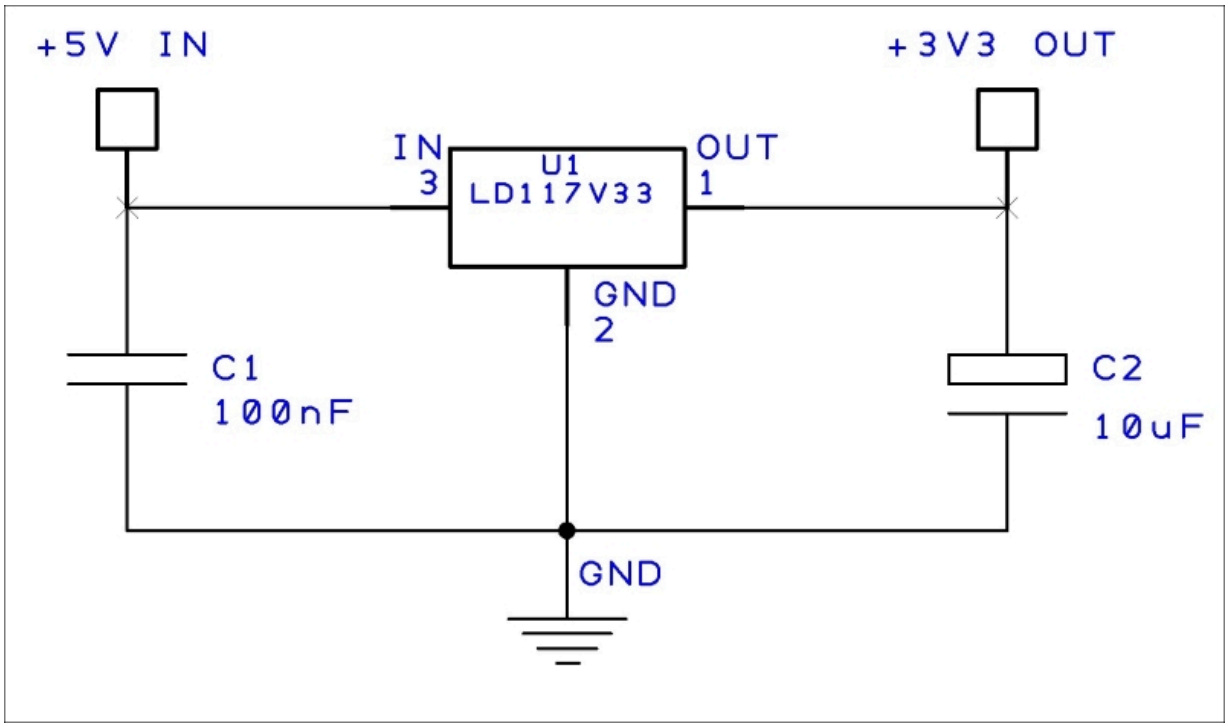
You'll also recall, however, that there's not much +3.3V juice available directly from the Raspberry Pi—in fact, just 50mA. This is really not enough for our interface. So, before we go any further, we're going to build our own +3.3V power supply, which is sufficient for our system.

For our power supply, we're going to use a basic 3.3V **voltage regulator** (type **LD1117V33**) that will take our slightly more plentiful +5V supply from the Raspberry Pi and regulate it to a nice smooth +3.3V supply. We should be able to draw a few hundred milliamps from this supply—enough for the I/O circuitry on our security system.

The parts required for our power supply are as follows:

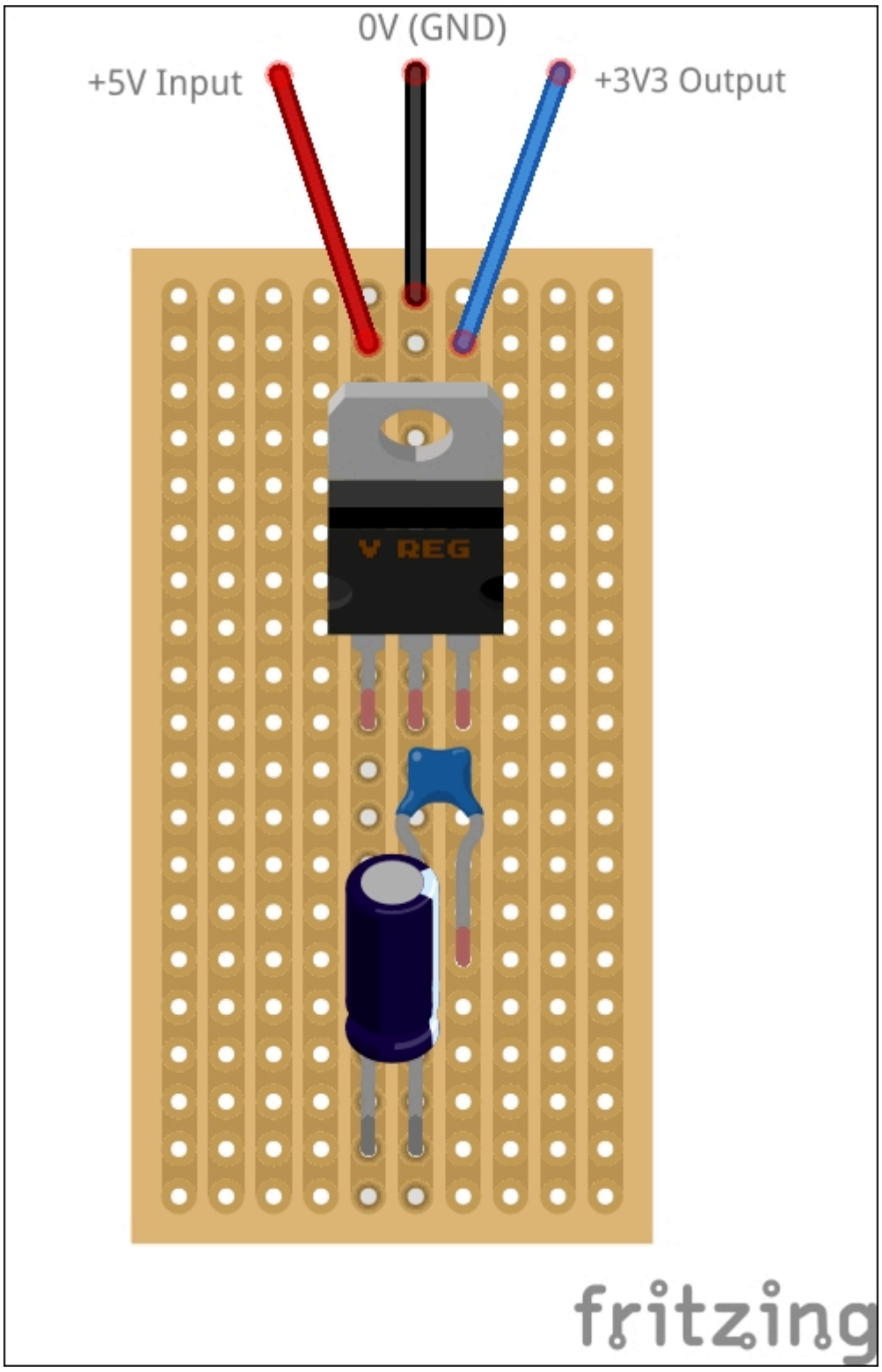
- A LD1117V33 voltage regulator
- A 100nF, 16V ceramic capacitor
- A 10uF, 16V electrolytic capacitor

Here's the circuit diagram for our +3.3V power supply:



As with all our components, the LD1117V33 regulator is widely available from many electronic component suppliers.

Our power supply can be easily built on a small piece of strip board like this:



Note

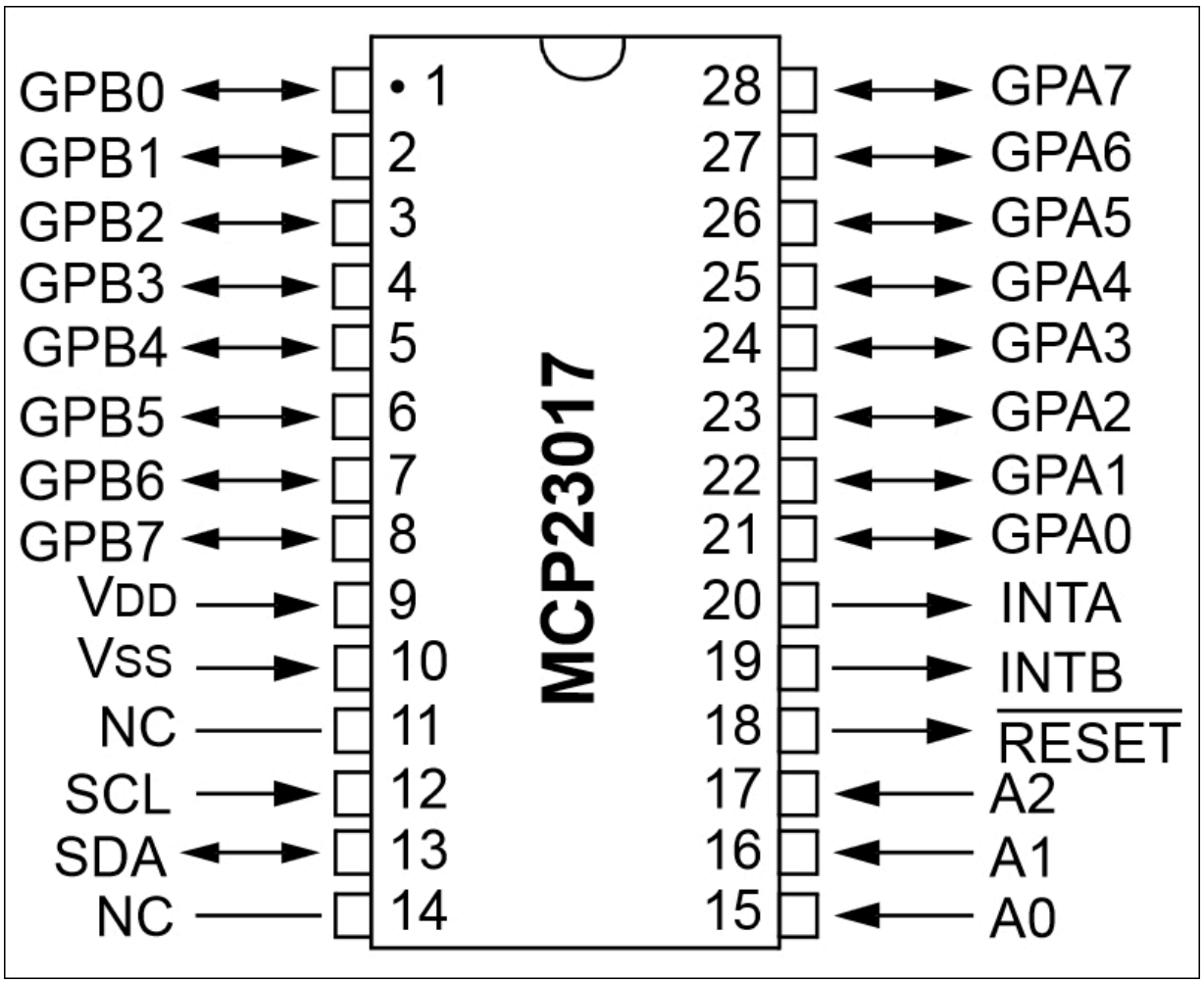
The strip board is shown from the top in the preceding layout. That is, the copper tracks are on the underside of the board and the components are inserted from the plain top-side and soldered to the strips underneath. In this layout, it's not necessary to cut any of the tracks on the strip board.

Building an I2C expander

Right, now that we've worked out what we need to do to give us more digital I/O pins, and built our power supply for it, we can build our expansion port.

To do this, we're going to use a chip designed exactly for the job: the **MCP23017**, manufactured by Microchip and widely available from electronic suppliers.

The MCP23017 is an integrated circuit that connects directly to the I2C bus (the SDA and SCL pins we talked about earlier) and gives us 16 bi-directional input and output pins. If required, we can connect up to 8 of these chips to the same bus, giving us up to 128 inputs and outputs (yes, I know that I said "virtually unlimited" previously, but I'll explain later).



An MCP23017 integrated circuit pinout

Note

The full datasheet for the MCP23017 is available on Microchip's site, which can be found at www.microchip.com/MCP23017.

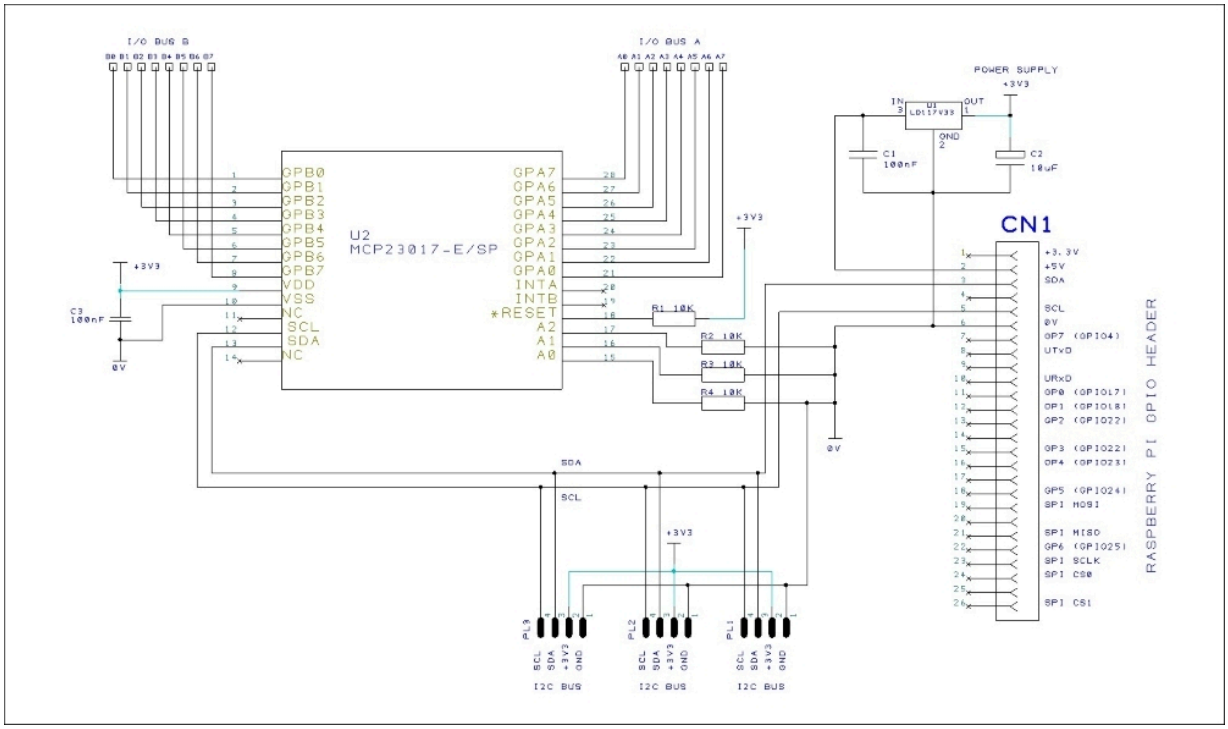
The I2C port expander circuit

The basic parts you will need to build your port expander are as follows:

- A 1 x MCP23017 16-bit port expander IC

- A 4 x 10K-ohm resistor
- A 1 x 100nF, 16V ceramic capacitor
- A copper strip board (or Veroboard–)
- A hook-up wire

Here's the circuit diagram for our I2C port expander circuit. It looks complicated, but actually most of the lines are for connections to the outside world:



Let's walk through the circuit

On the right-hand side, the connector, CN1, is our Raspberry Pi GPIO connector—note that we're only using four of the pins:

- The +5V Output (Pin 2)
- The I2C SDA (Pin 3)
- The I2C SCL (Pin 5)
- The 0V/GND (Pin 6)

You'll see my friend, the +3.3V regulator (U1, C1, and C2), discussed earlier. This takes the +5V output from the Raspberry Pi and gives us our +3.3V for use by the rest of the circuit.

The main component is U2—our **MCP23017 port expander** chip. Pins 9 and 10 on the chip are connected to the +3.3V supply and the GND, respectively, and C3 is used as a discoupling capacitor close to the chip to reduce any noise on the power supply.

The MCP23017 can be used as a 16-bit expander, or as 2 x 8-bit expanders. In our circuit, we have split the device to give us 2 x 8-bit busses: I/O Bus A and I/O Bus B. Each pin on the busses can be programmed to work as an input or output.

Tip

Connecting things to the input/output pins

The input and output pins on our busses can't usually be connected to things directly—they provide limited current and need to be interfaced correctly to things such as buzzers and lights; they must also be protected against damaging input signals. In the next chapter, we'll learn how to connect safely to our I/O ports.

The I2C SDA/SCL lines from the Raspberry Pi are connected to pins 12 and 13 of the chip. You'll see that there are also additional I2C outputs (PL1 to PL3) to illustrate that we can connect other devices to the I2C bus, such as another MCP23017 chip to give us a further 16 digital I/Os.

Resistor R1 is used to hold the RESET pin (18) high. By bringing this pin low, you can reset the chip.

Resistors R2 to R4 are used to hold the address pins A0 to A2 (pins 15-17) low.

Highs and lows

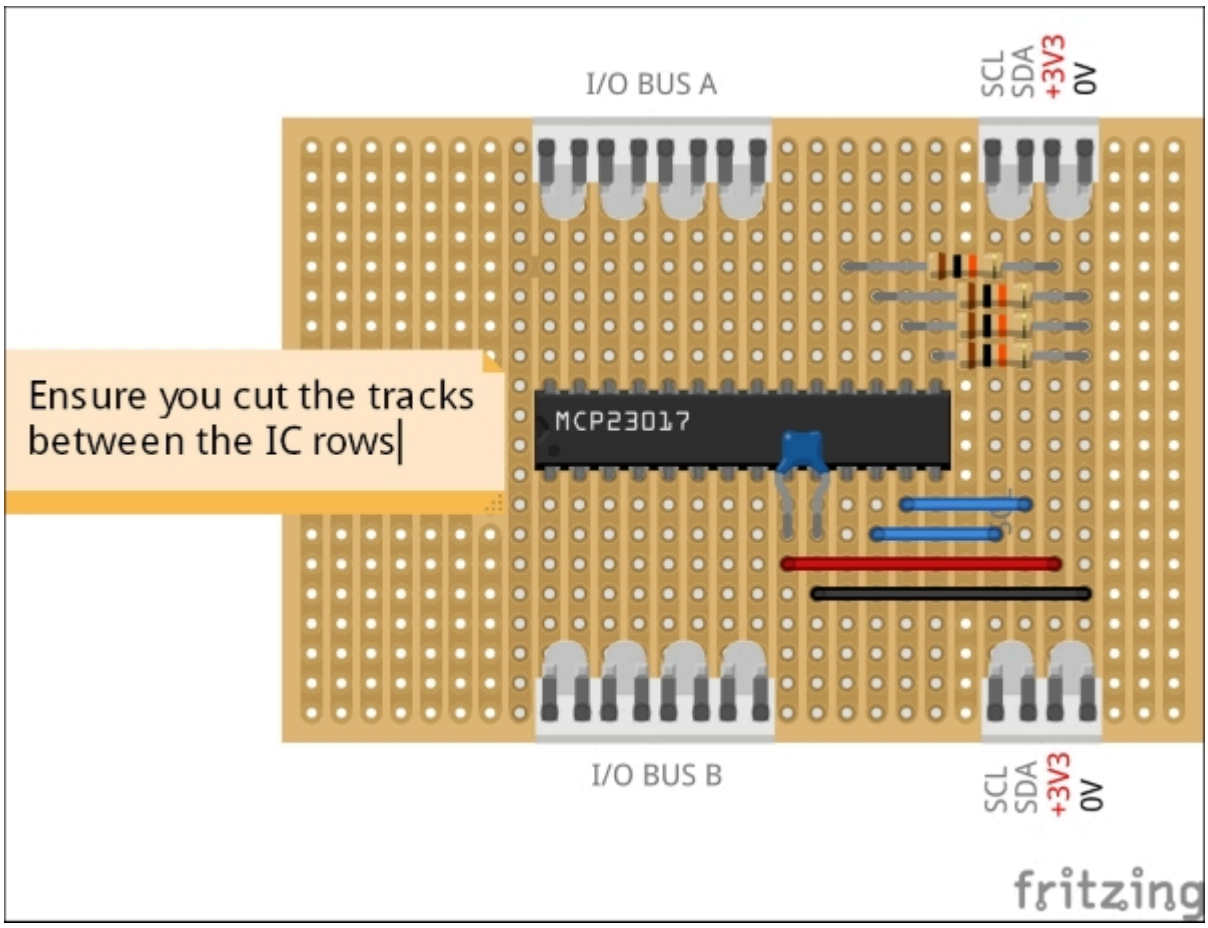
When we use the terms *high* and *low* in respect to digital pins or inputs, we are simply describing whether the logic level of the pin is at a binary 1 or 0, respectively. Digital pins don't like to be left *floating*—whereby they are neither high nor low—as this can cause unpredictable operations. Therefore, we always make sure they are held at a determined logic level. In general, connecting the pin to 0V (or ground) ensures that it's held at logic level 0, and connecting to the positive supply (e.g. 3.3V) ensures that it's held at logic level 1.

Remember I mentioned earlier that you can connect a large number of devices to the I2C bus in order to give us a virtually unlimited number of I/O pins? Well, actually in many cases, this is not strictly true. This is because of the addressing scheme for I2C devices, which makes all devices identifiable when they are all connected to the same two wires (their unique address). The address of each device is agreed upon in advance by manufacturers to make sure that everyone's devices will work together on the same bus without creating conflicts. As such, the address is pre-programmed into the device.

The MCP23017 has been given its unique base address, but can be modified by changing the address pins A0-A2 high or low; thus, in effect, it can be configured to be one of 8 addresses. This is why you can only have a maximum of 8 of these chips on the same I2C bus, giving us a theoretical maximum of 128 I/O pins (that is, 16 I/Os x 8 chips).

Building your expansion board

This circuit can easily be built on a small piece of stripboard. The following image shows an example of the layout, which looks a bit simpler than the circuit diagram. In the next chapter, we'll learn how to connect up our board and program it so we can check that it works.



Tip

When using stripboard, make sure that you cut the tracks between the two rows of pins on the MCP23017 so that they aren't shorted together. You can buy track cutters, which make this task easy, from many electronic suppliers. Again, on the preceding layout, the copper strips are underneath the board with the components on the plain side.

You might want to add the +3.3V power supply circuit to the same piece of stripboard too, to keep everything contained together.

Note

In the next chapter, we will learn how to program the device so that we can use it in our home security system.

Using ready-made expansion boards

While it's much more satisfying to build your own stuff, you might want to look at buying some readily available expansion boards for your home security system if you're not yet confident with your soldering iron, or if you just simply don't have the time.

Following are some ready-made expansion boards that you can obtain; they should work as part of our home security system with a bit of modification to our scripts to support the libraries that are required by the hardware.

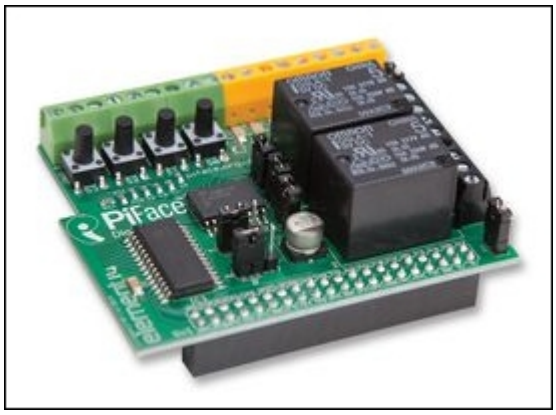
Hobbytronics MCP23017 expander port kit

This kit is almost identical to our own circuit in the previous section of this chapter. The kit comes with an MCP23017, a PCB, and various connectors. The boards are designed to be daisy-chained together so that you can have multiple expanders to give you more input/output ports. Note that this kit is not pre-built and requires soldering, but I thought I'd include it because it's the board that I use to build such systems when prototyping. You can get it directly from Hobbytronics at http://bit.ly/mcp23017.

PiFace Digital I/O expansion board

The **PiFace Digital I/O expansion board** is a pre-built version of our board, but it uses the **MSP23S17** chip variant that operates over the **SPI bus** instead of the I2C bus. The board is designed with 8 inputs and 8 outputs, as well as several additional pieces of hardware including a couple of relays, some LEDs, and some switches. Note that the code in this book for our system will need to be modified to

work with this board, since it uses a different interface and different libraries. It's available from Farnell element14 at http://bit.ly/2434230.



The PiFace Digital I/O Expansion Board

Gertboard

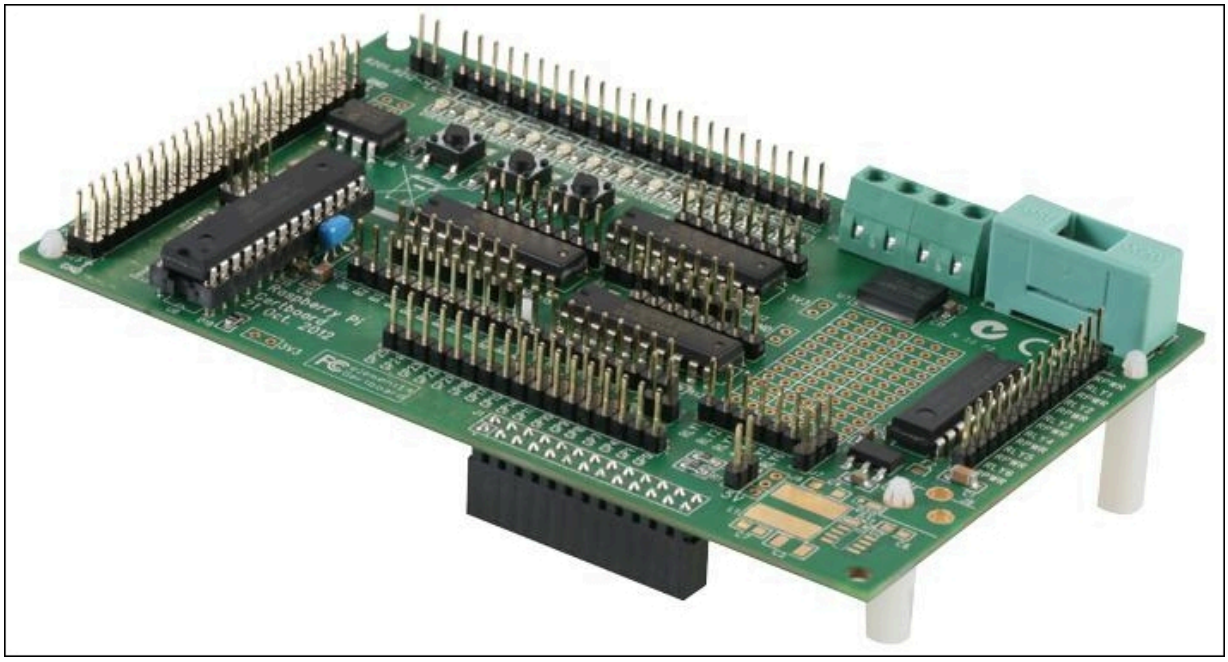
The **Gertboard** is a Raspberry Pi add-on board designed by Gert van Loo—one of the hardware engineers involved in the original design of the Raspberry Pi.

It's a very capable and reasonably-priced board that comes fully assembled and features 12 buffered input/output lines, open collector drivers for switching on devices that need a fair bit of current (such as sounders and lights), plus a digital-to-analog converter.

You can only connect one of these boards to your Raspberry Pi, so if you need more I/O lines you'll need to use something else as well. But it's a great board to experiment with. Interestingly, it features an **ATmega microcontroller**, which is the same as the one that the Arduino uses, and you can, in fact, use the **Arduino** development environment for the device.

Once again, the code in this book for our system will need to be modified to work with this board.

The Gertboard is available from Farnell element14 at http://bit.ly/2250034.



Assembled Gertboard

Summary

We've now looked at the I2C bus in detail, and learned how to build an expansion port using this interface so that we can connect many more things to our Raspberry Pi, rather than being restricted to just the 8 digital I/O pins offered by the Raspberry Pi's GPIO port. In addition to that, we explored other ready-made boards that can be used to connect lots of things to our Raspberry Pi. We have also built a power supply that will give us more +3.3V power than we can obtain from the Raspberry Pi directly.

In the next chapter, we'll start to actually connect things to our home security system, such as magnetic sensors and other types of contact devices, and learn how to program our I2C expansion port using Bash scripts so that we can read the state of our sensors and switch on warning LEDs. We'll also start developing the control scripts for our system, which will allow us to arm and disarm the system and add delay timers.

Chapter 4. Adding a Magnetic Contact Sensor

Now that we have built our port expander hardware, we need to learn how to program it so that our Raspberry Pi can detect the things that we connect to it as part of our home security system. We will begin by connecting switches to our system in the form of magnetic sensors—the most common component used in home security systems to detect intrusions through doors and windows.

In this chapter we will cover the following topics:

- Learning about reed switches and how they work as door sensors
- Enabling and setting up the I2C bus on the Raspberry Pi
- Connecting our sensor to an input on our port expander
- Learning how to access our I2C port expander from a Bash script
- Writing a script that will detect the state of our door sensor
- Looking at other types of contact sensors that can be connected and programmed in the same way

Prerequisites

You'll need the following parts for the exercises in this chapter:

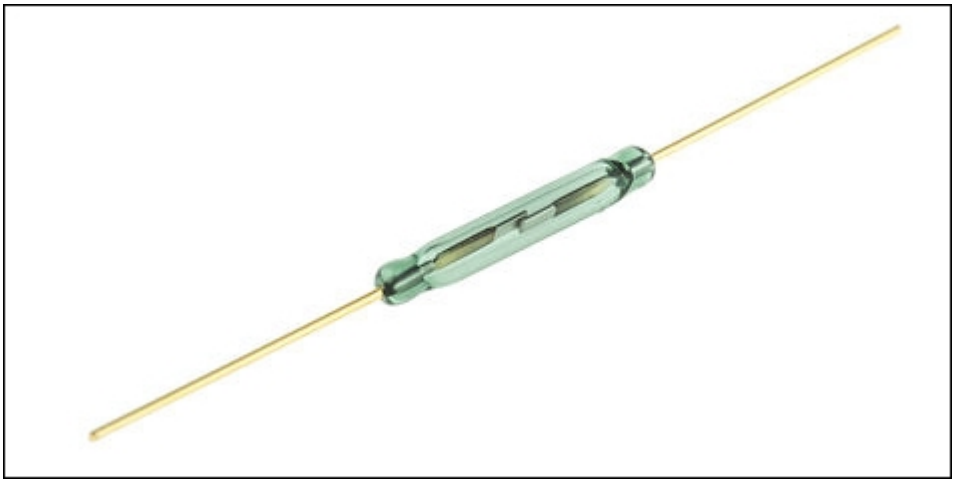
- Our Raspberry Pi and Port Expander board
- 8 x 10K ohm resistors
- A magnetic door sensor and magnet
- A hook-up wire
- A 4-core alarm wire

The working of magnetic contact sensors

A **reed switch** is essentially what makes up our **magnetic contact sensor**. A reed switch comprises two metal contacts made of magnetic material (called reeds) placed inside a glass envelope. When the contacts touch, the switch is on, and when they spring apart, the switch is off and the circuit is broken. The way to control these contacts is by means of a **magnetic field** that makes or breaks the circuit when it is near to the switch.

A normally open (NO) type of reed switch is normally switched off until a magnet comes close to the switch, which then pulls the contacts together.

A normally closed (NC) variety works the other way with the switch being normally on until the magnet comes close to the switch, pulling the two contacts apart.

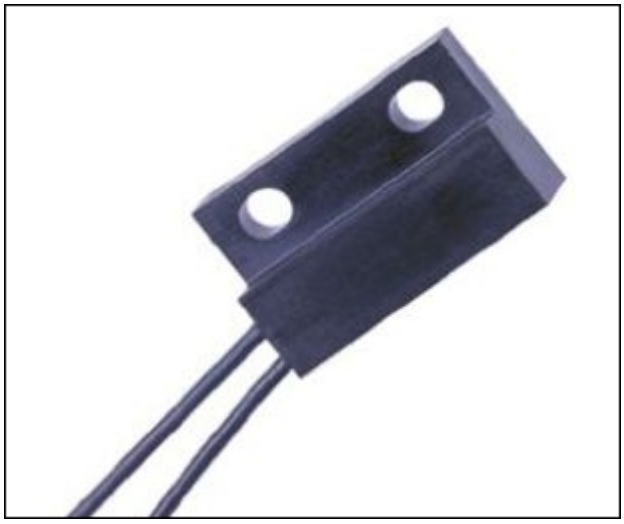


A typical type of reed switch

You can now see how a magnetic reed switch can be a useful sensor in security applications, and in particular for our home security system, to detect when **doors** and **windows** are opened

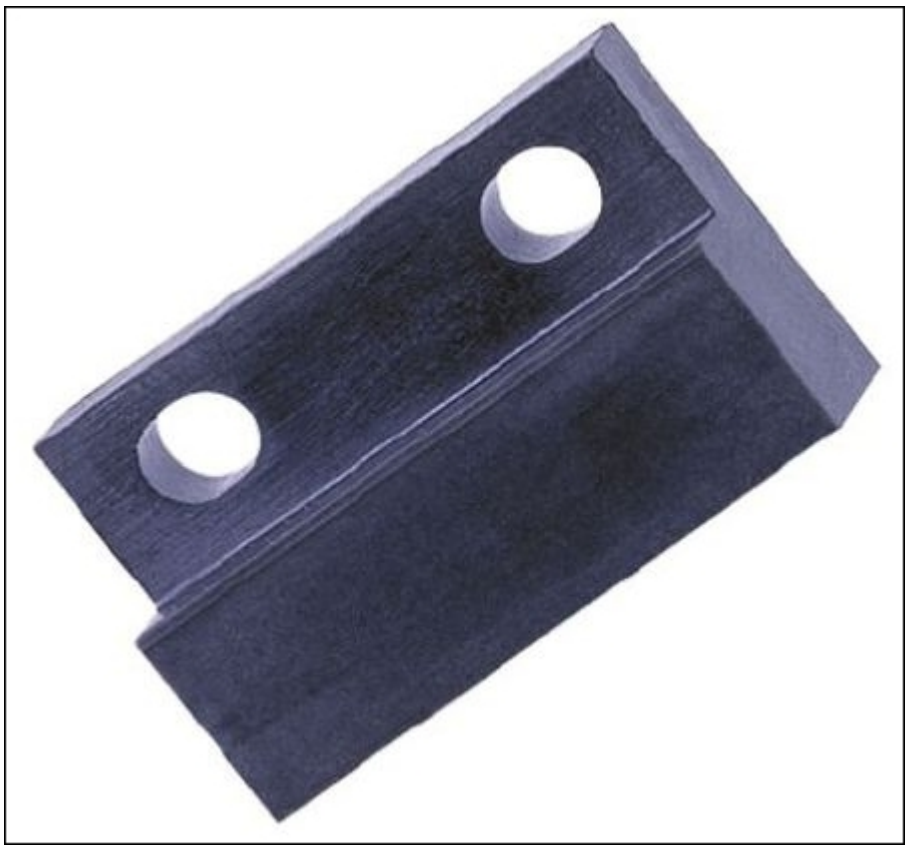
and closed. We simply put a reed switch on the door frame and connect it to our security system, with the magnet placed opposite the switch on the actual door. When the door opens and closes, it makes or breaks the contacts in our reed switch.

Reed switches and their magnets, which are designed for security systems, usually come enclosed in little plastic housings, making them easy to screw onto the door and frame.



A door-frame-mounted magnetic sensor containing a reed switch (Type: Cherry MP201801)

The magnetic sensor is mounted on the door frame (obviously, so it can connect to the alarm circuit wires), while the respective magnet will be attached to the door, close enough to the edge such that the sensor contacts connect (or break, depending on the type) when the magnet is directly opposite it.



A respective door-mounted magnetic actuator (Type: Cherry AS201801)

Setting up the I2C port expander

Now that we have built our port expander, we need to get it ready to connect our sensors to. First, we need to install the tools on the Raspberry Pi to allow us to use the I2C bus and program devices connected to it, including the MCP23017 chip that makes up our port expander.

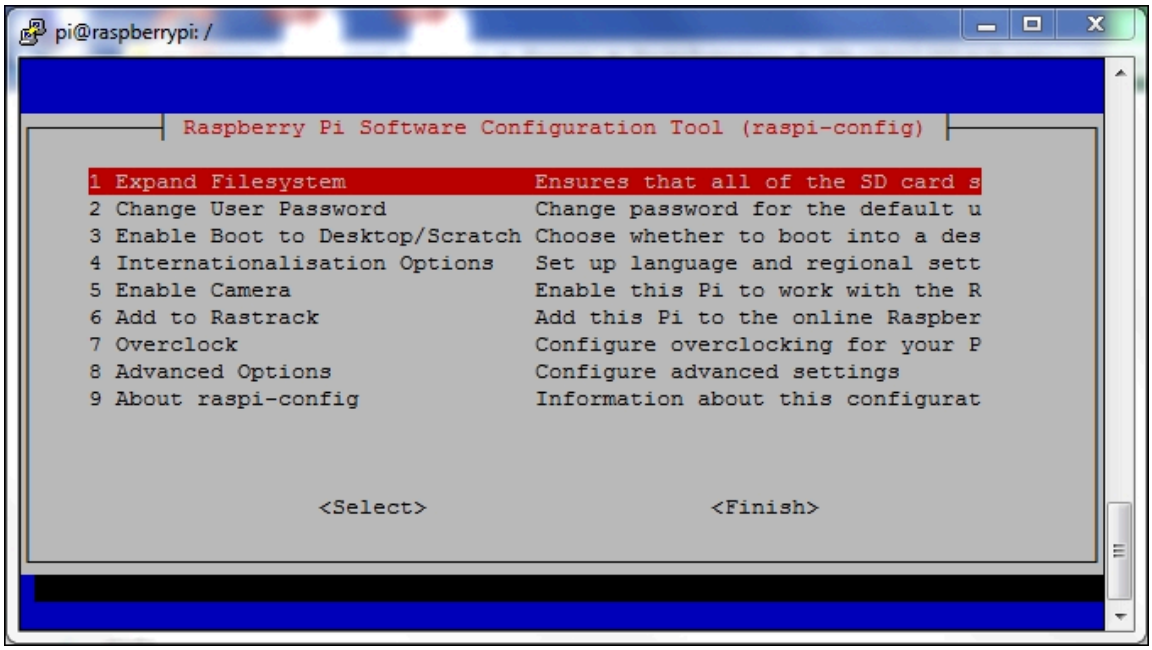
> **Note**
>
> Don't connect your port expander to the Raspberry Pi until after you've set up the I2C bus on your system.

Enabling the I2C Bus

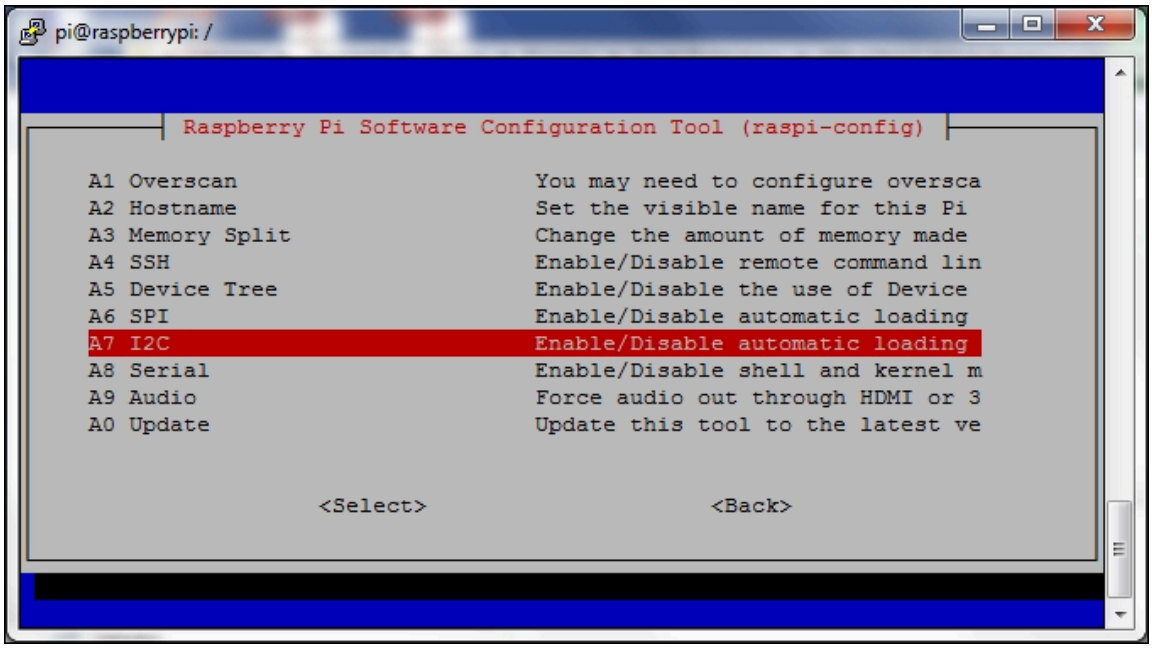
It's highly likely that the module for using the I2C bus hasn't been loaded by default. Fortunately, doing this is fairly straightforward and can be done using the Raspberry Pi configuration tool. Perform the following steps:

1. Launch the Raspberry Pi configuration tool with the following command:

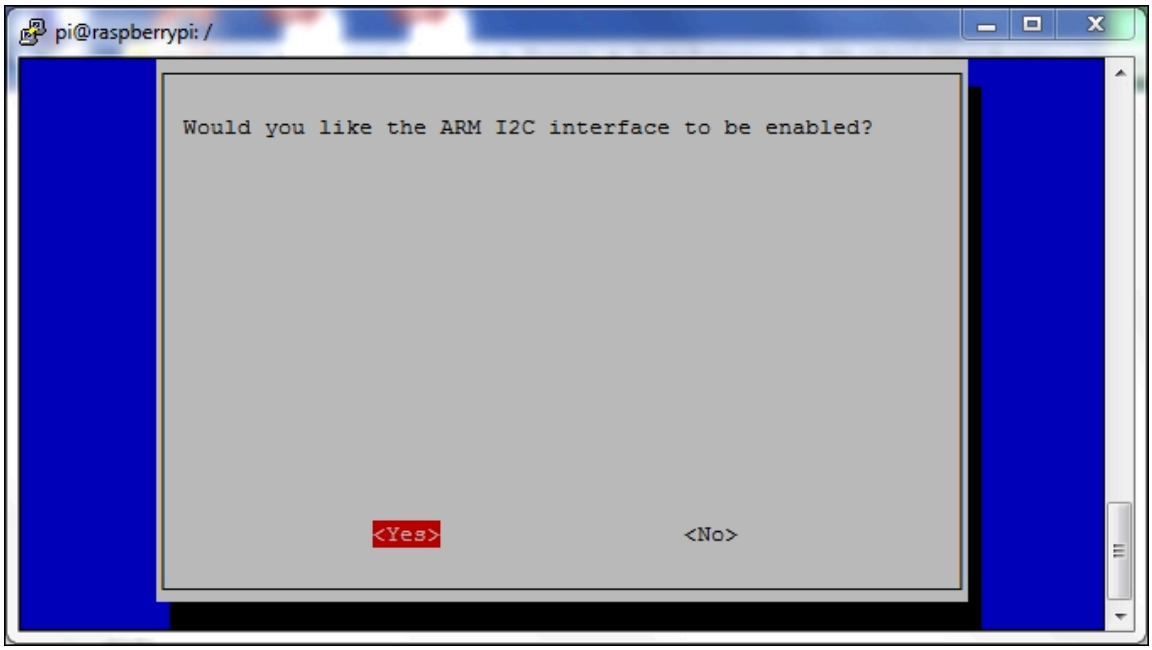
```
$ sudo raspi-config
```



2. Select option 8: `Advanced Options`.



3. Select Option A7: `I2C`.



4. Select `<Yes>`.
5. Reboot your Raspberry Pi for the setting to take effect.

Now that the I2C bus has been enabled, we need to set up the operating system so that the required modules are loaded each time the system boots. To do this, perform the following steps:

1. Edit the **Modules file** using the following line:

```
$ sudo nano /etc/modules
```

2. Add the following lines to the file:

```
i2c-bcm2708
i2c-dev
```

3. Save the file and exit Nano.

Installing the I2C tools package

So that we can easily access the I2C bus using Bash scripts, we need to install the `i2c-tools` package:

```
$ sudo apt-get install i2c-tools
```

Once installed, we should shutdown our system:

```
sudo shutdown –h now
```

After activity has stopped, switch off your Raspberry Pi, connect your port expander to the GPIO port, and power it back up so that we can start using it.

As a quick sanity check, you can see if I2C support has been loaded by typing:

```
$ ls /dev/i2c-*
```

This should give you a list of at least one bus—for example, `/dev/i2c-1`—if the module is loaded. If it's not, you'll probably get the following response:

ls: cannot access /dev/i2c-*: No such file or directory

In this case, you'll need to check back through the previous steps as something hasn't happened properly.

Finding our devices

The `i2c-tools` package installs several different tools to help us use our port expander attached to the bus. The `i2cdetect` tool allows us to find I2C buses and devices attached to the busses.

To get a list of I2C busses on our system, type the following:

```
$ sudo i2cdetect -l
```

You should get the following response:

pi@raspberrypi ~ $ sudo i2cdetect -l

i2c-1 i2c 20804000.i2c I2C adapter

The preceding output shows that we have one I2C bus, and this will be the one connected to our GPIO. *Note that earlier models of the Raspberry Pi may return the device ID as being i2c-0*.

We can now use the tool to scan for all of the devices attached to our bus. We do this by specifying the bus ID, as in the following command:

```
$ sudo i2cdetect 1
```

With nothing attached to the I2C bus (that is, without our port expander attached) we'd expect to see the following output:

```
pi@raspberrypi ~ $ sudo i2cdetect 1
WARNING! This program can confuse your I2C bus,
cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] Y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

Nothing found on the I2C bus

With our port expander attached, we should see the following output:

```
pi@raspberrypi ~ $ i2cdetect 1
WARNING! This program can confuse your I2C bus,
cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] Y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

```
70: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

Our I2C port expander slave device can be found at the address, 0x20 (32 decimal).

> **Note**
>
> The preceding address is the location of our MCP23017 chip connected to the I2C bus. If you don't see this, then there's probably a wiring issue and you'll need to go back and check.

You'll recall that we can add up to 8 of these devices to the I2C bus by setting the A0-A2 pins to a unique address. If A0 is set to `high`, then the address of the device will be shown as 0x21 (33 decimal)—and up to 0x27 (39 decimal), if all pins are high.

Setting up the port expander

As discussed in the previous chapter, we can have 2 x 8-bit busses on our port expander, with each pin being defined as an input or output. On the expander board we built, we called them I/O BUS A and I/O BUS B.

To configure the MCP23017 chip on the I2C bus, we can send it the appropriate commands using the **i2cset** tool we installed earlier.

On our home security system, we are going to assign all of the pins on BUS A as inputs for connecting our sensors to it. To do this, we use the following command:

```
$ sudo i2cset -y 1 0x20 0x00 0xFF
```

> **Note**
>
> What does this command mean?

- -y: This runs the command without user interaction.
- 1: This is the ID of the bus (for example, `i2c-1`).
- 0x20: This is the address of the chip.
- 0x00: This is the data register on the chip (in this case, the PORT pin assignment).
- 0xFF: This is the Value loaded into the data register (in this case, all pins as inputs—binary %11111111).

You can check that the data register has been set correctly by reading it using the following:

```
$ sudo i2cget –y 1 0x20 0x00
```

This should return a value of `0xFF`, which is the value we set earlier.

Connecting our magnetic contact sensor

Now that we've got our port expander working with the Raspberry Pi, we can start connecting things to it and create the scripts that will monitor the sensors on the input pins.

Let's go back to our port expander stripboard that was built in the previous chapter and connect our magnetic sensor. But first, we need to ensure that all of our inputs are pulled low by default using 10Kohm resistors. This prevents them from being in a *floating* state and giving us spurious data when we read the port's data.

> ## Note
>
> In the following diagram, I've connected the pull-down resistors externally, but you may want to include them directly on the stripboard. Toward the end of this book, we'll have a new board layout that brings everything that we've been prototyping so far together in a single solution.

To check the port's input value, we use the `i2cget` command:

```
$ sudo i2cget -y 1 0x20 0x12
```

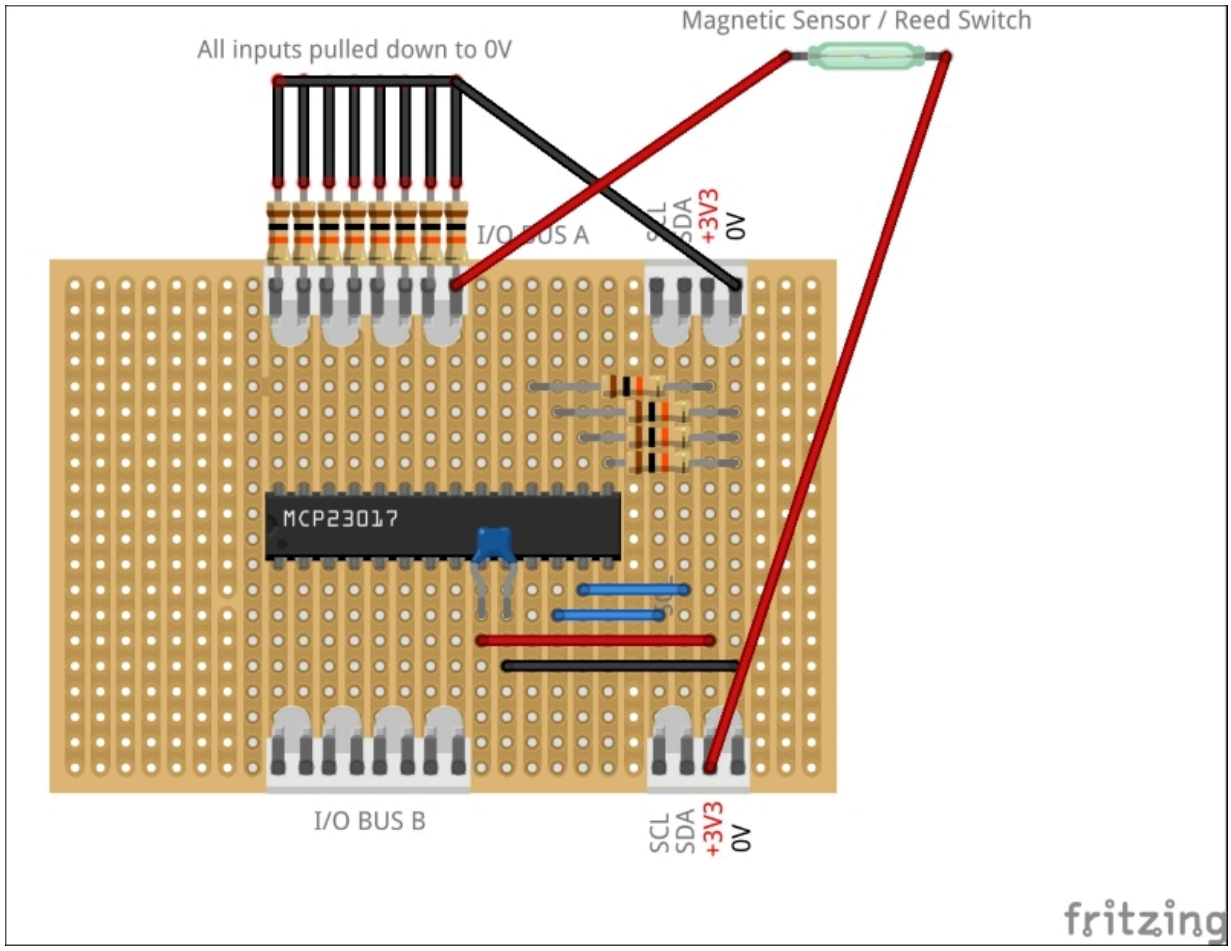
This should return `0x00`, which means all inputs are off (binary %00000000).

> ## Note
>
> What does this command mean?
>
> - -y: This runs the command without user interaction.

- 1: This is the ID of the bus (for example i2c-1).
- 0x20: This is the address of the chip.
- 0x12: This is the data register on the chip (in this case, the PORT read value).

Now let's connect one side of our magnetic sensor's reed switch to data pin 0 of BUS A (which we'll call GPA0 for reference), and the other side to our +3.3V line. By default, the switch is normally open (NO), which means that the input is still pulled low by the resistor.

But when you move the accompanying magnet near to the sensor switch (for example, if the door is closed), the switch will close, pulling the input high to the +3.3V line. If you read the port's input value now, by running the same command, you should see that it returns 0x01, indicating that the first bit is high (binary %00000001).



Monitoring the sensor

Now that we have everything in place and our magnetic sensor is detecting whether the door is closed, we can monitor this sensor with a simple Bash script that uses the I2C tool commands that we installed earlier.

The code listing for `poll-magnetic-switch.sh` is as follows:

```
#!/bin/bash
sudo i2cset –y 1 0x20 0x00 0xFF

# loop forever
while true
do
  # read the sensor state
  SWITCH=$(sudo i2cget –y 1 0x20 0x12)

  if [ $SWITCH == "0x01" ]
  then
    #contact closed so wait for a second
    echo "The door is closed!"
    sleep 1
  else
    #contact was opened
    echo "The door is open!"
  fi
done
```

When you run the script and then push the button, you should see "**The door is open!**" scrolling up the console screen until you stop pressing it.

By combining this with our elaborate light switch project in chapter 2, we can switch on the LED connected to GPIO17 when the door is opened:

```
#!/bin/bash

#set up the LED GPIO pin
```

```
sudo echo 17 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio17/direction

#set up port expander
sudo i2cset –y 1 0x20 0x00 0xFF

# loop forever
while true
do
  # read the sensor state
  SWITCH=$(sudo i2cget –y 1 0x20 0x12)

  if [ $SWITCH == "0x01" ]
  then
    #switch not pushed so turn off LED pin
    sudo echo 0 > /sys/class/gpio/gpio17/value
  else
    #switch was pushed so turn on LED pin
    sudo echo 1 > /sys/class/gpio/gpio17/value
  fi
  #short delay
  sleep 0.5
done
```

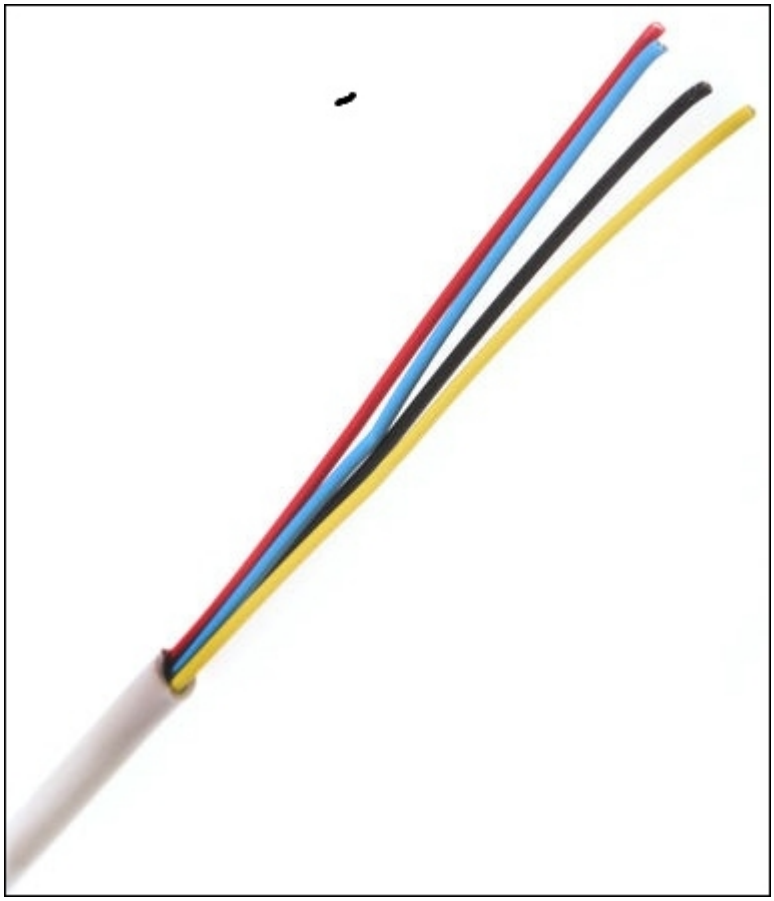
Note

Later, as we add more sensors to different input pins, we will need to be able to detect which one has been triggered. We'll look at writing a Bash function later in the book, which will parse the returned hex value from the i2cget command, and tell us exactly which of the 8 inputs is high.

Anti-tamper circuits

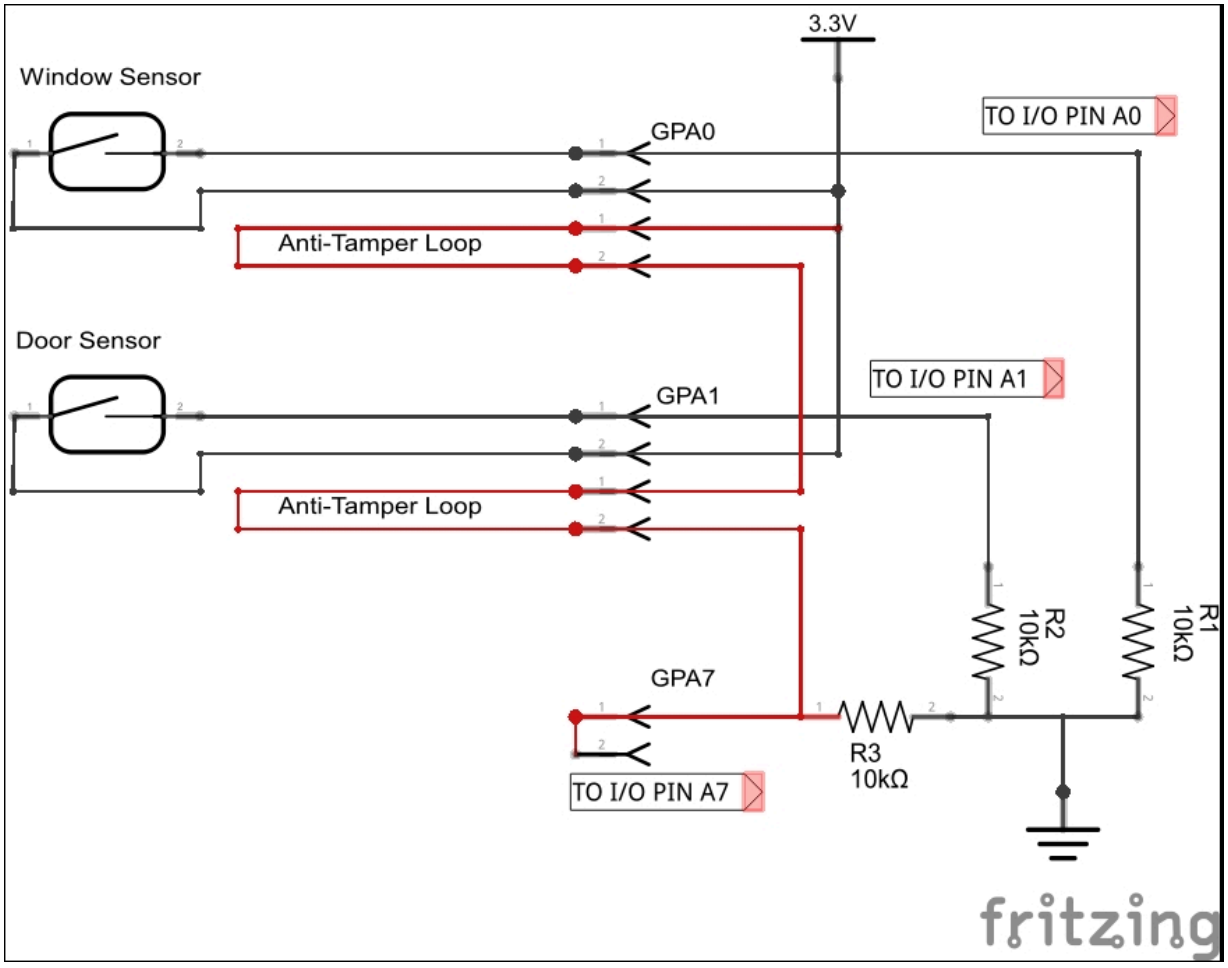
If you take a closer look at our system, you might realize that depending on whether you are detecting normally open or normally closed sensor switches, it is possible to tamper with the sensor channel by simply cutting the wire. So, in the case of a normally open switch, it wouldn't activate the monitoring system if the wires were cut, as it would always appear to be open, even if the switch was closed.

To mitigate this, most alarm systems feature a 4-core wiring system to connect the sensor devices to the main control board—two cores are used to connect the sensor and two are used to create an **anti-tamper loop**, which then itself forms a sensor input for monitoring.



4-core alarm cable

Take a look at the following circuit so that you see what I mean:



In this circuit, we have two sensors: one for monitoring a window and one for monitoring a door. These are connected to the I/O BUS A inputs, 0 and 1 (or GPA0 and GPA1, as we like to call them). As before, they are pulled down to 0V by resistors but, when switches are closed, the positive voltage rail takes the inputs high.

However, we've also added an anti-tamper loop throughout the whole system, which is connected to GPA7 for monitoring. The loop is daisy-chained through each of the cables connecting the sensors to the controller board. All the time the loop is intact, the input GPA7 is kept high, but if the cable is cut anywhere, the current will stop flowing through it and the resistor, R3, will pull the input low. This will then be detected by the monitoring script.

Many security sensor products provide a facility to terminate anti-tamper loop wires within them.

So, in our home security system, we're going to assign GPA7 as our anti-tamper loop.

Getting into the zone

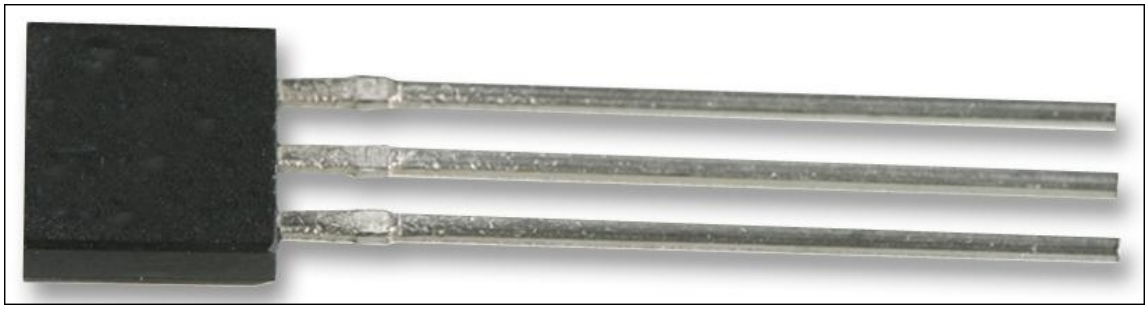
It may have occurred to you by now that even a modest-sized property could require plenty of door and window sensors; thus, if we used one input for each sensor, we'd soon run out unless we put more and more port expanders onto the system. The same is true for commercially available security systems.

So, the way this is dealt with is by creating **zones**, with each zone containing a group of sensors. A bedroom, for example, may be defined as one zone with a window sensor, a door sensor, and movement detector forming that zone. In this scenario, each sensor is connected to the next in a series (or daisy-chained); if one of them triggers, it will alert the monitoring system that there was a trigger in the zone. Obviously, though, it may not necessarily be the actual detector, which in most applications isn't really an issue.

However, this can introduce some challenges when we're considering mixing normally open and normally closed type sensors within a zone, but this is something we will explore later on in this book.

The other sensors you can use are listed as follows:

- **Hall Effect Sensor**: Hall-effect sensors are simple electronic chips that are used to detect magnetic fields placed near them. They are not dissimilar to the reed switch we've been using; however, because they are electronic devices, they are able to measure the degree of proximity in relation to the magnet (or the strength of magnetism), rather than being just on or off, as is the case with the reed switch. Also, because they are solid-state, they could be seen as being more reliable than mechanical switches.



A low-cost hall effect sensor—Allegro Microsystems A1302KUA-T

- Pressure Mat Sensors: Pressure mats are used to detect a person standing or walking on them, and can be placed under a floor mat to hide them from sight. They can even be used in a chair to detect people sitting on it. Essentially, they are switches, just like the reed switch, except that they are activated by the pressure of walking on them, and so, can be wired and used in exactly the same way as for our magnetic sensor circuits.



A pressure switch can be used under a front-door mat

Summary

In this chapter, we got our I2C-based port expander configured and working, and we experimented with it by connecting a magnetic sensor—one of the most commonly used sensors in security systems. We've also learned how to interact with I2C devices using Bash scripts, and how to read and write data to and from these devices.

In addition, we should now be beginning to understand the various elements and building blocks of a security system, including anti-tamper loops and zones. These are concepts that will prepare us for later on in the book, when we start to piece all of this together and build our final, all-encompassing system.

In the next chapter, we will look at passive infra-red motion detectors, how they work, and how we can connect the wired and wireless types to our home security system. We'll also learn how to create log files based on events using Bash scripts so that we can maintain a history of detector states as they change.

Chapter 5. Adding a Passive Infrared Motion Sensor

In the previous chapter, we started adding basic but commonly used magnetic switch sensors to our home security system and reading their status to protect doors and windows from intrusion. We also looked at how we can divide our home into zones, such as by individual rooms, so that we can group our sensors into logical circuits, which can then be identified as part of these zones rather than as individual sensor inputs.

We will now add **motion sensors** to our system in the form of **Passive Infra-Red** (**PIR**) detectors. These detectors come in a variety of types, and you may have seen them lurking in the corners of rooms. Fundamentally, they all work in the same way, which is detecting the presence of body heat within a certain range; so, they are commonly used to trigger alarm systems when somebody (or something, such as a pet cat) enters a room.



A typical PIR motion sensor (type GardScan QX-PIR)

In this chapter, we will:

- Learn how PIR detectors work and how they are set up
- Connect a wired PIR detector to an input on our port expander
- Start using a 12V power supply instead of 3.3V in our zone circuits
- Learn how to interface 12V circuits safely with our GPIO ports
- Learn how to connect a 433 MHz wireless receiver to our Raspberry Pi
- Connect a remote-controlled switch to our system using 433 MHz radio signals
- Write a script that will detect and log the state of our detector inputs when it changes

Prerequisites

You'll need the following parts for this chapter (apart from the components used in the previous chapter):

- A passive infrared detector, the wired type (this is available from any DIY store)
- A 4N25/4N35 opto-isolator
- A 1N4148 diode
- A 1-Kohm resistor
- A 10-Kohm resistor
- A 433 MHz receiver module and remote transmitter (this is optional)
- A 12V power supply
- A hook-up wire
- A 6 core alarm wire

Passive infrared sensors explained

You might not realize it, but all objects radiate heat energy (including your coffee table); it's just that you can't see it because heat consists essentially of infrared waves, which are invisible to the human eye (exactly the same as your TV remote control). These waves can, however, be detected by electronic devices designed for such a purpose, such as the infrared receiver in your TV that detects the energy emitted by your remote control when the buttons are pressed.

You probably do realize, however, that living things such as us, our cat, and the mouse under the floorboards generate quite a bit of heat. Passive infrared motion sensors used in security systems and automatic lights are designed to detect this level of heat. The term *passive* is used because the sensors themselves do not radiate any energy for detection purposes—instead, they just detect the infrared radiation emitted by objects. This is notably different from devices such as ultrasonic sensors and radars, which rely on detecting reflections from objects of the pulses of energy that the sensors send out.

PIR sensors need to be a little smart because they effectively have to cope with constantly varying temperatures in the room. They settle on the background temperature of the room they are in, such as that of a wall or floor that they point to. When an object, such as one of us or our cat, moves between the detector unit and the background object, the temperature in front of the sensor rises to the body temperature quickly, and this in turn triggers the system.

Setting up your PIR sensor

PIR sensor devices come in many formats, including different materials in sensor chips and the lens in front of the sensor view window that can widely affect the range, field of view, and sensitivity of the device. Therefore, your best guide to setting up a sensor will usually be in that little bit of instruction paper that comes with it.

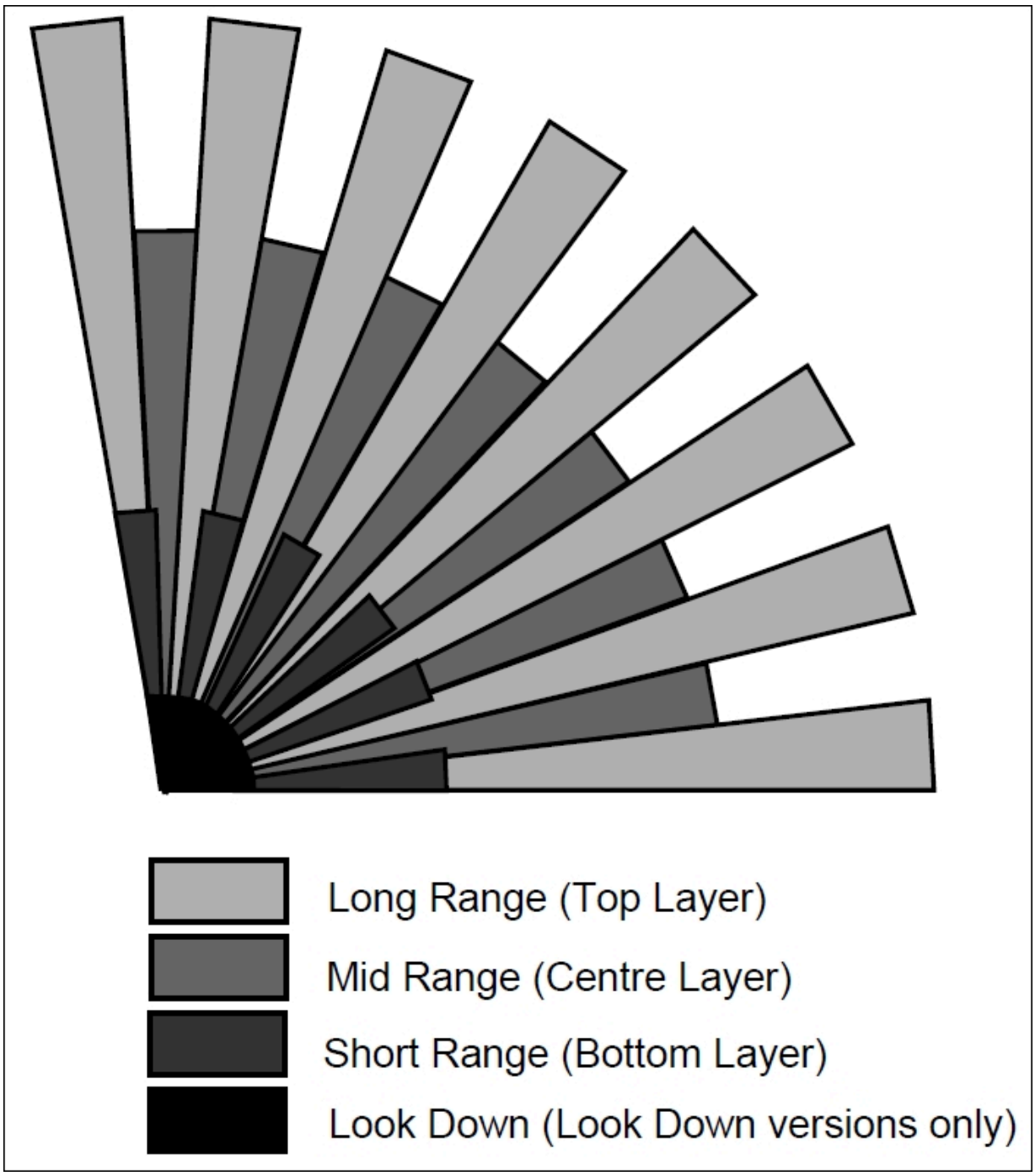
However, regardless of the type of PIR sensor you have, here are some general guidelines when considering where you mount your sensor in order to avoid false triggers:

- Ensure that the device is mounted on a solid foundation and not affected by vibration
- Never mount it in a location where direct or reflected sunlight can be picked up by the lens
- Similarly, never mount the device facing or above heat sources
- Don't mount the unit in draughty locations as this will affect its background temperature calibration

The location of the unit also depends on the area you want to protect. You may want to detect people entering your living room from the hallway, so your coverage area could be defined as being from the corner of the room where the device is mounted to the living room door.

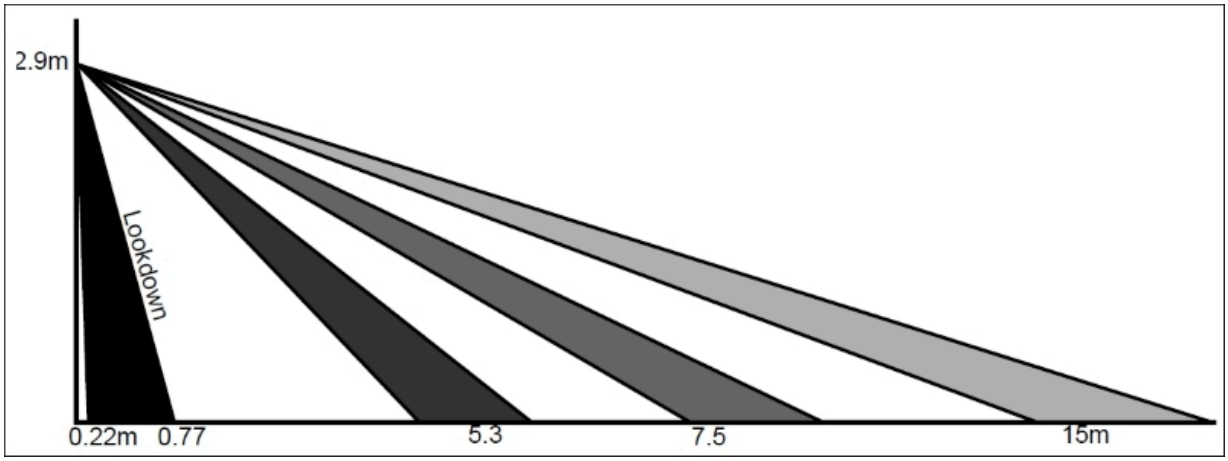
PIR sensors usually offer a fixed field of view (for example 90 or 110 degrees) but have a varying range, depending on the angle at which they are pointing down and the height at which they are located.

In my system, I will use a Gardscan QX PIR Detector for my wired units, which is a pretty good, low-cost unit available from RS Components (the order code is 493-1289). This unit has a field view of 110 degrees and a range of up to 12 meters, depending on the configurable down angle that it's mounted at. The coverage patterns for this particular unit, as taken from its datasheet, are shown in the following figure. Note that from these patterns not every part of the area in front of the device is covered, which is possibly not quite what you expect. This is why positioning the units in accordance with the device's datasheet is so important.



GardScan QX-PIR coverage pattern for its 110 degree field view (top/plan view)

Here is a diagram of the side view as well:



GardScan QX-PIR coverage pattern depending on the angle configured plus a "look-down" window (side view)

Give me power (again)

Before we can go on to connect off-the-shelf security devices to our alarm system, we need to have a power supply that's compatible with such devices. Typically, alarm circuits and their devices use a 12V supply with enough current to drive all the devices and the alarm control system itself.

Fortunately, this is not too difficult to sort out, but it is something we need to do now; otherwise, we won't be able to connect and power our PIR sensors. The easiest way to do this is to buy a high-quality 12V mains adapter that provides a nice regulated supply. These are readily available from online stores or electronics suppliers. Alternatively, you can build your own 12V regulated supply and add it to the power supply strip board that we built in Chapter 3, *Extending Your Pi to Connect More Things*.

> ## Note
>
> Another option is to use battery-powered PIR sensors, which means that you wouldn't have to power the unit from the security system's panel itself; however, it obviously also means that the batteries would need replacing from time to time. The wireless PIR we will look at later in this chapter is battery-powered.

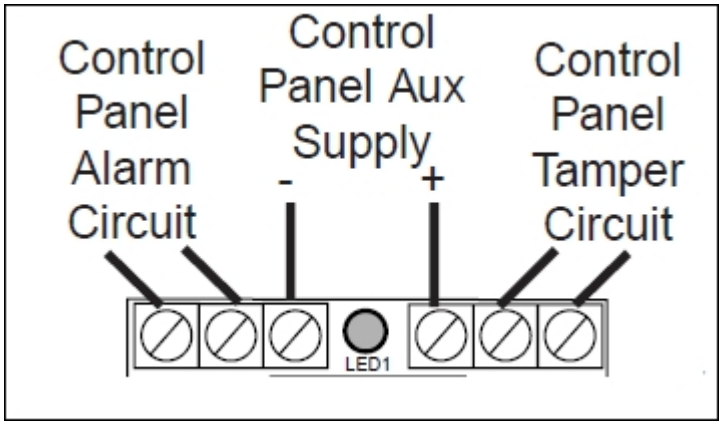
We'll take a look at handling higher-voltage sensor circuits later on in this chapter so that we don't blow up our home security control circuits or the Raspberry Pi.

Connecting our PIR motion sensor

Commercially available alarm systems connect to their devices using a 4 core or 6 core alarm cable. In the previous chapter, we used a 4 core cable because we were connecting a switch that needed two wires plus an antitamper loop, which needed another two wires.

For our PIR sensor circuit, we need the same four wires; however, we also need to send power to the device from the control panel, so an additional two wires are needed for this—hence the requirement for a 6 core cable.

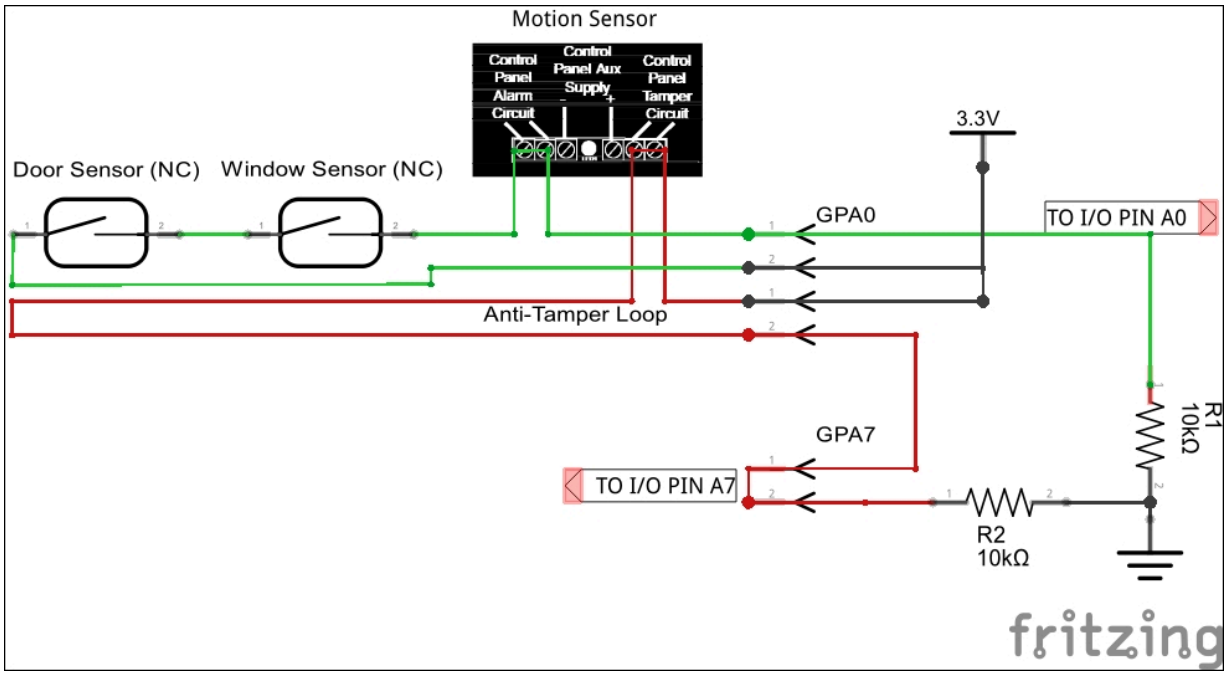
The following diagram shows the wiring connections for my GardScan PIR sensor, but this is in fact typical for most off-the-shelf security system devices:



Typical connections for security system sensor devices

Similar to the magnetic contact sensors that we looked at in the previous chapter, devices can come with either a **normally closed** (**NC**) or a **normally open** (**NO**) alarm. This particular device has a normally closed output, which means that the alarm circuit will be broken when the detector is triggered. This is the preferred configuration for our sensor devices as this means that they can be wired in a series within each of our zones.

We can now add this sensor device into the alarm circuit that we started putting together in the previous chapter. The following diagram shows the circuit for all our sensors so far wired into a single zone:



A schematic for our zone with all three sensors plus tamper loop in the same zone

Up until now, we used the +3.3V power supply to pass through the sensor switches and alarm circuit. In fact, this is not a good idea, and we've been doing this only for convenience to test out our GPIO inputs.

In reality, and in our final system, we really should use a 12V supply to pass through the sensor and antitamper circuits. This is because a higher voltage travels better through the system and is less susceptible to noise, which could prevent triggering or cause false triggering. This also makes it compatible with commercially available systems and accessories.

12V alarm zone circuits

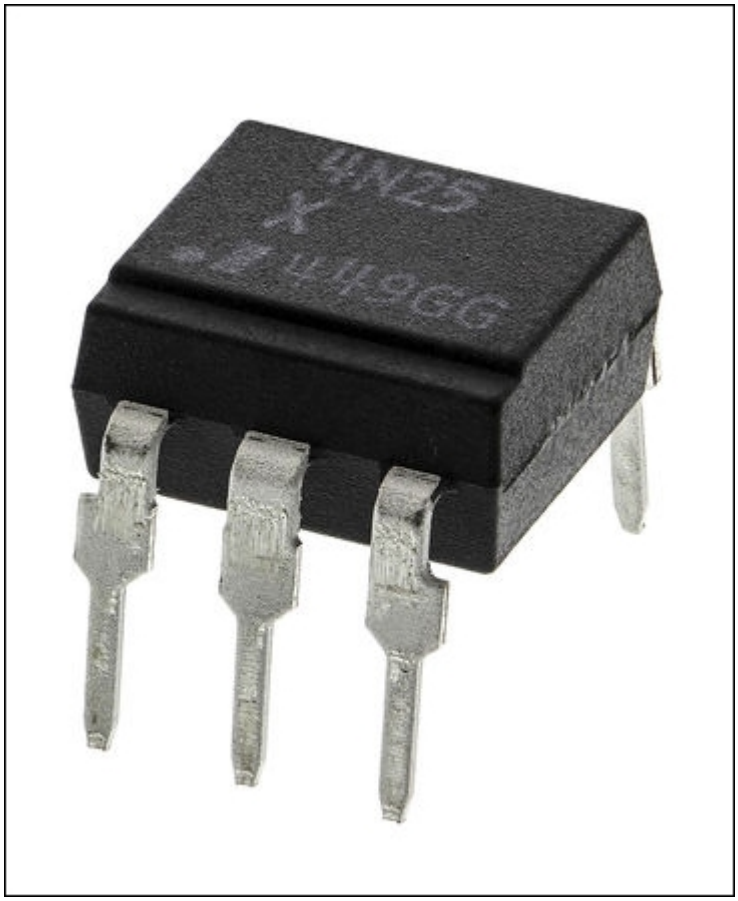
Making our zone circuits use 12V instead of 3.3V is as simple as changing the power supply, and in fact all of sensors we used so far can handle 12V power passed through their switches.

However, if we were to present the 12V circuit to the inputs on our GPIO port on the Raspberry Pi or our port expander, we would expect to see some magic smoke and smell something burning. So, we need to add some circuitry that allows us to use 12V alarm circuits as well as protect our control board inputs.

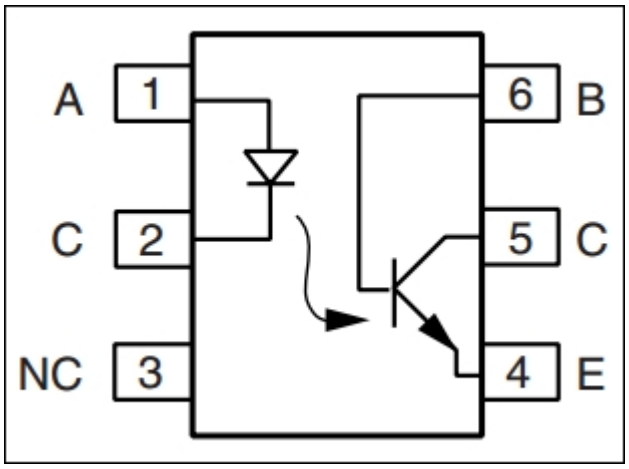
Alarm circuit protection

An effective way to protect our zone inputs from 12V alarm inputs is to use a little low-cost device called an opto-isolator. As the name suggests, this isolates the alarm circuit from the digital inputs of the control board using light.

Inside an opto-isolator (also called an opto-coupler) is an infrared LED, which transmits light to a photo-transistor when a current is passed through it, thus switching it on. The circuits are electrically isolated as they are controlled only by light.

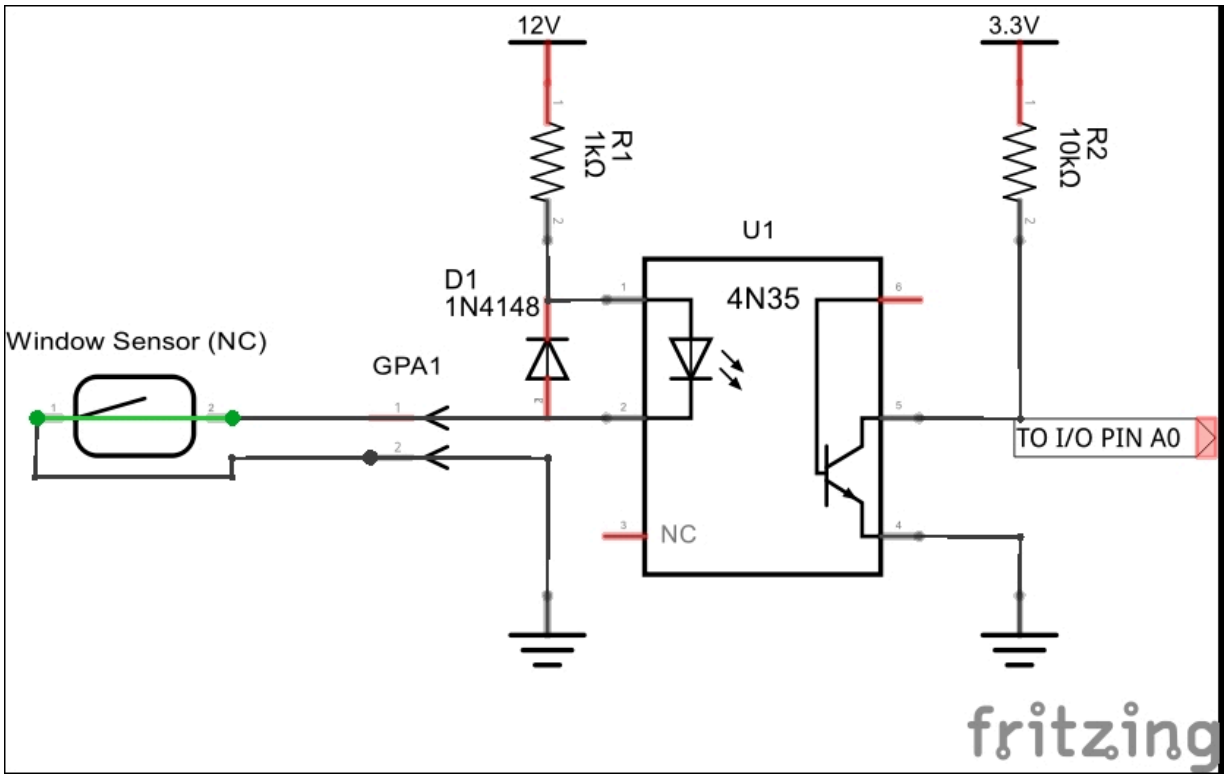


The 4N25 (shown in the preceding image) and 4N35 are low-cost, 6-pin opto-coupler devices, and most manufactures tend to use the pin layout shown in the following diagram:



Now that we know how we will couple our 12V alarm circuit with the inputs on our control panel, let's build the entire circuit, which we'll

use for each of the zones that we add to our system.



A 12V zone circuit optically isolated from the GPIO input

How it works

At this time, we're assuming that our zone circuits are normally closed—that is, the alarm triggers when the circuit is broken.

The 12V supply is passed through the LED of the opto-isolator with the current being limited by the 1-Kohm resistor. The 1N4148 diode, in reverse, is there to protect the opto-coupler from reverse-polarity voltages.

Note

The 1-Kohm resistor is calculated from the fact that we have a 12V supply and a forward voltage drop (Vf) of 1.2V across the LED with a current (If) of about 10 mA.

While the alarm circuit is closed, the current flows, and the LED is on. This keeps the transistor on and the input to the GPIO port is held low. If the alarm circuit is broken, the opto-coupler LED switches off, and this in turn switches off the transistor. The GPIO input is then pulled high by the 10-Kohm resistor.

This is quite simple but effective, eh?

The other advantage of this circuit is that it should fail positive—that is, if the opto-coupler should fail for any reason, the alarm input on the GPIO port should be pulled high, thus triggering it rather than it just failing silently.

Wireless PIR motion sensors

Wireless motion sensors are now commonly available at a low cost, allowing them to be installed practically anywhere without any wiring from the alarm control panel. Some of them still require an external power supply, but many operate on batteries. The alarm system must contain a wireless receiver compatible with the wireless sensor.

In this section, we'll take a look at how we can use our Raspberry Pi-based security system with wireless receiver devices.

433-MHz wireless alarm systems

Wireless systems use an unlicensed radio frequency to communicate between the various components of an alarm system. In the UK, the two most popular frequencies used are **433 MHz** and **868 MHz**. While the more recent systems now use the 868-MHz frequency, 433 MHz is still in widespread use as it has a slightly longer range than an 868-MHz system. However, the 433-MHz band is also used by many other devices, which makes it congested, whereas 868 MHz is generally used only for alarm systems.

While wireless security systems can be convenient, it's important to understand the advantages and disadvantages of using wireless rather than wired systems.

The advantages are as follows:

- Their ease and speed of installation
- Their ease of removal, which means that you can take them anywhere with you
- Expanding the system in the future can be easier, with most systems automatically detecting new units

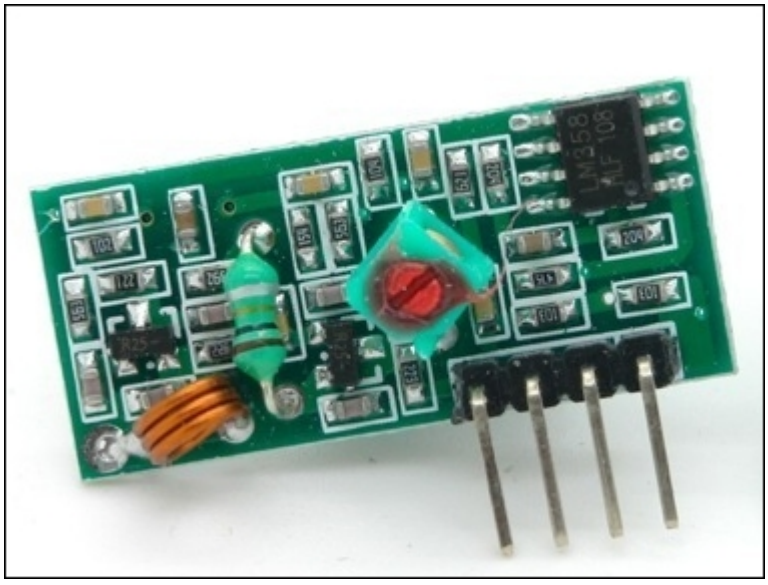
The disadvantages are as follows:

- They are more expensive than wired systems, sometimes three or four times the cost
- They are not as secure as wired systems and cannot achieve a security grading greater than two in accordance with European Standard BSEN 50131 (although, this grade is suitable for domestic properties)
- Wireless devices need to have their batteries replaced at regular intervals
- Wireless systems are less reliable and susceptible to interference and even radio jamming

Connecting a 433-MHz receiver

In the past, it was possible to roll out your own 433-MHz receiver for the Raspberry Pi using an inexpensive receiver, such as the XY-MV-5V module along with the **433-Util** library that was put together by a guy called Mark Wolfe, a contributor on GitHub. Essentially, he gathered together code relating to 433-MHz communications and put it all into this library. Originally developed for Arduino, this has now been ported to the Raspberry Pi.

You can then use a readily available transmitter, such as a key fob or any other 433-MHz transmitter, and take a look at the incoming code as you press each button on the transmitter.



A XY-MK-5V Generic 433-MHz receiver module

Finding a suitable 433-MHz receiver should be easy as websites such as Amazon and eBay are awash with them, and they cost as little as a couple of pounds.

Note

Note that the 433-MHz band is a free for many types of devices. As such, there are various different types of receiver, and although they may all state that they are 433-MHz receivers, they can operate using AM or FM, and some only detect certain types of data. Some, such as the Quasar QAM range, may also require special decoder chips in order to read transmitted data and may only work with paired transmitters.

The receiver module can pick up signals from a key fob remote control, such as the one shown in the following image (this can be picked up from the home security section of any local DIY store), which gets an output as a series of square waves. These square waves are then decoded by the 433-Util software.



A Novar/Blyss 433MHz wireless remote control

I liked this particular remote control because I thought it would be good as the **arm** and **disarm** device for our home security system. I will talk about arming and disarming in Chapter 8, *A Miscellany of Things*, where we will look at the ways to achieve this.

The alternative approach (because we have no choice)

I started off the previous section with the words "In the past–". This is because in recent times, I've not been able to get the 433-Util software working with receiver modules, which used to work in the past). I'm not entirely sure why this is so; however, I can only guess that because the software uses "bit banging" to decode incoming

data signals, the timing is no longer correct, perhaps because later Raspberry Pi boards are faster and therefore mess up the routines.

Note

What is bit banging?

Bit banging is a way of using software for serial communication instead of dedicated hardware. The software is responsible for all the parameters of the signal, including timing, levels, and synchronization. Bit banging can be seen as a bit of a hack, but it does allow the implementation of different protocols at a very low cost without any hardware changes.

So, in order to make our lives easier (and actually make the device work on all flavors of Pi), we will resort to using a dedicated receiver module that you can pick up for less than –5 on Amazon and doesn't require all this software bit banging nonsense. You'll notice from the following image that it still uses a similar XY-MK-5V radio receiver; it's just that the host board decodes the signals for us and switches a relay on or off in response to a command from the remote control.

Note

If you're still interested in the 433-Util software project and want to try and roll out your own receiver, you can find the original project at https://github.com/ninjablocks/433Utils.

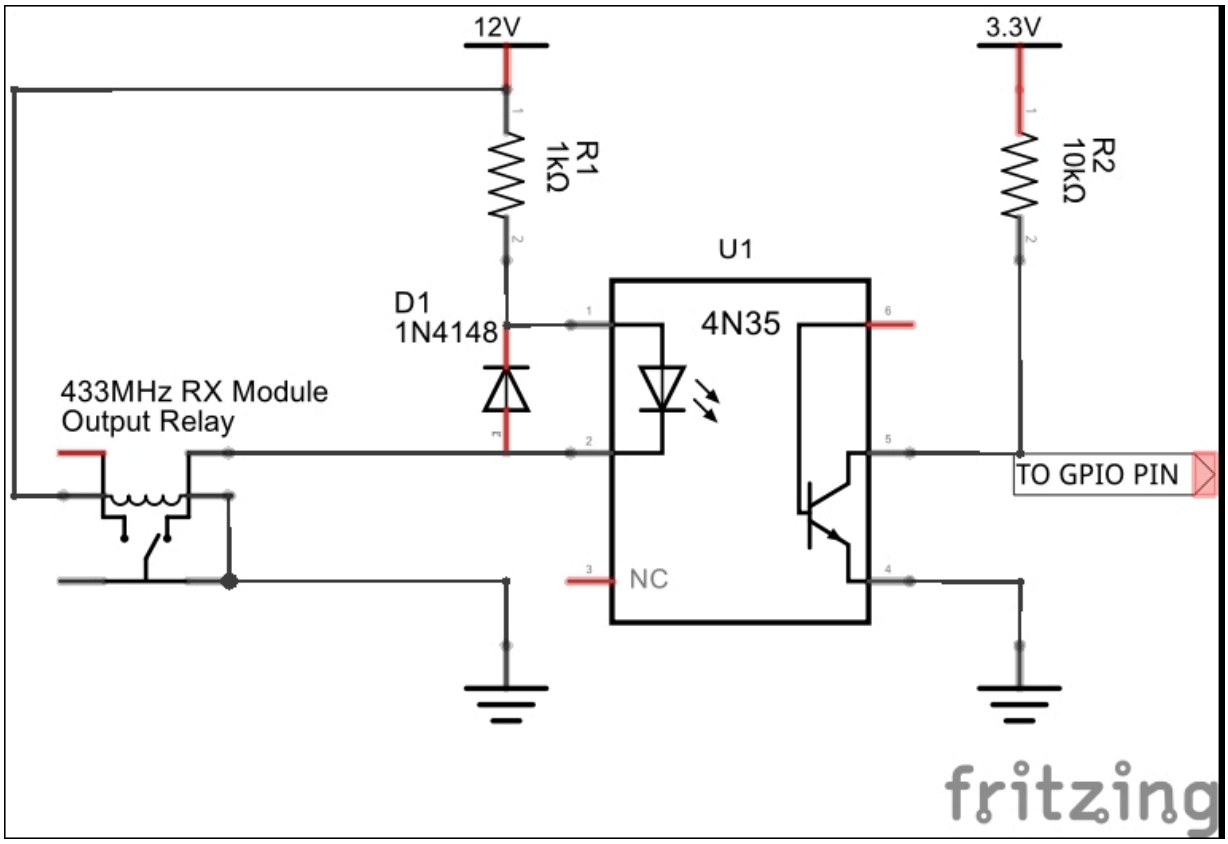


A Hielec transmitter fob and receiver module, available on Amazon

The fact that it just switches a relay on and off means that we can easily implement this in our home security system because it simply acts as a switch. When you press the button on the transmitter, the relay switches the contacts on; press it again, and the relay switches off. The screw terminals on the board provide us with access to the relay terminals.

The receiver wiring diagram

As we are just dealing with a switch input, we can use the same circuit as we did with the zone circuit earlier but connected to our arm/disarm GPIO input, which we'll determine in Chapter 9, *Putting It All Together*.



Circuit to interface the receiver module with a GPIO input

When the receiver module switches the relay on, this will complete the 12V circuit through the opto-coupler's LED by turning it on. This will make the transistor pull the GPIO pin down to ground, giving it a low input.

You can use this type of circuit for any paired receiver for the wireless security devices that you want to use in your system.

Logging detection data

With any system, it's useful to be able to log data when something happens. We can do this with our detectors too by writing to a log file every time a detector in a zone is triggered. This way, you can keep a log of every time someone enters a room, which you can review at a later date even if the system isn't armed. You can also keep a log of when the system is armed and disarmed.

Here's a simple script that shows you how to do this whenever an event happens on our zones connected to the GPIO inputs:

```
#!/bin/bash

#set up the I2C expansion port
sudo i2cset –y 1 0x20 0x00 0xFF

#reset status
CURR_STATE="0x00"
LAST_STATE="0x00"

#path to the log file
LOG_FILE="/etc/pi-alarm/zones.log"

# loop forever
while true
do
  # read the gpio inputs
  CURR_STATE=$(sudo i2cget –y 1 0x20 0x12)

   #check if state has changed
   if [ "$CURR_STATE" != "$LAST_STATE" ]
  then
    #write change to log file
      TIMESTAMP=`date "+%Y-%m-%d %H:%M:%S"`
     echo "$TIMESTAMP Zone Status Changed from
$LAST_STATE to $CURR_STATE" > $LOG_FILE
  fi
  $LAST_STATE = $CURR_STATE
  sleep 1
done
```

The preceding example is quite simple, but it can be made more useful by actually writing out the zone or zones that change by decoding the hex value that's returned by the `i2cget` command in the constituent zones.

Note

In Chapter 9, *Putting It All Together*, you'll learn how this is done in order to display the individual status of each zone on a web page. You can use exactly the same technique to do this for your log files and, in fact, output to the log file by expanding on the same script.

Summary

In this chapter, we started off by learning how passive infrared sensors are used to detect motion to protect a predefined coverage area from intrusion. We then looked at connecting these to the inputs on our port expander via opto-couplers as we will now use 12V to power the alarm zone circuits.

We then looked at wireless alarm systems that operate on the open 433-MHz band, which is commonly used for security devices. After exploring the possibility of using the legacy 433-Util bit-banging software on our Raspberry Pi to decode the signals transmitted by devices using a simple receiver, we opted to use a paired receiver device that will interface easily with our alarm circuit inputs.

Finally, we created a simple script that will log the changes in our alarm inputs to a text file, which can later be expanded to log exactly what's going on with the system in detail.

Chapter 6. Adding Cameras to Our Security System

Until now, we've been putting together the elements that will allow us to connect sensors to our alarm system to detect intrusions using either switches or passive infra-red motion detectors, which in turn will tell our Raspberry Pi that something has happened in a particular zone. These elements will all come together as a whole system later in this book.

Our system is now going to become a whole lot more sophisticated with the addition of cameras to take pictures and video clips, and e-mail them to us straightaway when it detects something.

We'll also use e-mail to send us alerts on our smart phone when we're out and about when any of the sensors in the system are triggered.

In this chapter we will cover the following topics:

- Setting up the Raspberry Pi camera module and learning how to capture stills and video images
- Learning how to overlay captured images with text and time-stamps
- Triggering image captures with a motion detector
- E-mailing the image and video files to us in real time
- Understanding the differences between capturing images during the day and during the night
- Switching on and off security lighting and other high-current devices when required
- Connecting a USB webcam instead of the native camera module

Prerequisites

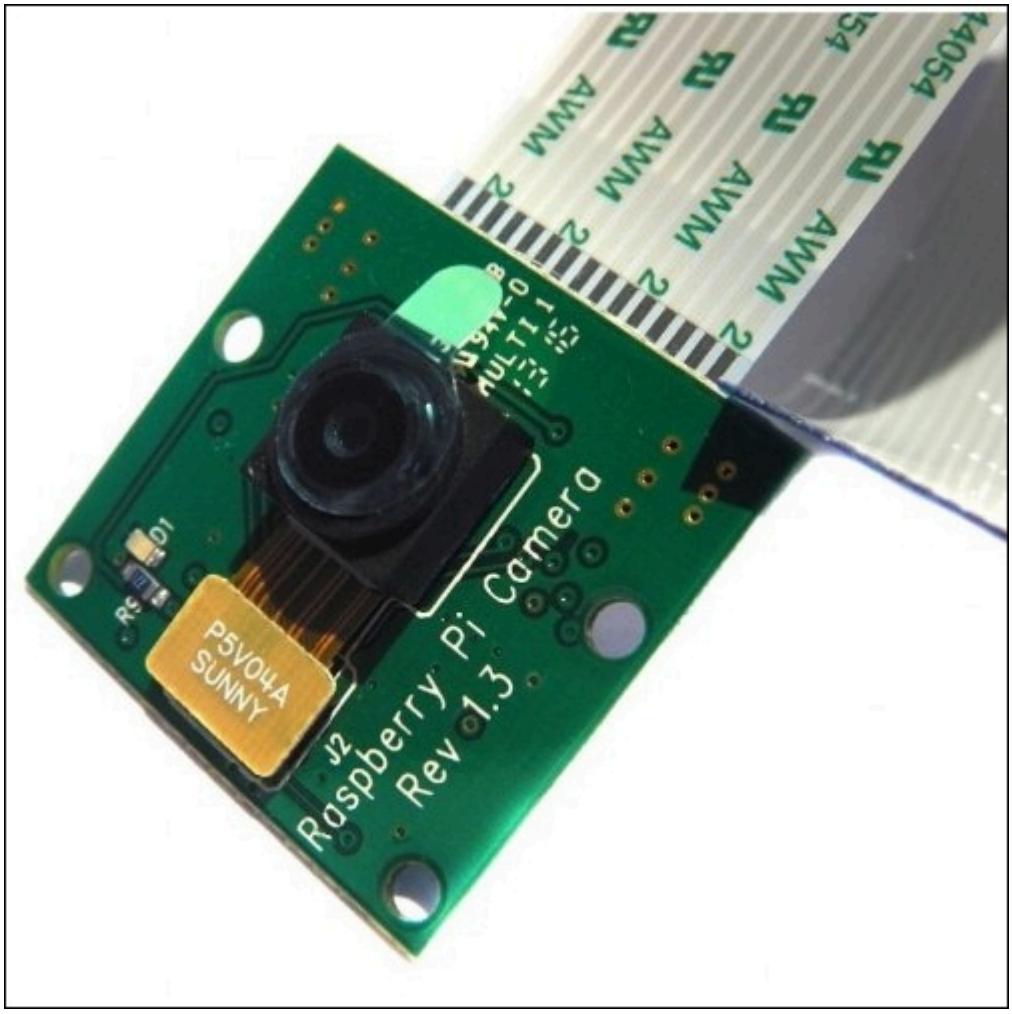
You'll need the following parts for this chapter, on top of the components used in the previous chapter:

- A Raspberry Pi standard camera module
- A Raspberry Pi NoIR camera module
- An Infra-Red LED array and/or visible LED array
- A USB webcam

The Raspberry Pi camera module

The Raspberry Pi Camera Module is an official Raspberry Pi accessory that works with all models of the Pi, and can be used to take high-definition stills and video images. It connects directly to the Pi board's **camera serial interface** (**CSI**) port, which is dedicated to these modules to enable high-speed operation.

The camera itself is a 5 megapixel fixed-focus sensor supporting 1080p, 720p, and VGA video modes and still captures.



The official Raspberry Pi Camera Module

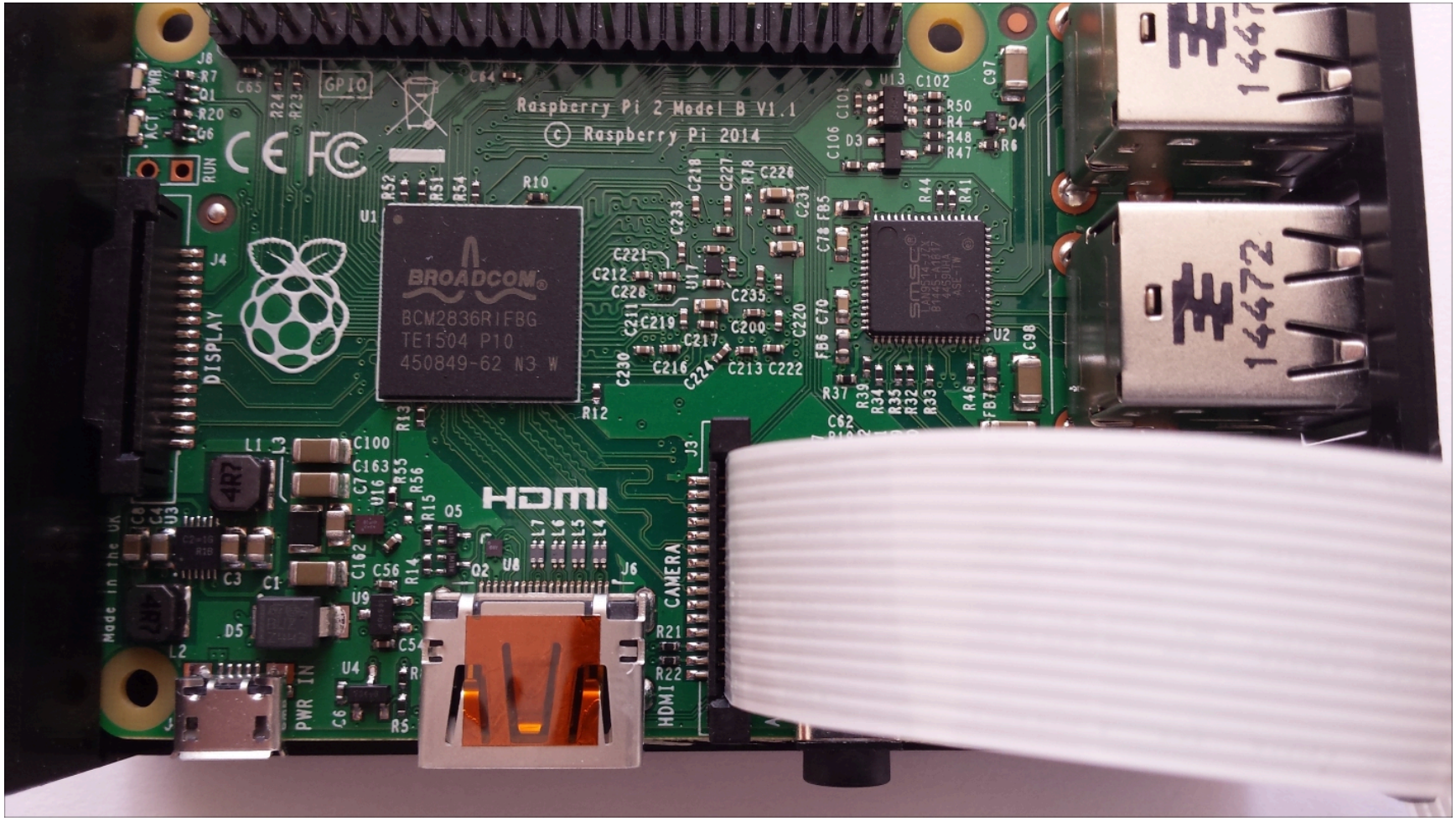
You can also obtain housings for the camera modules, which, unless you're going to build your own enclosure for the camera system, I recommend you use.



Raspberry Pi camera housings come in various colors and styles

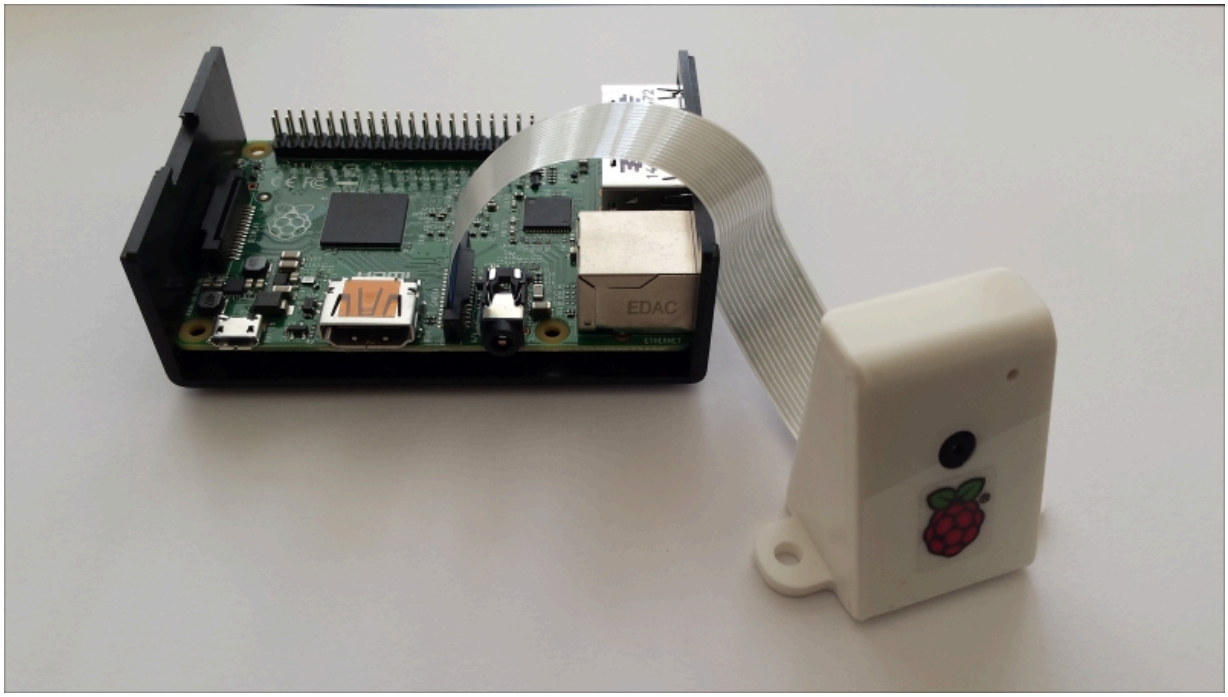
Connecting the camera module

As previously mentioned, the module connects directly to the Raspberry Pi board via its dedicated camera interfaces, as shown in the following image. When connecting the camera, the contact side of the ribbon cable is toward the HDMI connector and the blue side of the cable is toward the network connector.


Raspberry Pi 2 Model B V1.1
© Raspberry Pi 2014
BROADCOM
BCM2836RIFBG
TE1504 P10
450849-62 N3 W
HDMI
DISPLAY
CAMERA
PWR IN
GPIO
Made in the UK

Connect the camera module to the dedicated interface

As you can see in the following image, the ribbon connector is not that long, so the camera needs to be located close to the Raspberry Pi. By using a camera enclosure, you could actually mount the camera directly on top of the Raspberry Pi case itself, if that works for you.



The camera module, housed within an enclosure

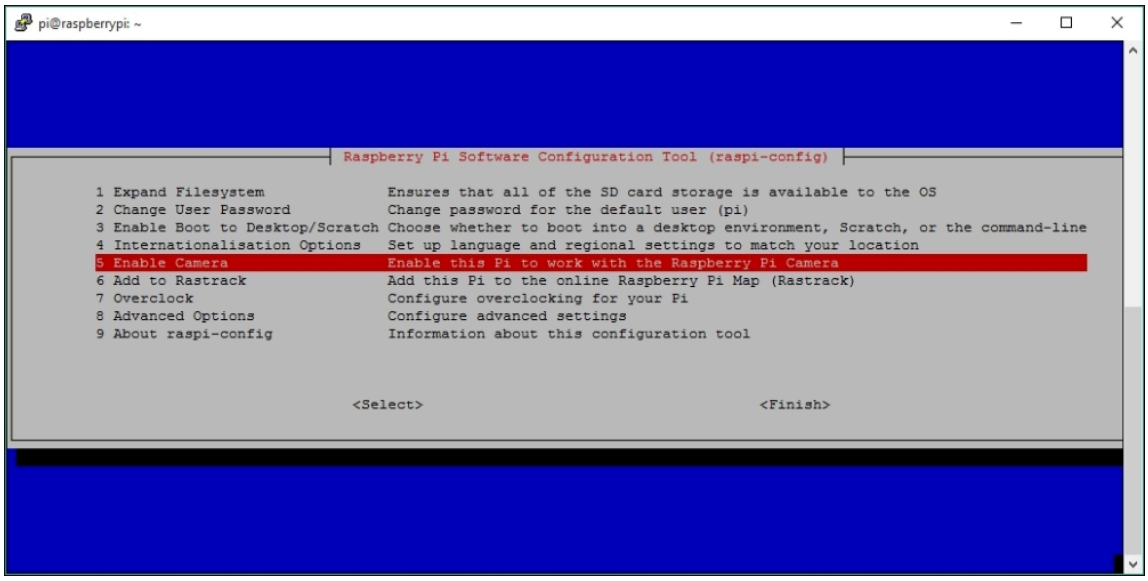
Setting up the camera module

Before we can use the camera module, we need to enable camera support on the Raspberry Pi. To do this, we use the `raspi-config` tool, as we did with the I2C bus earlier in our journey.

1. Connect to your Raspberry Pi the lazy way from your sofa using SSH, or directly using a keyboard and monitor.
2. Once you've logged in, launch the config tool with the following command:

```
$ sudo raspi-config
```

3. And then, select `5 Enable Camera`.



4. You'll then be asked to confirm whether you want to enable camera support.



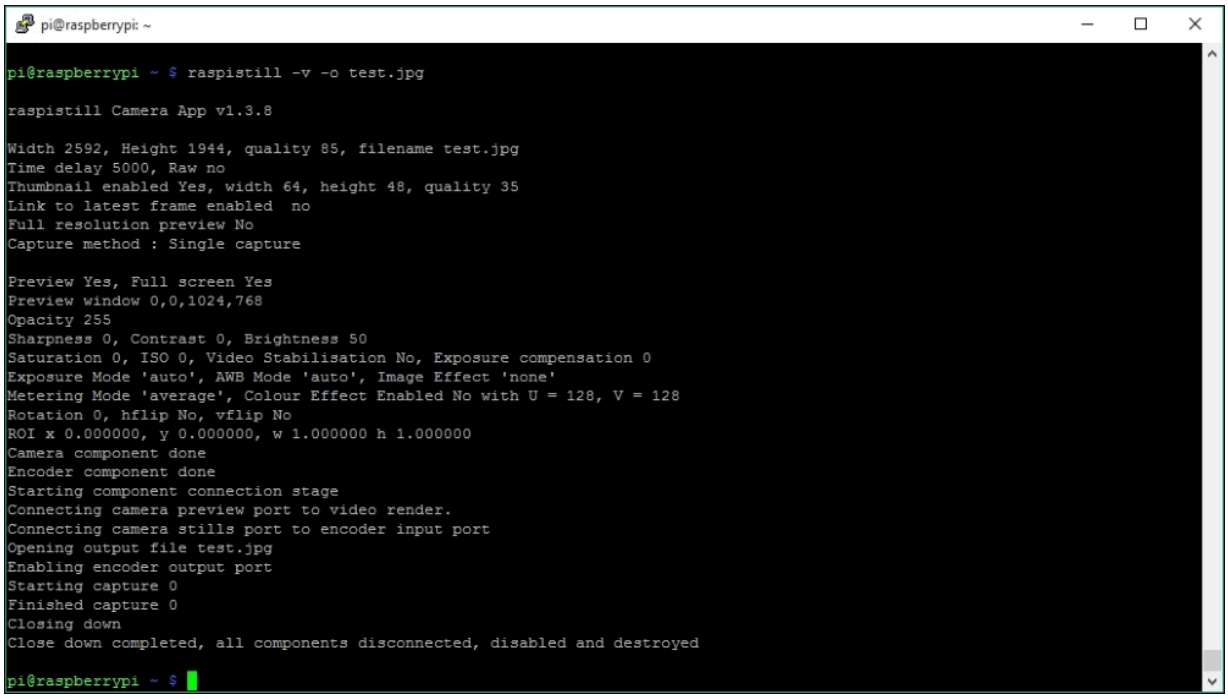
5. Select `<Enable>`.
6. Then, select **Finish** and reboot your Pi to enable the camera settings.

Testing the camera module

Once your Raspberry Pi has rebooted, your camera should be enabled. We can test this by taking a still image using the `raspistill` utility:

```
$ raspistill –v -o test.img
```

This will delay for 5 seconds then take a picture, while displaying various pieces of information, such as that shown in the following screenshot:



Note

The camera module needs at least 128 MB of GPU memory to operate properly on Raspian. If you experience any issues, first ensure that the the `gpu_mem` setting in the `/boot/config.txt` configuration file is set to at least `128`.

And if all goes well, you should find the file, `test.jpg`, in your home folder. As you're connected via the shell, you wouldn't have seen the 5 second preview image displayed when the command was running.

If you download the image file to your PC, you should see a nice quality snap taken by the camera module.



The test photo taken by the Raspberry Pi Camera Module

Tip

If you find that raspistill outputs errors when you run it, ensure that it is connected properly at both ends of the ribbon cable. One other catch is that sometimes the ribbon that connects the actual camera lens component to the tiny connector on the camera board can come loose. Just ensure that this is securely

connected too. I've had this issue a couple times after the camera modules have been taken out of my box of random test bits to be used.

The `raspistill` utility has loads of options for manipulating the images it captures, and we'll use some of them a bit later in our capture script. In the meantime, to see the available options, run `raspistill` without any options and they will be listed:

```
$ raspistill
```

Be a video star

Now that we know our camera module is working, we can try and capture some video. To do this, we'll use the `raspivid` utility. The following command will take 5 seconds of high-definition video and save the file to your Raspberry Pi:

```
$ raspivid –o test.h264 –t 5000
```

You'll notice that file is called `test.h264`—this is because the video is captured as a raw **H.264** video stream. Unfortunately, not many media players will handle these files (although VLC player will—it rocks and handles practically anything you throw at it—get it on your PC at www.videolan.org).

If you want to play the file on smartphones and conventional media players, then we will need to wrap it in a container format, such as MPEG-4, and give the file a `.mp4` extension.

To do this, we'll use the **GPAC** package, which is an open source multimedia framework. It comes with a utility called **MP4Box**, which is a tool we'll use to create an MP4 container for our video file:

1. First, install the GPAC package:

   ```
   $ sudo apt-get install gpac
   ```

2. Once it's installed, run the command to convert the test video we created:

   ```
   $ MP4Box -fps 30 -add test.h264 test.mp4
   ```

You should now have the file, `test.mp4`, which you can download and play on your PC or smartphone.

Tip

Another popular conversion tool is **ffmpeg**, which I use a lot on Windows to convert video files; however, it can be quite complex and although there is a package for the Raspberry Pi, I actually couldn't get it to convert properly on the Pi. MP4Box is much more straightforward and fitting for our needs.

Caught on camera

So, we now have a method of capturing still images and video, which we can put to use in our security system. If we want to have this running constantly, we could write a script to take video constantly, but this would soon fill up our memory card and wouldn't be particularly efficient. So, we'll combine our camera system with the motion detectors we connected earlier.

In the last chapter, we created an alarm zone which had a couple of sensors and a motion detector connected to our system on the input GPA0. So, let's write a script that will take a video clip whenever the motion detector is triggered:

```
#!/bin/bash


#set up port expander
sudo i2cset –y 1 0x20 0x00 0xFF

# loop forever
while true
do
  # read the GPA inputs
  GPA=$(sudo i2cget –y 1 0x20 0x12)

  # detect the zone on input 0
  if [ $GPA == "0x01" ]
  then
    #circuit normally closed so zone is OK
    #short delay
sleep 0.5

  else
```

```
    #zone is activated so take a 20 sec video clip

    #filename will be based on current timestamp
    sDate='date +%d%m%y'
    sTime='date +%T'
    echo "Zone 1 Activate at $sDate $sTime"

#take video clip
raspivid –o $sDate$sTime.h264 –t 20000

#convert to MP4
MP4Box -fps 30 -add $sDate$sTime.h264
$sDate$sTime.mp4
  fi
done
```

You have new mail

Having the images stored on your Raspberry Pi is not really much use—ideally, you would want the images sent to you straightaway, as soon as they are captured, so that you can view them on your smartphone.

An easy, quick, and reliable way to do this is to simply have them e-mailed to you. Hence we're going to add an e-mailing functionality to our home security system so that image captures are attached to a message and sent to your e-mail address straightaway, which you can access from your smartphone. The images can then be removed from your Raspberry Pi to prevent the SD card space from being clogged up with these reasonably large files.

Setting up the e-mail sender client

Fortunately, there are some good packages available that will help us with this. Carry out the following steps to install the email packages we need:

1. Update the package installer with the following command:

```
$ sudo apt-get update
```

2. Install and set up the SMTP client with the following command:

```
$ sudo apt-get install ssmtp
```

   You'll now need to set up the client to send emails through your email account. In the following configuration file, I've assumed that you have a Gmail account. The settings may be different if you use another email provider.

3. Open the `ssmtp` configuration file using **Nano** or another text editor:

```
$ sudo nano /etc/ssmtp/ssmtp.conf
```

4. Replace the entries with the following configuration:

```
root=<your-username>@gmail.com
mailhub=smtp.gmail.com:587
rewriteDomain=gmail.com
AuthUser=<your-username>@gmail.com
AuthPass=<your-password>
FromLineOverride=YES
UseSTARTTLS=YES
```

5. `ssmtp` can be used on its own but can be a bit of a faff while automatically sending emails (by default, you manually type the email in with the command line, or create a text file), so we're also going to install the `mailutils` package:

```
$ sudo apt-get install mailutils
```

6. Once it's installed, we can use the `mail` command to send emails more easily. Send a test email through the (G)mail account that we set up earlier, using the following command to make sure your settings are working:

```
$ echo "Test Email" | mail –s "Test Pi-Mail"
me@mydomain.com
```

If all goes well, you should receive the test email in your mailbox within a few seconds or so.

Sending attachments

Now that we can send basic emails from our home security system, let's try sending the still image taken from our camera earlier. But first, we need to install yet another package to help us with this:

```
$ sudo apt-get install mpack
```

Once that's installed, you can send the test image file we took previously by using the following command:

```
$ sudo mpack –s "Security Photo" test.jpg
me@mydomain.com
```

We now have all of the elements needed to send alerts and images from our home security system directly to our smartphone using email.

Where was that taken?

Ordinarily, you could just annotate the email message with where and when the attached image was taken, but that wouldn't be as cool as actually overlaying the image with some text, would it? So let's do some magic with the help of `imagemagick`, which is a popular command-line image manipulation tool. Install it with the following line:

```
$ sudo apt-get install imagemagick
```

We'll now use the command line to take the test photo that we took earlier, overlay some text using one of the `imagemagick` utilities, and save it to another file:

```
$ convert test.jpg –fill red –pointsize 48
annotate +20+60 'Camera 1' annotated.jpg
```

After a few seconds, this will have generated a file called `annotated.jpg` containing our image with **Camera 1** written in red in the top corner. When we put all of this together in our final system, we'll also overlay the image with a time stamp.



Note

At the moment, the images generated by the `raspistill` tool are pretty large, being high resolution photos. This makes manipulating and sending them a bit time-consuming as far as processing time is concerned, so when we build our final system, we'll be using the `raspistill` options, `-w`, `-h` and, `-q`, to reduce the size and quality of the images to make the system more efficient.

To capture smaller image files, try using the following command:

```
$ raspistill -o test.img -h 768 -w 1024 -q 25
```

Night vision

The standard Raspberry Pi camera is great for taking daytime snaps of people walking up the garden path, but when it comes to night time shots, it's not really suitable. There are two ways of dealing with this: the first is to illuminate the capture area with a bright light when the PIR detector is triggered, and the second is to use the Raspberry Pi **NoIR camera module** and an infra-red LED array to let the camera see in the dark. More about that in a minute.

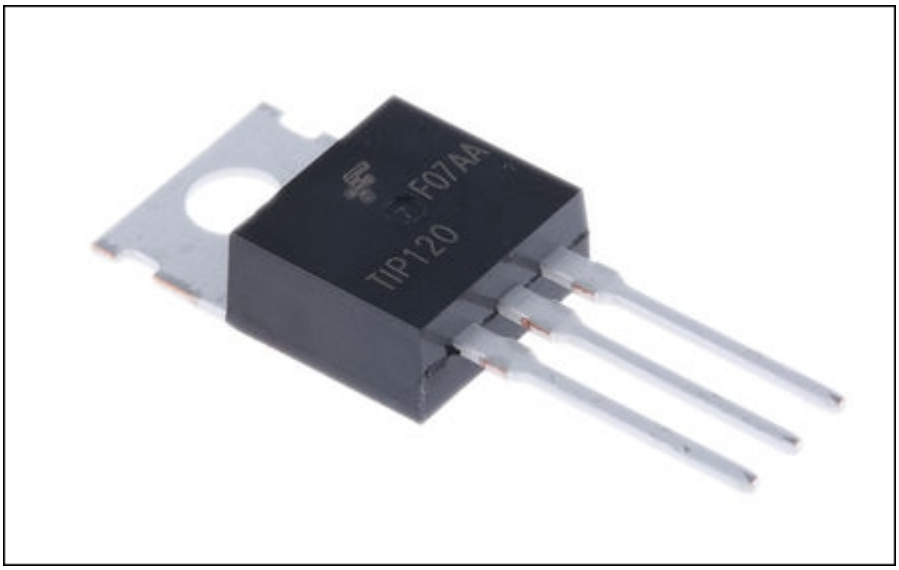


The Raspberry Pi NoIR camera module; it looks similar to the standard model

An illuminating experience

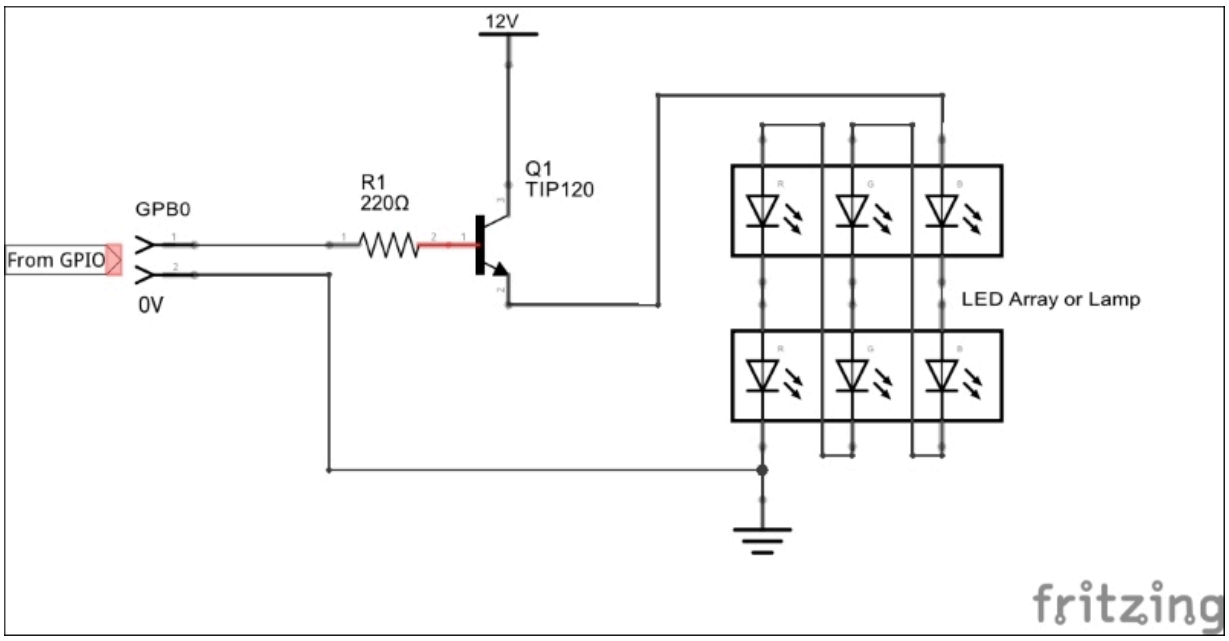
In order to switch on a light or LED array from the Raspberry Pi GPIO or our port expander circuit, we need something that will allow us to drive higher currents and voltages than can be provided by the GPIO ports alone.

A good candidate for this is the **TIP120 Darlington transistor**, which will allow us to switch on and off loads of up to 80V and 5 A from our GPIO pins. In our full system later on, we're going to use Port B of our MCP23017 port expander to control outputs, but the principle stands for any of the GPIO outputs available to us.



TIP120 transistors can be bought cheaply but can drive large loads

The following circuit shows how we can drive big loads from our GPIO port outputs.



In our example circuit, we're using a GPIO output pin to control the base of our transistor via a 220 ohm resistor. When the GPIO pin goes high, the transistor is switched on and allows the 12V circuit to flow through the LED array.

In the preceding circuit, there is no current limiting for the LEDs because they are connected in series, and so with nine of them, each dropping about 1.5V across, this is about right for a 12V supply (yes I know I've only included six LEDs here but it's just for illustration). Remember to adjust for your particular needs. This circuit could easily drive other loads, such as bulbs or sounders.

Note

If you intend to drive high power loads, you will probably need to attach the TIP120 to a heat sink that will dissipate any heat and prevent it from over-heating and burning out. In our circuit that was demonstrated previously, however, you probably won't need one as we're only driving a couple of hundred milliwatts at most.

The Elaborate light switch re-visited

Expanding once again on our elaborate light switch from previous chapters, we can once again write a Bash script that will switch on our camera light, take a snap with the camera, and e-mail it to us when a PIR detector is triggered.

For the following script, we're assuming that the output controlling the TIP120 transistor is the Raspberry PI GPIO17 pin (D0 or pin 11 of our connector), which replaces the LED in our earlier set-up. The input from the PIR trigger is, again, connected to the GPA0 (port A, data pin 0) of our MCP23017 port expander. All the other inputs are tied low, as before, using 10 K resistors:

```
#!/bin/bash
```

```
#set up the High Load GPIO pin
sudo echo 17 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio17/direction

#set up port expander Port A for inputs
sudo i2cset –y 1 0x20 0x00 0xFF

#clear the output by default to switch light off
sudo echo 0 > /sys/class/gpio/gpio17/value


# loop forever
while true
do
  # read the sensor state
  SWITCH=$(sudo i2cget –y 1 0x20 0x12)

  #PIR is normally closed so pin is held high
  if [ $SWITCH != "0x01" ]
  then
    #PIR was triggered – pin taken low

    #switch on lamp driver
    sudo echo 1 > /sys/class/gpio/gpio17/value
    sleep 0.5

#take a still image
    sudo raspistill –o –image.jpg –h 768 –w 1024 –
q 25

    #email the image
    mpack –s "Security Alert Photo" test.jpg
me@mydomain.com

    #switch off the lamp driver
    sudo echo 0 > /sys/class/gpio/gpio17/value

  fi
  #short delay
  sleep 0.5
done
```

pir-camera-trigger.sh

You'll now see that we've started developing the foundations of the software that will control our home security system.

Is that a badger?

If you don't want to illuminate an area before capturing an image, you can use **infra-red lighting** in conjunction with a compatible camera. The standard Raspberry Pi camera module won't work with infra-red lighting because it contains an infra-red filter, but we can use the NoIR version of the camera module instead.

The Raspberry Pi NoIR camera module is exactly the same as the standard one, except that it doesn't have an infra-red filter built in, which means it will see in the dark with the aid of infra-red lighting. This makes it good for watching badgers at night as well as for use in our home security system.

You will need an infra-red LED array or cluster to invisibly illuminate the area you want to capture with the camera. These are readily available in various form factors and intensities, or you can build your own using individual infra-red LEDs purchased from an electronics store.



The Kingbright infra-red LED cluster runs from a 6V supply, which means you can connect two in series—one on either side of the camera.

Connecting and driving the LED cluster modules works exactly the same as our illuminating light above, using the TIP120 driver circuit. The only difference is that we humans can't see when the LEDs are on.

Using USB cameras

Instead of using the Raspberry Pi Camera Module, it's also possible to use a standard USB **webcam** to take still images. You should be aware though that the dedicated camera module is far superior to a USB webcam in terms of image quality. Although, you may already have a webcam knocking about in your box of bits, so why not try it?

Installing the webcam

After you've connected your webcam to a USB port on your Pi, you can check whether it's been recognized using the `lsusb` command:

```
$ lsusb
```

I'm using a Logitech webcam that gets reported as follows with `lsusb` (Device 006):

```
pi@raspberrypi ~ $ lsusb
Bus 001 Device 002: ID 0424:9514 Standard
Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation
2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard
Microsystems Corp.
Bus 001 Device 006: ID 046d:08d8 Logitech, Inc.
QuickCam for Notebook Deluxe
```

Note

Not all webcams will work with the Raspberry Pi. Even though it may be recognized as a USB device, it might not actually work properly with the operating system and create a video device (for example, `/dev/video0`). For example, an old cheap Trust webcam I had appeared as a USB device but wouldn't capture any images.

You can check whether your webcam is likely to work with the Pi by checking your make and model at http://elinux.org/RPi_USB_Webcams.

So, now that the Pi knows that we have a webcam device attached, we can use the `fswebcam` utility to capture image frames. You can find out more about `fswebcam` from the developer's site at http://www.sanslogic.co.uk/fswebcam.

Install `fswebcam` with the following:

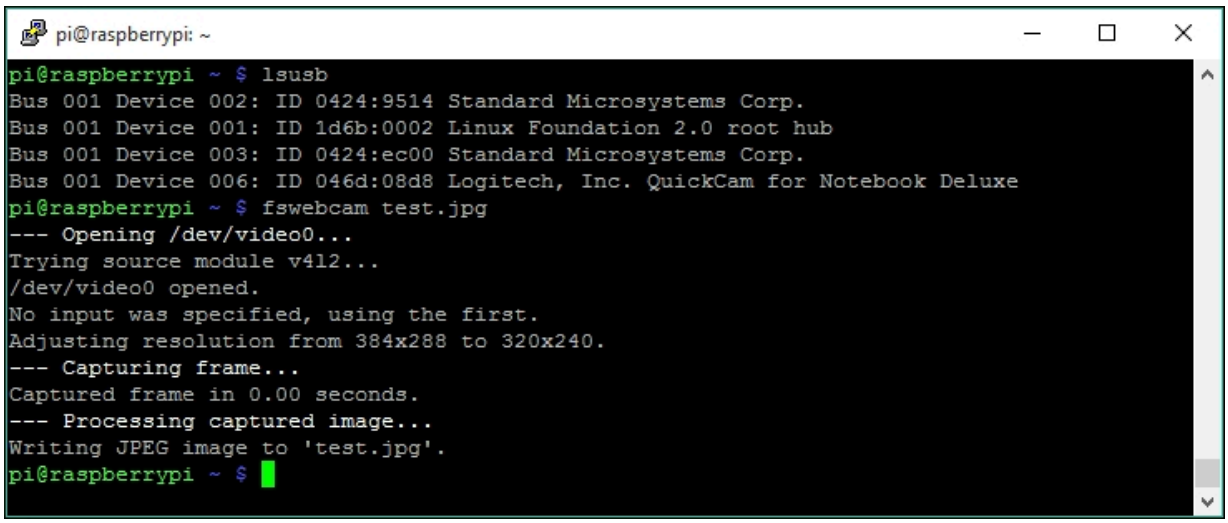
```
$ sudo apt-get install fswebcam
```

Taking a snap

You can now test the webcam by capturing a still image, which can be done by running the following command:

```
$ fswebcam test.jpg
```

You should expect to see output similar to the following:



Note

`fswebcam` has lots of options for things like the resolution and quality of the image. Use the command `fswebcam -?` to get a list of all options.

Snap snap snap

`fswebcam` doesn't take video streams, but you can set it up to take a series of **frames** at regular intervals. For example, to take a snap every 10 seconds, you can use the following command:

```
$ fswebcam frame.jpg -l 10
```

An example of how this would be useful can be demonstrated by setting the webcam to take a snap every few seconds in the background (the `-q` switch runs `fswebcam` in the background). When our security system is triggered, we could then take the latest image snapped with the webcam which could be looking down your pathway.

For the purpose of putting together our entire system later in this book, we'll be focusing on the Raspberry Pi Camera Module, but you can always replace the code with the previous examples if you want to use USB webcams instead.

You'll notice that `fswebcam`, unlike `raspistill`, has the ability to overlay the images with timestamp information, so you don't need to worry about overlaying text as we did previously. Look at the `fswebcam` command line options for more information.

The multicamera setup

It may have occurred to you that the Raspberry Pi has only one camera module input. Now, this is obviously limiting if you want to have multiple cameras around your property that are triggered by motion detectors.

However, there is nothing stopping us from building standalone units that have a separate Raspberry Pi board with a PIR detector, Camera Module, and network connection, either using a Wi-Fi dongle or Ethernet.

Because you only need a single input to the Raspberry Pi to detect when the PIR motion sensor is triggered, you can use the on-board GPIO port to connect the sensor, rather than using a port expander. The Raspberry Pi will email the alert over the network, and could alert the main controller Pi if required—making it a slave sensor device.

You can readily obtain small PIR detectors, such as the Parallax one shown next, which you can mount onto a Raspberry Pi Case along with the camera module, creating a self-contained unit.



A Parallax PIR motion sensor (type 555-28027)

The Slave driver

While it may seem quite elaborate to have a Raspberry Pi for each camera—think about it—you can actually build each camera unit with all of the components for around –50, which is significantly cheaper than buying a wireless *smart* camera. If you really want to be clever, you could also use this as a slave device to accept further sensor inputs local to the unit.

There is nothing to stop you from connecting a GPIO output pin on the slave unit to drive an input on the main controller and control the pin depending on the state of its local sensors. By running a 6-core cable between the units, you could even power the slave unit if your power supply is man enough (you'd need to have a supply of 5V @ 1A for the slave Pi running along the wire).

I'm not going to go into any more detail about this configuration at this time, but you could set yourself a challenge to create a fully distributed home security system using multiple Raspberry Pis and the building blocks and concepts learned in this book.

Summary

In this chapter, we learned how to connect both Raspberry Pi camera modules and USB cameras to our Pi board in order to take image and video captures when required by our home security system. We also learned how to overlay our images with informative text and have the files immediately emailed to us.

In order to capture images from our camera at night, we also looked at ways to illuminate the capture area using both visible and infra-red lighting, with the ability to switch the lighting on and off as required by using a high-current Darlington transistor driver.

In the next chapter, we're going to get down to the business of putting together modules by building a mobile-optimized web-based control panel for our home security system. We'll learn how to set up a Web server on our Raspberry Pi and manipulate files using our Web control panel, which means that we'll start to explore how all of the elements we've encountered so far can come together as part of our final system.

Chapter 7. Building a Web-Based Control Panel

We've now got all of our hardware elements together for us to create a complete home-security system featuring contact switches for our doors and windows, and motion detectors and cameras to take happy snaps of wannabe intruders! I've deliberately guided you through this in a modular fashion so that you can pick and choose and expand on the hardware sensor elements that suit your requirements. In Chapter 9, *Putting It All Together* we will be wiring all of this together to form the complete system based on zones that we looked at earlier.

One thing that all home security systems require is a **control panel** that allows us to **arm** and **disarm** the system and monitor the status of the zones within our system. We might also want to do things such as only arm certain zones, or have the system automatically arm and disarm at certain times of the day.

The hardware required for this, such as switches, LEDs, and LCD displays, can be quite expensive and time-consuming to put together; they can also make the system less configurable and flexible. So, in our system, we're going to build a Web-based control panel that we can access from our mobile phone browser. This also means that we can control the system remotely, when we are out of the house.

In this chapter, we will cover the following:

- Defining the scope of our home security in terms of the number of zones we will be monitoring and the I/O ports we will use
- Learning how to install and configure a web server on our Raspberry Pi
- Developing a basic HTML5 web page for our alarm control panel

- Learning how to use PHP scripts to dynamically configure our system from the web page

Installing the web server

There are several **web servers** readily available that we could install on our Raspberry Pi, and they would all be suitable for our system. But I like the **lighttpd** web server as it's easy to use and lightweight. lighttpd is often referred to, and affectionately known as, "Lighty"—which to be honest is less of a mouthful than lighttpd.

As well as the Web server itself, we're also going to install **PHP** support, which will allow us to write dynamic web pages to interact with the Linux system. Now, to be honest, I'm not a massive fan of PHP for commercial Web-based deployments for many reasons, but for a small embedded-Linux system such as our home security system, it's perfect and works really well. It's also quite straightforward to get into if you've never done server-side Web-scripting as well.

To perform the following steps, you'll need to be logged into your Raspberry Pi via the terminal console (for example, PuTTY):

1. Update the package installer:

   ```
   $ sudo apt-get update
   ```

2. Install the `lighttpd` Web server:

   ```
   $ sudo apt-get install lighttpd
   ```

   Once installed, it will automatically start up as a background service, and will do so each time your Raspberry Pi starts up.

3. Install PHP5 support:

   ```
   $ sudo apt-get install php5-cgi
   ```

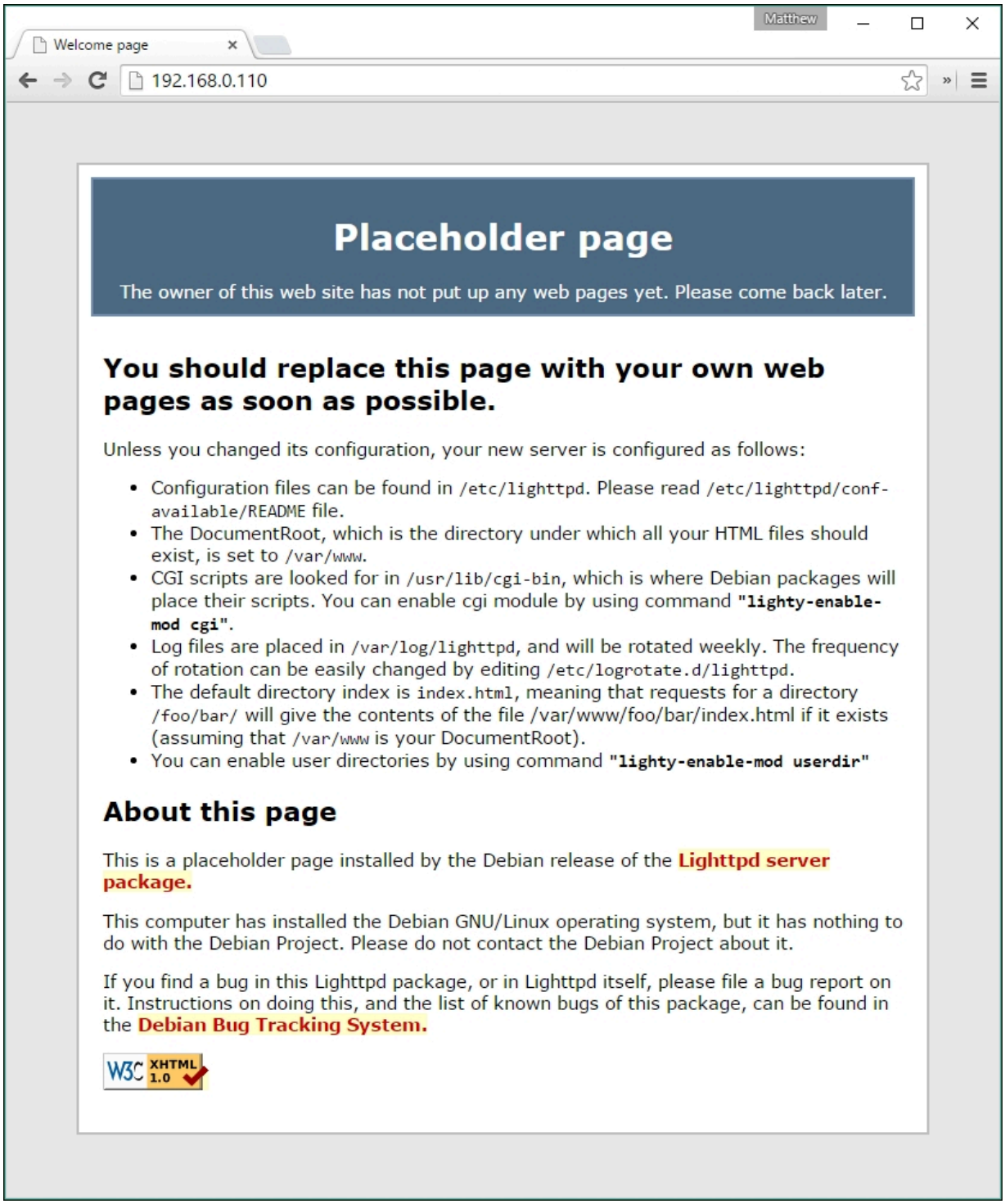
4. Now, we need to enable the PHP FastCGI module in our web server:

```
$ sudo lighty-enable-mod fastcgi-php
```

5. And finally, we need to restart the Web server:

```
$ sudo /etc/init.d/lighttpd
```

That's it! You should now have your PHP Web server installed. By default, the web content files get installed in the location, `/var/www`, and Lighty installs a test placeholder page in this location, which you can access from your browser by simply entering the IP address of your Raspberry Pi, as shown in the following screenshot:



The Lighttpd placeholder page

Testing the PHP5 installation

While we're at it, we should also test our PHP installation, as this is fundamental to building our console. This can be done by writing a simple PHP script page that, if PHP is installed correctly, will return information about its environment and configuration:

1. First, go to the web content folder:

```
$ cd /var/www
```

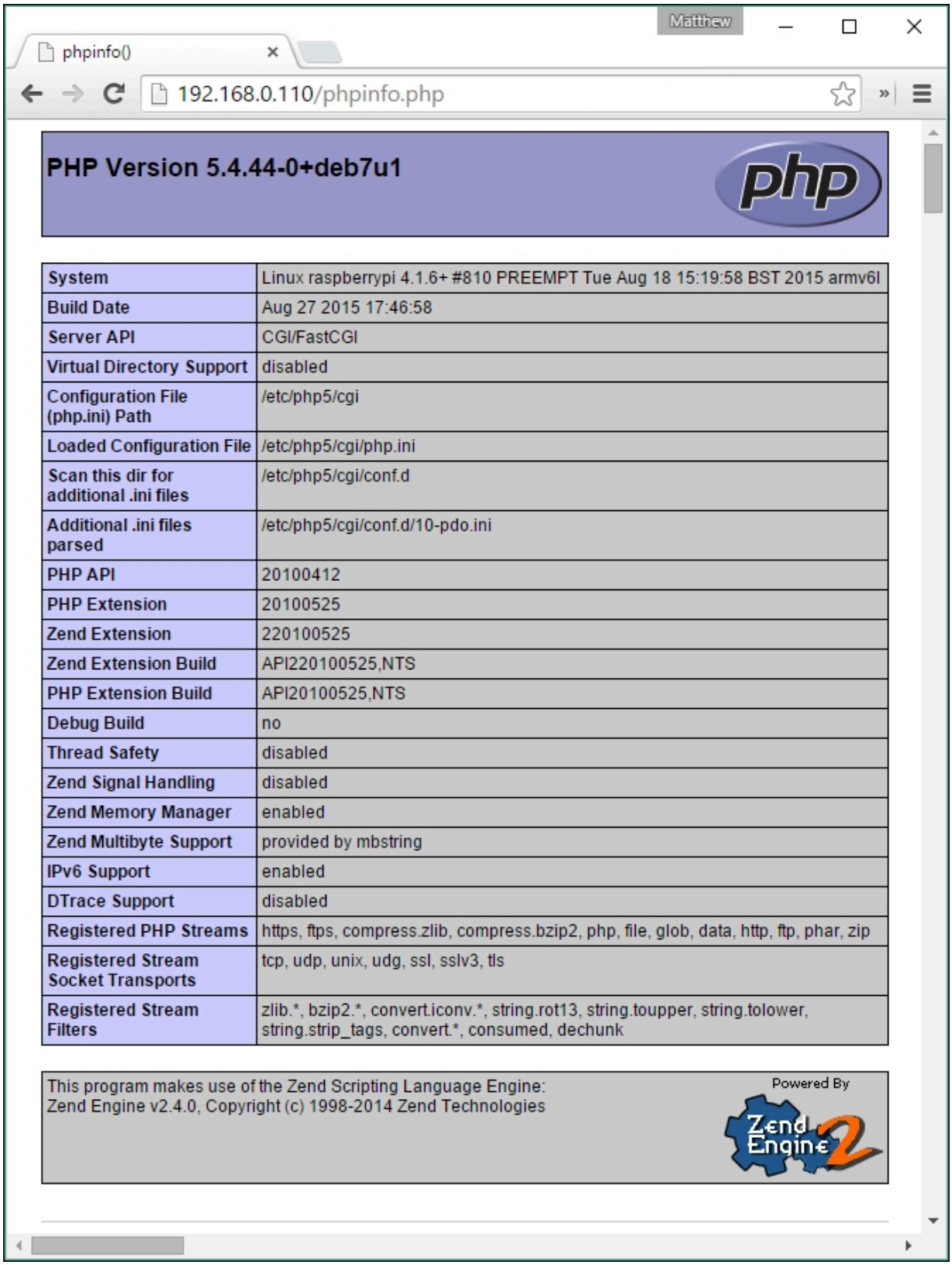
2. In Nano, create a file called `phpinfo.php`:

```
$ sudo nano phpinfo.php
```

3. In the editor, enter just the following single line, then save and exit from Nano:

```
<?php phpinfo(); ?>
```

Now, in your browser, enter the IP address of your Raspberry Pi followed by `/phpinfo.php`, for example, http://192.168.0.110/phpinfo.php, and you should be presented with the following page:



The PHP info page generated by the Web server

Now that we know our web server is working properly, we can start creating our console web page.

Being in control

So that we know what controls we want on our alarm control panel, we need to map out our system with the number of zone inputs and control inputs and outputs. As you'll remember from Chapter 3, *Extending Your Pi to Connect More Things* we can essentially have up to 16 zones in our system using the two I/O ports on our port expander. We also have the eight GPIO pins at our disposal on the Raspberry Pi board itself. So, let's now allocate these outputs and document them in the table that follows.

I'm going to set up an 8-zone system for my alarm inputs using port A on the I/O expander board, using the native GPIO pins for things such as buttons and alert outputs. One reason for doing it in this configuration is that the system can always fail-safe—so if the expander board fails, the Raspberry Pi can still communicate alerts and buzzers connected to it.

Port	**I/O Pin**	**Label/Purpose**
Expander A	0 (A0)	Zone 1 Input (Entry/Exit Channel)
	1 (A1)	Zone 2 Input
	2 (A2)	Zone 3 Input
	3 (A3)	Zone 4 Input

Port	I/O Pin	Label/Purpose
	4 (A4)	Zone 5 Input
	5 (A5)	Zone 6 Input
	6 (A6)	Zone 7 Input
	7 (A7)	Zone 8 – Anti-Tamper Loop Input
Expander B	0 (B0)	
	1 (B1)	
	2 (B2)	
	3 (B3)	
	4 (B4)	
	5 (B5)	
	6 (B6)	

Port	I/O Pin	Label/Purpose
	7 (B7)	
R-Pi GPIO	0 (GP0)	Arm/Disarm Switch (Input)
	1 (GP1)	
	2 (GP2)	
	3 (GP3)	
	4 (GP4)	Armed LED (Output)
	5 (GP5)	Arm/Disarm Buzzer (Output)
	6 (GP6)	Alarm LED (Output)
	7 (GP7)	Alarm Bell (Output)

Arming yourself

The terms *arm* and *disarm* are alarm system-speak for switching the alarm monitoring on (**arming** the system) and off (**disarming** the system). Zone 1 of our system is going to be linked to the arming and disarming part of the system as it will be connected to the

sensors on the door that we leave or enter from; this will be a special zone for **entry** or **exit** purposes.

When we set the alarm, we need a bit of time to get out of the house. The way that the system knows we've left the property is by monitoring the *exit* zone to see if we've opened and then closed the front door behind us within the time allowed.

Similarly, when we return, we will open the front door, but we don't want the alarm to go off straightaway—we need a chance to disarm the system within a given amount of time. We will arm and disarm the system via our web-based control panel, or by using a switch of some sort on the input GP0.

The master configuration file

Our system will use a **master configuration file** that will tell it how everything is set up and connected. This configuration file will be used by both the web control panel and the main alarm control scripts so that the two sub-systems can "talk" to each other. Let's create the file with our initial settings.

The settings file will be stored in the same location as where we will create our control scripts in Chapter 9, *Putting It All Together*, which is in the folder. `/etc/pi-alarm`. So, let's create this folder, and give it execute rights so that our scripts can be run:

```
$ cd /etc
$ sudo mkdir pi-alarm
$ sudo chmod 777 pi-alarm
```

We'll now create the master configuration file, to be used by our system, in this folder:

```
$ cd pi-alarm
$ sudo nano alarm.cfg
```

Tip

As before, you don't have to create your files in Nano on the Raspberry Pi—you can create them on your desktop computer, and then transfer them to your Pi using SCP.

```
# ALARM MASTER CONFIG FILE #

#Number of zones in the system
NUM_ZONES=8

#Display labels for each zone
ZONE_LABEL_1="Zone 1 - Entry/Exit"
```

```
ZONE_LABEL_2="Zone 2"
ZONE_LABEL_3="Zone 3"
ZONE_LABEL_4="Zone 4"
ZONE_LABEL_5="Zone 5"
ZONE_LABEL_6="Zone 6"
ZONE_LABEL_7="Zone 7"
ZONE_LABEL_8="Zone 8"

#Zones that are enabled
#Set to 0 to Disable or 1 to Enable
ZONE_ENABLE_1=1
ZONE_ENABLE_2=1
ZONE_ENABLE_3=1
ZONE_ENABLE_4=1
ZONE_ENABLE_5=1
ZONE_ENABLE_6=1
ZONE_ENABLE_7=1
ZONE_ENABLE_8=1

SYSTEM_ARMED=0

#Zone status
#Set to 1 if zone is triggered
ZONE_STATUS_1=0
ZONE_STATUS_2=0
ZONE_STATUS_3=0
ZONE_STATUS_4=0
ZONE_STATUS_5=0
ZONE_STATUS_6=0
ZONE_STATUS_7=0
ZONE_STATUS_8=0
```

`alarm.cfg` file

Creating the web page

Our Web-based control panel is going to be a single PHP-driven HTML5 web page which will be **mobile optimized**. HTML5 is the latest mark-up standard for web pages and is supported by most modern smartphones and browsers. We will also create a **cascading style-sheet** (**CSS**) that will make our page look half reasonable on mobile devices.

To create the web files, I recommend that you use something like the excellent Notepad++ on your desktop computer, rather than doing it directly on the Raspberry Pi. Alternatively, if you are a seasoned web developer, you may already have your IDE of choice.

The control panel HTML template

The first thing we'll do is create an HTML file that we can use to test our layout before we put the HTML into a PHP file to make it interact with our system. This makes it easier to tweak the way we want it to look beforehand, without the PHP scripts getting in the way.

Note

This is not a tutorial on Web development—there is a plethora of books out there on that subject—but I hope the code is clear enough for you to work out what's going on. The code I'm going to show you is fully functional, so you can just use what I give you without having to do any more. Hopefully, it makes your control panel look OK too!

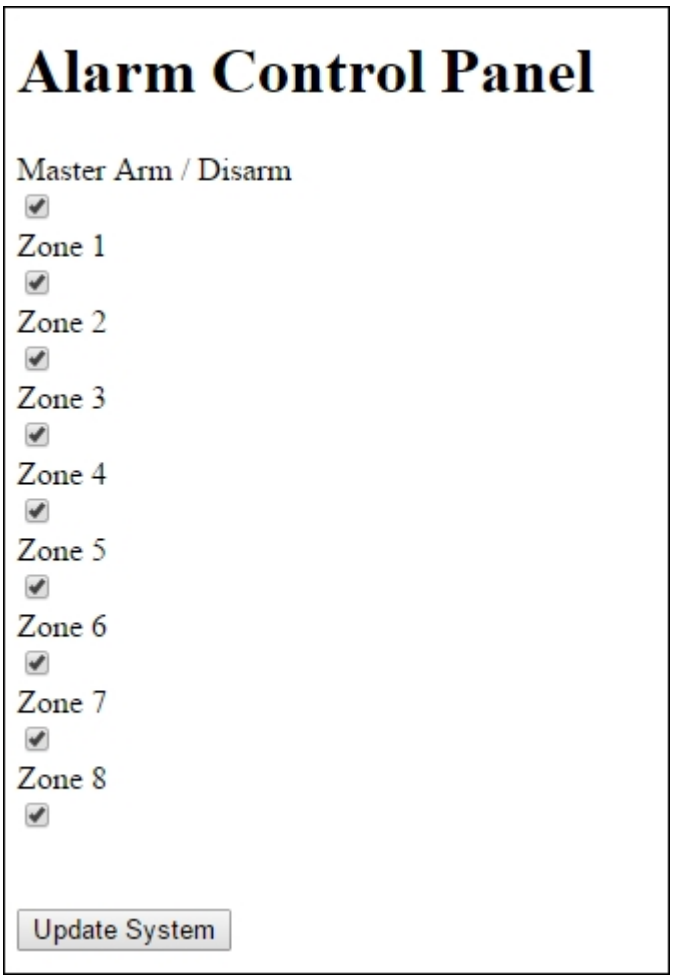
The following mark-up gives you a basic control panel with status for our 8 zones, a master arm and disarm switch, and switches to enable or disable any of our zones.

The `<head>` section of the code contains some `<meta>` tags that help mobile devices know that it's a mobile-friendly site. In the main `<body>` mark-up, we have a section for each zone that contains the zone's name and an on/off switch. Each zone is in its container so that we can also highlight a particular zone that needs our attention, for example, if it's triggered.

You can find the full HTML5 markup for our control panel in the `alarm-panel.html` file located inside the code folder of `chapter 7`.

Giving it some style

At the moment, this page doesn't look that great (in fact, it looks awful, like something from the 1990s); it isn't particularly good for mobile devices and would most certainly fail the *sausage test*. So, we're going to apply some styling to make it look not half bad. Although the preceding mark-up contains a reference to a CSS file—we haven't created that file—so this is what our page currently looks like (as I said: it looks awful):



The web control panel without any styling

The following CSS3 mark-up is designed specifically for our control panel, and it makes it look quite nice while also making it usable on **touch-screen** mobile devices. The CSS is quite long and seems overwhelming, but you don't need to do anything with it, or understand it, if you don't want to—the only thing you need to know is that it's been designed for modern browsers and smartphones, so don't expect it to work on Internet Explorer 7, or probably even IE9!

In essence, it contains the styling for the following:

- Preparing the browser for our mobile layout
- Our text and zone areas
- Creating cool switches instead of boring checkboxes
- Making an area flash on and off when we need it to

```
/* Clear browser margin and padding defaults */
body, div, dl, dt, dd, ul, ol, li, h1, h2, h3, h4,
h5, h6, pre, form, fieldset, input, textarea, p {
margin:0;padding:0;-webkit-text-size-adjust:none;
}

body {
  background: #ffffff;
  color: #4A5651;
  font-family: "Trebuchet MS", Helvetica, sans-
serif;
  font-size:10px;
  height: 100%;
  padding:0;
  margin:0 auto;
  max-width:320px;
  min-width:240px;
  text-align: left;
  width:100%;
  -webkit-box-shadow: 0px 20px 40px 0px
rgba(0,0,0,0.50);
  -moz-box-shadow: 0px 20px 40px 0px
rgba(0,0,0,0.50);
  box-shadow: 0px 20px 40px 0px rgba(0,0,0,0.50);
}

p, .zoneLabel {
  font-size:16px;
  margin:5px;
  line-height:1.4;
  color:#4A5651;
}

#header h1 {
  font-size:20px;
  line-height:40px;
  margin:0;
  padding:0 0 0 15px;
  text-align:center;
  text-overflow: ellipsis;
  font-weight:bold;
}

.zoneControl, .masterControl{
  border-bottom:1px solid #dddddd;
  margin-top:5px;
  margin-bottom:0px;
```

```
  padding:5px;
  display:block;
  width:100%;
}

.zoneLabel {
  font-weight:bold;
  text-overflow:ellipsis;
}

input[type="submit"] {
  border: none;
  background-color: #0b70cc;
  color: white;
  height: 32px;
  display: block;
  padding: 4px 7px;
  float: left;
  border-radius: 8px;
  position: relative;
  bottom: 1px;
  margin-left: 4px;
  text-align: center;
}
input[type="submit"]:hover {background-color:
#b2ceec;color: #0b70cc;border: none;border: 1px
solid #b2ceec;}

/* Flashing animation */
@-webkit-keyframes flash{0%, 50%, 100% {opacity:
1;} 25%, 75% {opacity: 0;}}
@keyframes flash {0%, 50%, 100% {opacity: 1;} 25%,
75% {opacity: 0;}}
.flash {
  -webkit-animation-name:
  flash;animation-name:
  flash;color:#f00000;
}
.animated {
  -webkit-animation-duration: 1s;
  animation-duration: 1s;
  -webkit-animation-fill-mode: both;
  animation-fill-mode: both;
  animation-iteration-count:infinite;
  -webkit-animation-iteration-count:infinite;
}

/*
```

```
  ON/OFF SWITCH STYLES
  The rather cool On/Off switch styling was
generated on
  https://proto.io/freebies/onoff/
*/
.onoffswitch {
  position: relative;
  width: 90px;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
}

.onoffswitch-checkbox {
  display: none;
}

.onoffswitch-label {
  display: block;
  overflow: hidden;
  cursor: pointer;
  border: 2px solid #FFFFFF;
  border-radius: 20px;
}

.onoffswitch-inner {
  display: block;
  width: 200%;
  margin-left: -100%;
  transition: margin 0.3s ease-in 0s;
}

  .onoffswitch-inner:before, .onoffswitch-
inner:after {
    display: block;
    float: left;
    width: 50%;
    height: 30px;
    padding: 0;
    line-height: 30px;
    font-size: 14px;
    color: white;
    font-family: Trebuchet, Arial, sans-serif;
    font-weight: bold;
    box-sizing: border-box;
  }

  .onoffswitch-inner:before {
```

```
    content: "ON";
    padding-left: 10px;
    background-color: #34C290;
    color: #FFFFFF;
  }

  .onoffswitch-inner:after {
    content: "OFF";
    padding-right: 10px;
    background-color: #EEEEEE;
    color: #999999;
    text-align: right;
  }

.onoffswitch-switch {
  display: block;
  width: 18px;
  margin: 6px;
  background: #FFFFFF;
  position: absolute;
  top: 0;
  bottom: 0;
  right: 56px;
  border: 2px solid #FFFFFF;
  border-radius: 20px;
  transition: all 0.3s ease-in 0s;
}

.onoffswitch-checkbox:checked + .onoffswitch-label
.onoffswitch-inner {
  margin-left: 0;
}
.onoffswitch-checkbox:checked + .onoffswitch-label
.onoffswitch-switch {
  right: 0px;
}

.masterswitch {
  position: relative;
  width: 90px;
  -webkit-user-select: none;
  -moz-user-select: none;
  -ms-user-select: none;
}

.masterswitch-checkbox {
  display: none;
}
```

```
.masterswitch-label {
  display: block;
  overflow: hidden;
  cursor: pointer;
  border: 2px solid #FFFFFF;
  border-radius: 20px;
}

.masterswitch-inner {
  display: block;
  width: 200%;
  margin-left: -100%;
  transition: margin 0.3s ease-in 0s;
}

.masterswitch-inner:before, .masterswitch-
inner:after {
  display: block;
  float: left;
  width: 50%;
  height: 30px;
  padding: 0;
  line-height: 30px;
  font-size: 12px;
  color: white;
  font-family: Trebuchet, Arial, sans-serif;
  font-weight: bold;
  box-sizing: border-box;
}

.masterswitch-inner:before {
  content: "ARMED";
  padding-left: 10px;
  background-color: #F00000;
  color: #FFFFFF;
}

.masterswitch-inner:after {
  content: "OFF";
  padding-right: 10px;
  background-color: #EEEEEE;
  color: #999999;
  text-align: right;
}

.masterswitch-switch {
  display: block;
```

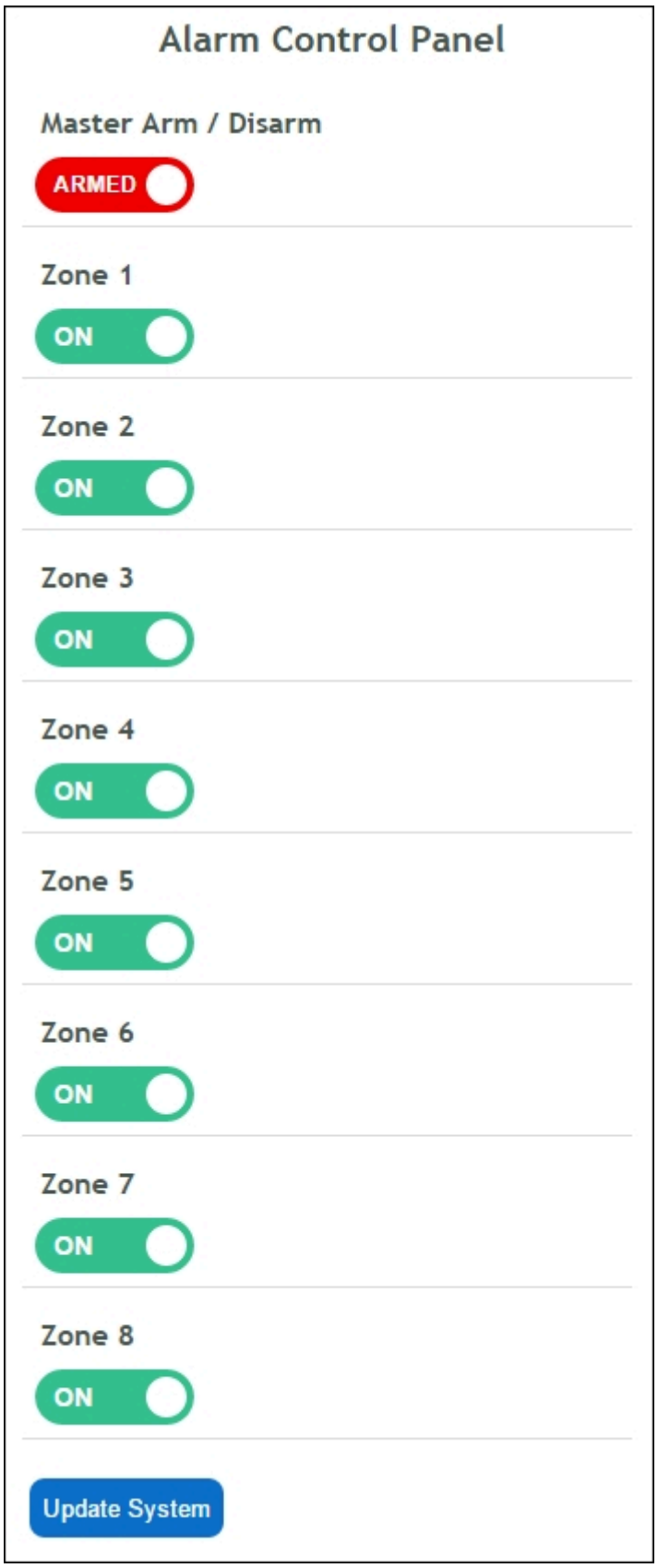
```
  width: 18px;
  margin: 6px;
  background: #FFFFFF;
  position: absolute;
  top: 0;
  bottom: 0;
  right: 56px;
  border: 2px solid #FFFFFF;
  border-radius: 20px;
  transition: all 0.3s ease-in 0s;
}

.masterswitch-checkbox:checked + .masterswitch-
label .masterswitch-inner {
  margin-left: 0;
}

.masterswitch-checkbox:checked + .masterswitch-
label .masterswitch-switch {
  right: 0px;
}
/* END OF SWITCH STYLES */
```

Web control panel style sheet – `alarm-panel.css`

Apply the stylesheet and this is what you end up with (a little bit nicer, I think you'll agree):


Alarm Control Panel
Master Arm / Disarm
ARMED
Zone 1
ON
Zone 2
ON
Zone 3
ON
Zone 4
ON
Zone 5
ON
Zone 6
ON
Zone 7
ON
Zone 8
ON
Update System

Making it dynamic

Now that we have the layout code defined for our control panel page, we can insert it in our PHP page so that it can be modified dynamically by the PHP script on the Web server, depending on the status of our home security system.

The PHP script will help us achieve the following basic functions:

- Updating the configuration file with the position of the on/off switches for zones
- Arming and disarming the system
- Telling us which zone has been triggered when an intrusion has been detected

Again, I'm not going to go into detail about how the PHP code works, but hopefully the comments within the code will help you follow what's going on, and also help you modify it if you want to change its behavior.

Getting a bit of help first

Unless you change some of the PHP configuration, it can be a nightmare trying to work out what's gone wrong if you have a small bug in your code, as basically you are presented with–nothing!

So, before we create and build our PHP page, we'll change a couple of settings in the PHP configuration file to make sure we know if there are any issues:

1. Open the configuration file with **Nano**:

```
$ sudo nano /etc/php5/cgi/php.ini
```

2. The file is a bit large and unwieldy, but battle your way through it, find these settings, and change them as follows:

```
error_reporting = E_ALL
display_errors = On
```

3. Save the file and exit Nano.
4. Finally, restart Lighty:

```
$ sudo /etc/init.d/lighttpd restart
```

The main PHP code

And here it is– But don't run it yet—there's still a bit more to do–

You can find the full main PHP code in the `index.php` file located inside the code folder of `chapter 7`. In our Web server content folder, we should now have the following files:

```
pi@raspberrypi ~ $ ls -1 /var/www
alarm-panel.css
alarm-panel.html
index.lighttpd.html
index.php
phpinfo.php
```

I'm someone else

Now, before we can actually open this PHP web page successfully, we need to be aware of the fact that the Web server, by default, actually runs as a different user called `www-data`. This means that it doesn't ordinarily have the right to perform certain operations; in particular, those that interact with the file system.

If you worked through the previous PHP script, you'll see that it actually executes some Linux commands to read and update our `alarm.cfg` file.

In the same way that we have to put `sudo` in front of many commands because we're not the root user, it is true for other users as well, including `www-data`. So, to give the Web server rights to

execute certain commands, we need to add it as a **sudoer**, using the **visudo** utility.

Run the utility to open the sudoer configuration file:

```
$ sudo visudo
```

At the bottom of the file, add the following line:

```
www-data ALL=(ALL) NOPASSWD:/bin/cat,/etc/pi-
alarm/update-alarm-setting.sh
```

Then save the file and exit.

The final thing we have to do is create a small **Bash script** that will handle the task of updating settings in our `alarm.cfg` file. The reason why we need to do this is because we're going to use the Linux `sed` command to update the file. The way that we are invoking the `sed` command means that it needs to create a temporary file. Unless we do a bit of work with configuring the Web server because of its file location context, it won't work. So, it's easier to create a stub Bash script that is called by the PHP script. In this way, the Bash environment deals with the temporary file context.

So, we'll create the following Bash script and save it in our `/etc/pi-alarm` folder:

```
#!/bin/bash
#/etc/pi-alarm/update-alarm-setting.sh
##########################################
# Provides access to the sed command from  #
# PHP as it needs write access to a temp   #
# folder.                                  #
# $1 - Setting Name                        #
# $2 - Setting Value                       #
##########################################

sed -i "s/^\($1\s*= *\).*/\1$2/" /etc/pi-
alarm/alarm.cfg
```

update-alarm-setting.sh

And then we need to give the script execution rights:

```
$ sudo chmod 777 /etc/pi-alarm/update-alarm-
setting.sh
```

This is what we should see in our `/etc/pi-alarm` folder at this time:

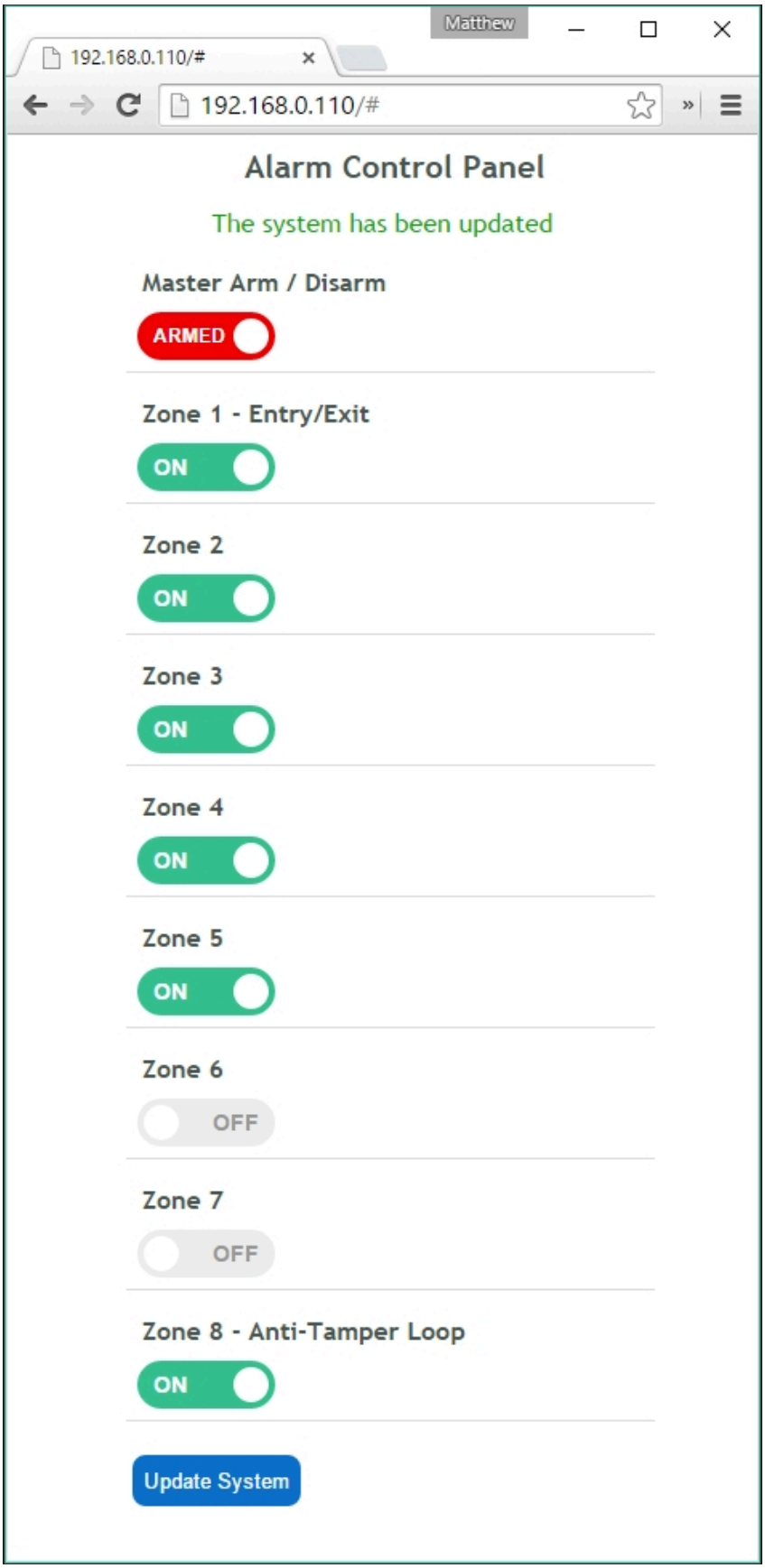
```
pi@raspberrypi ~ $ ls -1 /etc/pi-alarm
alarm.cfg
update-alarm-setting.sh
```

Right, after all that, I think we can now launch the control panel page in our browser at

`http://<my-pi-ip>`.

`index.php` is configured as a default page in Lighty's config, so you don't need to add it to the end of the URL; just the IP address will suffice.

By changing the switch positions and then clicking on the **Update System** button, you should find that the setting values get updated accordingly in `alarm.cfg`. You can now see how this file will be the way for the status to be exchanged between our web console and the security system scripts that we'll develop in Chapter 9, *Putting It All Together*.


Matthew
192.168.0.110/#
192.168.0.110/#
Alarm Control Panel
The system has been updated
Master Arm / Disarm
ARMED
Zone 1 - Entry/Exit
ON
Zone 2
ON
Zone 3
ON
Zone 4
ON
Zone 5
ON
Zone 6
OFF
Zone 7
OFF
Zone 8 - Anti-Tamper Loop
ON
Update System

The final operational control panel

Remote access to our control panel

While we can set up our system to receive email alerts when our system detects an intrusion, it would be really useful to be able to access our Web-based control panel wherever we are so that we can perhaps arm and disarm the system or switch off certain zones when we're not there.

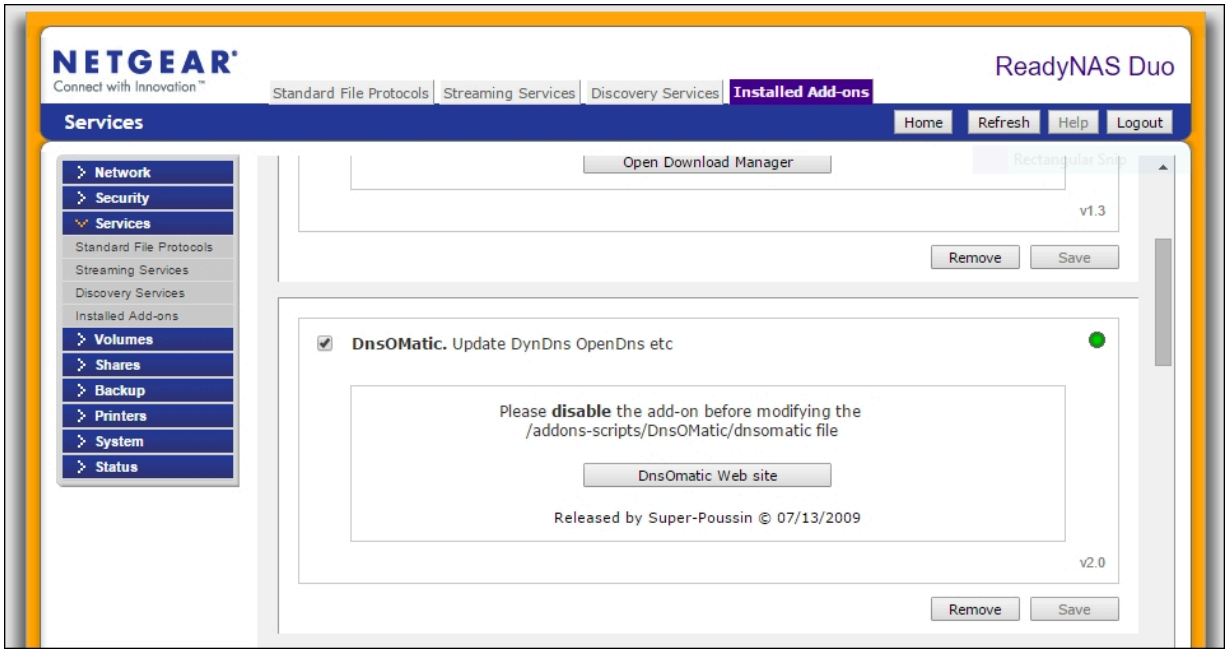
However, in order to make this possible we need to do a few things:

Setting up a dynamic DNS account

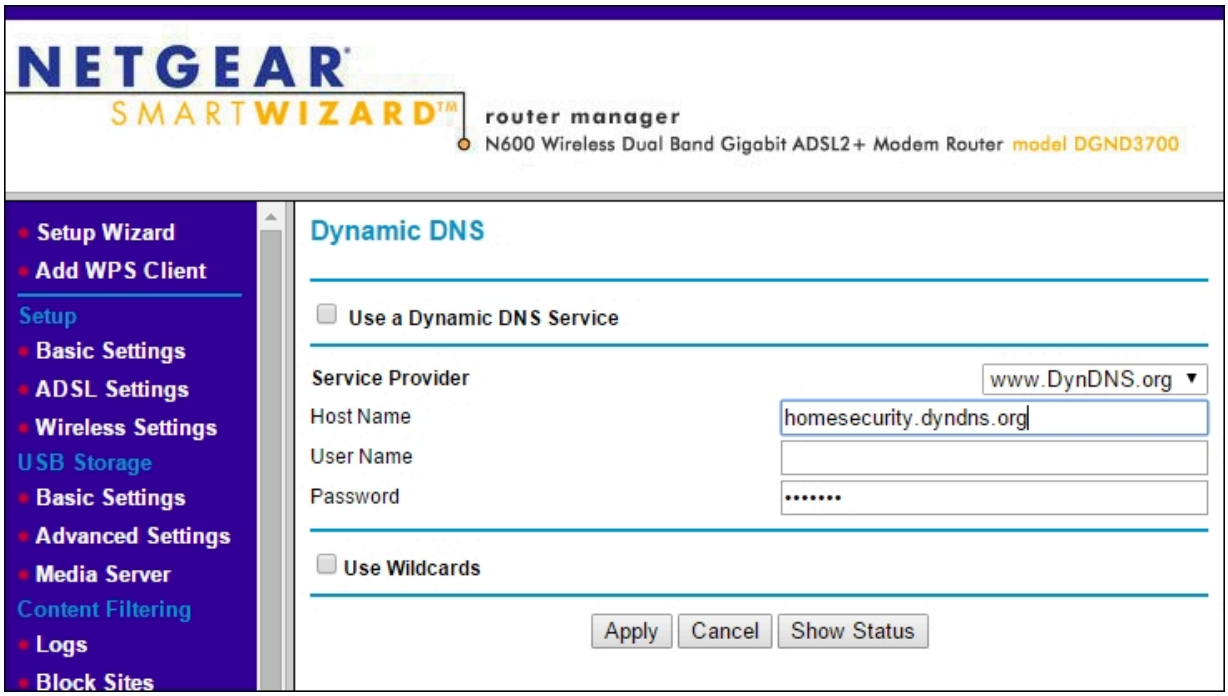
Most of us won't have a **fixed IP address** for the Internet connection in our home; it is likely to change from time to time, especially when we reboot or unplug our router, whereby our Internet service provider assigns us a new one when we next connect to them. Because of this, we can't rely on using the IP address to get to our home network when we're out and about. To solve this, we need to set up a **dynamic DNS** account that will allow us to set up a domain name for our home network (for example, *myhomenetwork.com*).

It works by having a service that runs inside your network, such as on your router or laptop, that updates the dynamic DNS service hosting your domain name with the current IP address of your Internet connection. Then, when you use your domain name in your browser, it will take you to a Web server on your home network.

Popular dynamic DNS providers out there include No-IP (www.noip.com) and DynDNS (www.dyn.com). You can also get a free DnsOMatic account with OpenDNS to manage your services (www.dnsomatic.com).



My Netgear NAS device has a DnsOMatic updater service add-on



My Netgear Router has the option of updating a Dynamic DNS service

The Raspberry Pi dynamic DNS client

Since your Raspberry Pi-based home security system is likely to always be on, you might want to install the **ddclient** updater service on there instead:

```
$ sudo apt-get install ddclient
```

Once installed, you can set it up for your particular service and account details using the following config file:

```
$ sudo nano /etc/ddclient.conf
```

Setting up a static IP on your Raspberry Pi

So that our home network always knows where to find your Raspberry Pi, we need to set up a **static IP address** on it, assuming that it currently acquires an IP address from your router's DHCP server each time it boots up.

1. To do this, we need to edit the network settings on the Raspberry Pi. In Nano, open the following configuration file:

   ```
   $ sudo nano /etc/network/interfaces
   ```

2. You'll probably find the Ethernet port configuration set to something like this:

   ```
   auto eth0
   allow-hotplug eth0
   iface eth0 inet manual
   ```

3. Change this configuration to be an unused static IP address on your network. In my case, I've set it to `192.168.0.99`. The gateway setting is the IP address of my Internet router:

   ```
   auto eth0
   allow-hotplug eth0
   iface eth0 inet static
           address 192.168.0.99
   ```

```
netmask 255.255.255.0
gateway 192.168.0.1
```

4. Now, we need to restart the networking service—note that you'll be disconnected from your terminal session. You'll need to reconnect using the new IP address:

```
$ sudo /etc/init.d/networking restart
```

If you have any issues, simply restart the Pi with `sudo reboot` and all should be good when it comes back up.

Port-forwarding

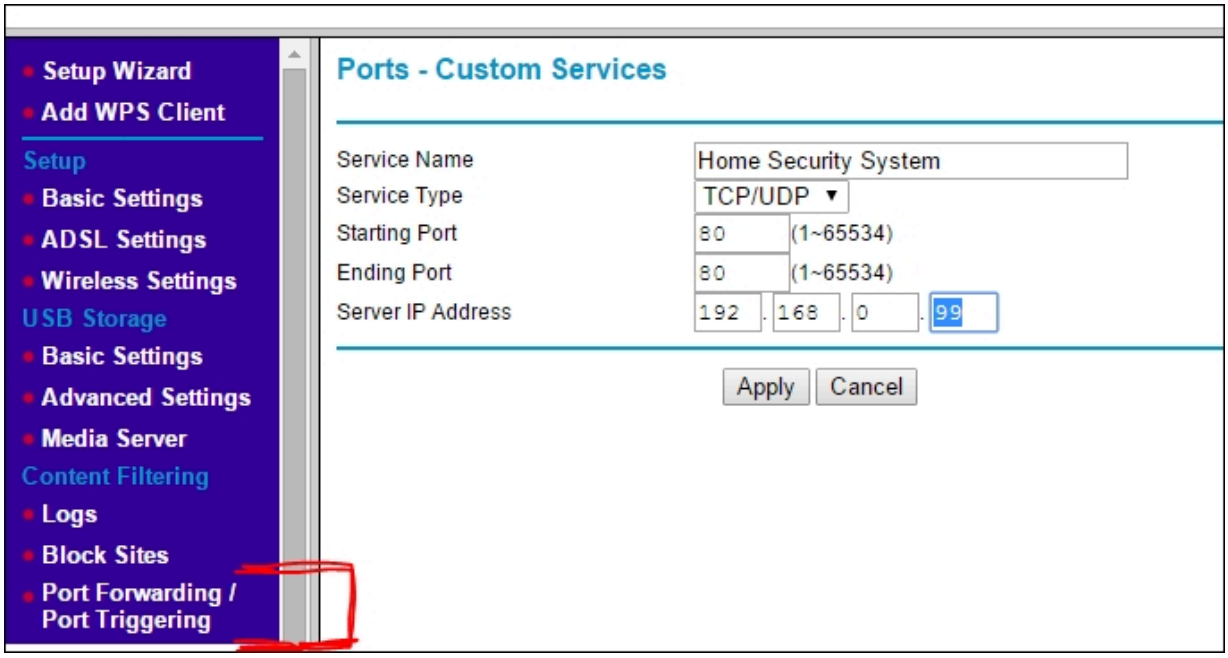
The final piece of this puzzle is to make sure that our Internet router will direct incoming traffic on a given port to our Raspberry Pi's Web server. For the purpose of this example, I'm going to assume that we are going to stick to the default, port 80, on our Web server.

Tip

A word about security

Given that our Web server will now be accessible from the outside world, we need to be mindful about securing our system properly. The two main ways to do this are to change the Web server port to a random number other than 80 (for example, 8799) and add password protection to your site by applying basic authentication. Both of these can be done in the `lighttpd` configuration file.

Most routers will allow you to set up **port-forwarding** as part of their **firewall** configuration. Essentially, setting this means that any incoming traffic from the Internet on a given TCP port will be allowed to pass through the router and will be directed to the device with the specified IP address. On my Netgear router, it's set up as shown in the following screenshot:



Setting up port-forwarding on a Netgear router

Now, when you enter your personal domain name in your browser, when you're away from home you should be taken to your alarm control panel.

Tip

You might also want to consider opening up port 22 so that you can access the Raspberry Pi directly using PuTTy and SSH from outside your network.

Summary

We've now started building the software that will control our home security system by determining the format of the main configuration file. We've also installed a Web server and built a basic single-page control panel with PHP, HTML5, and CSS3, which can be accessed nicely on our mobile phone, allowing us to configure our system and view the status.

In addition, we've learned how to configure our home network and Raspberry Pi so that we can access our control panel when we're away from home.

In Chapter 9, *Putting It All Together*, we'll put all of the electronic elements together and write the main scripts that will run the home security system. But before that, in the next chapter, we're going to look at a few other bits and pieces, such as adding other sensors, not necessarily related to intruder detection, to our home security system. We'll also look at how we can administer our entire Raspberry Pi system remotely using a Web browser, in addition to accessing our home security control panel.

Chapter 8. A Miscellany of Things

The previous chapters have provided us with the foundation and elements to design and put together our entire home security system, which we will do in the next chapter. I hope that I've guided you through this journey in a fairly structured and logical way so that you are ready to do that.

Beforehand, though, I'm including this chapter dubbed a *Miscellany of Things*, as that's exactly what it is. It comprises a few optional, but useful, extras that we should consider for our system, but that don't really warrant a whole chapter in their own right. I guess you could refer to them as footnotes to previous chapters.

As such, we will take a look at the following topics:

- Ways to arm and disarm the system without the web-based panel
- Driving inductive loads safely from our GPIO outputs
- Adding an escaped water sensor input to our system
- Adding a temperature sensor input to our system
- How carbon monoxide detectors could be added to our system
- Remotely managing our Raspberry Pi using Webmin

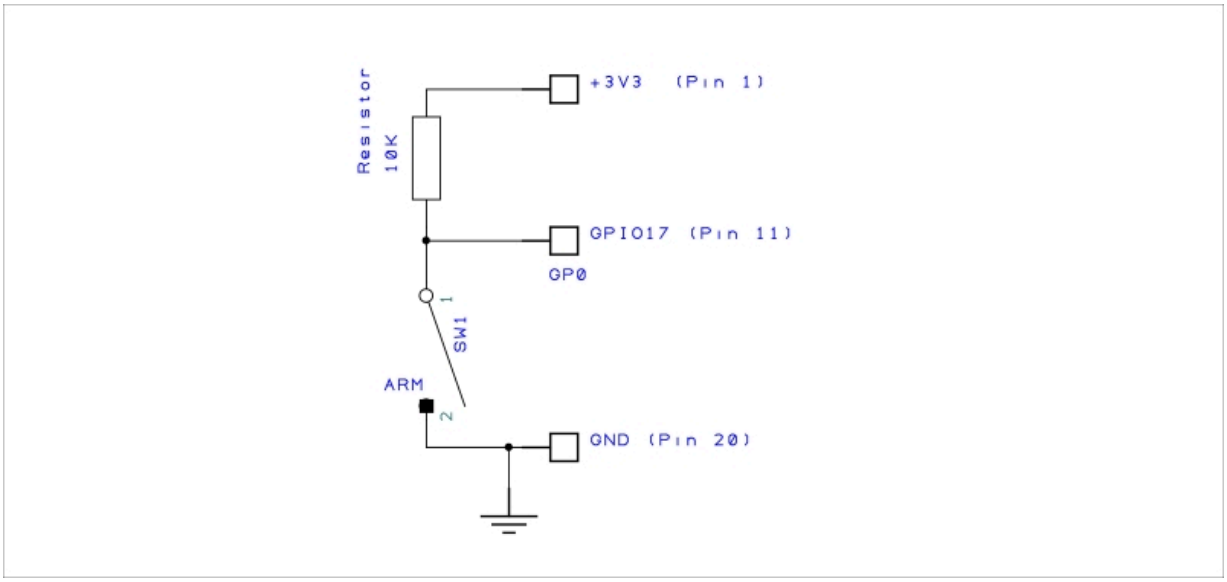
Arming and disarming the system

We've included a switch on our Web-based control panel so that you can arm and disarm the system from your smartphone. However, this is probably not the most convenient way of doing it, especially when you're rushing out of the house, or you've returned home with a phone whose battery is flat. So, we need to find an additional way

of arming and disarming our system at the entry and exit point of our property.

In the zone list table in the previous chapter, you'll notice that I assigned input GP0 on the Raspberry Pi GPIO as our arm/disarm switch input. This input will work in conjunction with our control panel switch.

This input can as be a simple as a toggle switch, or a bit more secure, such as a **key switch** or **electronic keypad**. Either way, it will be wired to ground GP0 (GPIO17) on our Raspberry Pi when the system is armed.



The circuit diagram for our arm/disarm switch

If you have switches or other such devices that will be outside and exposed to the elements, you'll need to ensure that they are suitable for outdoor use so that they don't get damaged and compromise the integrity of the system.



The IP67-rated key switch, suitable for outdoor use (type Lorlin WRL-5-E-S-2-B)

By using a standalone **security keypad**, you can allow each user to have their own code to arm and disarm the system. For example, the CDVI ECO 100 is a low-cost keypad that allows up to a 100 users. When the correct code is entered, it will arm the system by closing an internal switch. When the code is entered again, the keypad will disarm the system by opening the switch.

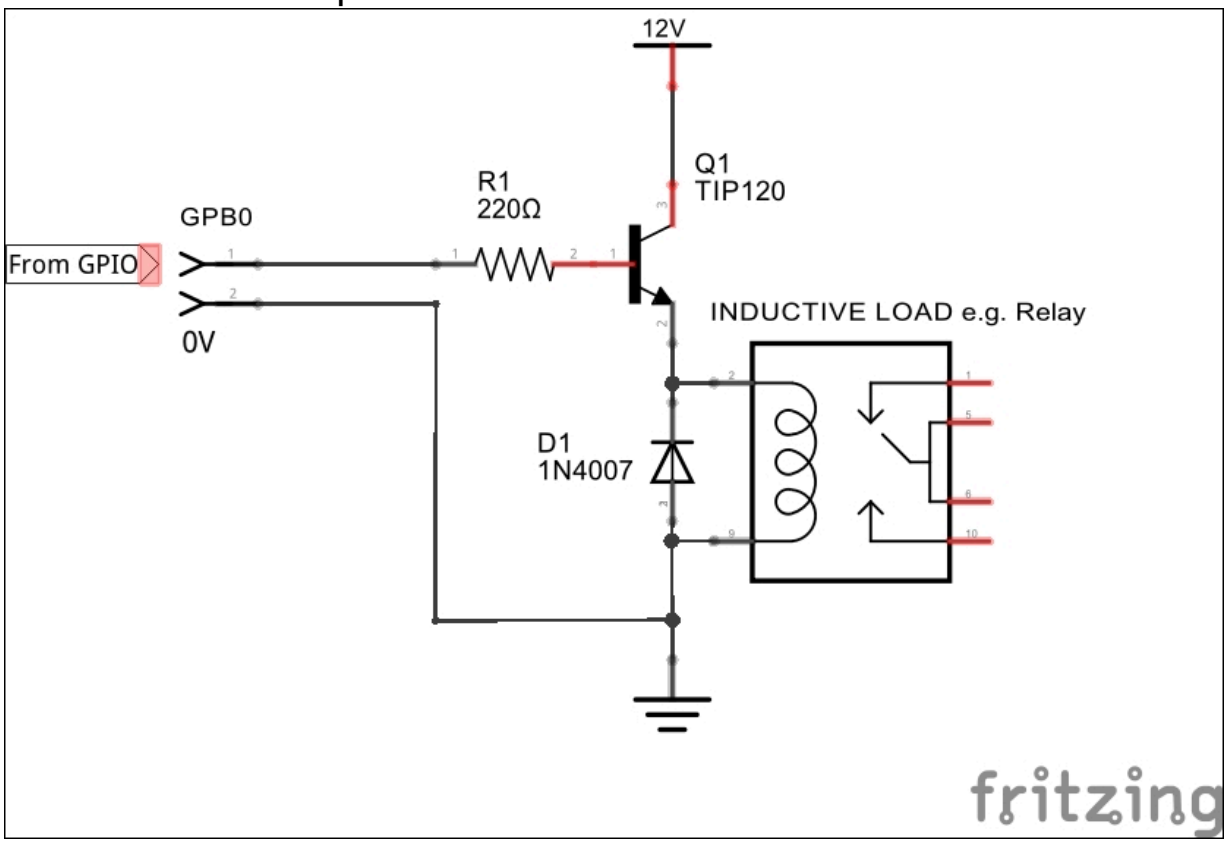


The CDVI ECO 100 programmable keypad

Driving inductive loads

I talked about driving large loads in Chapter 6, *Adding Cameras to Our Security System*, but now is probably a good time to expand on this a bit and talk about driving **inductive loads** such as **bells** and incandescent **lamps**. In the previous circuit example, I used the TIP120 Darlington transistor to drive an LED array that was not inductive. With inductive loads, you need to add a bit of diode protection to protect the circuit against spikes generated by the coils within relays and bells as they switch on and off.

Here's the modified circuit for our digital load driver with a 1N4007 **rectifier diode** for protection:



The digital load driver with diode protection

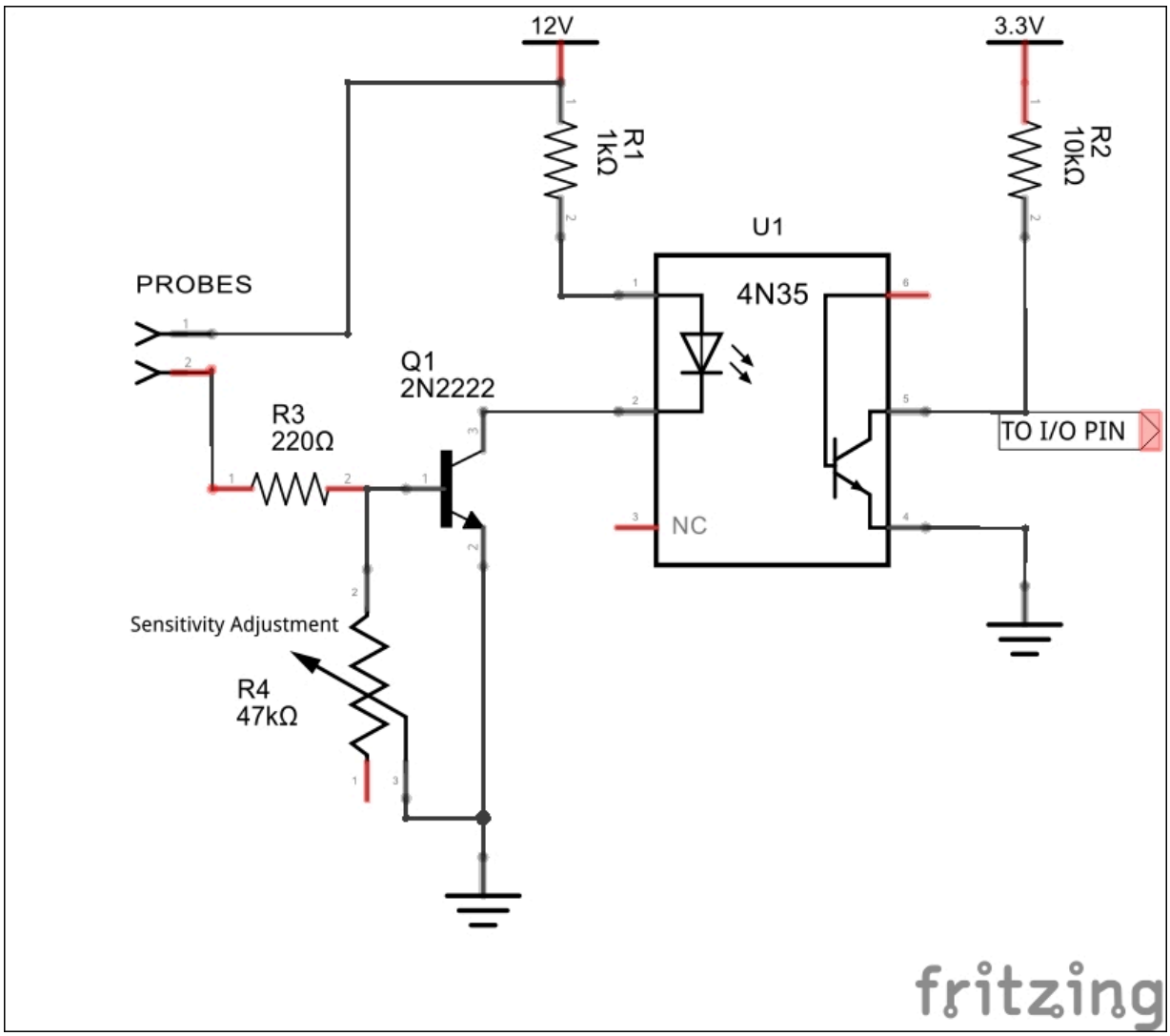
Beyond intrusion

Home security is not just about protecting our property against intrusion, it's also about protecting against other risks too, such as flood, fire, carbon monoxide leaks, and so on. So, it makes sense to extend our home security system to detect these other risks too.

You may choose to set up the system so that certain types of alerts only come to your phone as emails, rather than triggering all of the outside bells, lights, and whistles. This can be done by adapting the scripts in the next chapter so that they operate how you want them to.

A simple water detector

There's nothing worse than being away for a few days and coming home to a flooded kitchen because a leak has developed under the sink. Our simple circuit will detect the presence of water and trigger an input on our home security system, which can then alert you. You can also buy kits and ready-built modules to do this, but the following circuit is cheap and features our opto-isolator as we're going to have a different voltage for our actual detector.



The circuit for a simple water detector, isolated from our GPIO input

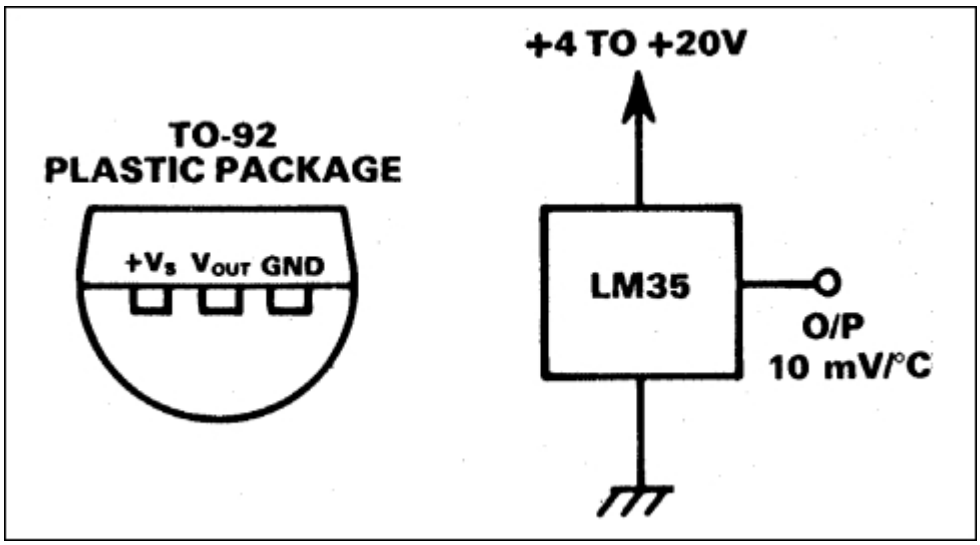
How it works

When water is placed across the probes, current flows through the water, and so, through the R3/R4 potential divider on the base of transistor Q1. When the current at the base is high enough to saturate it, the transistor will switch on fully, allowing the LED inside the opto-coupler to switch on. This in turn will pull down the input pin to our system to ground via the photo-transistor inside the optocoupler.

You can use the trimmer, R4, to calibrate the sensor by adjusting its sensitivity. Any generic NPN bipolar transistor should work here, but obviously, they all have different operating parameters, so choose a suitable one.

A simple temperature sensor

If we want to be alerted when the ambient temperature reaches a certain threshold, then we can build a circuit using the commonly used LM34/LM35 temperature sensors. It's a simple device with just three pins: power, ground, and output, providing a voltage proportional to the temperature. The difference between the LM34 and LM35 is that the LM34 produces an output of 10mV/–F, whereas the LM35 produces 10mV/–C. There is also an LM335 variant that produces an output of 100mV/–K.



Pinout taken from the Texas Instruments LM35DZ datasheet

It may have occurred to you at this point that this is an analogue device—so how do we interface that with our wholly digital system? One way is to incorporate an analogue-to-digital interface onto our input control board and read the data coming in from that so that we know the exact temperature, but that's probably a bit beyond the scope of this book. So, we're going to implement a circuit that will alert us when the temperature exceeds a pre-defined threshold,

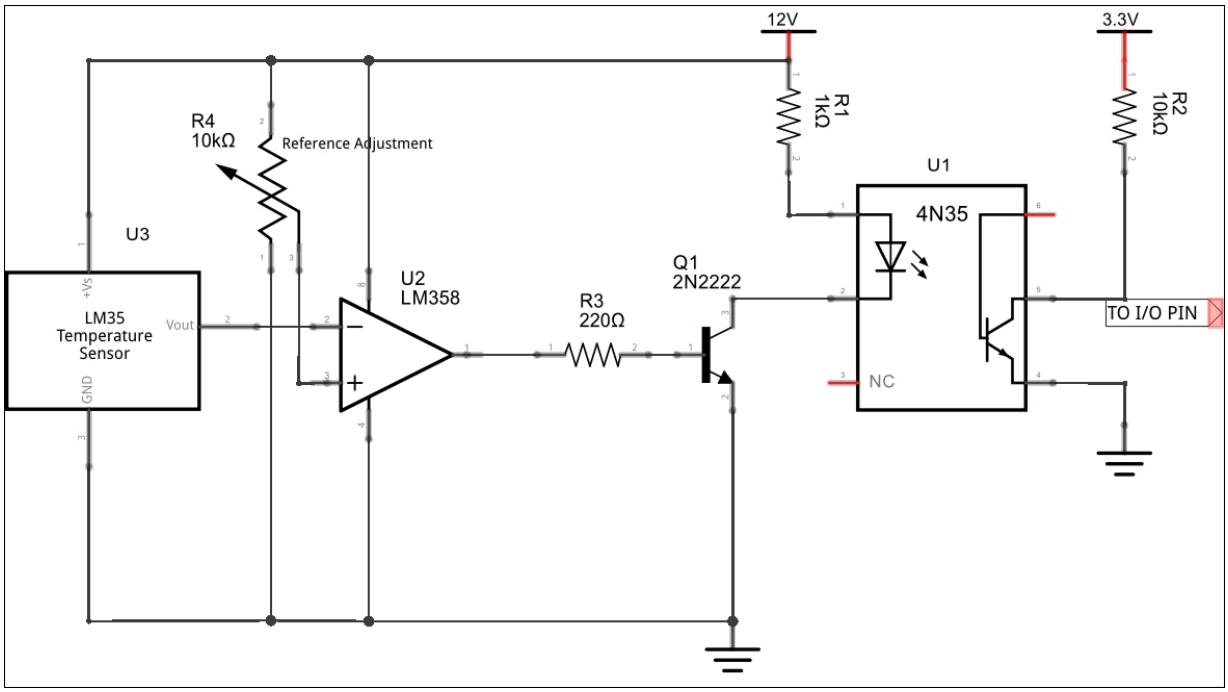
which is probably all we need in the context of our home security system.

Note

If you're interested in building an analogue-to-digital module to extend your home security, then take a look at something such as the PCF8591 chip from NXP, which is an I2C-based analogue-to-digital converter. This will connect to the I2C bus that we're already using, and so it is effectively just an add-on.

http://bit.ly/NXPPCF8591T

For our temperature detector circuit, we will use an operational amplifier configured as a comparator that will trigger our opto-coupler input when the pre-set temperature is reached. So, for fire detection, we might want to detect when the ambient temperature has exceeded 50–C.



The temperature threshold sensor to drive our digital input

How it works

The reference voltage is set by the variable resistor, R4, which forms a voltage divider between the 12V and the ground. This essentially means that the reference voltage on the +ve input of the op-amp comparator can be between 0 and 12V. Assuming that we want to detect when 50– is reached, we will need the op-amp to trigger when the –ve is 500mV (10mV/–C).

In our circuit, the output of the op-amp is high in its normal state, which keeps the opto-coupler on. However, when the threshold is reached, the output of the op-amp is driven low, switching off the transistor Q1, and hence, the opto-coupler. This pulls our alarm input high via resistor R2.

A carbon monoxide detector

It's entirely possible to build smoke and carbon-monoxide detectors that we can connect to our home security system in a similar way to the previous sensors, although they are a little bit more complex as they can require special handling. The SparkFun MQ-7 **Carbon Monoxide** (**CO**) detector (which is actually made by Winsen Electronics) can be implemented in a similar way to our temperature sensor, triggering an alarm input when a particular threshold is reached.



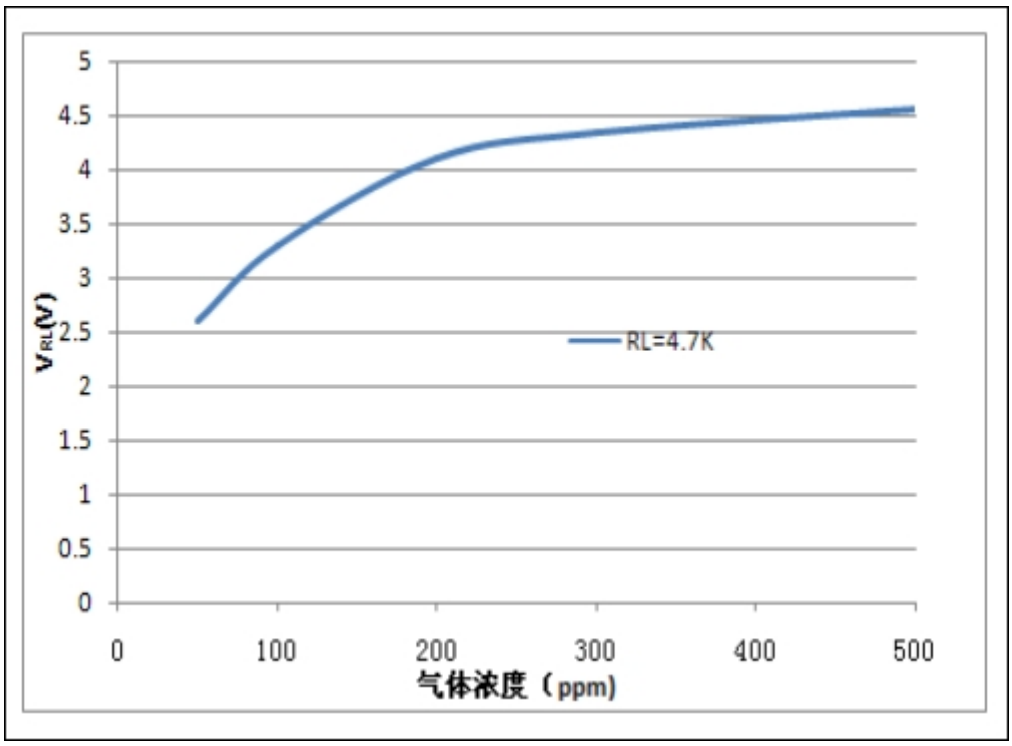
The Winsen MQ-7 carbon monoxide gas detector, available from SparkFun.

Note

The maximum safe continuous exposure to carbon monoxide (CO) is 9ppm (parts-per-million) according to ASHRAE (www.ashrae.org), and you should certainly not be exposed to

CO higher than this for prolonged periods of time, with 35ppm being the absolute maximum for a normal 8-hour working day.

The MQ-7 detector has a sensitivity of between 10 and 500ppm, so in my mind, I'd want to be alerted as soon as it picks up anything, therefore we should set our comparator's reference voltage to the lower end of the scale, in accordance with the sensitivity curve taken from the datasheet, shown as follows:



The sensitivity curve taken from the Winsen MQ-7 manufacturer's datasheet.

Note

Warning

I've included this section on carbon monoxide detection more for interest than anything else. It's nasty stuff, and while rolling out your own detector is OK for interest's sake, please keep it just for that. It's useful to have this in our home security system to alert

us when we're out of the house as an addition, but this *should not* be a replacement for a commercially available one that sits next to your boiler with all of the certifications, standards, and so on, and makes a very loud noise when we're in the house.

Remote administration for our Raspberry Pi

In the previous chapter, we learned how to set up our system and home network so that we can remotely access the alarm control panel from wherever we are. I'm now going to show you how to extend this to be able to administer and monitor our entire Raspberry Pi system.

Getting Webmin

Webmin is a rather fine and well established web-based interface for administering Unix/Linux systems. You can find everything about Webmin on its website at www.webmin.com. I'm assuming, as throughout this book, that you are using the Raspbian distribution on our Pi when it comes to installing Webmin.

There are a couple of ways to install Webmin: either by manually downloading and unpacking it, or by updating our repository sources so that we can use `apt-get`. I'm going to opt for the latter, so any dependencies are automatically installed and updates can be managed more easily in the future. There are a few steps, but it's pretty straightforward:

Updating the repository sources

1. The first thing we need to do is update our repository sources to include the Webmin repositories:

```
$ sudo nano /etc/apt/sources.list
```

2. Add the following two lines to the end of the file:

```
deb
http://download.webmin.com/download/repository
sarge contrib
```

```
deb
http://webmin.mirror.somersettechsolutions.co.
uk/repository sarge contrib
```

3. Save and exit Nano.

Importing the signing key

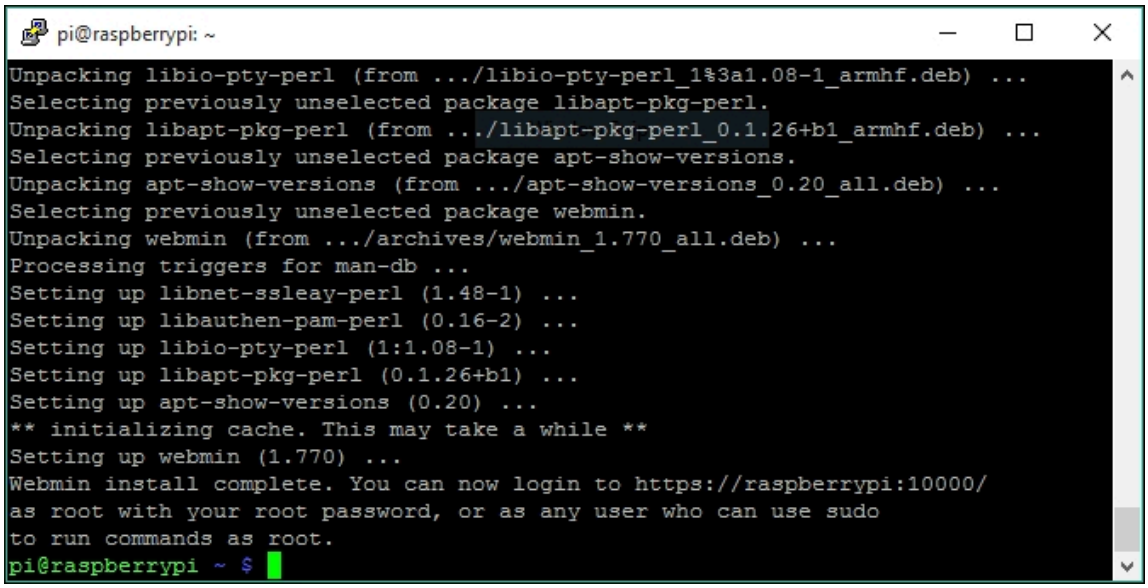
1. Next, we need to download and import the repository's signing key:

```
$ cd ~
$ sudo wget http://www.webmin.com/jcameron-
key.asc
$ sudo apt-key add jcameron-key.asc
```

2. Now that we have everything we need, we can update the package installer and install Webmin. It can take a while, so you might want to go and make yourself a cup of tea or coffee:

```
$ sudo apt-get update
$ sudo apt-get install webmin
```

3. Once it's installed, you should see the following message in the shell window:



Webmin installation

Accessing Webmin locally

Webmin, by default, runs on port 10000 and uses the secure HTTP**S** protocol; so, to access it, you need to enter the following URL in your browser:

```
https://<my-ip>:10000
```

Where `<my-ip>` is the IP address of your Raspberry Pi.

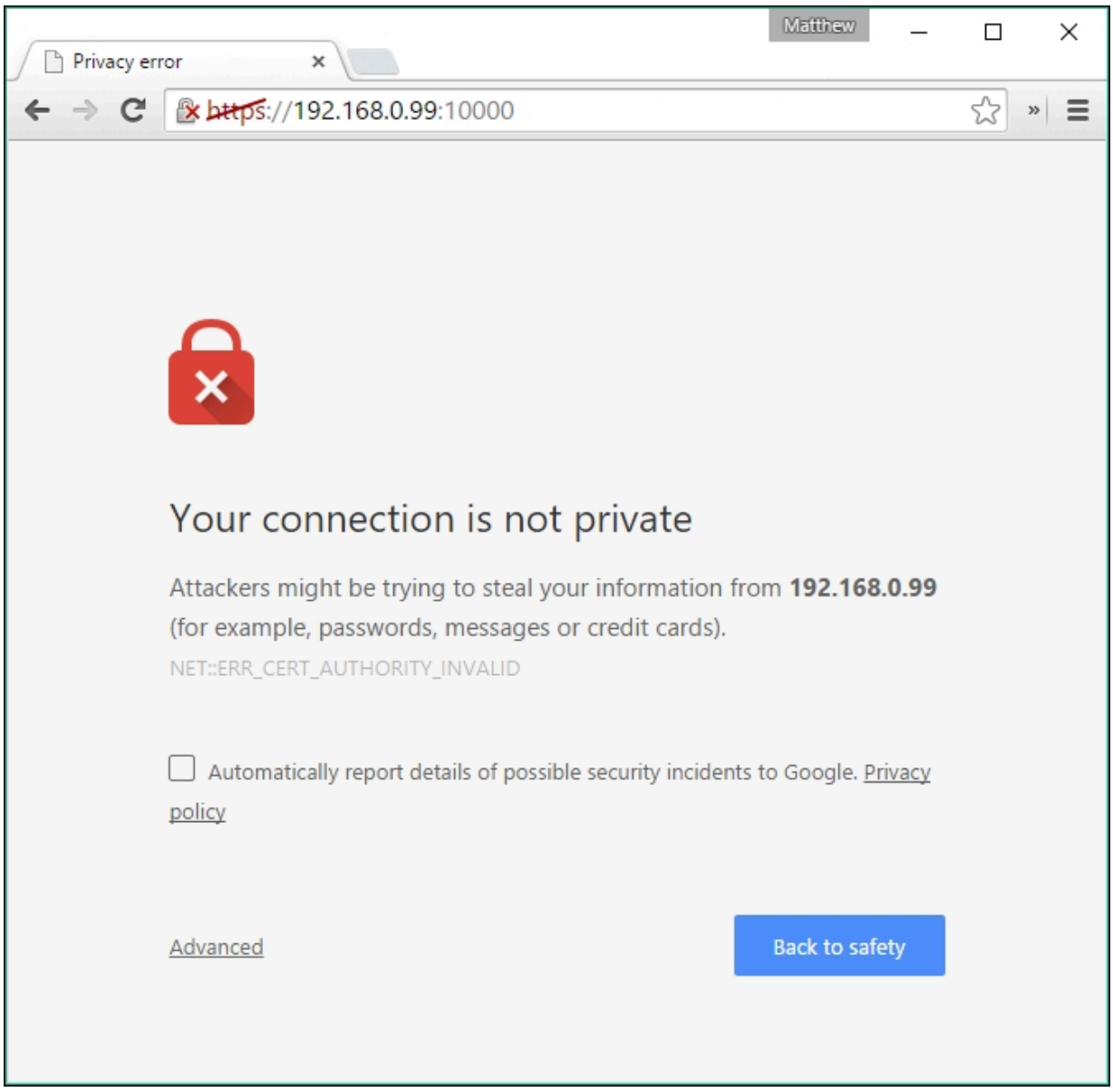
In the previous chapter, we set up a static IP address on our system; in my case, I set up the address as 192.168.0.99. So, to access Webmin on my system, I would use:

```
https://192.168.0.99:10000
```

Note

HTTPS Privacy Errors

In some browsers, such as Google Chrome, you might see a privacy error as you try to access the Webmin Web page. This is because the SSL certificate behind the HTTPS connection is not signed by a known authority. This is fine—just tell your browser that you want to accept this and proceed (in Chrome, you need to click on the **Advanced** link first to access that option).

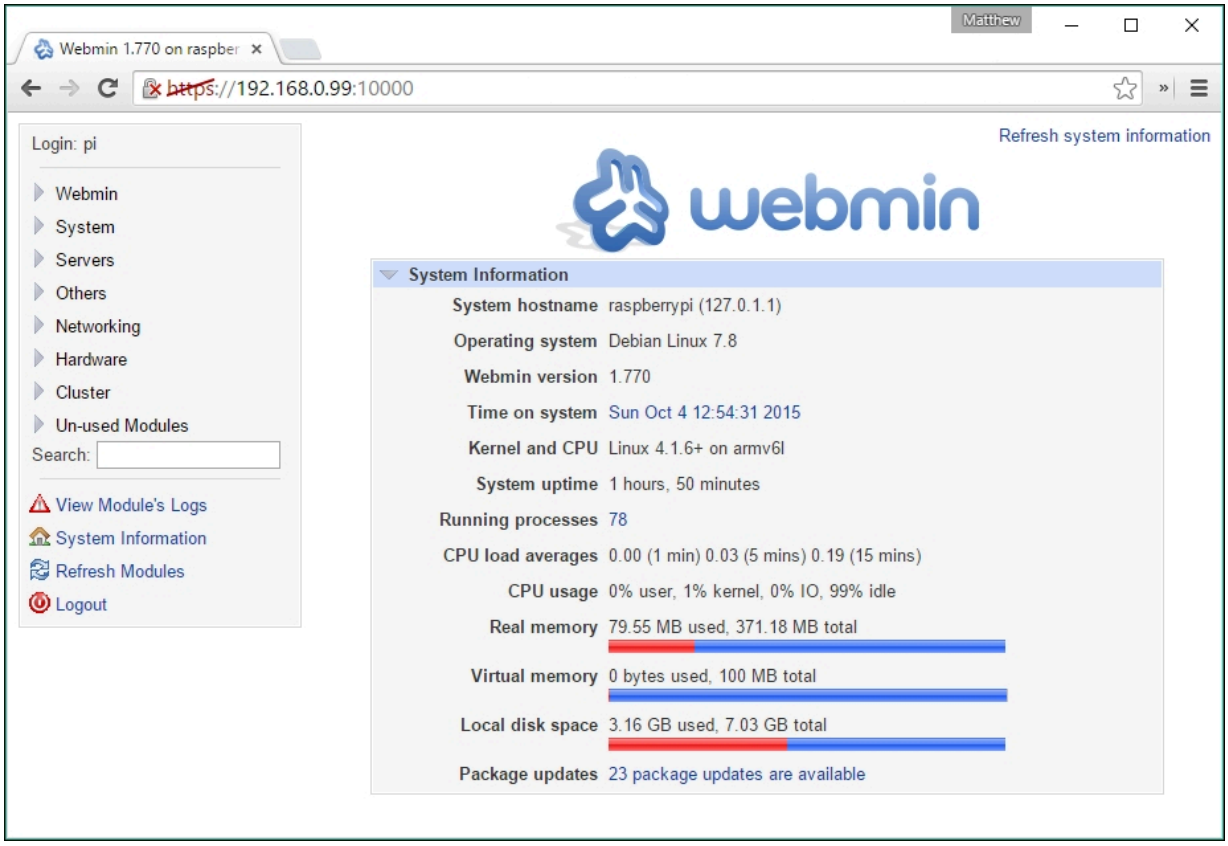


You can log into Webmin using the **root** or **pi** user account, or any other account that has **sudo** rights:



Webmin login

Once logged in, you'll be presented with the main system information page. Have a good poke around in it because there's lots of useful stuff you can see and do.



Webmin system information view

Webmin comes with a lot of modules, and not all of them are installed; therefore, you might want to explore the **Un-used Modules** section of the panel to see if there is anything you want to add to Webmin.

Remotely accessing Webmin

In the same way that we set up remote access for our alarm control panel in the previous chapter, you can do it with Webmin—just set up port-forwarding on your router for port 10000. You can then access Webmin from anywhere using `https://<my-public-ip>:10000`.

Summary

Well, this has been a bit of a mix-and-match of various topics to end on before we put together our home security system framework. I hope you enjoyed these various footnotes to previous chapters, and that it's given you some ideas on how far you can take your home security system.

We started by looking at ways we can arm and disarm our system without having to access the Web-based control panel, by adding a mechanical or digital switch to an arm/disarm input.

We then looked at adding analogue-type sensors to our system, which can alert us when a threshold has been reached by using operational amplifiers set up as voltage comparators. The idea behind these comparator circuits can be implemented for different types of sensors where you want to know when a certain voltage threshold has been reached at the analogue sensor output.

Finally, we learned how to install Webmin on our Raspberry Pi so that we can monitor and configure many aspects of the Linux operating system.

The next chapter is the moment we've all been waiting for; we're going to take all of the elements and concepts from the previous chapters and put together our full system comprising the elements we want to feature. The star of the show will be our Bash scripts, which will glue together all of these elements and provide the control logic for the entire system.

Chapter 9. Putting It All Together

Over the past eight chapters, we've explored the elements and concepts of a full-featured home security system that you'd expect to have installed in your property. It's been presented in a modular fashion so that you can choose which features you want for your system, to allow you to make it as compact and basic or large and complex as you require.

Fundamentally, the idea behind a home security system is to detect whether particular zone inputs are triggered high or low by an external sensor, be that a switch, motion detector, or water detector. At the end of the day, as far as the control software is concerned, the type of sensor is irrelevant and the system software's job is to simply check the state of its inputs and alert accordingly.

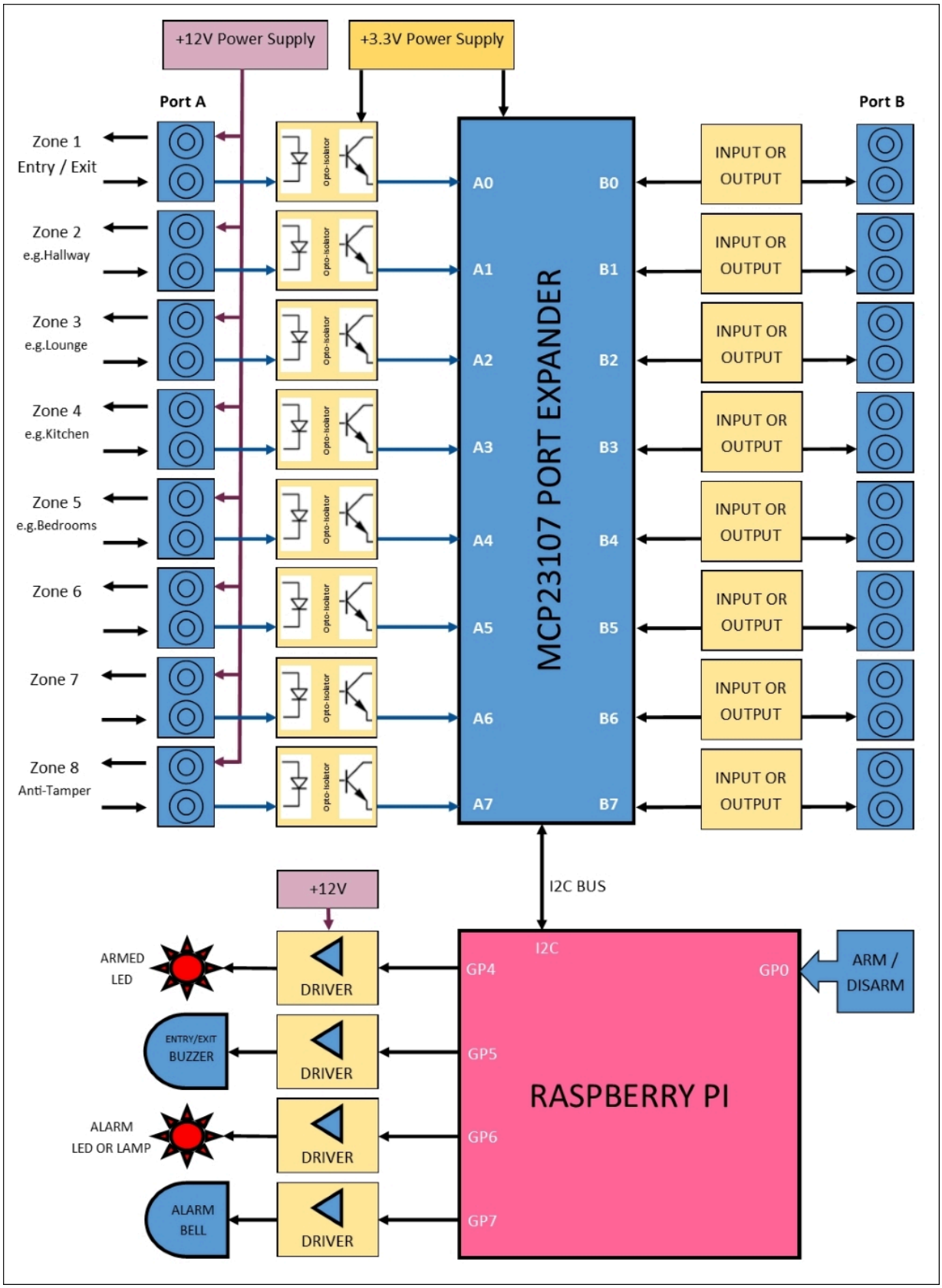
In this final chapter, we're going to put all of the concepts together to come up with a security system framework and write the control scripts around it. This is what we will cover:

- Defining a high-level overview of our system, detailing the connected elements
- Building the entire modular security system framework control script, exploring the code in detail
- Delving into some detailed shell scripting techniques to perform certain tasks
- Learning how to make our system automatically start at boot-time
- Preventing the burning out of our SD card by creating a RAM-based file system

Alarm system diagram

So that we don't get lost in this process, the first thing I recommend is to come up with a complete system diagram that we can follow. I do this for any system I design and put together so that it can be built in a structured way, and easily documented and modified.

For the home security system in this chapter, I have come up with the following system diagram that we will look to as a framework. The whole concept is designed to be modular, so you can come up with your own system to suit your requirements and implement it accordingly, using the scripts presented in this chapter.


+12V Power Supply
+3.3V Power Supply
Port A
Port B
Zone 1
Entry / Exit
Zone 2
e.g.Hallway
Zone 3
e.g.Lounge
Zone 4
e.g.Kitchen
Zone 5
e.g.Bedrooms
Zone 6
Zone 7
Zone 8
Anti-Tamper
Opto-Isolator
A0
A1
A2
A3
A4
A5
A6
A7
MCP23107 PORT EXPANDER
B0
B1
B2
B3
B4
B5
B6
B7
INPUT OR OUTPUT
I2C BUS
+12V
ARMED LED
ENTRY/EXIT BUZZER
ALARM LED OR LAMP
ALARM BELL
DRIVER
I2C
GP4
GP5
GP6
GP7
GP0
RASPBERRY PI
ARM / DISARM

The final home security system diagram

Overview of the system elements

The preceding system diagram comprises the elements and modules that we have discussed in previous chapters. Here's a quick recap of these:

A +12V power supply

This is the primary power supply to our system, which we will obtain from an external mains adapter that could be **battery-backed**. This supply needs to be smooth and regulated to ensure that it remains stable for the system as currently drawn.

All of the alarm wiring and sensors will be supplied with this power, as will peripherals such as sounders and bells, which usually operate from a 12V supply. Chapter 5, *Adding a Passive Infrared Motion Sensor* discussed the merits of using a 12V supply for the alarm circuits.

A +3.3V power supply

This supply is a regulated +3.3V supply for the digital port expander circuit; it also provides the logical alarm zone inputs via an opto-coupler. The +3.3V power supply can be derived from either the +12V supply (recommended), or the +5V supply from the Raspberry Pi's GPIO connector, using a voltage regulator chosen according to how much current you need.

Chapter 3, *Extending Your Pi to Connect More Things*, showed you how to build a +3.3V regulated supply.

The opto-isolator input module

This will isolate the +12V zone input power lines from the port expander and GPIO digital inputs, which should only have a maximum of +3.3V presented to them when triggered high.

The circuit for these opto-isolated input modules was discussed and shown in Chapter 5, *Adding a Passive Infrared Motion Sensor*.

The port expander

The port expander is our main digital input/output system that will take the alarm zone inputs and transmit them to the Raspberry Pi using the I2C bus, or allow the Raspberry Pi to switch outputs on and off.

We built our MCP23017-based port expander circuit in Chapter 3, *Extending Your Pi to Connect More Things* and configured the software for it in Chapter 4, *Adding a Magnetic Contact Sensor*.

An arm/disarm switch

The arm/disarm input overrides the arm/disarm **soft-switch** function on our web-based control panel, and is a switch (key, digital keypad, or otherwise) connected to GP0 directly on the Raspberry Pi's GPIO connector.

Remember to connect any switch circuit appropriately to the GPIO pin to avoid damage to your Raspberry Pi. This was discussed in Chapter 2, *Connecting Things to Your Pi with GPIO*.

Alarm outputs

In our system, we have several output devices that are controlled by our Raspberry Pi via output driver circuits. We have an output for an entry/exit buzzer, an armed status LED, an alarm bell, and an alarm LED indicator.

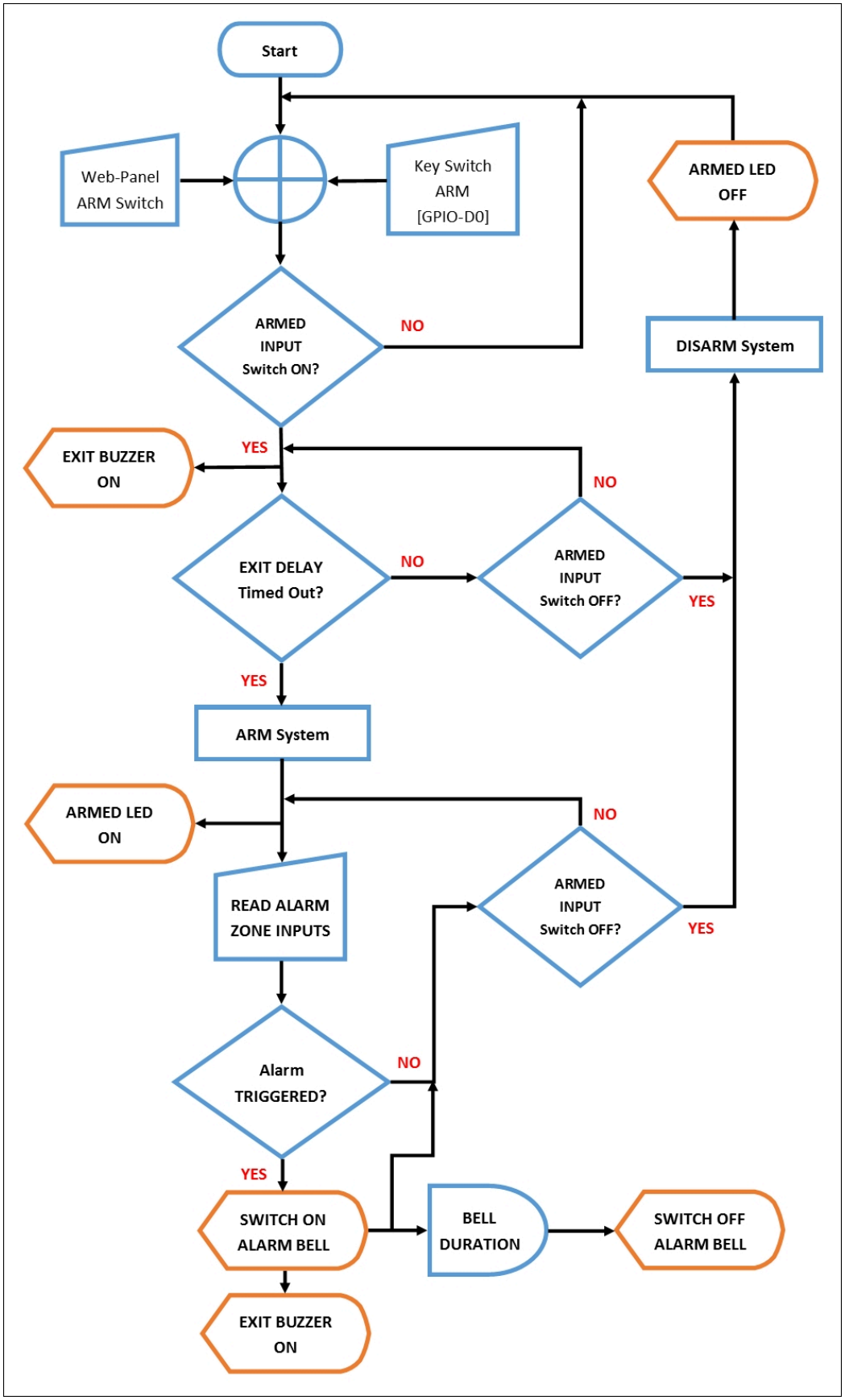
These are switched on and off by our Raspberry Pi GPIO connector via driver circuits that allow us to drive high current and inductive loads using the GPIO pins. These driver circuits, based around TIP120 Darlington transistors, were discussed in Chapter 6, *Adding Cameras to Our Security System* and Chapter 8, *A Miscellany of Things*.

Designing the control scripts

Before we start writing the scripts to control our alarm systems, it is probably a good idea to outline the **high-level** process for the system. The following **flow-chart** helps us picture how our system should work, and the various logical decisions our script needs to make.

The flowchart might look a bit complicated with all its lines in different directions, but it's actually pretty linear and in a downward direction. Referring to the flowchart, it shows the following tasks that the control script will be doing:

- Sitting quietly until the system is armed either by the hardware key switch or the web-based panel's soft switch.
- When the system is first armed, it will sound the exit buzzer for a pre-determined amount of time before actually arming the system. This gives you a chance to leave the property or disarm the system again, before it starts monitoring the inputs.
- Once the system is armed, the armed LED will be switched on and the system will wait to see if any of the alarm zone inputs are triggered. It will also wait to see if the alarm is disarmed on your return to the property. We can optionally put an entry timer in here on the entry zone to delay before triggering the alarm.
- If the alarm is ultimately triggered, then the main alarm bell will be switched on, as well as the exit buzzer. The main bell should only sound for a while, depending on environmental restrictions in your neighborhood, and so, this will be switched off after a pre-defined period, but the internal buzzer will stay on.
- When triggered, the system will then wait for you to disarm it, before resetting it.


Start
Web-Panel
ARM Switch
Key Switch
ARM
[GPIO-D0]
ARMED LED
OFF
ARMED
INPUT
Switch ON?
NO
DISARM System
EXIT BUZZER
ON
YES
NO
EXIT DELAY
Timed Out?
NO
ARMED
INPUT
Switch OFF?
YES
YES
ARM System
ARMED LED
ON
NO
READ ALARM
ZONE INPUTS
ARMED
INPUT
Switch OFF?
YES
Alarm
TRIGGERED?
NO
YES
SWITCH ON
ALARM BELL
BELL
DURATION
SWITCH OFF
ALARM BELL
EXIT BUZZER
ON

The control script flowchart

Building the control script

Now that we have designed our system the way we want it to work, we can start writing our Bash **control script**. As before, we'll locate our scripts in the folder, `/etc/pi-alarm`, which, you'll remember from Chapter 7, *Building a Web-Based Control Panel*, is also where our Web-based control panel writes its configuration status file, `alarm.cfg` to. We'll be referring to that file in our scripts too.

In this script, we are going to use the **bc** tool (the Bash command-line **calculator**) to convert **hex** values to **binary**. It's not installed by default, so you'll need to get the package:

```
$ sudo apt-get update
$ sudo apt-get install bc
```

Tip

Our script file is quite long so, as before, you might want to sit on the sofa and write it on your laptop using something such as Notepad++. Remember, however, if you're using a PC, ensure that the end-of-line (EOL) format is converted to the Unix format, otherwise the Bash script won't run on the Pi when you copy it across. Notepad++ will do this for you.

Exploring the script code

I'm now going to walk you through the various sections of the control script code I've written, which will be used as a framework for our system. I say "framework" because, while it will provide you with a fully functional control script for the system, it can be modified and extended to suit your particular requirements.

The following code listings are all part of the single bash script, `alarm-control.sh`, that can be downloaded in full with comments from the Packt Publishing website.

Declarations

We'll start off by setting up the various **control variables** needed to track the system's state:

```
#!/bin/bash
#/etc/pi-alarm/alarm-control.sh

ALM_BELL_DURATION=600    #duration in seconds the
alarm bell should sound for
ALM_EXIT_DELAY=30    #entry/exit zone delay in
seconds
ALM_KEY_ARMED=0    #status of the arm/disarm key
switch
ALM_SYS_ARMED=0    #armed status of the system

ALM_ZONE_INPUT_READ=""      #this will store the
value of the zone inputs read
ALM_ZONE_INPUT_STAT="00000000"    #binary
representation of the inputs (b7-b0)
ALM_ZONE_INPUT_PREV=""      #previous zone input
status
ALM_ZONE_TRIGGER=0    #this will be set to 1 if
one or more zones is triggered
ALM_ZONES_STAT=(0 0 0 0 0 0 0 0)    #dynamic array
of normalised zone status (z1 to z8 order) - 1 is
triggered

STAT_RET_VAL=""    #return value from functions
```

Because we could face the situation whereby a HIGH or a LOW input could represent a triggered zone, depending on its configuration and wiring, I have introduced an array of *normalized* status flags in the variable, `ALM_ZONES_STAT`, which will be the definitive state as far as the script is concerned. We'll look at the function that deals with this later.

Updating config settings

In Chapter 7, *Building a Web-Based Control Panel*, we introduced the configuration file, `alarm.cfg`, which stores the system status and configuration for the benefit of the Web-based control panel. This file not only needs to be read by the main control script to get any settings made using the control panel, but also needs to be updated with status values from the main control script so that they can be presented back to the control panel, essentially exchanging data between the two sub-systems.

Therefore, we're going to include a helper function that contains the same code called by the Web page PHP script to update this file from the control panel:

```
#This helper function will update the alarm config
#file with the specified value (alarm.cfg) so that
#the Web panel can know the latest status
function almUpdateConfigSetting()
{
  #$1 - Setting Name
  #$2 - Setting Value
  sudo sed -i "s/^\($1\s*= *\).*/\1$2/" /etc/pi-
alarm/alarm.cfg
}
```

Setting up the GPIO

We now need to set up the Raspberry Pi's GPIO pins for our purposes, as outlined by the earlier system diagram. The following commands were first discussed in Chapter 2, *Connecting Things to Your Pi with GPIO*:

```
# GPIO SET UP ###################################
#Set up the Raspberry Pi GPIO pins
#Refer to Chapter 2 for info
#D0 (GPIO17) Arm/Disarm Key Input
sudo echo 17 > /sys/class/gpio/export
sudo echo in > /sys/class/gpio/gpio17/direction
```

```
#D4 (GPIO23) Armed LED Output
sudo echo 23 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio23/direction
sudo echo 0 > /sys/class/gpio/gpio23/value

#D5 (GPIO24) Exit Buzzer Output
sudo echo 24 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio24/direction
sudo echo 0 > /sys/class/gpio/gpio24/value

#D6 (GPIO25) Alarm LED Output
sudo echo 25 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio25/direction
sudo echo 0 > /sys/class/gpio/gpio25/value

#D7 (GPIO4)  Alarm Bell Output
sudo echo 4 > /sys/class/gpio/export
sudo echo out > /sys/class/gpio/gpio4/direction
sudo echo 0 > /sys/class/gpio/gpio4/value
```

Note

Note that you can only **export** a GPIO pin once, unless it has been subsequently **unexported**. Therefore, you might see the error, `echo: write error: Device or resource busy`, if you re-run the script when it tries to export the pin again. You can safely ignore this.

We'll also throw in a few helper functions that will easily allow us to switch on or off various outputs to simplify the main code. I'm a big fan of implementing functions, however simple, as they keep the code modular, reusable, and simpler to read in most cases:

```
#This helper function will switch a specified GPIO
output on or off
function almSetGPIOValue()
{
  #$1 - GPIO pin number
  #$2 - Value
  sudo echo $2 > /sys/class/gpio/gpio$1/value
}
```

```
#Helper functions to switch on and off the outputs
function almSetArmedLED()
{
  #$1 - 0 or 1 (Off or On)
  almSetGPIOValue 23 $1
  echo "[ALM] Armed LED set to $1"
}
function almSetExitBuzzer()
{
  #$1 - 0 or 1 (Off or On)
  almSetGPIOValue 24 $1
  echo "[ALM] Exit Buzzer set to $1"
}
function almSetAlarmLED()
{
  #$1 - 0 or 1 (Off or On)
  almSetGPIOValue 25 $1
  echo "[ALM] Alarm Trigger LED set to $1"
}
function almSetAlarmBell()
{
  #$1 - 0 or 1 (Off or On)
  almSetGPIOValue 4 $1
  echo "[ALM] Alarm Bell set to $1"
}
```

And, we'll add a helper function that will read the ARM switch status from the D0 (GPIO17) of the Raspberry Pi and from the web-console to see if the ARM soft switch has been set:

```
#this function returns whether the system is armed
via
#either the web console or key switch
function almGetArmedSwitchStatus()
{
  STAT_RET_VAL="0"
  #read arm key switch input from
  local L_VAL=$(sudo cat
/sys/class/gpio/gpio17/value)
  if [ $L_VAL -eq 1 ]; then
    #system has been armed with key switch
    echo "[ALM] System ARMED with key switch"
    ALM_KEY_ARMED=1
    almUpdateConfigSetting "SYSTEM_ARMED" "1" #set
system armed console flag
```

```
    STAT_RET_VAL="1"
  else
    #read system armed value from web console
config file
    if [ $SYSTEM_ARMED == 1 ]; then
      echo "[ALM] System ARMED with web console"
      STAT_RET_VAL="1"
    fi
  fi
}
```

Setting up the I2C port expander

The next few lines of code set up the I2C port expander to set all of the pins, on both Port A and Port B, as inputs. In our system here, we're only using Port A, but this allows us to have another 8 inputs if we want to expand our system. We originally looked at this in Chapter 4, *Adding a Magnetic Contact Sensor*:

```
# PORT EXPANDER SET UP ############################
#Refer to Chapter 4 for more information about the
I2C bus

#We will set up I/O BUS A as all inputs
sudo i2cset -y 1 0x20 0x00 0xFF

#Whilst we're not using BUS B in our system,
#we can set that up as all inputs too
sudo i2cset -y 1 0x20 0x01 0xFF
```

Note

If you don't have your I2C port expander attached, then you'll see the following error when you try to run these commands: *Error: Write failed*

Decoding the zone inputs status

The next function is a big one—and key to our system. It will read the Port A value from the I2C port expander. It'll be returned as a hexadecimal value, so we need to convert this to a binary value with a 0 or 1 flag representing each input bit. We'll use the `bc` tool installed earlier to do this.

Once we have the status of each input bit, we then normalize the status by determining whether a 0 or a 1 determines a positive trigger. The resulting output is the array, `ALM_ZONES_STAT`, which contains the status of each zone—with a 1 representing a positive triggered zone de-facto:

```
#This function will read the port inputs and set
the
#status of each zone
function almReadZoneInputs()
{
  #preserve previous zone status
  ALM_ZONE_INPUT_PREV=$ALM_ZONE_INPUT_STAT
  #read the 8-bit hex value of port a
  ALM_ZONE_INPUT_READ=$(sudo i2cget -y 1 0x20
0x12)

  if [[ $ALM_ZONE_INPUT_READ = *"Error"* ]]; then
    #An error occurred reading the I2C bus - set
default value
    ALM_ZONE_INPUT_READ="0x00"
  fi

  #remove the 0x at the start of the value to get
the hex value
  local L_HEX=${ALM_ZONE_INPUT_READ:2}
  #convert the hex value to binary
  local L_BIN=$(echo "obase=2; ibase=16; $L_HEX" |
bc )
  #zero pad the binary to represent all 8 bits
(b7-b0)
  ALM_ZONE_INPUT_STAT=$(printf "%08d" $L_BIN)

  echo "[ALM] Zone I/O Status:
$ALM_ZONE_INPUT_STAT ($ALM_ZONE_INPUT_READ)"

  #check each zone input to see if it's in a
triggered state
```

```
  #a triggered state may be either 1 or 0
depending on the input's configuration
  #you'll need to set the logic here accordingly
for each input
  #the ALM_ZONES_STAT array contains the
definitive trigger value for each input

  #zone 1 test (bit 0)
  local L_FLG=${ALM_ZONE_INPUT_STAT:7:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[0]=0;
else ALM_ZONES_STAT[0]=1; fi

  #zone 2 test (bit 1)
  local L_FLG=${ALM_ZONE_INPUT_STAT:6:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[1]=0;
else ALM_ZONES_STAT[1]=1; fi
  #zone 3 test (bit 2)
  local L_FLG=${ALM_ZONE_INPUT_STAT:5:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[2]=0;
else ALM_ZONES_STAT[2]=1; fi

  #zone 4 test (bit 3)
  local L_FLG=${ALM_ZONE_INPUT_STAT:4:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[3]=0;
else ALM_ZONES_STAT[3]=1; fi

  #zone 5 test (bit 4)
  local L_FLG=${ALM_ZONE_INPUT_STAT:3:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[4]=0;
else ALM_ZONES_STAT[4]=1; fi

  #zone 6 test (bit 5)
  local L_FLG=${ALM_ZONE_INPUT_STAT:2:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[5]=0;
else ALM_ZONES_STAT[5]=1; fi

  #zone 7 test (bit 6)
  local L_FLG=${ALM_ZONE_INPUT_STAT:1:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[6]=0;
else ALM_ZONES_STAT[6]=1; fi

  #zone 8 test (bit 7)
  local L_FLG=${ALM_ZONE_INPUT_STAT:0:1}
  if [ $L_FLG -eq 0 ]; then ALM_ZONES_STAT[7]=0;
else ALM_ZONES_STAT[7]=1; fi

  echo "[ALM] Zone Trigger Status:
```

```
$ALM_ZONES_STAT[*]"
}
```

Initialization

Now that we have declared our module-level variables and helper functions, we will start our main routine. First, we'll initialize the system that clears the `SYSTEM_ARMED` status and reads in the initial settings from the config file:

```
# initialise system #########
echo "[ALM] Initialising system..."
almUpdateConfigSetting "SYSTEM_ARMED" "0" #clear
system armed console flag
sleep 1
sudo cat /etc/pi-alarm/alarm.cfg
sleep 1
echo "[ALM] Initialising done"
##############################
```

The system monitoring loop

The script then jumps into a never-ending loop that will be the main control system, monitoring the arm/disarm status and, when armed, monitoring the zone input status and responding accordingly:

```
# loop continuously###########
while true
do

  # wait for system to be armed ###############
  echo "[ALM] Alarm now in STAND-BY state -
waiting to be armed"
  almSetArmedLED 0 #switch off armed LED
  STAT_RET_VAL="0"
  while [[ $STAT_RET_VAL = "0" ]]; do
    sleep 1
    #read the control panel status file
    . /etc/pi-alarm/alarm.cfg
    almGetArmedSwitchStatus #result is returned in
STAT_RET_VAL
```

```
    echo -n "*" # indicate standby mode
  done
  ##############################################
```

Arming the system

When the system goes into the ARMED state, it will first switch on the exit buzzer and then wait for a pre-determined amount of time. This will give you time to leave the property or disarm the system:

```
# perform exit delay ########################
  echo "[ALM] Alarm now in EXIT DELAY state"
  almSetExitBuzzer 1 #switch on exit buzzer
  COUNTER=$ALM_EXIT_DELAY
  while [[ $STAT_RET_VAL = "1" && $COUNTER -gt 0
]]; do
    sleep 1
    #read the control panel status file
    . /etc/pi-alarm/alarm.cfg
    almGetArmedSwitchStatus #result is returned in
STAT_RET_VAL
    COUNTER-=1
    echo -n "X$COUNTER " # indicate exit mode
  done
  almSetExitBuzzer 0 #switch off exit buzzer
  ##############################################

  # system now armed - monitor inputs #########
  ALM_SYS_ARMED=1
  echo "[ALM] Alarm now in ARMED state"
  almSetArmedLED 1 #switch on armed LED

  #read the control panel status file
  . /etc/pi-alarm/alarm.cfg
  almReadZoneInputs  # > ALM_ZONES_STAT[x]
```

Monitoring the zones

Once armed, the system will monitor the zone inputs in a continuous loop until either the system is disarmed, or a zone input is triggered. When a zone is triggered, it will check against the `ZONE_ENABLE_n` configuration to see if that zone has been disabled (this is done in

the Web-based control panel). If the zone is not disabled, then the alarm system is deemed triggered.

The `ZONE_STATUS_n` setting is also updated here so that the web-based control panel indicates which zone or zones have been triggered:

```
#check each zone input to set if it's enable
  #and has been triggered
  #NUM_ZONES setting is stored in alarm.cfg

  while [[ $ALM_SYS_ARMED -eq 1 ]]; do
    echo -n "A" #indicate armed mode

    ALM_ZONE_TRIGGER=0
    for (( i=$NUM_ZONES; i>0; i-- )); do
      if [[ $ALM_ZONES_STAT[$i-1] -eq 1 ]]; then
        #zone has been triggered
        echo "[ALM] Zone $i TRIGGERED"
        E_VAR="ZONE_ENABLE_$i"
        E_VAL=`echo "$E_VAR"` #get zone enabled
status loaded from alarm.cfg

        if [[ $E_VAL -eq 1 ]]; then
          #zone is enabled
          ALM_ZONE_TRIGGER=1 #set alarm triggered
flag
          echo "[ALM] Zone $i ENABLED - alarm will
be triggered"
          almUpdateConfigSetting "ZONE_STATUS_$i"
"1"

          ## YOU CAN INSERT CODE HERE TO TAKE
CAMERA IMAGE IF YOU WANT##
          ## REFER BACK TO CHAPTER 6 ##

        fi
      fi
    done

    . /etc/pi-alarm/alarm.cfg
    almGetArmedSwitchStatus #result is returned in
STAT_RET_VAL
```

Entry delay

When an alarm zone is triggered, it will first check whether it was the entry/exit zone that was triggered. If it was, then the system will delay before sounding the main alarm to give you a chance to disarm the system. Only the entry buzzer will sound at this time:

```
if [[ $ALM_ZONE_TRIGGER -eq 1 ]]; then
      # alarm has been triggered
      almSetAlarmLED 1
      echo "[ALM] A zone has been triggered"

      ######################################
      # ZONE 1 is the ENTRY zone - if that's
triggered then delay
      if [[ $ALM_ZONES_STAT[0] -eq 1 ]]; then
        # perform entry delay ###########
        echo "[ALM] Alarm now in ENTRY state"
        setExitBuzzer 1 #switch on entry/exit
buzzer

        COUNTER=$ALM_EXIT_DELAY
        STAT_RET_VAL="0"
        while [[ $STAT_RET_VAL = "1" && $COUNTER -
gt 0 ]]; do
          echo -n "E$COUNTER " #indicate entry
mode
          sleep 1
          #read the control panel status file
          . /etc/pi-alarm/alarm.cfg
          almGetArmedSwitchStatus #result is
returned in STAT_RET_VAL
          COUNTER-=1
        done
      fi
      ######################################
```

Sounding the main alarm

If, at this point, the system hasn't been disarmed, then we need to sound the main bell. We have a duration limit for sounding the bell to cater to environmental noise restrictions; we wouldn't want the alarm sounding for hours, annoying the neighbors until we got home. At

this point, you can also add code from Chapter 6, *Adding Cameras to Our Security System*, if you want to be sent an alert email to your mobile device:

```
#####################################
      # STAY in TRIGGERED mode until system has
been disarmed
      if [[ $STAT_RET_VAL = "1" ]]; then
        #alarm has not been disabled
        almSetAlarmBell 1 #switch on alarm bell
        echo "[ALM] Alarm now in TRIGGERED state"

        ## YOU CAN INSERT CODE HERE TO SEND YOU AN
EMAIL IF YOU WANT##
        ## REFER BACK TO CHAPTER 6 ##

        COUNTER=0
        STAT_RET_VAL="0"
        while [[ $STAT_RET_VAL = "1" ]]; do
          echo -n "T$COUNTER " #indicate triggered
mode
          sleep 1
          #read the control panel status file
          . /etc/pi-alarm/alarm.cfg
          almGetArmedSwitchStatus #result is
returned in STAT_RET_VAL

          COUNTER+=1
          if [[ $COUNTER -gt $ALM_BELL_DURATION
]]; then
            almSetAlarmBell 0 #switch off alarm
bell
            echo "[ALM] Bell has been switched
OFF"
          fi
        done
      fi
      #####################################
```

Disarming and resetting the system

When we disarm the system, we need to reset its status and complete the monitoring loop so that we can start all over again and wait for it to be re-armed:

```
# alarm has been disarmed ##########
      echo "[ALM] Alarm has been DISARMED"
      ALM_SYS_ARMED=0
      almSetAlarmBell 0 #switch off alarm bell
      almSetExitBuzzer 0 #switch off exit buzzer
      almSetAlarmLED 0
      almSetArmedLED 0 #switch off armed LED

      ####################################
    fi

  done
  ##############################################

done
##############################################
```

We're done (almost)–

And there we have it: a framework for an entire alarm control script on our Raspberry Pi. Additional features that you may want to implement within your script could include the following:

- Sending a photo or video clip from a zone's camera when it's triggered
- Sending an email alert with status details when the alarm has been triggered
- Writing a regular log file recording historical status information
- Adding additional environmental sensors to port B

> ### Tip
>
> Each of the script blocks is taken from the single script file, `alarm-control.sh`, so you should be able to put all of the described pieces together into one file to have a fully functional script.

As always, before we can run it we need to give the script execute rights:

```
$ sudo chmod 777 /etc/pi-alarm/alarm-control.sh
```

After we copy the script to our Raspberry Pi, this is what we should see in our `/etc/pi-alarm` folder:

```
pi@raspberrypi ~ $ ls -1 /etc/pi-alarm
alarm.cfg
alarm-control.sh
update-alarm-setting.sh
```

Automatically starting the system

Now, obviously, we don't want to have to manually start the alarm control script each time the Raspberry Pi boots up, for example, after a power failure—for a start, we may not even be there. Therefore, we need to set up our operating system so that it will automatically start up the `alarm-control.sh` script at boot time.

To do this, we need to edit the `rc.local` file using Nano:

```
$ sudo nano /etc/rc.local
```

Before the line containing `exit 0`, insert the following line:

```
sudo /etc/pi-alarm/alarm-control.sh &
```

> ## Note
>
> The `&` symbol at the end of the line is important because it will then make the script run in a different process, otherwise the `rc.local` script would never exit.

Your `rc.local` file should now look something like this:

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each
multiuser runlevel.
# Make sure that the script will "exit 0" on
success or any other
# value on error.
#
```

```
# In order to enable or disable this script just
change the execution
# bits.
#
# By default this script does nothing.

# Print the IP address
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
  printf "My IP address is %s\n" "$_IP"
fi

sudo /etc/pi-alarm/alarm-control.sh &
exit 0
```

The operating system runs the `rc.local` script after the system boots up, so you can put anything in there that you want to happen automatically at this time.

Preserving the SD card

One final topic I want to share with you is that of preserving your Raspberry Pi's SD card. SD cards have a finite write cycle, and continuous writing to the card will eventually burn it out. If we're going to be writing lots of log file entries and taking lots of camera images, we will want to protect our SD card in order to maintain the integrity and reliability of our system; using the system RAM instead can help us with this.

Creating a RAM-based file system

Our Raspberry Pi has plenty of fast system RAM available to us (1Gb on the latest models) that isn't susceptible to this write burn-out issue. Therefore, I'm going to show you how to allocate some of it to create a temporary disk in memory, which we can write files to that we don't need kept on the SD card. Such files would include the, quite large, camera image files that will be emailed out of the system — which, therefore, don't need to be stored permanently. You should also consider any log files that are regularly written to, which would then be shipped off the system at regular intervals.

> **Note**
>
> Remember that this is a RAM-based file system, so content will be lost when the Raspberry Pi shuts down or reboots. So, don't store any data here that you want to persist after a restart.

Let's create a Bash script file called `setup-ramfs.sh`, and copy it to our `/etc/pi-alarm` folder:

```
#!/bin/bash
#/etc/pi-alarm/setup-ramfs.sh
```

```
RAM_DISK="/ramfs"
RAM_DISK_SIZE=64M

# Create RAM Disk ############################
if [ ! -z "$RAM_DISK" ]; then
  echo "[INIT] Creating RAM Disk... $RAM_DISK"
  mkdir -p $RAM_DISK
  chmod 777 $RAM_DISK
  mount -t tmpfs -o size=$RAM_DISK_SIZE tmpts
$RAM_DISK/
  echo "[INIT] RAM Disk created at $RAM_DISK"
fi
##############################################
```

`setup-ramfs.sh` RAM disk creation script

Running the preceding script will create a RAM disk folder at `/ramfs`—you can treat it just like any other folder; it's just that it resides in the system memory rather than on the SD card:

```
$ cd /ramfs
$ ls
```

You can call this script from the `alarm-control.sh` script as part of the initialization process by including the line:

```
. /etc/pi-alarm/setup-ramfs.sh
```

Conclusion

The Raspberry Pi is a powerful little beast and a great platform for building low-cost, but highly capable, embedded systems. The interfaces built into its GPIO connector make it easy to bolt on modules using simple low-cost electronics and a bit of configuration to create very functional and flexible systems. The inclusion of a dedicated camera interface and networking interfaces give you everything you could possible need for an Internet-connected home security system.

I've covered a lot of topics in this book, and I could have gone on and on, but I hope that what I have presented has been done in a structured and methodical way, and has given you the tools and techniques to carry on this journey so that you are able to create the perfect home security system for your needs.

Tips for building systems

As a systems guy who has to work with many different technologies and disciplines on a day-to-day basis, I just want to leave you with the following thoughts to consider, if you choose to build upon the system we've put together in this book, which, of course, I hope you will:

- Create a high-level diagram of your proposed system first—a bit like the one I produced earlier in this chapter.
- Define everything in a modular way so that you can build and test your system in small chunks. This makes it much easier to spot issues early on.
- Building the system using smaller modules makes it easier to re-use and replace circuits and code, and don't be afraid to mix-and-match technologies using what's best for the individual module.
- Don't try to re-invent the wheel—use existing code and circuit resources that are proven to work. This makes it much quicker

to get things working and minimizes the number of times you have to hit your head against a brick wall. I call it blagging.

Summary

Well, we've reached the end of our journey to build a fully functional and extensible home security system using the mighty Raspberry Pi mini-PC. In this final chapter, we put together all of the elements and concepts from the previous chapters to create a home security framework, both from a hardware and software perspective.

In particular, this chapter guided us toward building a modular framework for our home security system, implementing features that you would find in any commercially available system, and also things that you don't see out there. We walked through the complete control script, exploring its various sections and understanding how they fit into our system.

We also learned how to automatically start-up our home security system script when our Raspberry Pi boots up, and how data is shared between the Pi and the web-based control panel in real-time via the configuration file. Finally, we looked at how to prevent our SD card from burning out by creating a rather useful RAM-based temporary file system.

Part 3. Module 3

Raspberry Pi Robotics Essentials

Harness the power of Raspberry Pi with Six Degrees of Freedom (6DoF) to create an amazing walking robot.

Chapter 1. Configuring and Programming Raspberry Pi

Robots are beginning to infiltrate our world. They come in all shapes and sizes, with a wide range of capabilities. And, just like the evolution of the personal computer before them, much of what is happening in the robot development world is coming from hobbyists and do-it-yourselfers that are using a new generation of inexpensive hardware and free, open source software to build machines with all kinds of amazing capabilities. In this book, you will learn how to build robots by building a robot, a four-legged quadruped with sensor and vision capabilities. The skills you will learn, however, can also be used on a wide variety of walking, rolling, swimming, or flying robots.

In this chapter, you'll learn:

- How to configure your Raspberry Pi, the control center of your robot, with the Raspbian operating system
- How to set up a remote development environment so you can program your robot
- Basic programming skills in both Python and C so you can both create and edit the programs your robot will need to do all those amazing things

Configuring Raspberry Pi – the brain of your robot

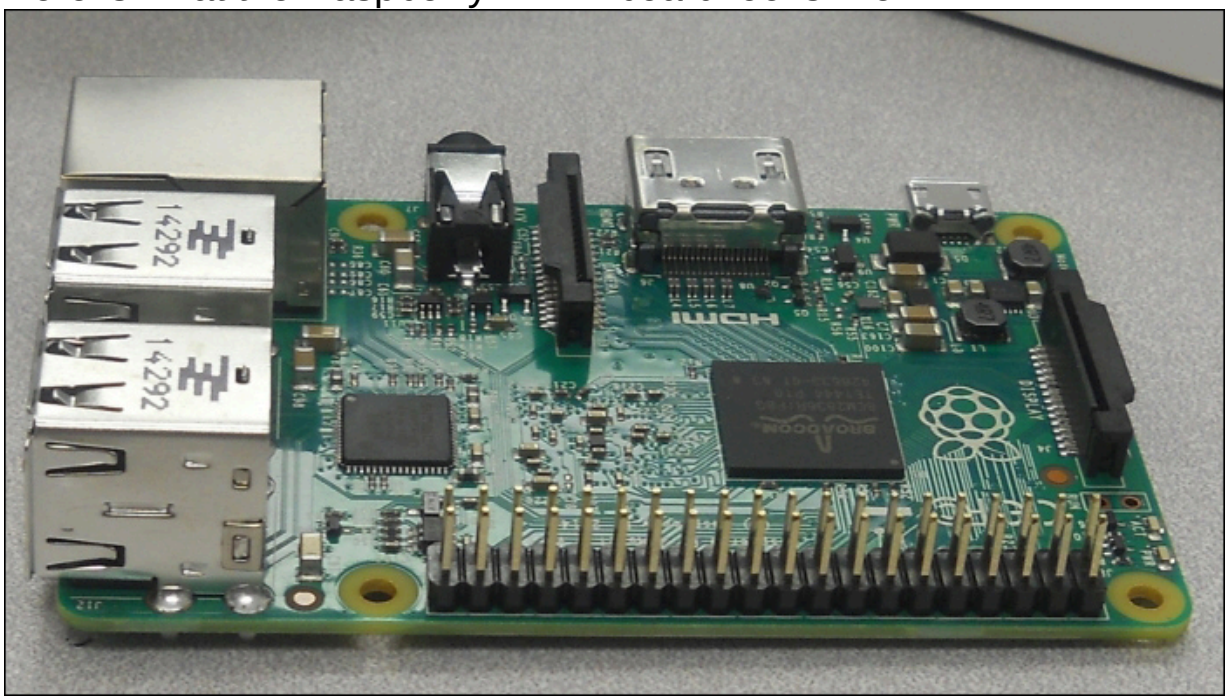
One of the most important parts of your robot is the processor system you use to control all the different hardware. In this book, you'll learn how to use Raspberry Pi, a small, inexpensive, easy-to-use processor system. Raspberry Pi comes in several flavors – the original A and B model, and the new and improved A+ and B+ model. The B+ flavor is the most popular and comes with additional

input/output capability, four USB connections, more memory, and will be the flavor we'll focus on in this book.

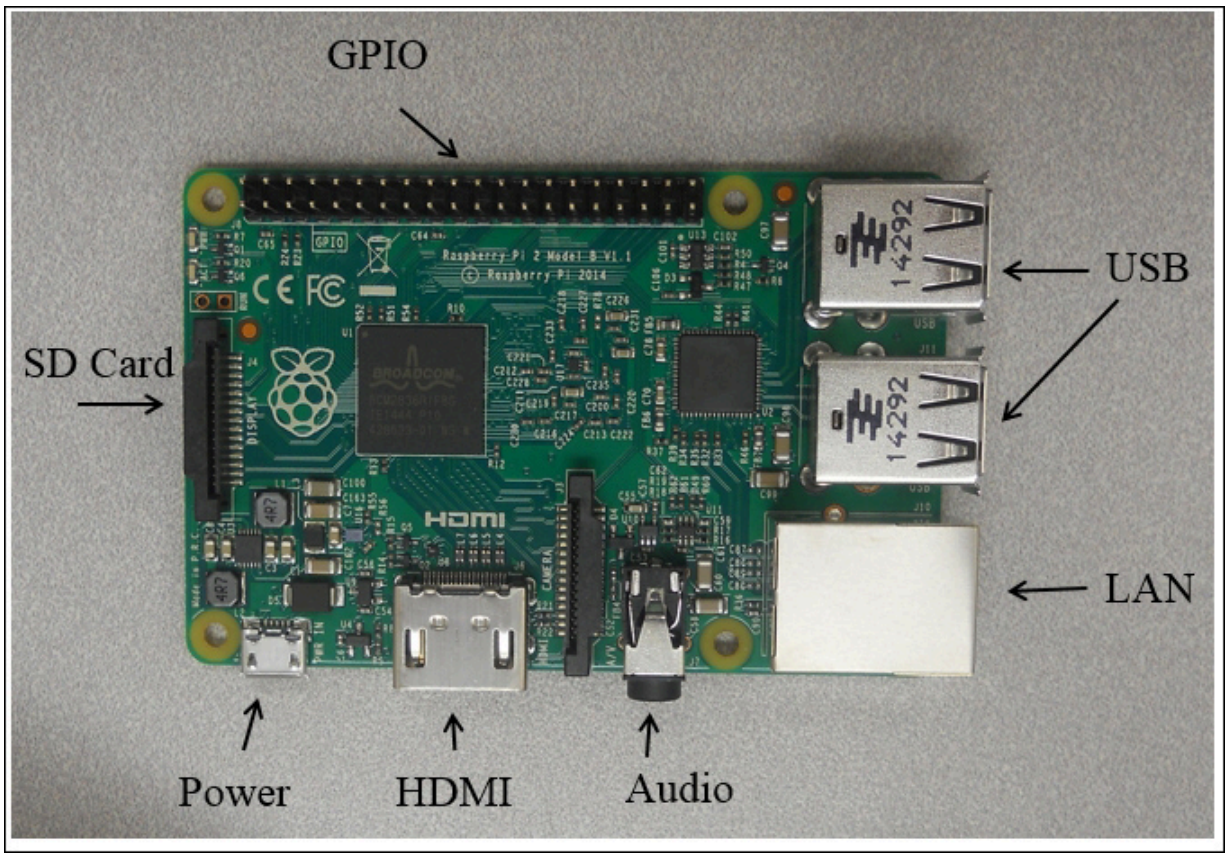
Here are the items you'll need to set up an initial Raspberry Pi development environment:

- A Raspberry Pi, Model B 2. There are three other Raspberry Pi models, the B+, the B, and the A. These are models with less processing power and different hardware configurations. In this book, we'll focus on the Raspberry Pi Model B 2; it has the best processing power and the most useful input/output access. However, many of the items in this book will also work with the Raspberry Pi B+ and A versions, perhaps with some additional hardware.
- The USB cable to provide power to the board.
- A microSD card — at least 4 GB.
- A microSD card writer.
- Another computer that is connected to the Internet.
- An Internet connection for the board — for the initial configuration steps, you'll need a LAN cable and wired LAN connection.
- A wireless LAN device.

Here is what the Raspberry Pi B 2 board looks like:



You should also acquaint yourself with the different connections on the board. Here they are on the B 2, labelled for your information:


GPIO
SD Card
USB
LAN
Power
HDMI
Audio

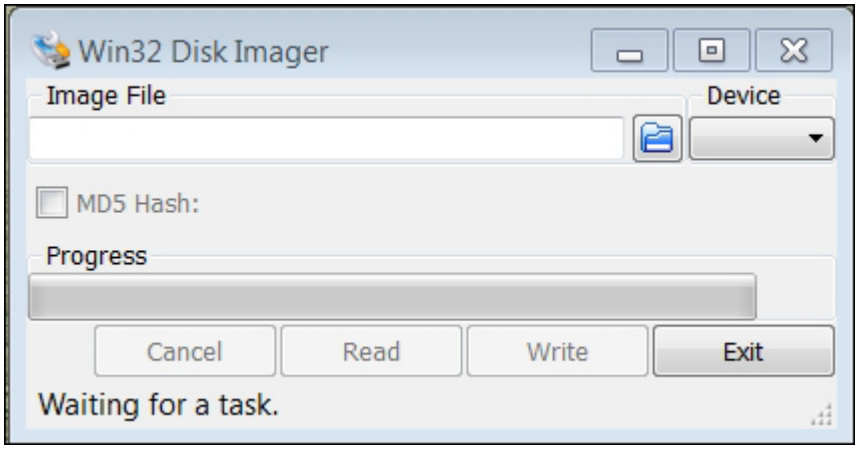
Installing the operating system

Before you get started, you'll need to download and create a card with the Raspbian operating system. You are going to install Raspbian, an open source version of the Debian version of Linux, on your Raspberry Pi.

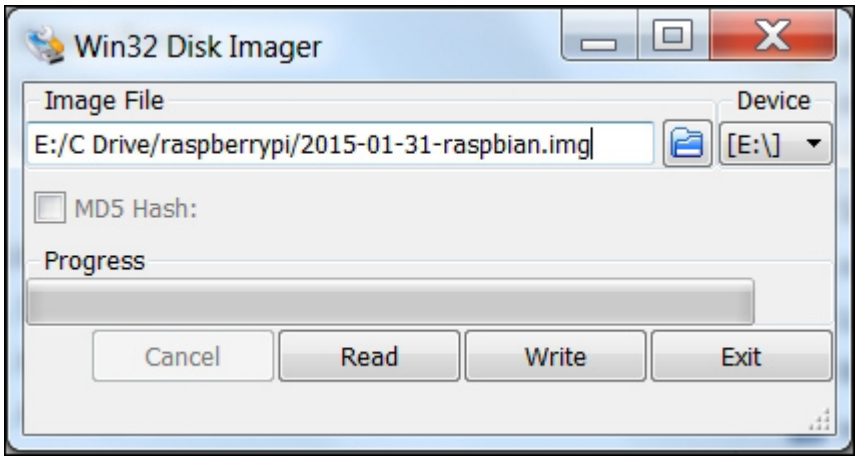
There are two approaches to getting Raspbian on your board. The board is getting popular enough that you can now buy an SD card that already has Rasbpian installed, or you can download it onto your personal computer and then install it on the card. If you are going to download a distribution, you need to decide if you are going to use a Windows computer to download and create an SD card, or a Linux machine.

No matter which machine you are going to use, you'll need to download an image. Open a browser window. Go to the Raspberry Pi site, www.raspberrypi.org, and select **Downloads** from the top of the page. This will give you a variety of download choices. Go to the **Raspbian** section and select the `.zip` file just to the right of the image identifier. This will download an archived file that has the image for your Raspbian operating system. Note the default username and password; you'll need those later.

If you're using Windows, you'll need to unzip the file using an archiving program like 7-Zip. This will leave you with a file that has the `.img` extension, a file that can be imaged on your card. Next, you'll need a program that can write the image to the card. Use Image Writer if you are going to create your card using a Windows machine. You can find a link to this program at the top of the download section on the www.raspberrypi.org website. Plug your card into the PC, run this program, and you should see this:

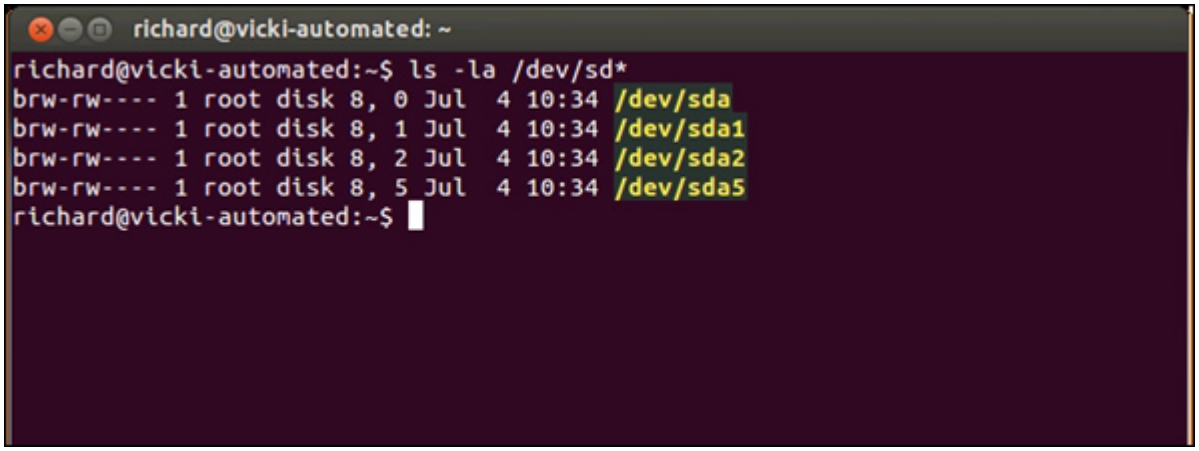


Select the correct card and image; it should look something like this:

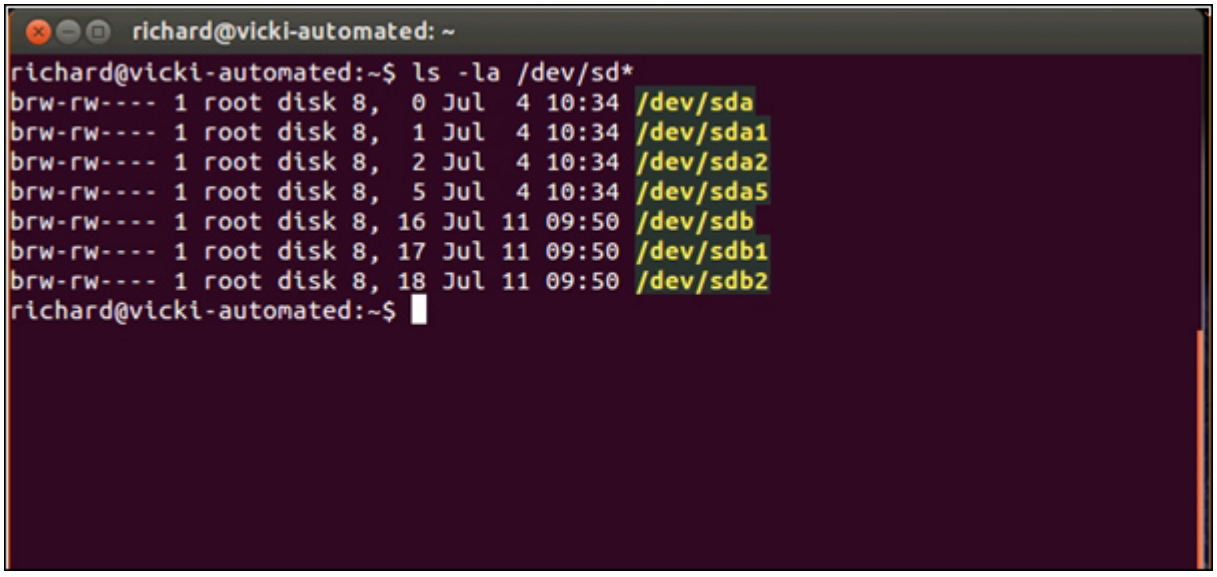


Then select **Write**. This will take some time, but when it is complete, eject the card from the PC.

If you are using Linux, you'll need to unarchive the file and then write it to the card. You can do all of this with one command. However, you do need to find the `/dev` device label for your card. You can do this with the `ls -la /dev/sd*` command. If you run this before you plug in your card, you might see something like the following:



After plugging in your card, you might see something like the following:



Note that your card is `sdb`. Now, go to the directory where you downloaded the archived image file and issue the following command:

```
sudo dd if=2015-01-31-raspbian.img  of=/dev/sdX
```

The `2015-01-31-raspbian.img` command will be replaced with the image file that you downloaded, and `/dev/sdX` will be replaced with your card ID; in this example, `/dev/sdb`.

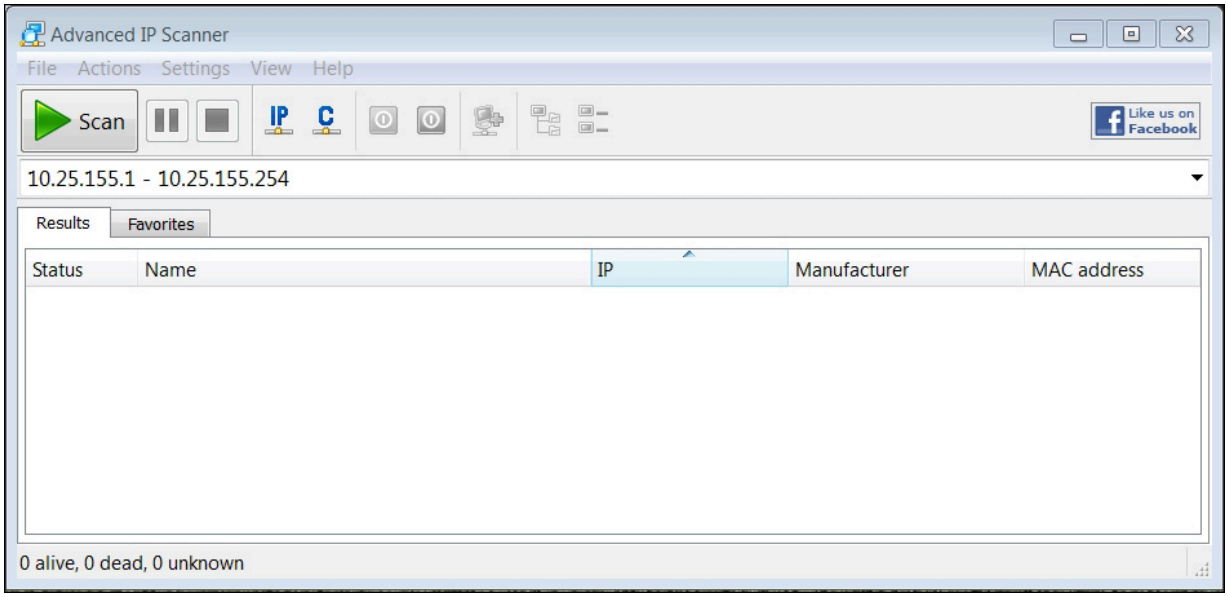
Once your card image has been created, install it on the Raspberry Pi. You'll also need to plug your Raspberry Pi into the LAN cable, and the LAN cable into your wired LAN network.

Note

If you don't have a wired connection, you can complete the following steps by connecting your Raspberry Pi directly to a monitor, keyboard, and mouse.

Power the device. The **POWER LED** should light and your device should boot from the card. To configure the card, you'll need to access it remotely. To do this, you'll now need to connect to the device via SSH, a secure protocol that allows you to control one computer remotely from another computer.

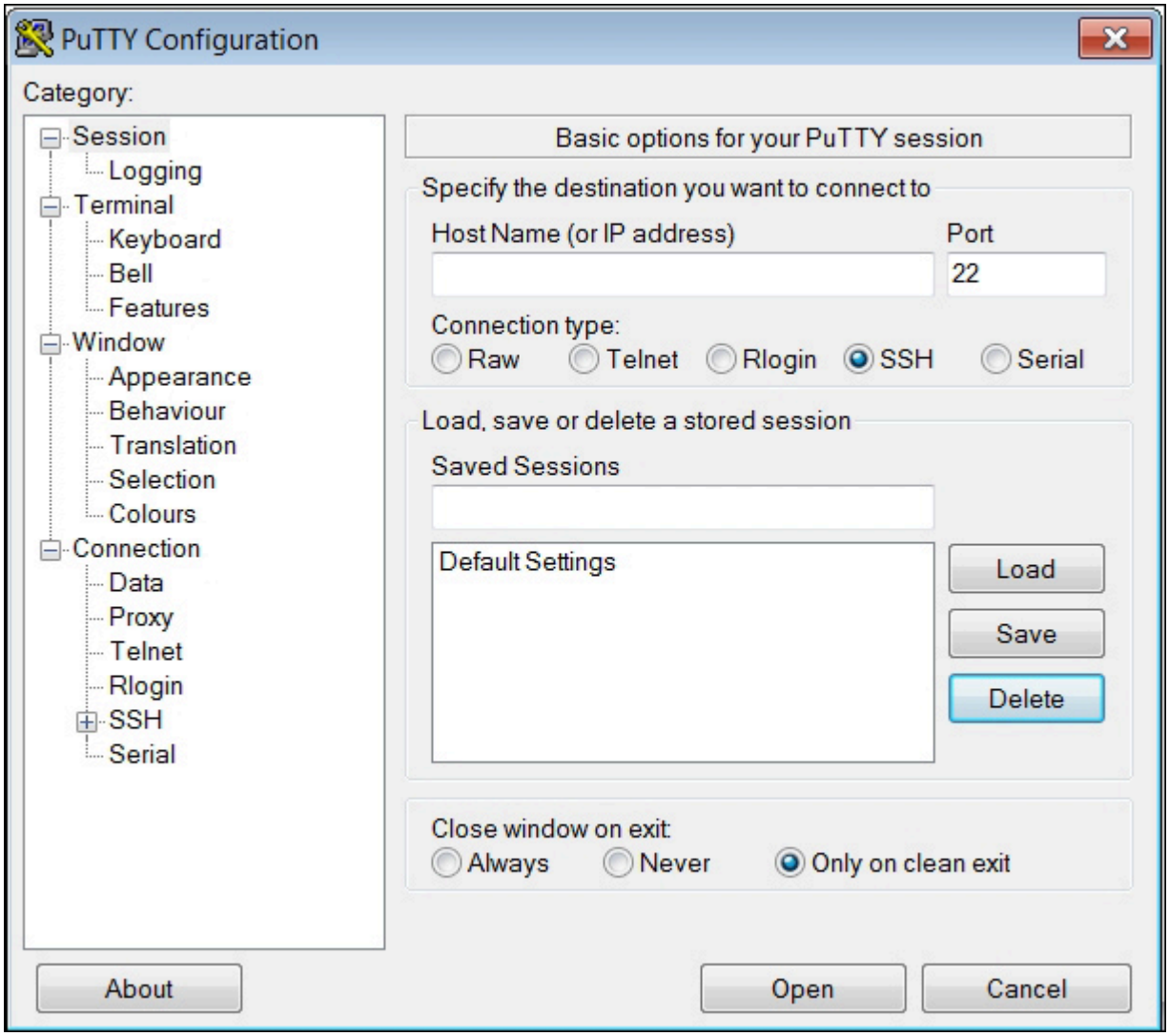
One of the challenges of accessing the system remotely is that you need to know the IP address of your board. There is a way to discover this by using an IP scanner application. There are several scanners available for free; on Windows, a possible choice is Advanced IP Scanner, which is available from http://www.advanced-ip-scanner.com/. Here is what the program looks like when it is run:



Clicking on the **Scan** selector scans for all the devices connected to the network. You can also do this in Linux; one application for IP scanning in Linux is called Nmap. To install Nmap, type in `sudo apt-get install nmap`. To run Nmap, type in `sudo nmap -sP 10.25.155.1/154` and the scanner will scan the addresses from 10.25.155.1 to 10.25.155.154.

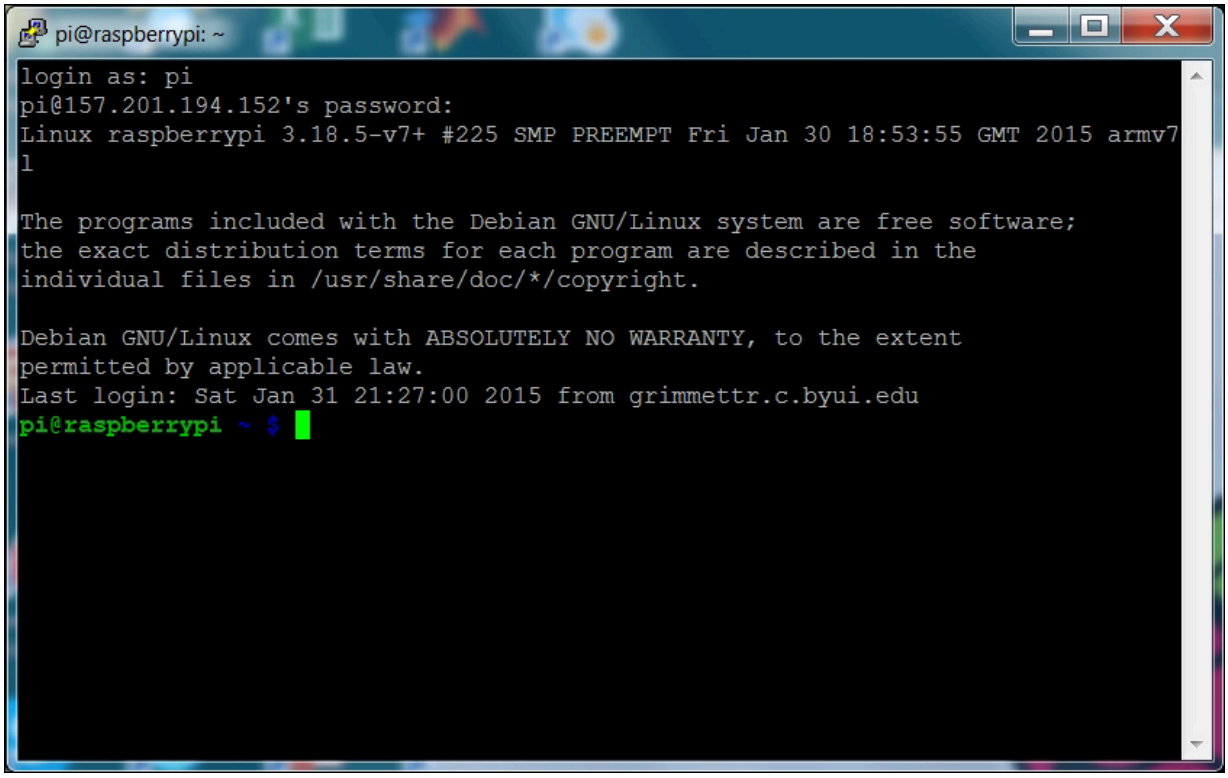
These scanners can let you know which addresses are being used, and this should then let you find your Raspberry Pi IP address. Since you are going to access your device via SSH, you'll also need an SSH terminal program running on your remote computer. If you are running Microsoft Windows, you can download such an application. One simple and easy choice is Putty. It is free and does a very good job of allowing you to save your configuration so you don't have to type it in each time. This program is available at www.putty.org.

Download Putty on your Microsoft Windows machine. Then run `putty.exe`. You should see a configuration window. It will look something like the following screenshot:



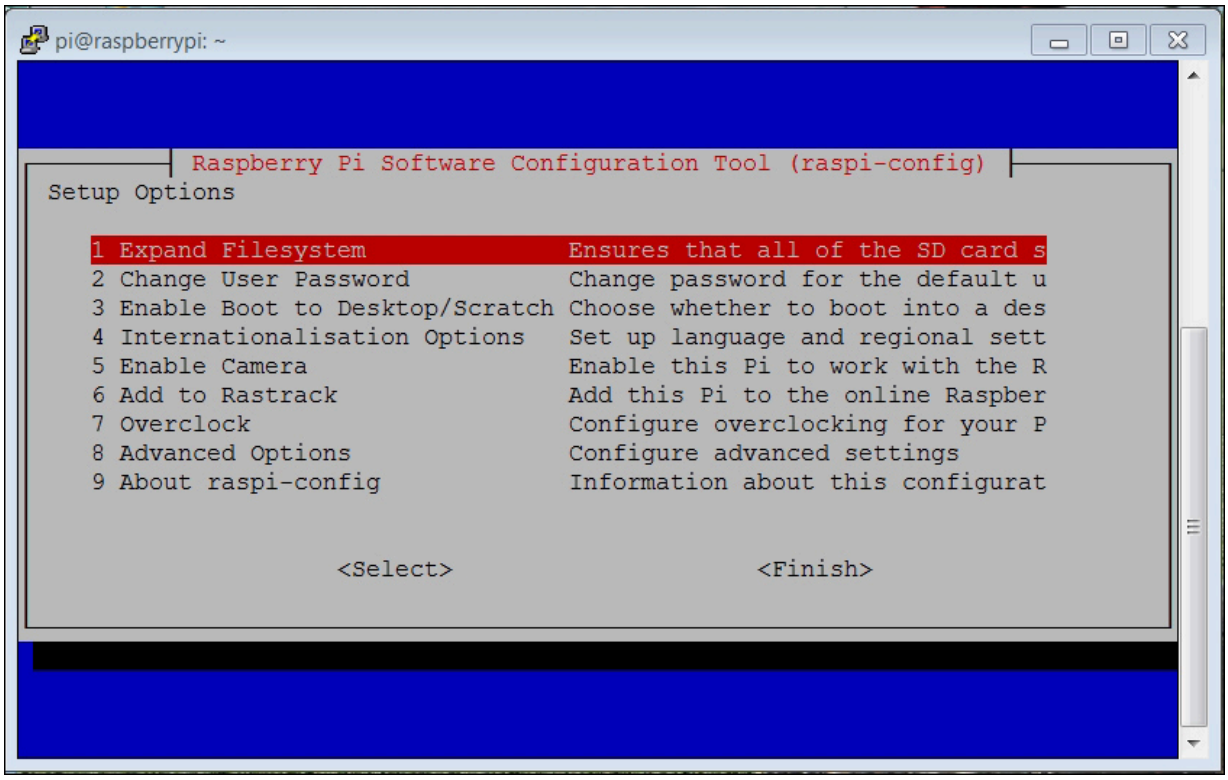
Type in the `inet addr` from the IP Scanner in the **Host Name** space and make sure that the SSH is selected. You may want to save this configuration under Raspberry Pi so you can reload it each time.

When you click on **Open**, the system will try to open a terminal window onto your Raspberry Pi via the LAN connection. The first time you do this, you will get a warning about an RSA key, as the two computers don't know about each other; so Windows is complaining that a computer it doesn't know is about to be connected in a fairly intimate way. Simply click on **OK**, and you should get a terminal with a login prompt, like the following screenshot:

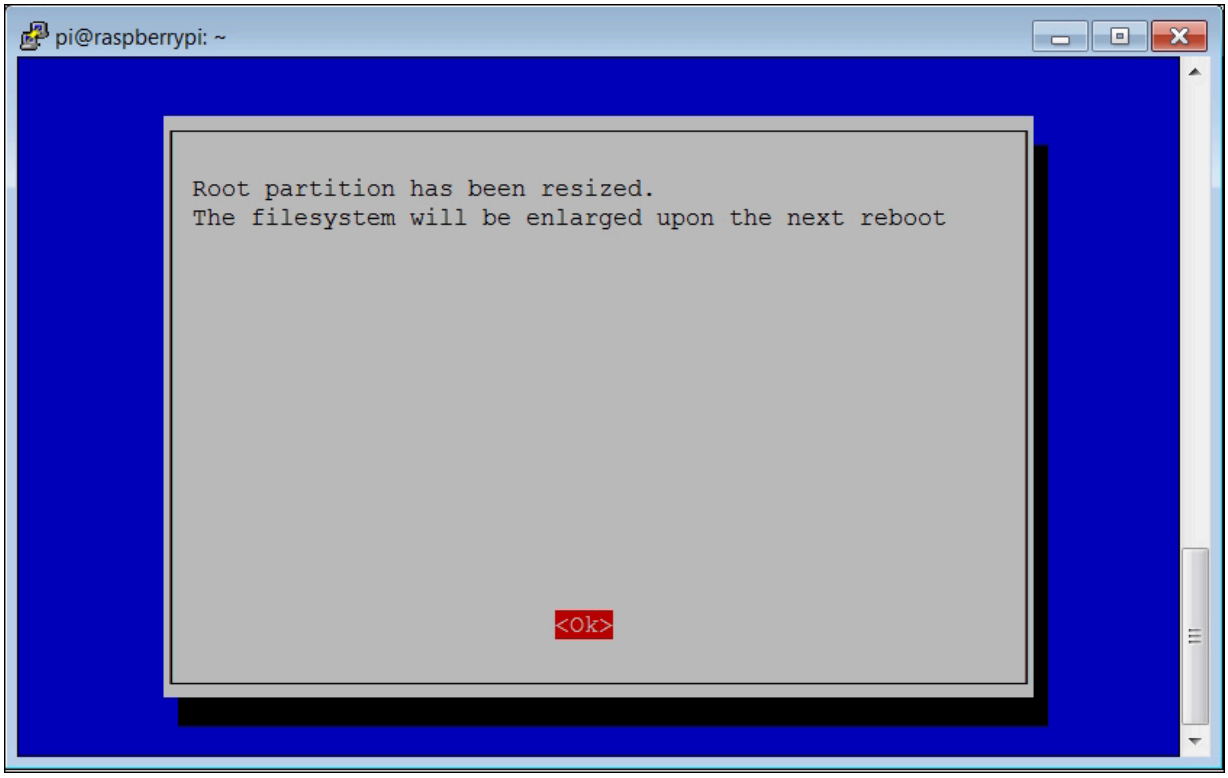


Now you can log in and issue commands to your Raspberry Pi. If you'd like to do this from a Linux machine, the process is even simpler. Bring up a terminal window and then type in `ssh pi@xxx.xxx.xxx.xxx -p 22`, where `xxx.xxx.xxx.xxx` is the `inet addr` of your device. This will then bring you to the login screen of your Raspberry Pi, which should look similar to the previous screenshot.

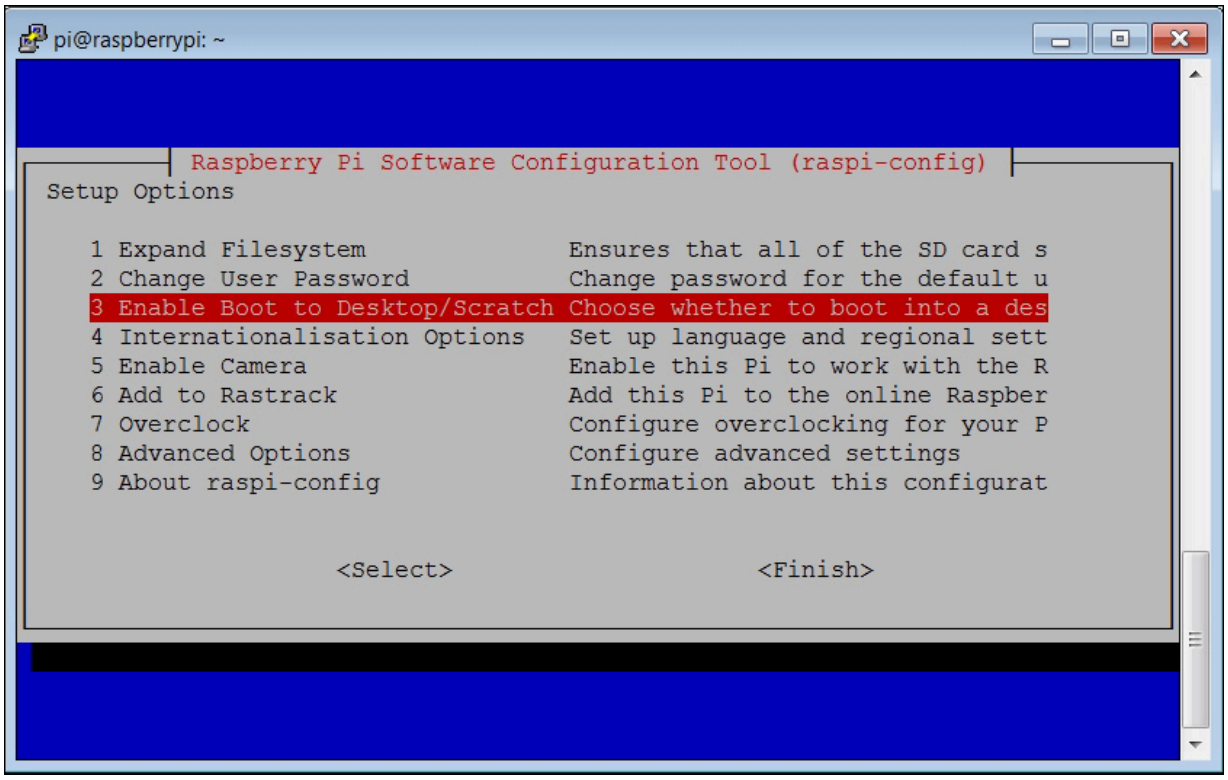
After your log in, you should get a screen that looks like the following:



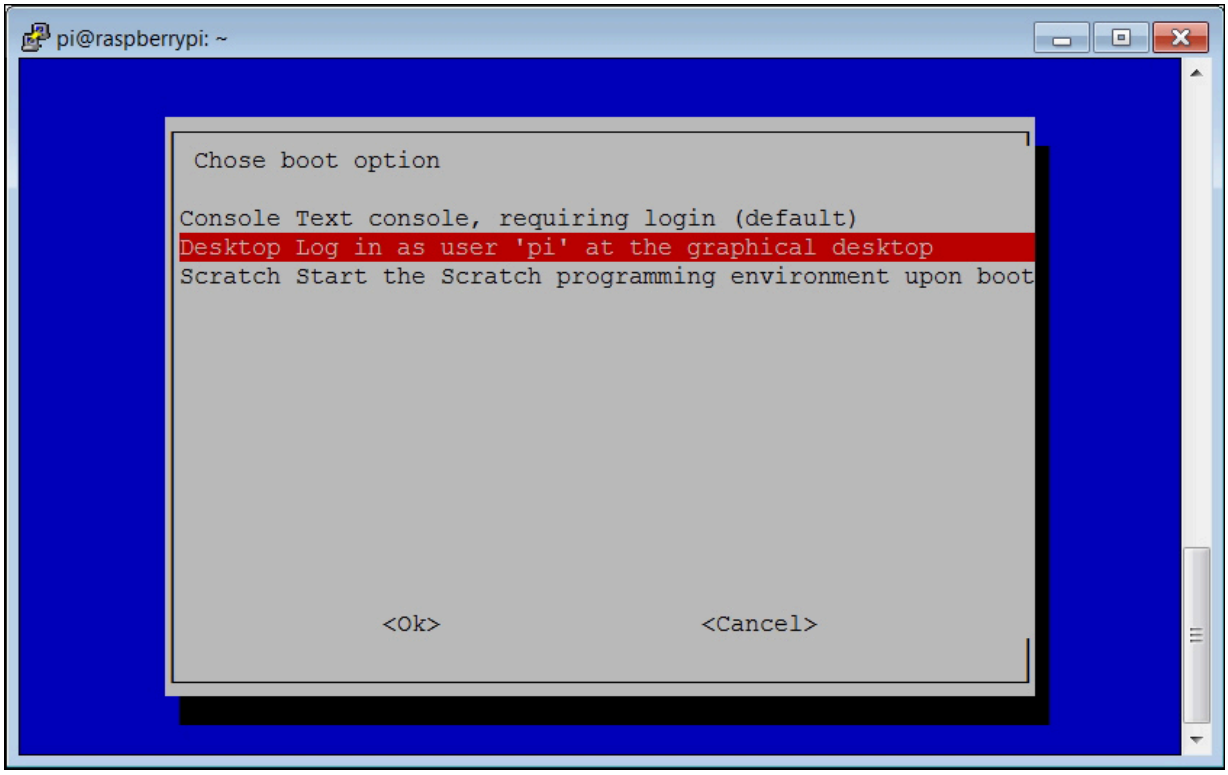
First, you'll want to expand the file system to take up the entire card. So, hit the *Enter* key, and you'll see the following screen:



Hit *Enter* once again and you'll go back to the main configuration screen. Now, select the **Enable Boot to Desktop/Scratch** option.



When you hit *Enter*, you'll see the following screen:



You can also choose to overclock your device. This is a way for you to get higher performance from your system. However, there is a risk that you can end up with a system that has reliability problems.

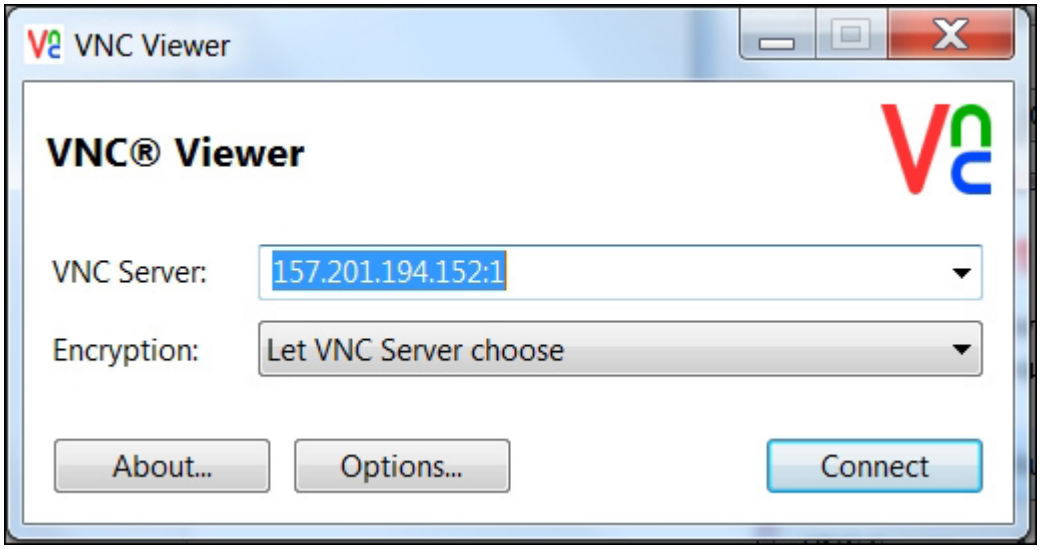
Once you are done and are back at the main configuration menu, hit the *Tab* key until you are positioned over the **<Finish>** selection, then hit *Enter*. Then, hit *Enter* again so that you can reboot your Raspberry Pi. This time, when you log in, you will not see any configuration selections. However, if you ever want to change your configuration choices, you can run the configuration tool by typing in `raspi-config` at the command prompt.

Adding a remote graphical user interface

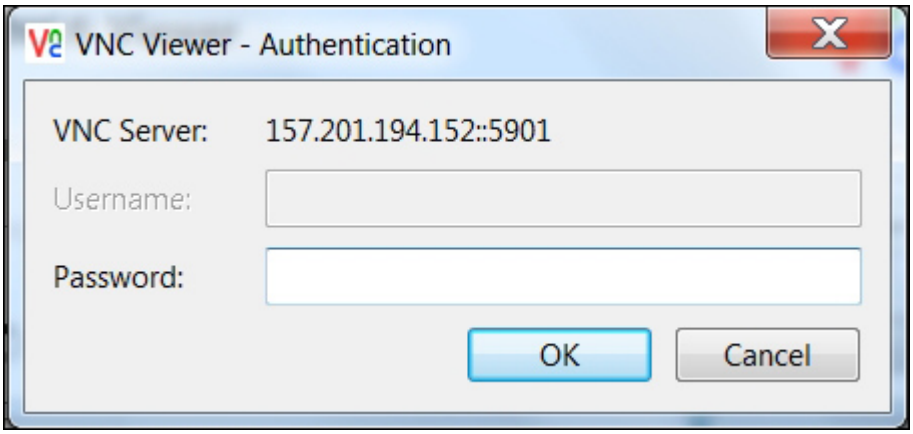
For some steps in your robot build, you will need a graphical look at your system. You can get this on your Raspberry Pi using an application called vncserver. You'll need to install a version of this on your Raspberry Pi by typing in `sudo apt-get install tightvncserver` in a terminal window on your Raspberry Pi.

Tightvncserver is an application that will allow you to remotely view your complete graphical desktop. Once you have it installed, you can do the following:

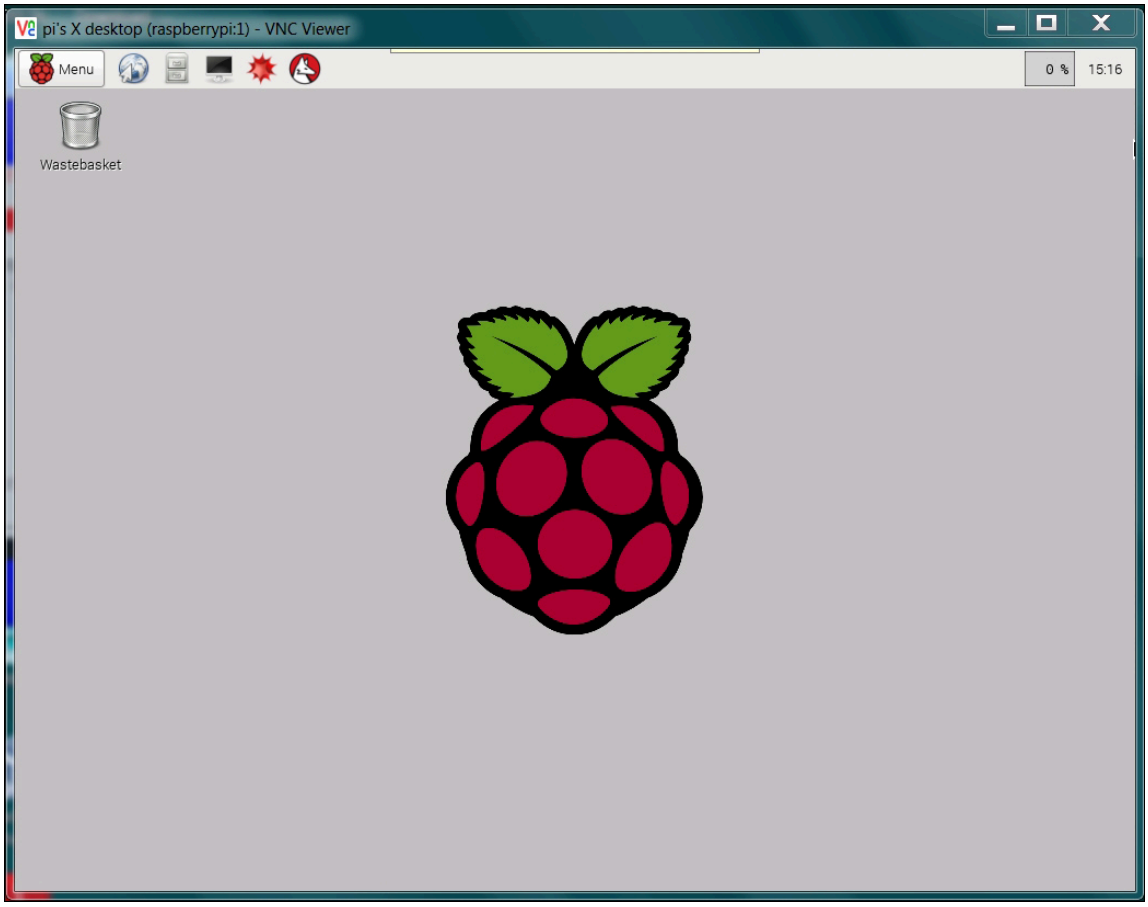
1. You'll need to start the server by typing in `vncserver` in a terminal window on the Raspberry Pi.
2. You will then be prompted for a password, prompted to verify the password, and then asked if you'd like to have a view only password. Remember the password you entered; you'll need it to remotely log in via a VNC viewer.
3. You'll need a VNC viewer application for your remote computer; a good choice is Real VNC, available from http://www.realvnc.com/download/viewer/. When you run it, you should see this:



4. Enter the VNC server address, which is the IP address of your Raspberry Pi, and click on **Connect**. You will get a warning about an unencrypted connection; select **Continue** and you will get this pop-up window:



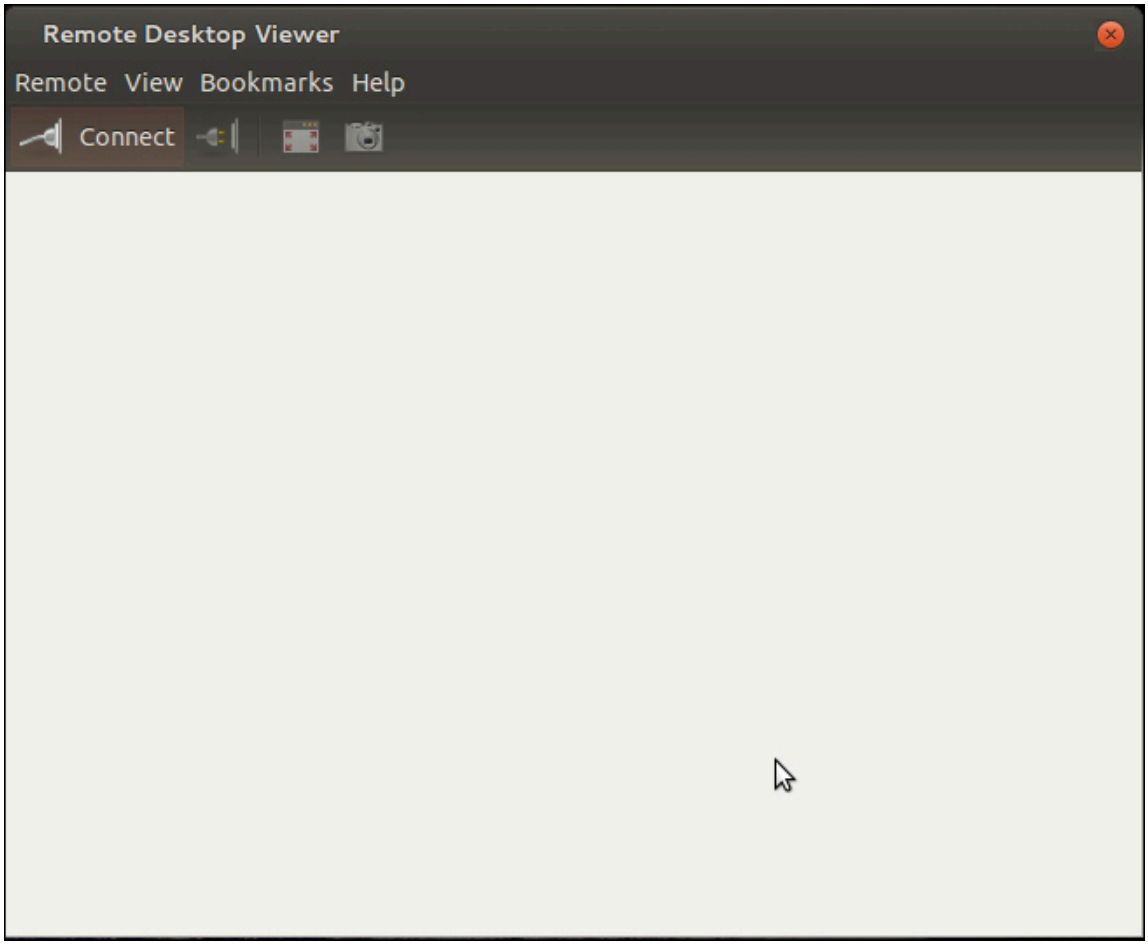
5. Type in the password you just entered while starting the vncserver, and you should then get a graphical view of your Raspberry Pi, which looks like the following screenshot:



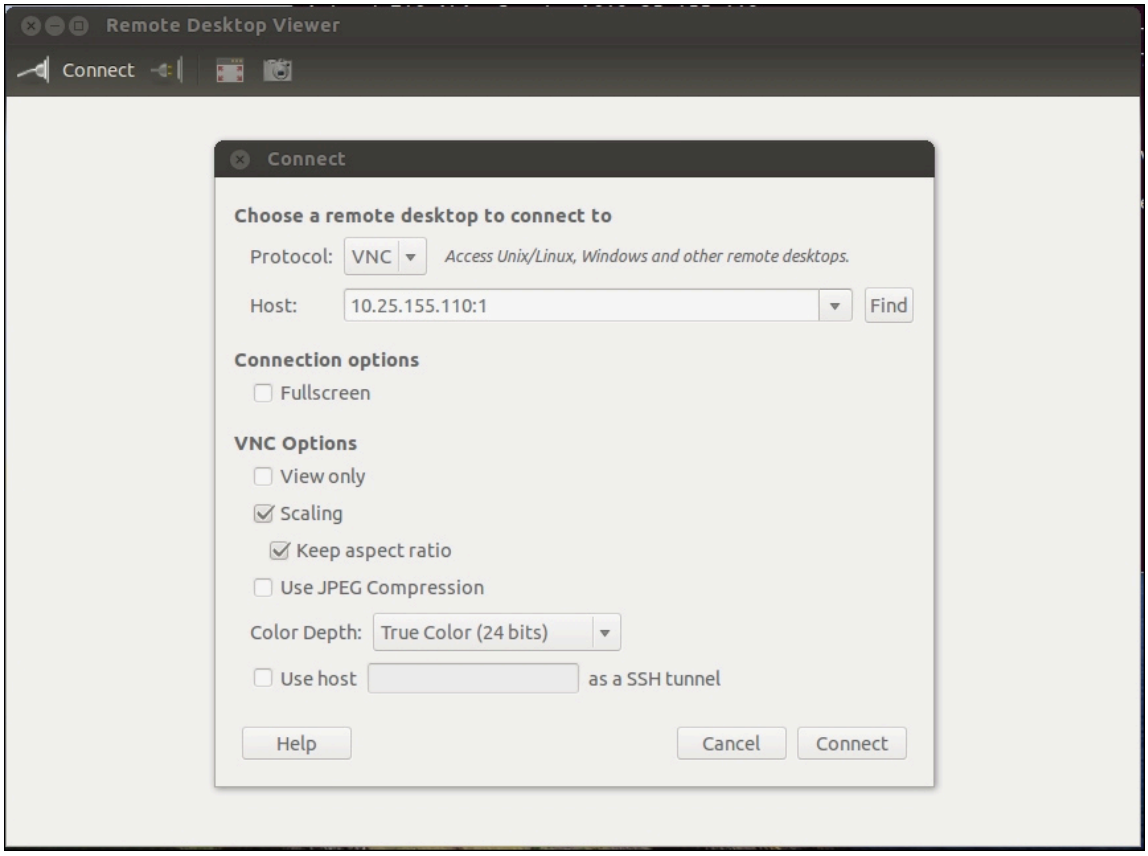
You can now access all the capabilities of your system, albeit they may be slower if you are doing a graphics-intense data transfer. To avoid having to type in vncserver each time you boot your Raspberry Pi, use the instructions at http://www.havetheknowhow.com/Configure-the-server/Run-VNC-on-boot.html.

Vncserver is also available via Linux. You can use an application called Remote Desktop Viewer to view the remote Raspberry Pi Windows system. If you have not installed this application, install it using the updated software application based on the type of Linux system you have. Once you have the software, do the following:

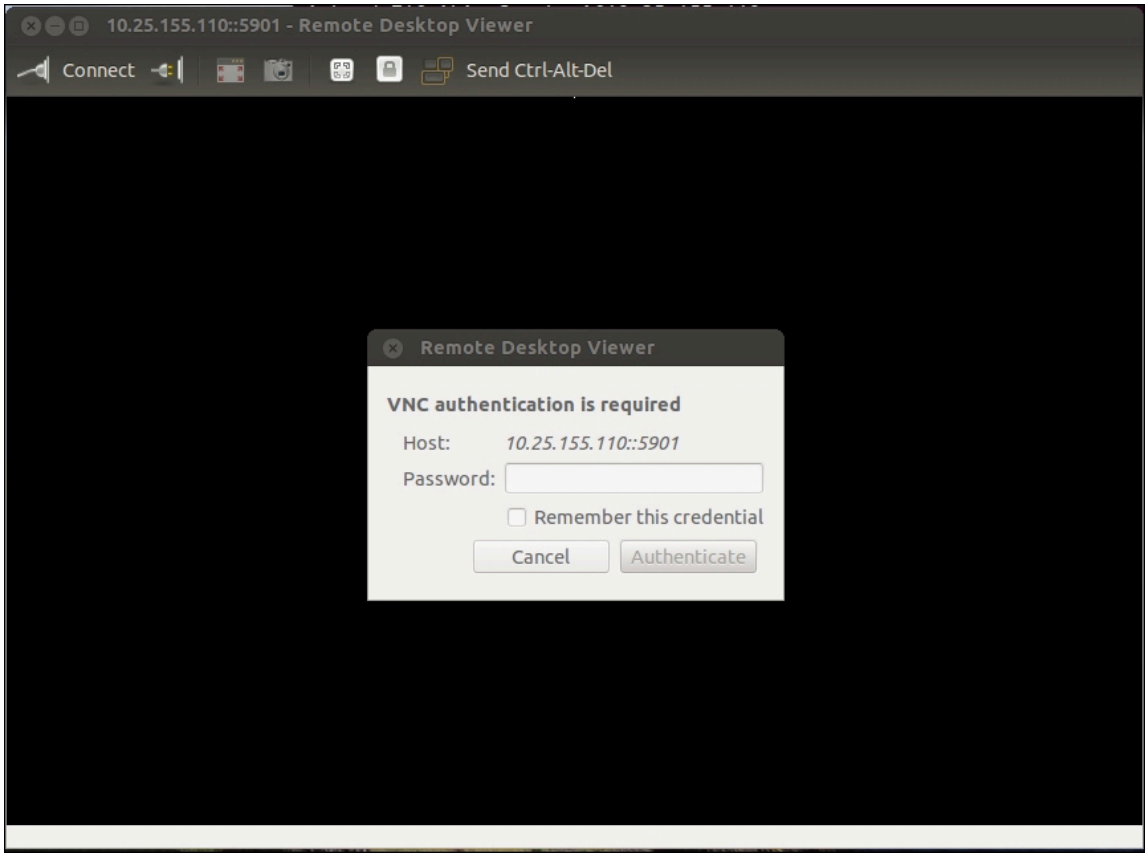
1. Run the application, and you should see the following screenshot:



2. Make sure that vncserver is running on the Raspberry Pi; the easiest way to do this is to log in using SSH and run vncserver at the prompt. Now, click on **Connect** on the **Remote Desktop Viewer**. Fill in the screen as follows. Under the **Protocol** selection, choose **VNC**, and you should see the following screenshot:



3. Now, enter the Host inet address—make sure to include `:1` at the end, and then click on **Connect**. You'll need to enter the vncserver password you set up, like the following screenshot:



You should now see the graphical screen of the Raspberry Pi. You are ready to start interacting with the system!

Establishing wireless access

Now that your system is configured, the next step is to connect your Raspberry Pi to your remote computer using wireless. To do this, you'll add a wireless USB device and configure it. See http://elinux.org/RPi_USB_Wi-Fi_Adapters to identify wireless devices that have been verified to work with Raspberry Pi. Here is one available at many online electronics outlets:



To connect to your wireless LAN, boot the system and edit the network file by typing in `sudo nano /etc/network/interfaces`. Then, edit the file to look like this:

```
pi@raspberrypi: ~
  GNU nano 2.2.6         File: /etc/network/interfaces

auto lo

iface lo inet loopback
iface eth0 inet dhcp

allow-hotplug wlan0
iface wlan0 inet dhcp
        wpa-ssid "walkPi"
        wpa-psk "12345678"

                [ Read 9 lines (Warning: No write permission) ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Reboot your device and it should now be connected to your wireless network.

Note

If you are using a US keyboard, you may need to edit the keyboard file for your keyboard to use nano effectively. To do this, type in `sudo nano /etc/default/keyboard` and change `XKBLAYOUT="gb"` to `XKBLAYOUT="us"`.

Your system has lots of capabilities. Feel free to play with the system, which will give you an understanding of what is already there and what you'll want to add from a software perspective.

Programming on Raspberry Pi

One last bit of introduction. You'll need some basic programming skills to be successful on your project. This section will touch a little on Python and C programming on the Raspberry Pi.

Creating and running Python programs on the Raspberry Pi

You'll be using Python for two reasons. First, it is a simple language that is intuitive and very easy to use. Second, a lot of open source functionality in the robotics world is available in Python. To work the examples in this section, you'll need a version of Python installed. Fortunately, the basic Raspbian system has a version already installed, so you are ready to begin.

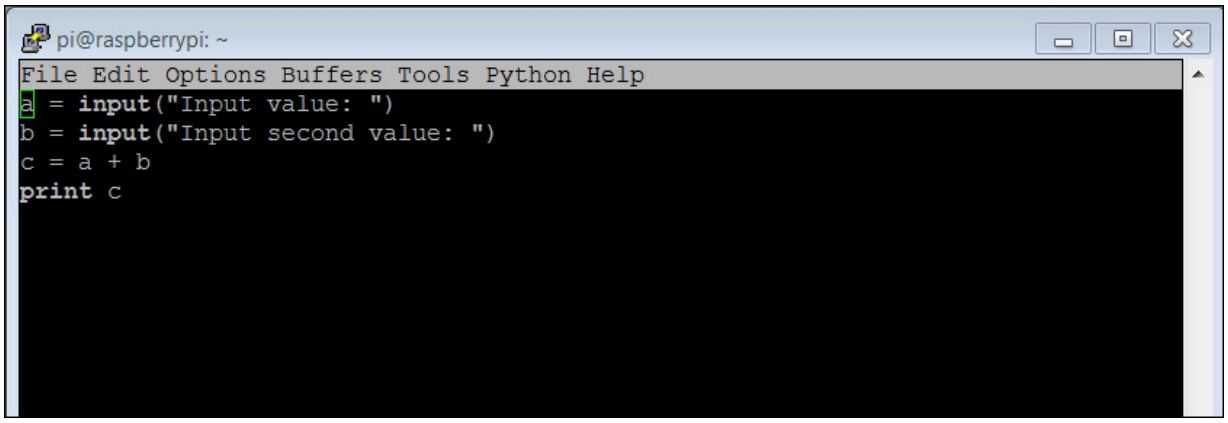
Note

If you are new to programming, there are a number of different websites that provide interactive tutorials. If you'd like to practice some of the basic programming concepts in Python using these tools, go to www.codeacademy.com or http://www.learnpython.org/ and give it a try.

But, to get you started, let's first cover how to create and run a Python file. It turns out that Python is an interactive language, so you could run Python and then type in commands one at a time. However, you want to use Python to create programs, so you are going to create Python programs and then run these programs from the command line by invoking Python.

Open an example Python file by typing in `emacs example.py`. Now, put some code in the file. Start with the lines shown in the following

screenshot:



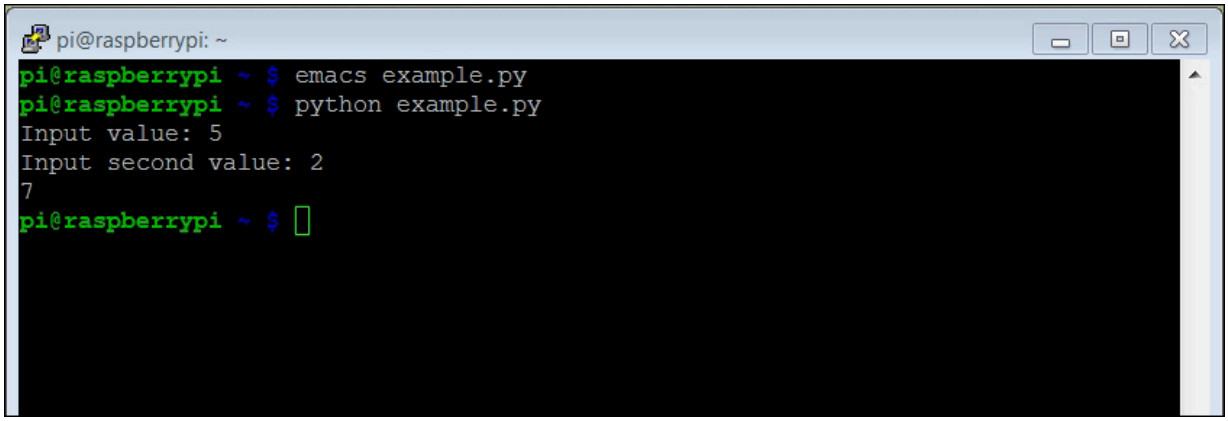
Note

Your code may be color coded. I have removed the color coding here so that it is easier to read.

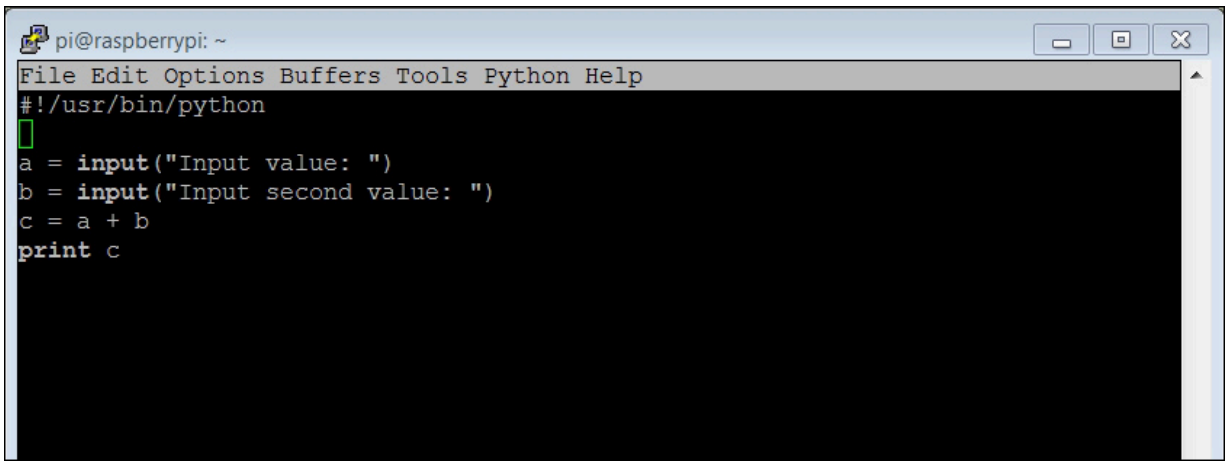
Let's go through the code to see what is happening:

1. `a = input("Input value: ")`: One of the basic needs of a program is to get input from the user. The `raw_input` part allows us to do that. The data will be input by the user and stored in `a`. The prompt `"Input value:"` will be shown to the user.
2. `b = input("Input second value: ")`: This data will also be input by the user and stored in `b`. The prompt `"Input second value:"` will be shown to the user.
3. `c = a + b`: This is an example of something you can do with the data; in this example, you can add `a` and `b`.
4. `print c`: Another basic need of our program is to print out results. The print command prints out the value of `c`.

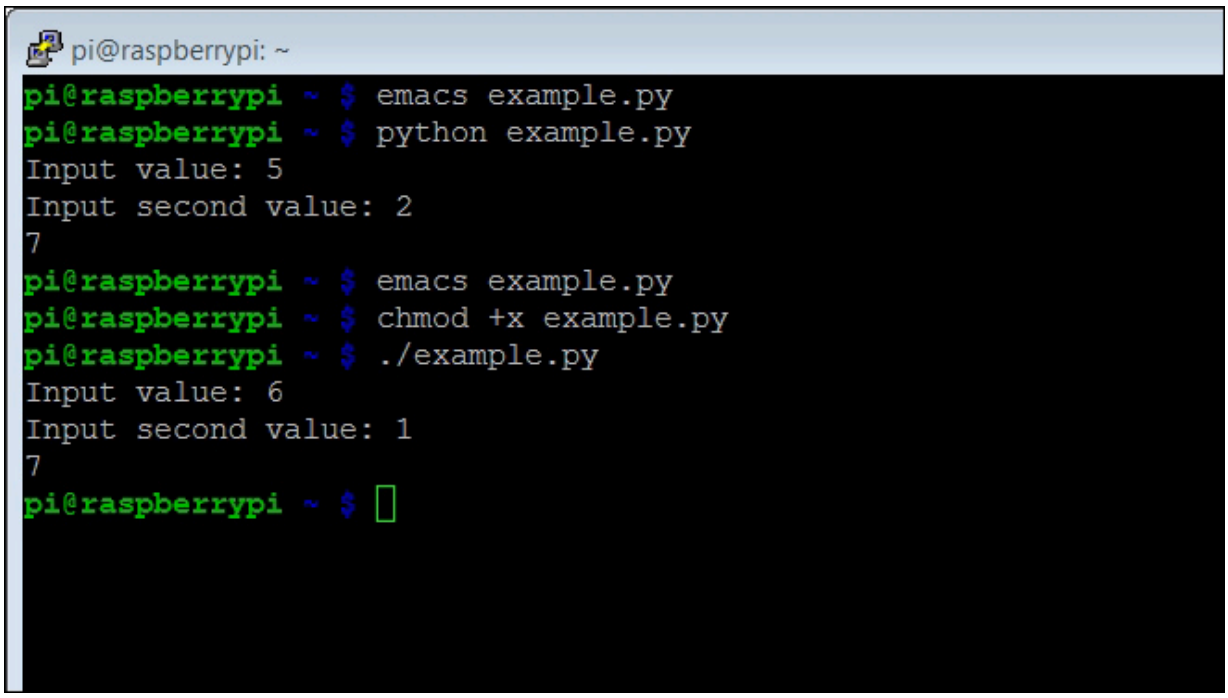
Once you have created your program, save it (using `ctrl-x ctrl-s`) and quit emacs (using `ctrl-x ctrl-c`). Now, from the command line, run your program by typing in `python example.py`. You should see something similar to the following screenshot:



You can also run the program right from the command line without typing in `python example.py` by adding one line to the program. Now, the program should look like the following screenshot:



Adding `#!/usr/bin/python` as the first line simply makes this file available for us to execute from the command line. Once you have saved the file and exited emacs, type in `chmod +x example.py`. This will change the file's execution permissions, so the computer will now believe it and execute it. You should be able to simply type in `./example.py` and the program should run, as shown in the following screenshot:



Note that if you simply type in `example.py`, the system will not find the executable file. In this case, the file has not been registered with the system, so you have to give it a path to the file. In this case, `./` is the current directory.

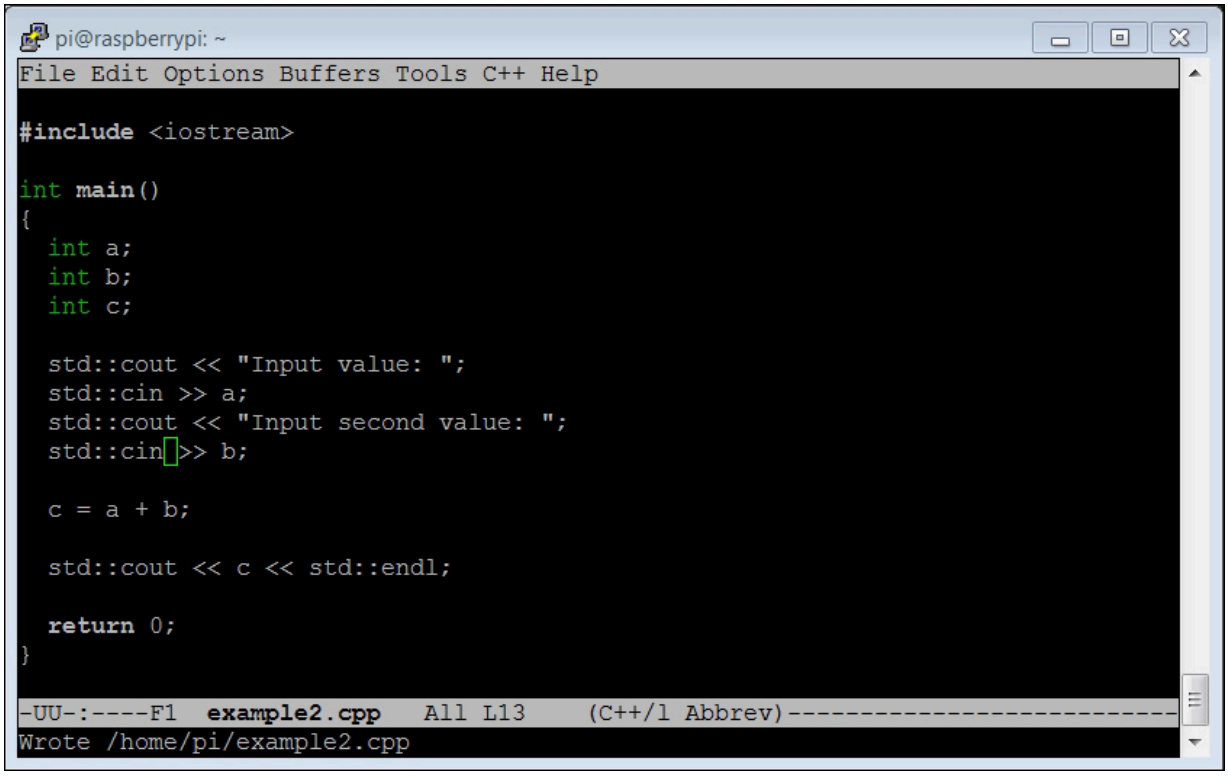
An introduction to the C/C++ programming language

Now that you've been introduced to a simple programming language in Python, you'll also need a bit of exposure to a more complex, but powerful, language called C. C is the original language of Linux and has been around for many decades, but is still widely used by open source developers. It is similar to Python, but is also a bit different, and since you may need to understand and make changes to C code, you should be familiar with it and know how it is used.

As with Python, you will need to have access to the language capabilities. These come in the form of a compiler and build system, which turns your text files into ones that contain programs to machine code that the processor can actually execute. To do this, type in `sudo apt-get install build-essential`. This will install the

programs you need to turn your code into executables for the system.

Now that the tools are installed, let's walk through some simple examples. Here is the first C/C++ code example:



The following is an explanation of the code:

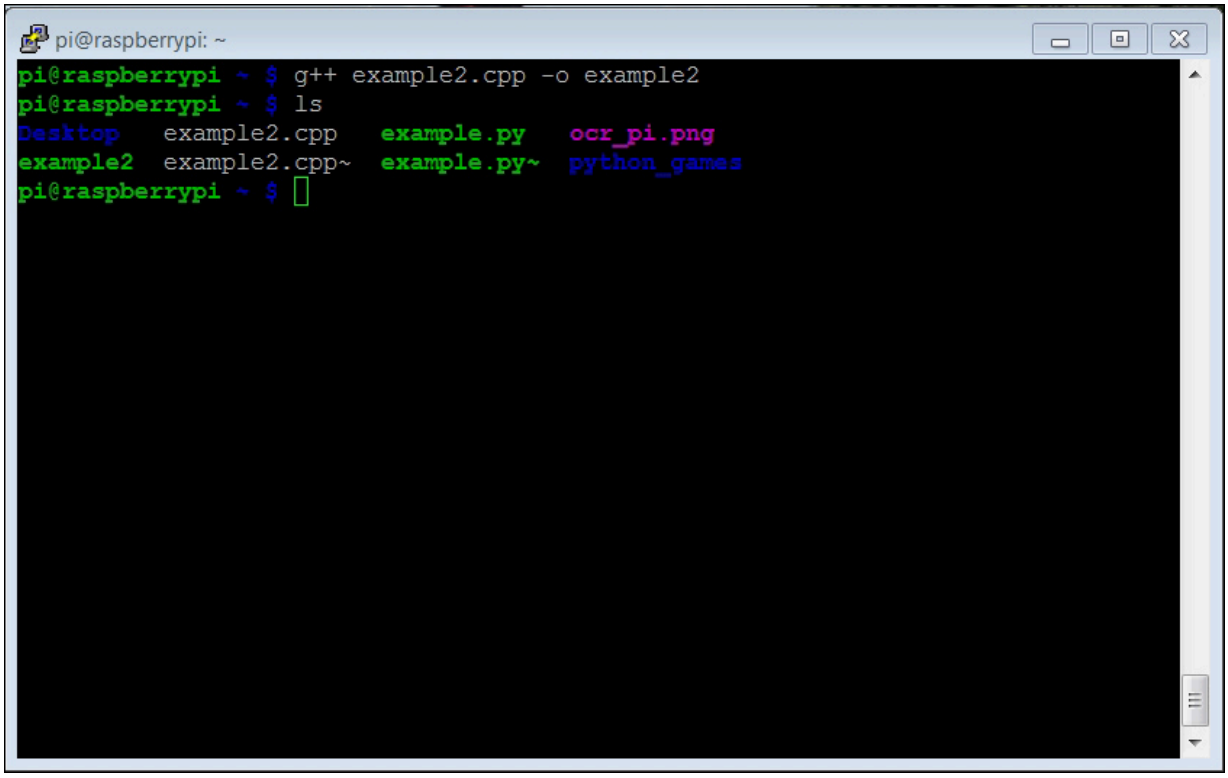
- `#include <iostream>`: This is a library that is included so that your program can input data from the keyboard and output information to the screen.
- `int main()`: As with Python, we can put functions and classes in the file, but you will always want to start execution at a known point; C defines this as the `main` function.
- `int a;`: This defines a variable named `a`, of type `int`. C is what we call a strongly typed language, which means that we need to declare the type of the variable we are defining. The normal types are `int`, a number that has no decimal points; `float`, a number that requires decimal points; `char`, a character of text,

and `bool`, a `true` or `false` value. Also note that every line in C ends with the `;` character.

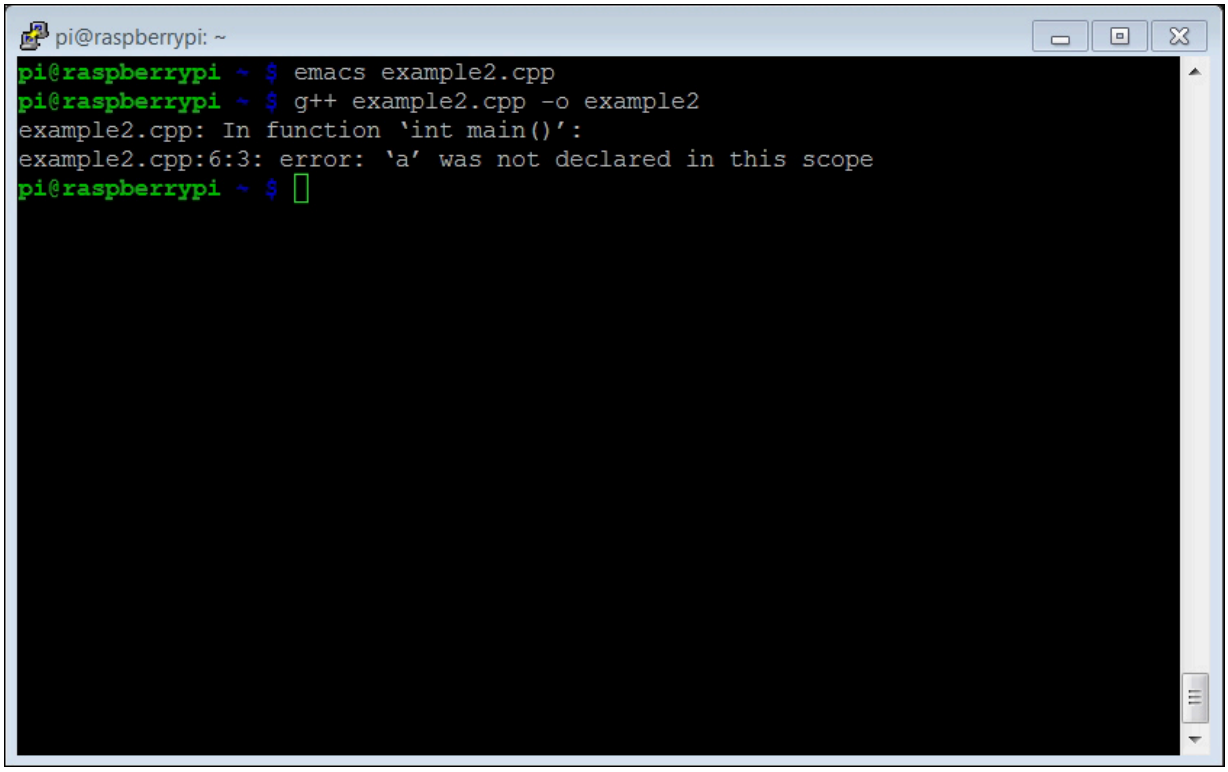
- `int b;`: This defines a variable named `b`, of type `int`.
- `int c;`: This defines a variable named `c`, of type `int`.
- `std::cout << "Input value: ";`: This will display the string `"Input value: "` on the screen.
- `std::cin >> a;`: The input that the user types will go into the variable `a`.
- `std::cout << "Input second value: ";`: This will display the string `"Input second value: "` on the screen.
- `std::cin >> b;`: The input that the user types will go into the variable `b`.
- `c = a + b`: The statement is a simple addition of two values.
- `std::cout << c << std::endl;`: The `cout` command prints out the value of `c`. The `endl` command at the end prints out a carriage return so that the next character appears on the next line.
- `return 0;`: The main function ends and returns `0`.

To run this program, you'll need to run a compile process to turn it into an executable program that you can run. To do this, after you have created the program, type in `g++ example2.cpp -o example2`. This will then process your program, turning it into a file that the computer can execute. The name of the executable program will be `example2` (as specified by the name after the `-o` option).

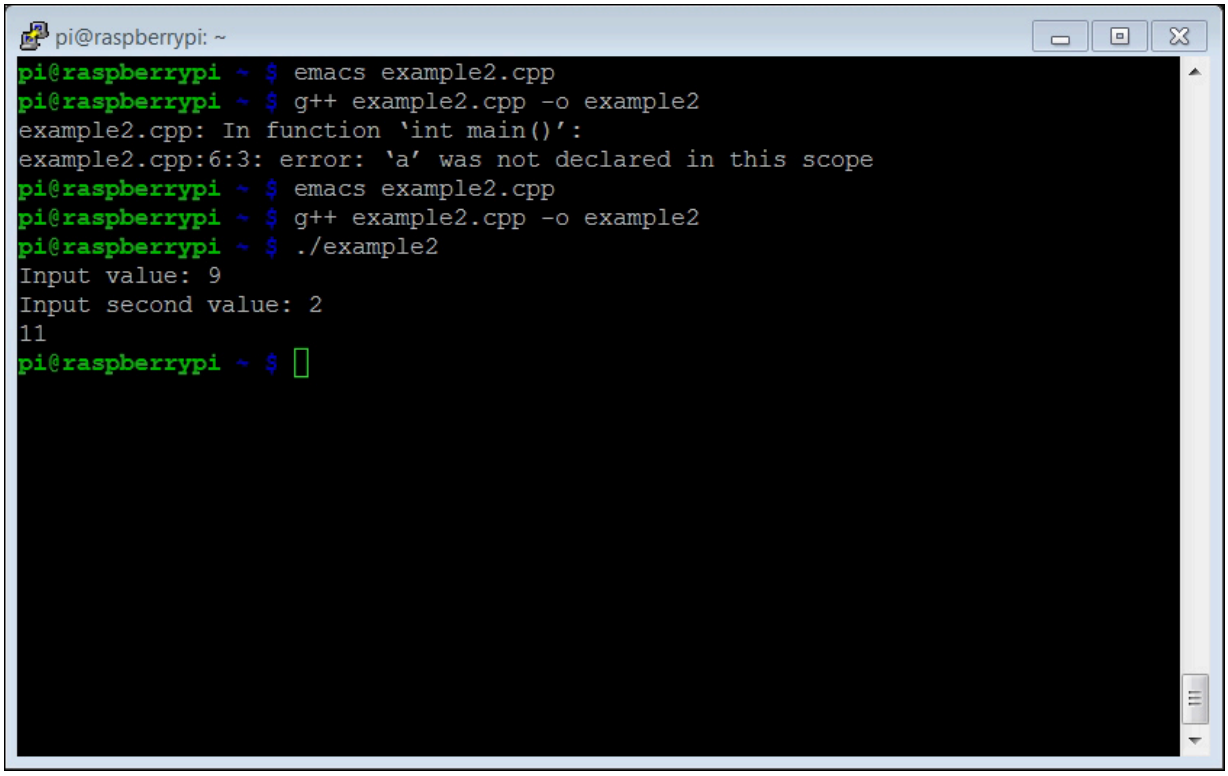
If you run an `ls` on your directory after you have compiled this, you should see the `example2` file in your directory, as shown in the following screenshot:



If you run into a problem, the compiler will try to help you figure out the problem. If, for example, you forgot the `int` before `a` in the expression `int a`, you would get the following error when you try to compile:



The error message indicates a problem in the `int main()` function and tells you that the variable `a` was not successfully declared. Once you have the file compiled, to run the executable, type in `./example2`, and you should be able to create the following result:



Note

If you are interested in learning more about C programming, there are several good tutorials out on the Internet that can help- for example, at http://www.cprogramming.com/tutorial/c-tutorial.html and http://thenewboston.org/list.php?cat=14.

There is one more aspect of C you will need to know about. The compile process that you just encountered seemed fairly straightforward. However, if you have your functionality distributed between a lot of files or need lots of libraries, the command-line approach to executing a compile can get unwieldy.

The C development environment provides a way to automate this process; it is called the make process. When using this, you create a text program named `makefile` that defines the files you want to include and compile, and then, instead of typing a long command or set of commands, you simply type in `make` and the system will

execute a compile based on the definitions in the `makefile` program. There are several good tutorials that talk more about this system-for example, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ or http://mrbook.org/tutorials/make/.

Now you are equipped to edit and create your own programming files. The next chapters will provide you with lots of opportunities to practice your skills as you translate lines of code into cool robotic capabilities.

Summary

Congratulations! You have your Raspberry Pi up and working. No gathering dust in the bin for this piece of hardware, Now, you are ready to start commanding your Raspberry Pi to do something.

The next chapter will show you how to construct your biped robot.

Chapter 2. Building the Biped

Now that you've got your Raspberry Pi 2 Model B all configured and ready to go, you'll need to add some hardware to control and interface. In this chapter, you'll learn:

- How to build a basic 10 **Degrees of Freedom** (**DOF**) biped
- How to use a servo motor controller connected to the USB port of the Raspberry Pi to control the servos to make your robot move

Building robots that can walk

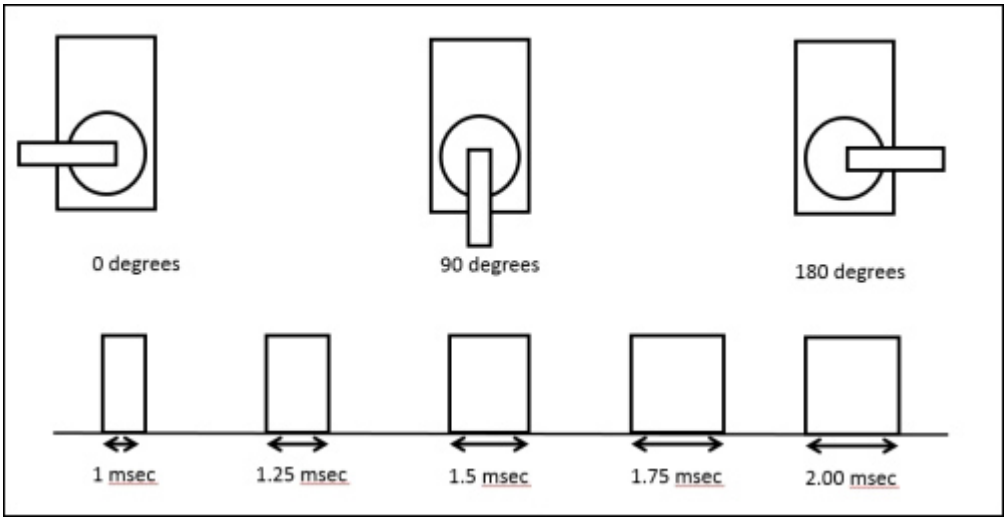
There are several choices when considering how to create a mobile robot. One of the more interesting choices is a robot that can walk. This normally comes in three versions: a biped robot with two legs, a biped robot with four legs, and a hexapod robot with six legs. While each offers an interesting and different set of possibilities, in this chapter, you'll build a basic, 10 DOF biped.

You'll be using a total of 10 servos for your project, as each leg has 5 points that can move, or 5 degrees of freedom (DOF). As servos are the most critical component of this project, it is perhaps useful to go through a tutorial on servos and learn how to control them.

How servo motors work

Servo motors are somewhat similar to DC motors. However, there is an important difference. While DC motors are generally designed to move in a continuous way—rotating 360 degrees at a given speed—servos are generally designed to move to a limited set of angles. In other words, in the DC motor world, you generally want your motors to spin with a continuous rotation speed that you control. In the servo world, you want your motor to move to a specific position that you control.

This is done by sending a **Pulse-Width-Modulated** (**PWM**) signal to the control connector of the servo. The length of this pulse will control the angle of the servo like this:



These pulses are sent out with a repetition rate of 60 Hz. You can position the servo to any angle by setting the correct control pulse.

Building the biped platform

There are several approaches to building your biped platform. Perhaps the most simple is to purchase a set of basic parts; this is the example you'll see in this chapter. There are several kit possibilities out there, including one at http://www.robotshop.com/en/lynxmotion-biped-robot-scout-bps-ns-servos.html, a kit like the one offered at http://www.ebay.com/itm/10-DOF-Biped-Robot-Mechanical-Leg-Robot-Servo-Motor-Bracket-NO-Servo-Motor-good-/131162548695?pt=LH_DefaultDomain_0&hash=item1e89e5a9d7, or the one at http://www.amazon.com/gp/product/B00DR7GA4I/ref=oh_aui_detailpage_o04_s00?ie=UTF8&psc=1. This is the specific kit we'll use in this chapter.

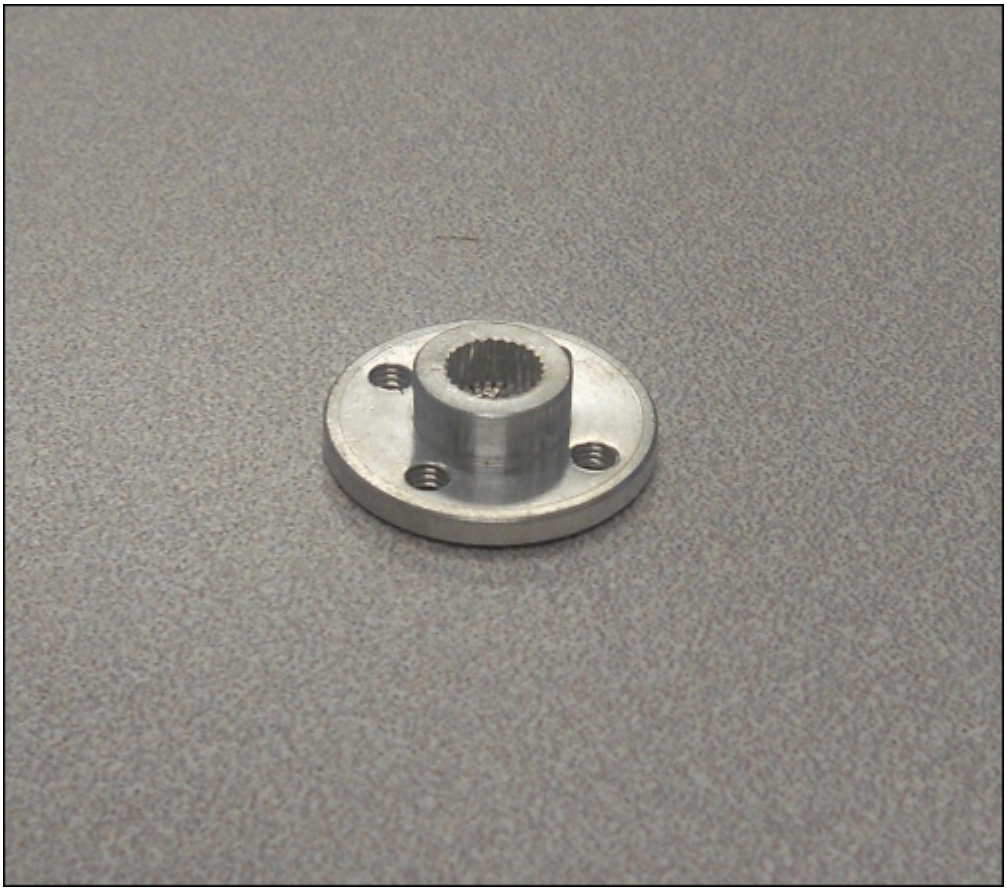
In the end, your biped will work more like the legs of a Tyrannosaurus Rex of a human, but this will make it easier to program, and it will power down more gracefully. It will also be a bit more stable.

You'll also need 10 standard size servos. There are several possible choices, but **Hitec** servos are relatively inexpensive servos and you can get them at most hobby shops and online electronics retailers. One of the important steps in this process is to select the model of the servo. Servos come in different model numbers, primarily based on the amount of torque they can generate.

Torque is the force that the servo can exert to move the part connected to it. In this case, your servos will need to lift and move the weight associated with your biped, so you'll need a servo with enough torque to do this. However, there are different torque needs for your biped robot. The angle joints will not lift the entire leg, so they can be servos with a lower servo rating, for example, the HS-422 servos. For the knee servo, you'll need a more powerful servo. In this case, I suggest that you use model HS-645MGHB servos. The hip joint that lifts the leg is where you will need the most torque

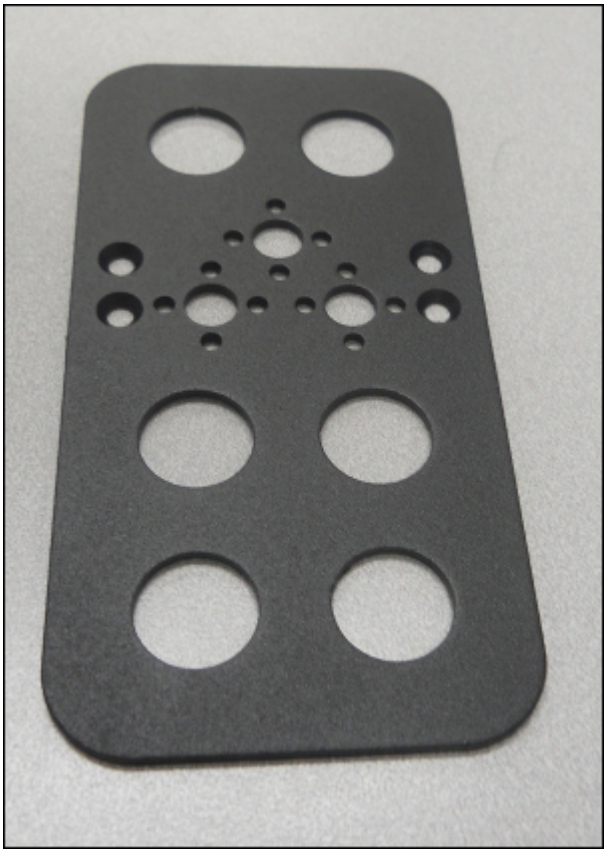
to be able to lift the leg. Here, too, I suggest that you use the model HS-645MG servos. You can also just use 10 HS-645MG servos for all the servos, but they are more expensive, so using different servos will save you some money.

One final piece that you'll need is some metal servo horns. These servo horns are optional, but they will make your biped robot much more solid than the plastic servo horns that normally come with the servos. Here is a picture of one of these horns:

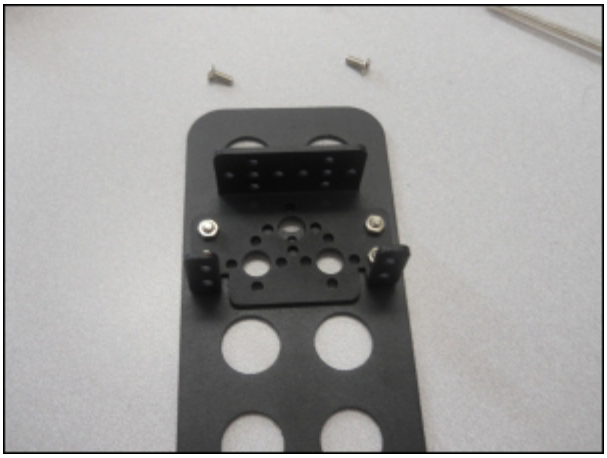


Here are the steps to assemble the biped:

1. Attach the first ankle servo to the foot. To do this, find the foot plate, as shown here:



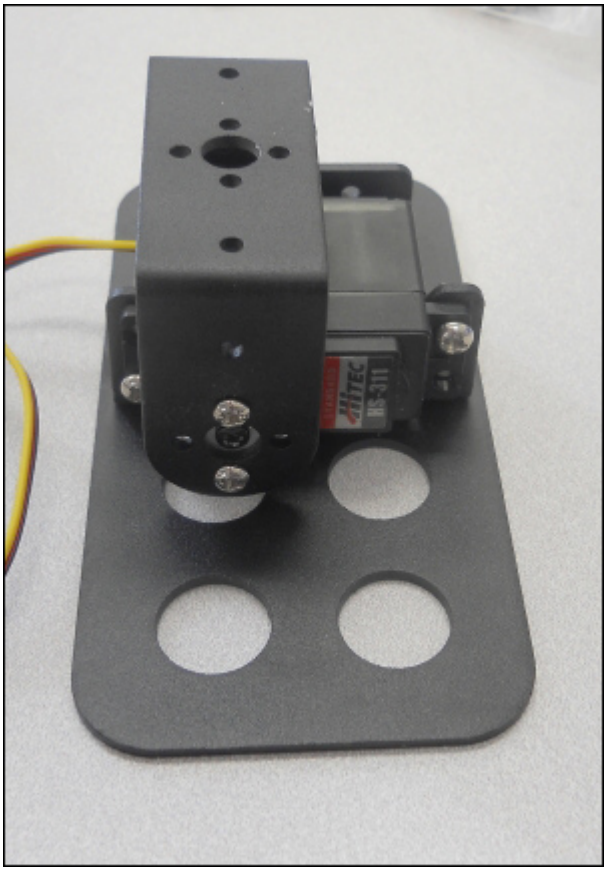
This is the bottom of the foot. Notice the beveled holes-you'll be using bevel-headed bolts to connect a servo bracket to the foot plate, as shown here:



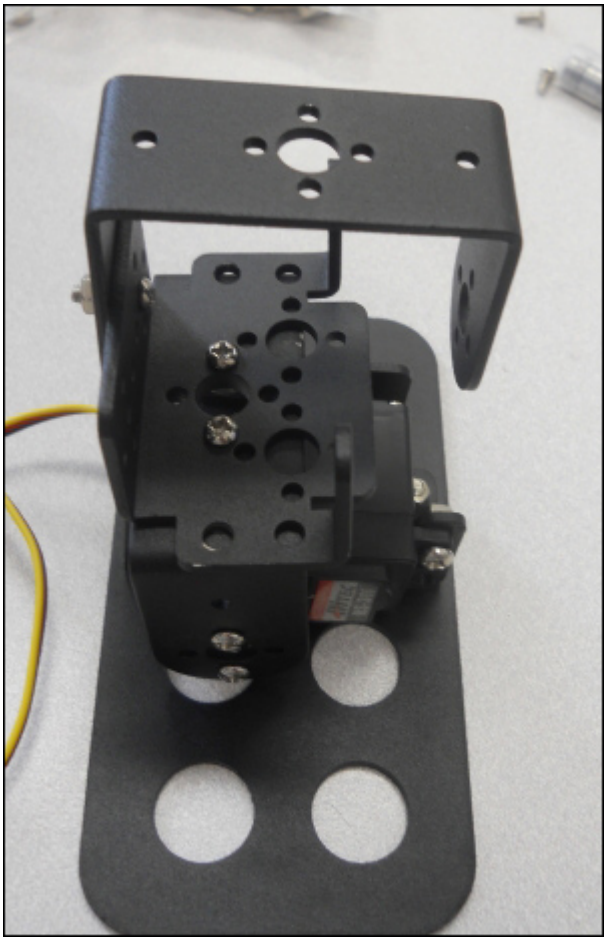
2. Make sure that the bottom of the foot is flat. Before you mount the servo to this bracket, you'll first connect a U-shaped bracket to this servo bracket using one of the bearings in the kit, like this:



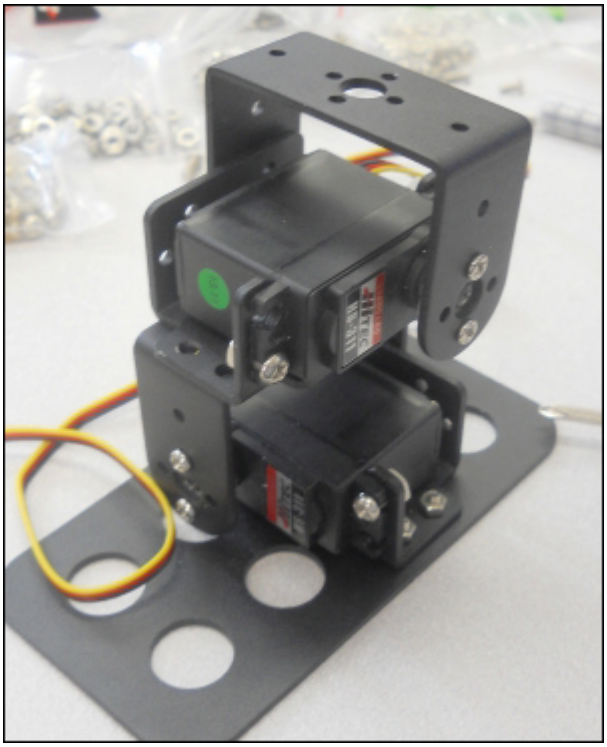
3. Finally, mount one of the servos in the bracket and connect the U-shaped bracket to the servo horn, as shown here:



4. This first servo should allow your biped to move the foot, tipping it side to side. Now, add the second ankle servo to the foot. This will allow your biped to tip the ankle front to back. To do this, connect a servo bracket to the assembly you just created, and then add a U bracket to this assembly, like this:



5. Now, add the bracket to this assembly, like this:



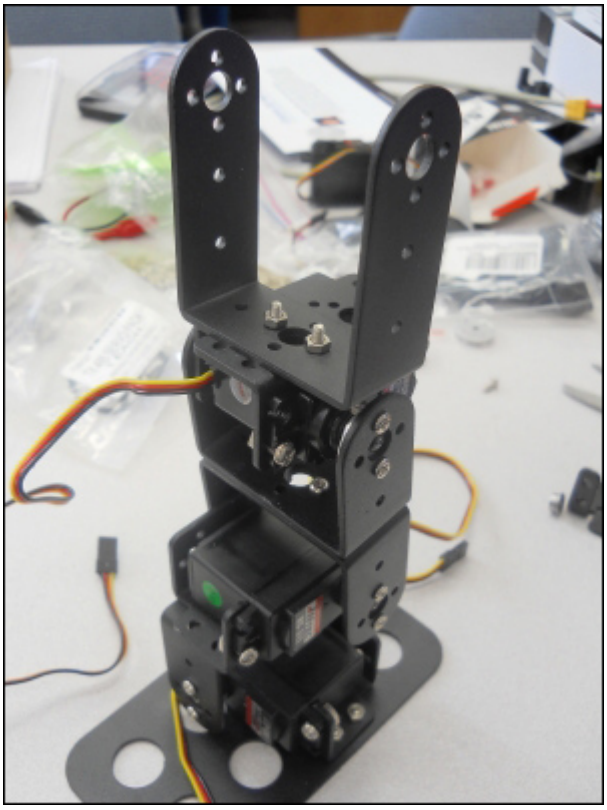
6. Now, you can add the knee servo to your biped. However, you'll first want to connect the upper leg, the longest U bracket, to a servo bracket, like this:



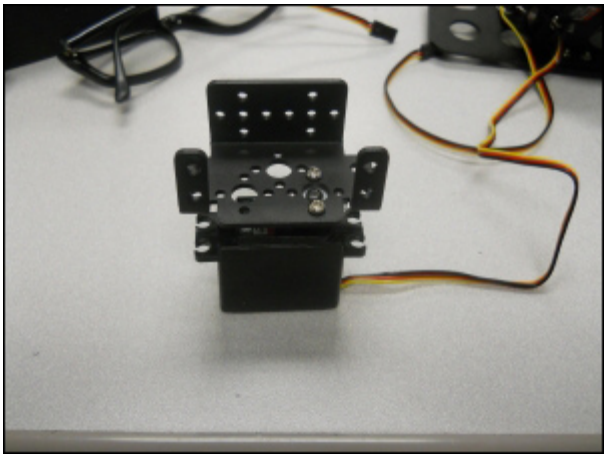
7. Now, connect this assembly, using another bearing, to the lower leg that you have already built, like this:



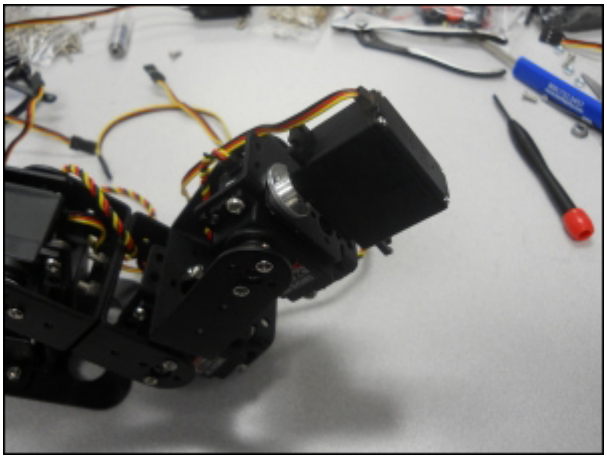
8. Now, you can mount the knee servo in this place. If you have different servos, use a more powerful servo in the knee joint. Here is a picture:



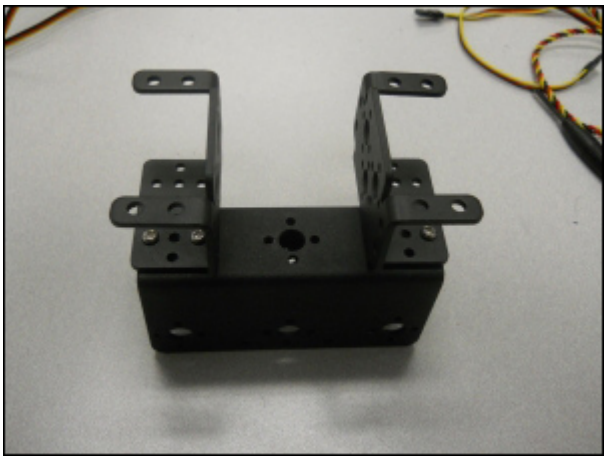
9. The last step is to put the hip together. First, you'll put the servo that turns the leg, connecting it to a servo bracket, as shown here:



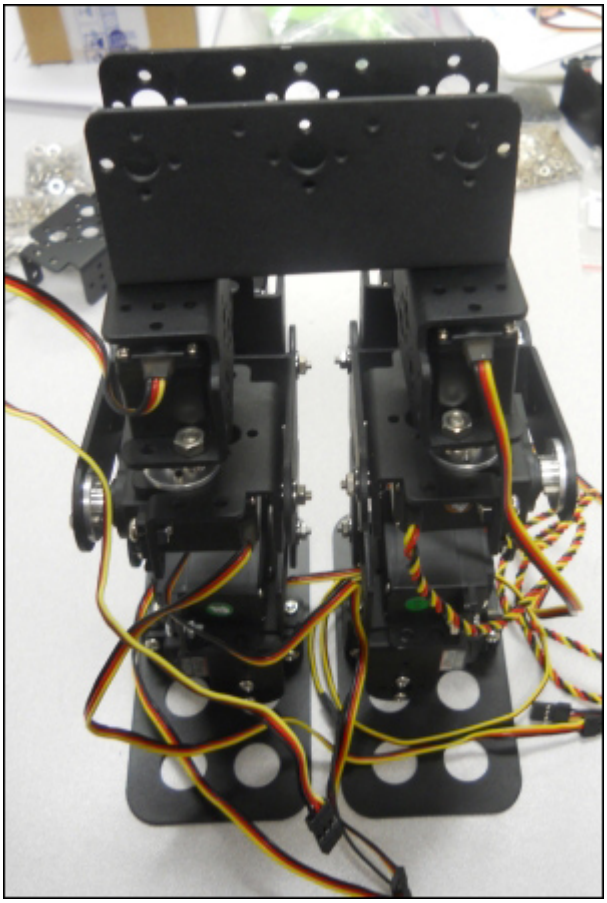
10. Now, connect this servo bracket to the long U bracket, and mount the servo that lifts the entire leg. This is another place; if you are using different servos, you'll want to use a servo with a significant torque. The entire assembly should look like this:



11. Put the other leg together. It will be a mirror image of the first leg.
12. Now, you'll connect both legs to the hip by first connecting a servo bracket connector to the hip piece, in two places, like this:



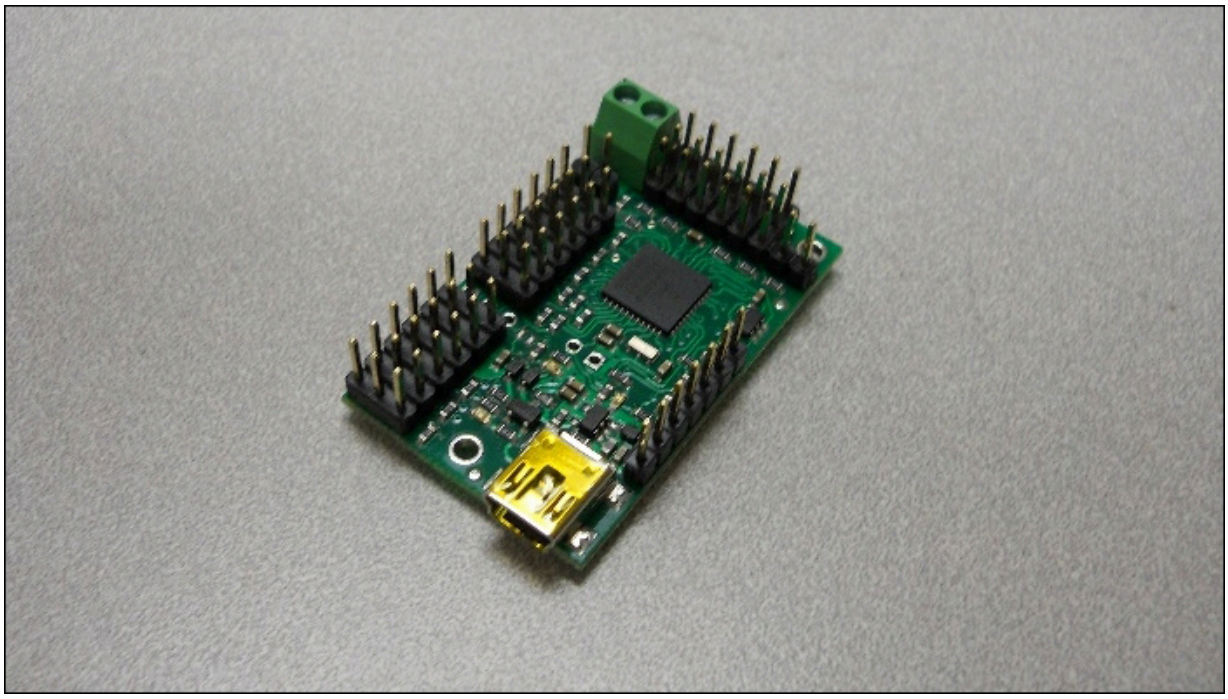
13. Finally, mount the top of the leg servos into the brackets, like this:



Your biped is now ready to walk. Now that you have the basic hardware assembled, you can turn your attention to the electronics.

Using a servo controller to control the servos

To make your biped walk, you first need to connect the servo motor controller to the servos. The servo controller you are going to use for this project is a simple servo motor controller utilizing the USB from Pololu—Pololu item number 1354 is available at pololu.com—that can control 18 servo motors. Here is a picture of the unit:



Make sure that you order the assembled version. This piece of hardware will turn USB commands from Raspberry Pi into signals that control your servo motors. Pololu creates a number of different versions of this controller, and each one is able to control a certain number of servos. In this case, you may want to choose the 18 servo version, so that you can control all 12 servos with one controller, and you may also add an additional servo to control the direction of a camera or sensor. You could also choose the 12 servo version. One advantage of the 18 servo controller is the ease of connecting power to the unit via screw-type connectors.

There are two connections you'll need to make to the servo controller in order to get started: the first to the servo motors and the second to a battery.

First, connect the servos to the controller. In order to be consistent, let's connect your 12 servos to the connections marked 0 through 11 on the controller using this configuration:

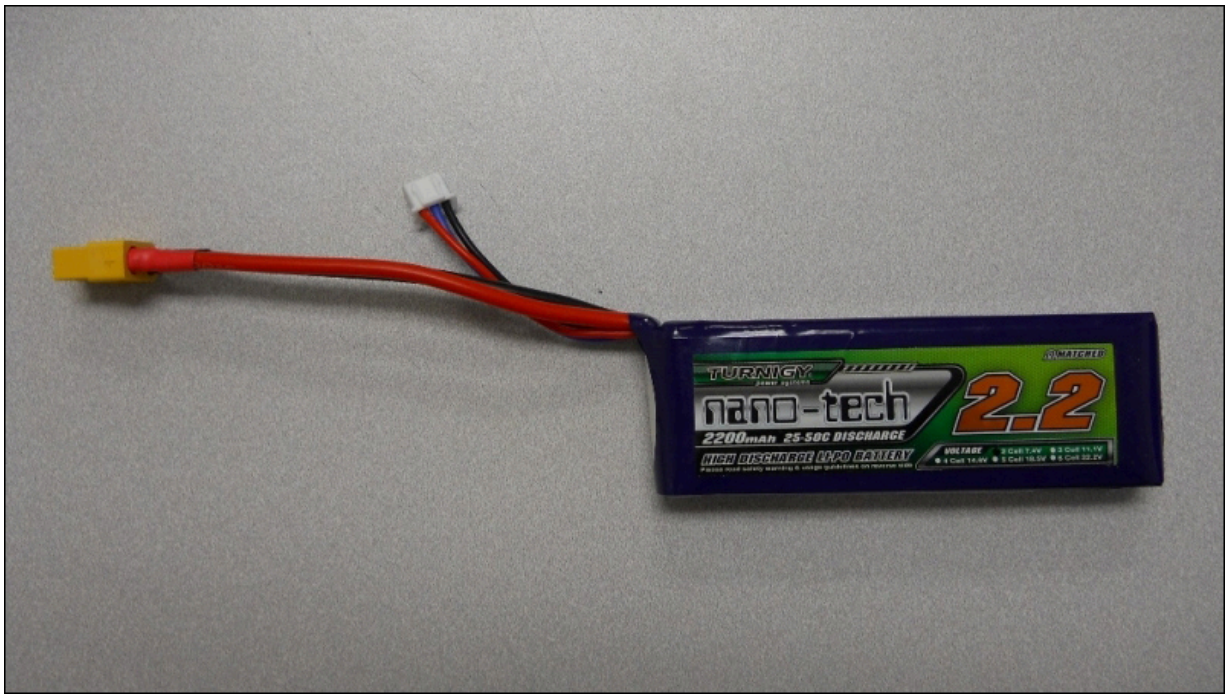
Servo Connector	**Servo**
0	Right ankle in/out
1	Right ankle front/back
2	Right knee
3	Right hip up/down
4	Right hip turn
5	Left ankle in/out
6	Left ankle front/back
7	Left knee

Servo Connector	Servo
8	Left up/down
9	Left hip turn

Here is a picture of the back of the controller; this will tell us where to connect our servos:



Now, you need to connect the servo motor controller to your battery. For this project, you can use a 2S RC LiPo battery; it will supply the 7.4 volts and the current required by your servos, which can be on the order of 2 amps. Here is a picture:

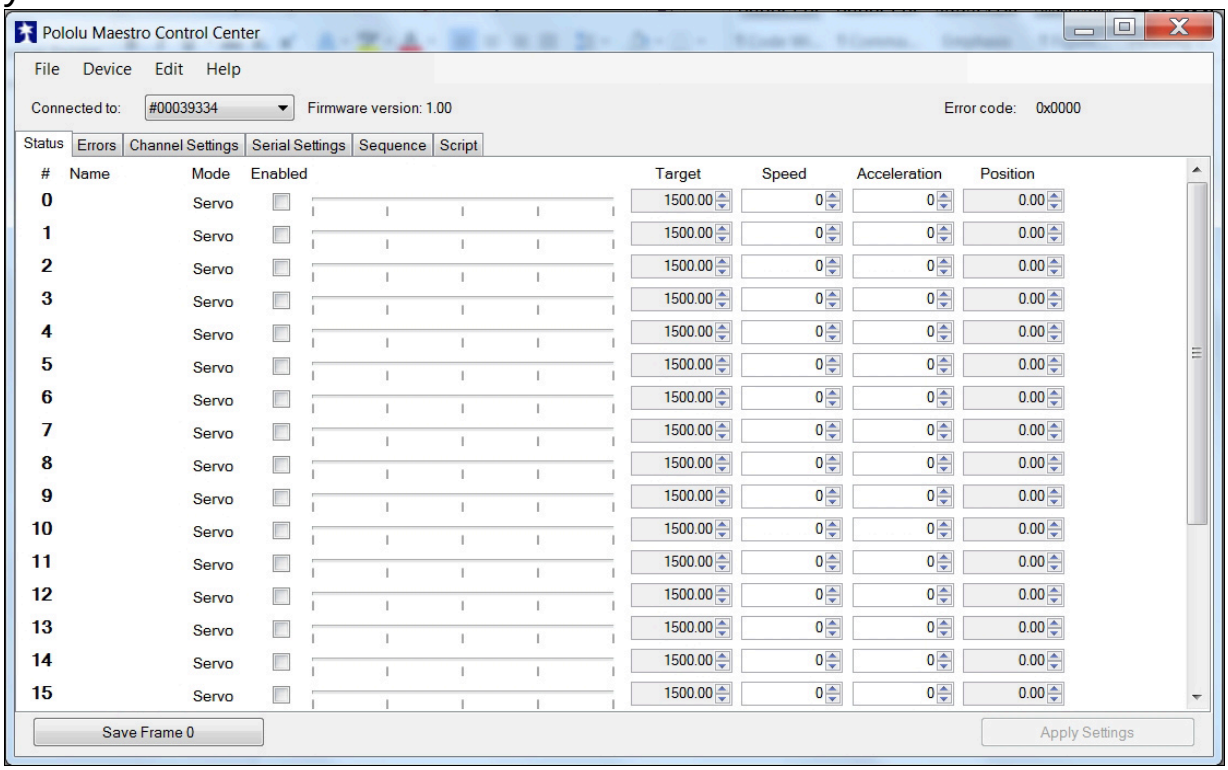


This battery will come with two connectors, one with larger gauge wires for normal usage and a smaller connector to connect to the battery recharger. You can use the XT60 Connector Pairs, solder some wires to the mating connector of the battery, and then insert the bare end of the wires into the servo controller.

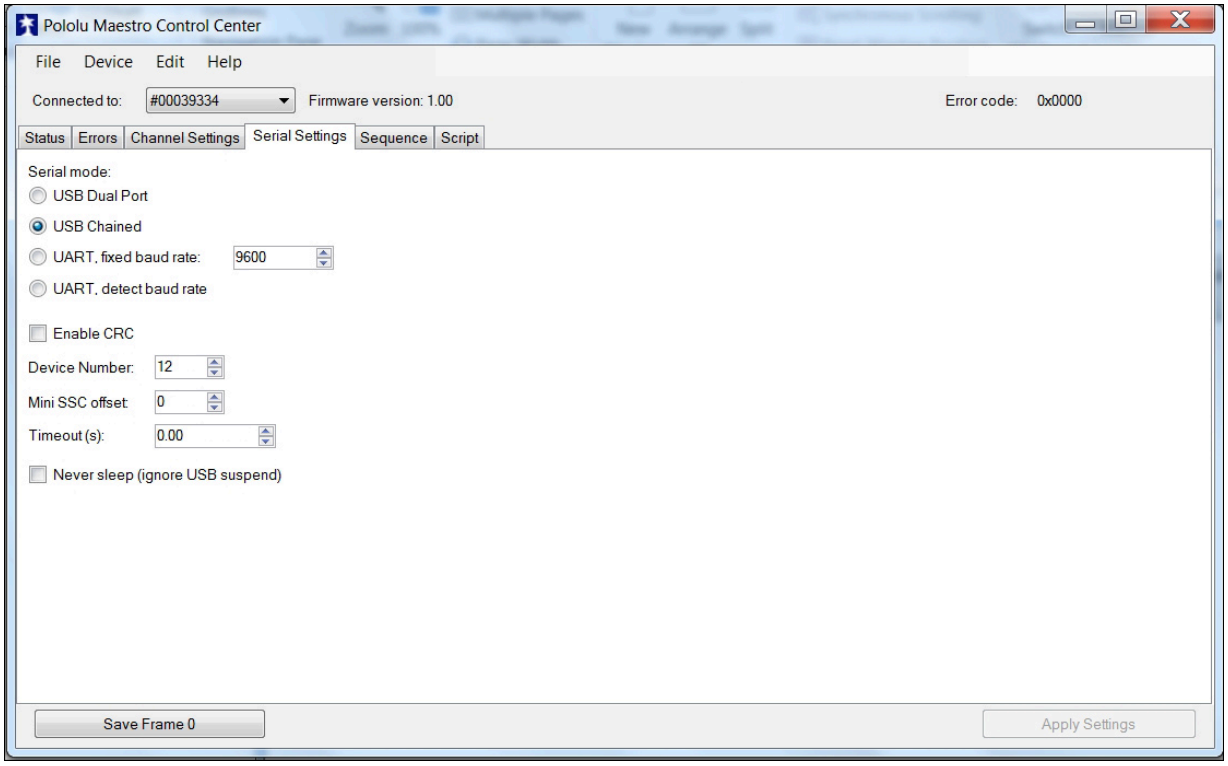
Your system is now functional. You can connect the motor controller to your personal computer to check whether you can communicate with it. To do this, connect a mini USB cable between the servo controller and your personal computer.

Communicating with the servo controller with a PC

Now that the hardware is connected, you can use some software provided by Polulu to control the servos. Let's do this using your personal computer. First, download the Polulu SW from www.pololu.com/docs/0J40/3.a, and install it based on the instructions on the website. Once it is installed, run the software, and you should see this screen:

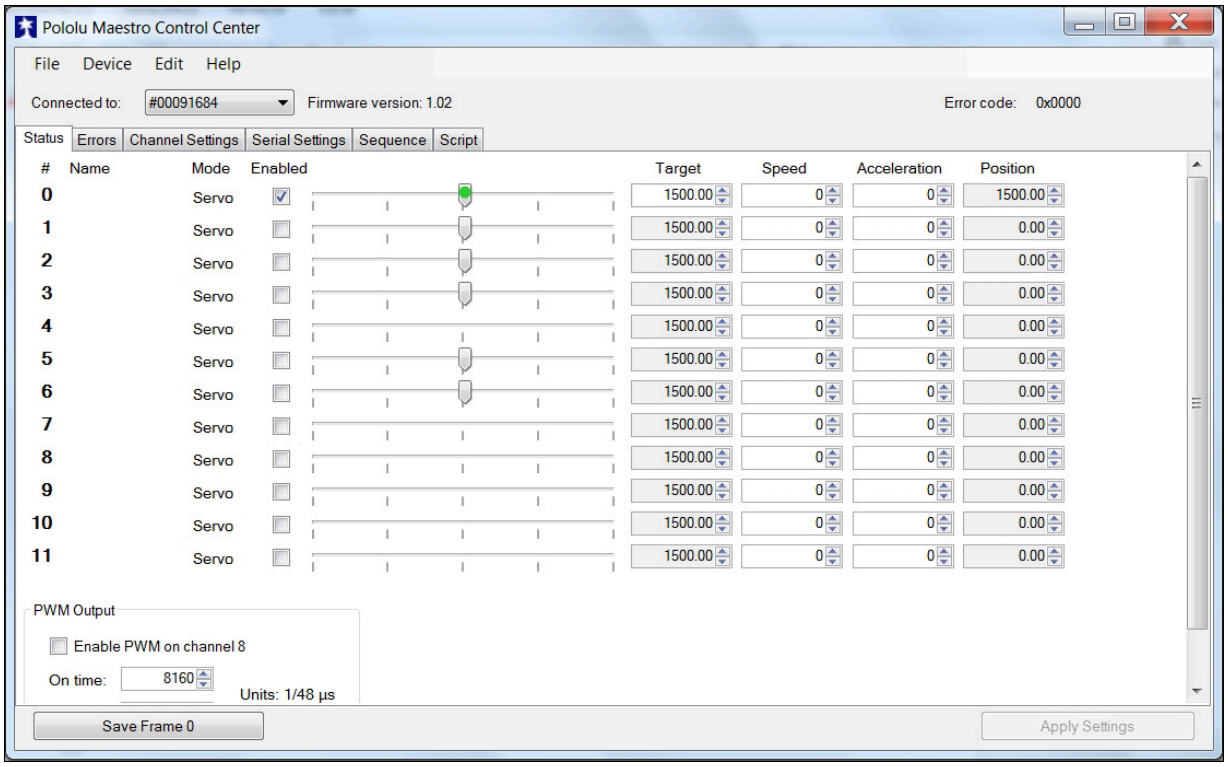


First, you will need to change the configuration in Serial Settings, so select the **Serial Settings** tabs, and you should see this:

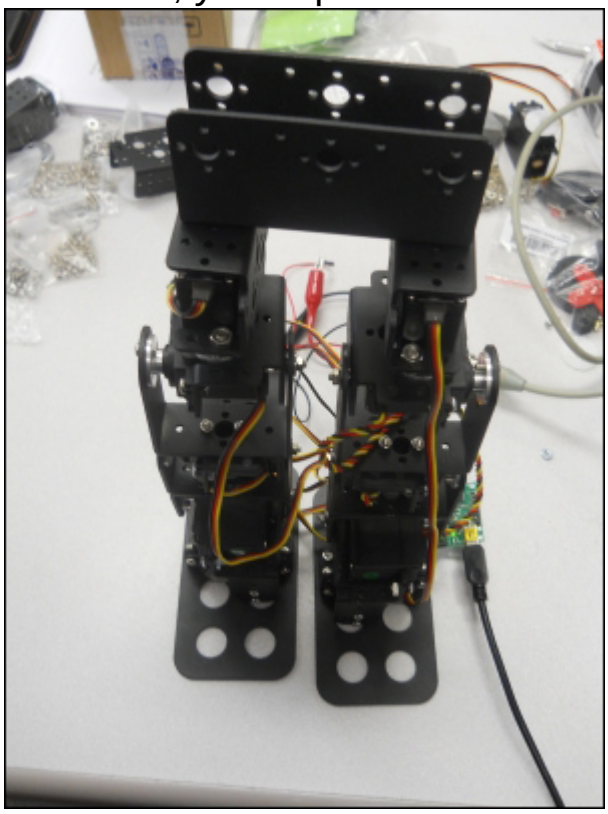


Make sure that USB Chained is selected; this will allow you to connect and control the motor controller over USB. Now, go back to the main screen by selecting the **Status** tab; now, you can actually

turn on the 10 servos. The screen should look like this:



Now you can use the sliders to actually control the servos. Turn on servo 0. Make sure that servo 0 moves the lower-right ankle servo. You can also use this to center the servos. Set servo 1 so that the slider is in the middle. Now, unscrew the servo horn on the servo until the servos are centered at this location. At the zero location of all servos, your biped should look like this:

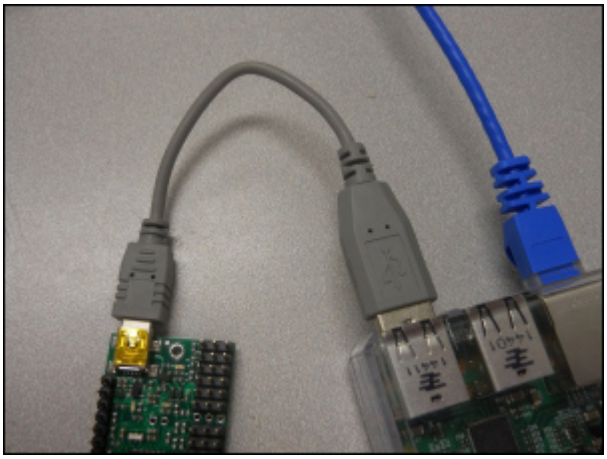


Connecting the servo controller to the Raspberry Pi

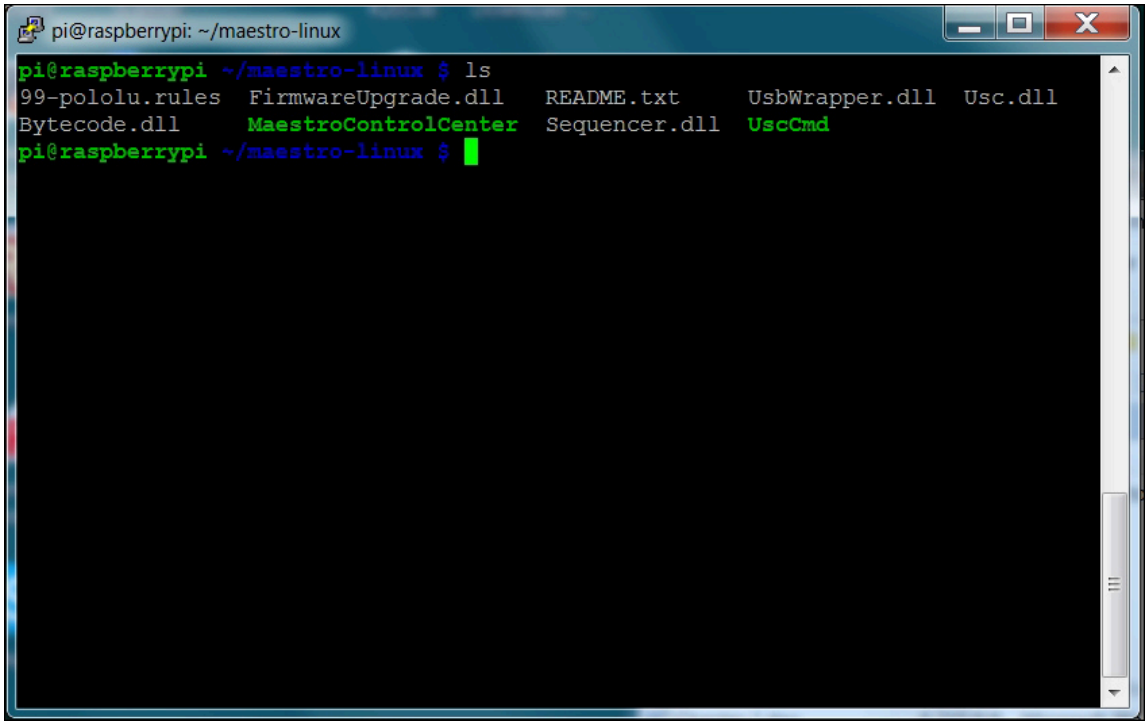
You've checked the servo motor controller and the servos. You can now connect the motor controller up to the Raspberry Pi and make sure that you can control the servos from it.

Now, let's talk to the motor controller. Here are the steps:

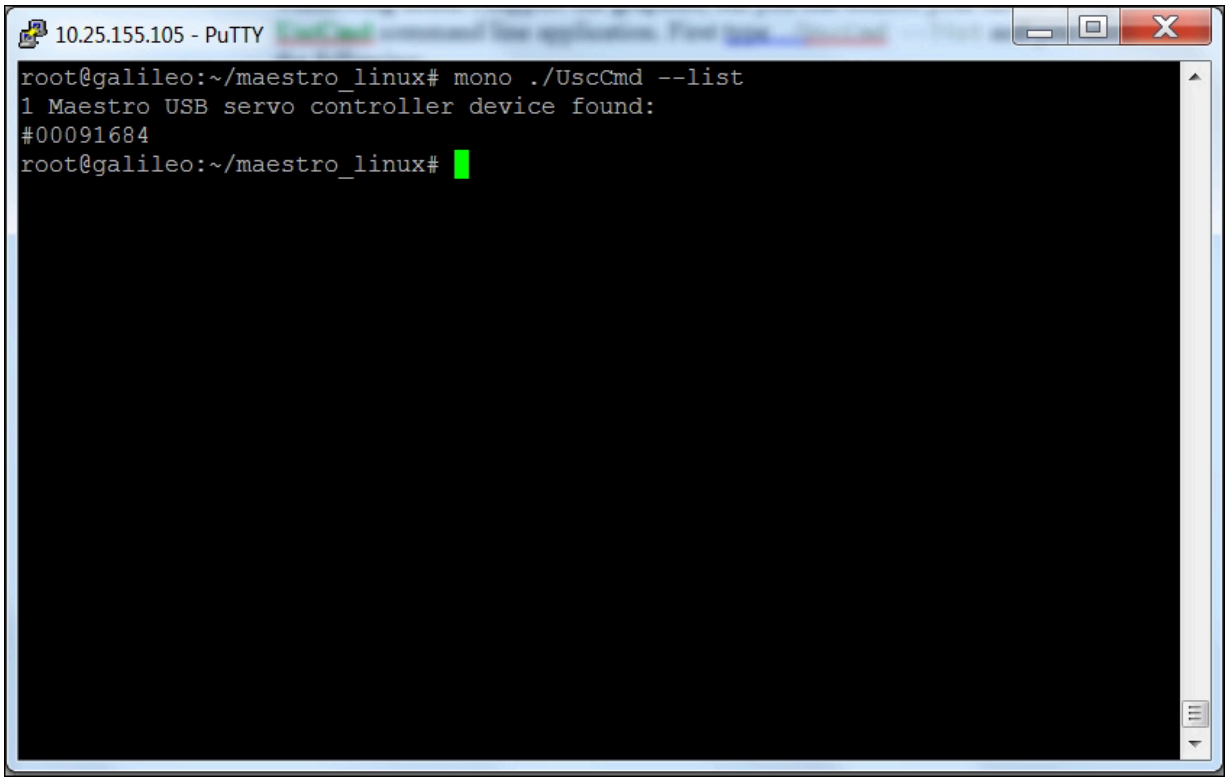
1. Connect Raspberry Pi to the motor controller by connecting a mini USB to a mini USB cable. Connect the cable to the USB host connection on the Raspberry Pi, like this:



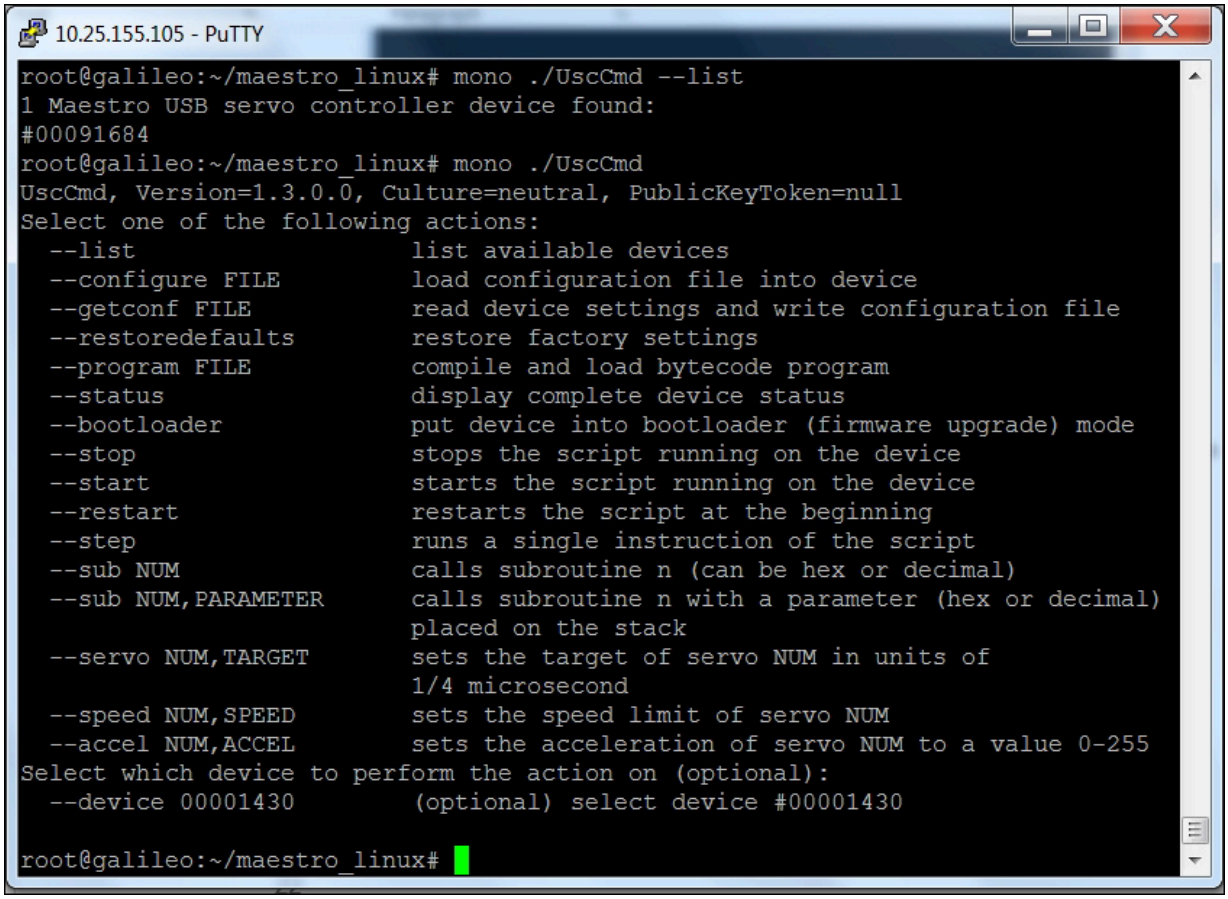
2. Download the Linux code from Pololu at https://www.pololu.com/docs/0J40/3.b. Perhaps the best way to do this is to log on to your Raspberry Pi and then type `wget http://www.pololu.com/file/download/maestro-linux-100507.tar.gz?file_id=0J315`.
3. Then, move the file using `mv maestro-linux-100507.tar.gz\?file_id\=0J315 maestro-linux-100507.tar.gz`.
4. Unpack the file by typing `tar –xvf maestro_linux_011507.tar.gz`. This will create a directory called `maestro_linux`. Go to this directory by typing `cd maestro_linux`, and then type `ls`; you should see something like this:



The `README.txt` document will give you explicit instructions on how to install the software. Unfortunately, you can't run **MaestroControlCenter** on your Raspberry Pi; the version of Windows it uses doesn't support the graphics, but you can control your servos using the **UscCmd** command-line application. First, type `./UscCmd --list`, and you should see the following:



The unit sees your servo controller. If you just type `./UscCmd`, you can see all the commands you could send to your controller:



Note that you can send a specific target angle to a servo, although the target is not in angle values, so it makes it a bit difficult to know where you are sending your servo. With a servo and battery connected to the servo controller, try to type `./UscCmd --servo 0, 10`. The servo will move to its full angle position. Type `./UscCmd – servo 0, 0`, and it will stop the servo from trying to move. In the next section, you'll write some Python code that will translate your angles to the commands that the servo controller will want to see in order to move it to specific angle locations. If you are struggling with the USB connection, refer to http://www.linux-usb.org/FAQ.html for more information.

Note

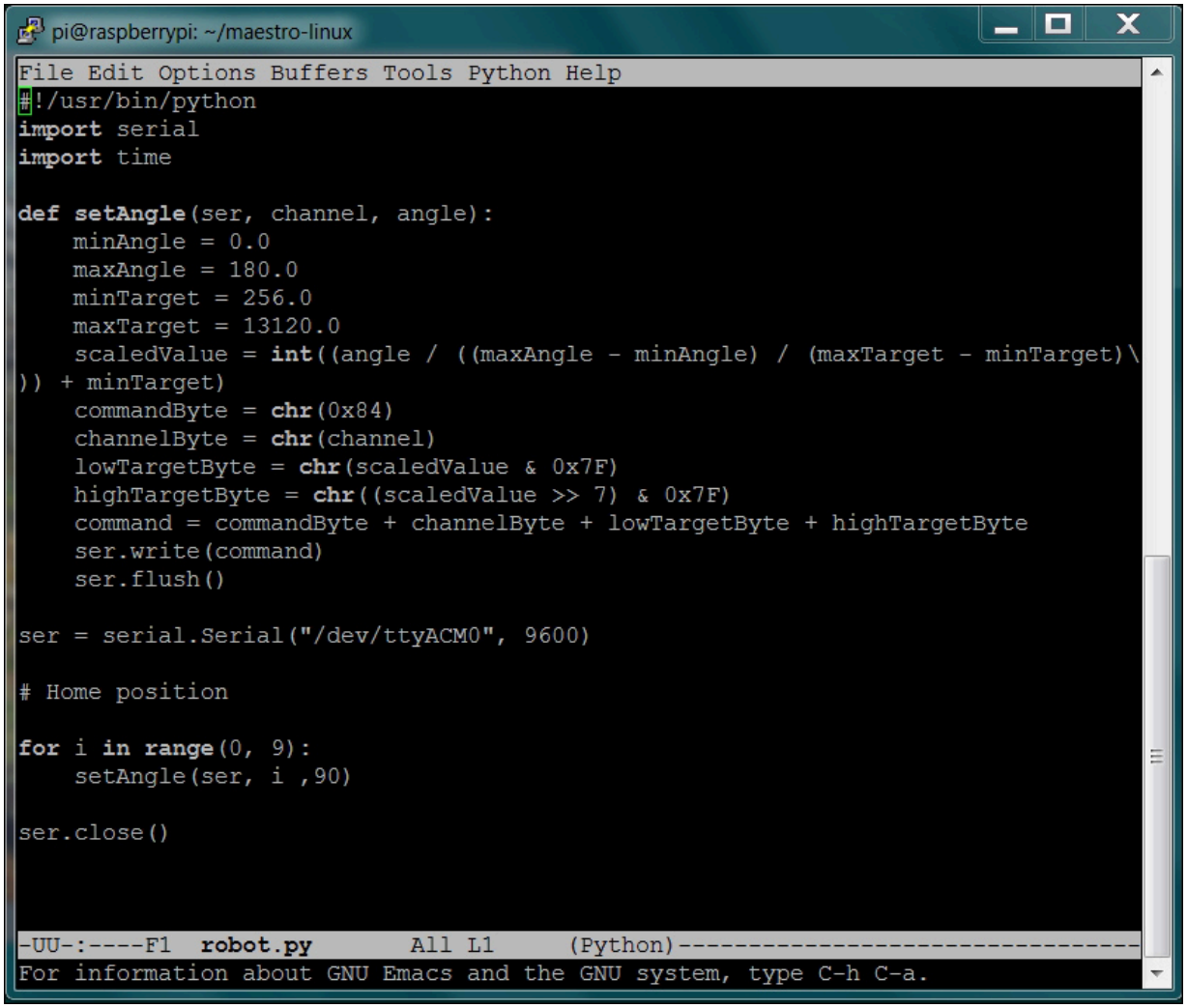
If you didn't run the Windows version of Maestro Controller and set **Serial Settings** to **USB Chained**, your motor controller may

not respond. Rerun the MaestroController code and set **Serial Settings** to **USB Chained**.

Creating a program to control your biped

Now you know that you can talk to your servo motor controller and move your servos. In this section, you'll create a Python program that will let you talk to your servos to move them at specific angles. You can use Python as it is very simple and easy to run.

Let's start with a simple program that will make your legged mobile robot's servos go to 90 degrees (which should be somewhere close to the middle between the 0 to 180 degrees you can set). Here is the code:



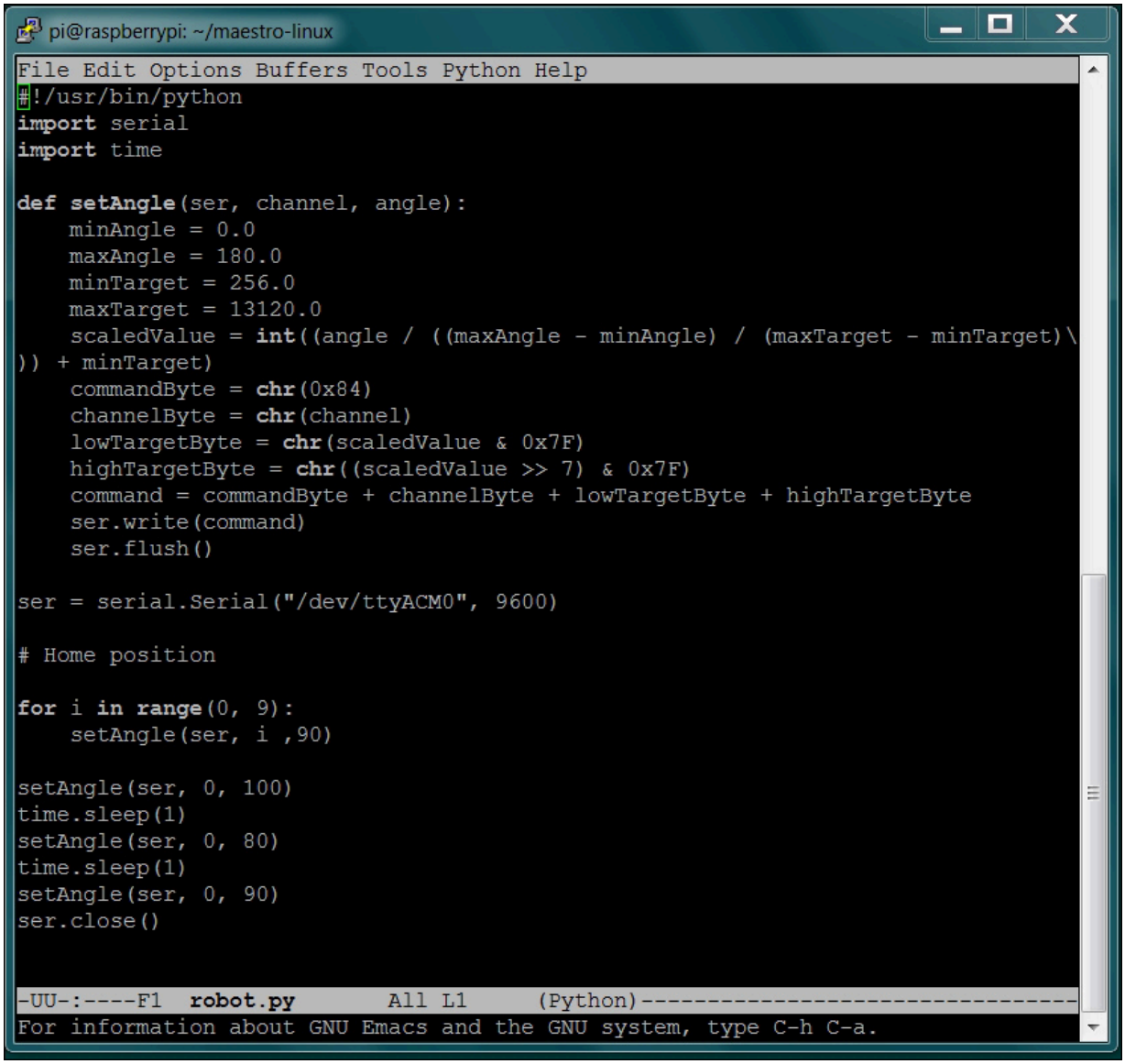
Here is an explanation of the code:

- `#!/usr/bin/python`: This first line allows you to make this Python file execute from the command line.
- `import serial`: This line imports the serial library. You need the serial library to talk to your unit via the USB.
- `def setAngle(ser, channel, angle):`: This function converts your desired setting of servo and angle into the serial command that the servo motor controller needs. To understand the specifics of the code used to control the servos, refer to https://www.pololu.com/docs/0J40.
- `ser = serial.Serial("/dev/ttyACM0", 9600)`: This opens the serial port connection to your servo controller.
- `for i in range(0, 9)`: For loop to access all nine servos.

- `setAngle(ser, i, 90)`: Now, you can set each servo to the middle (home) position. The default would be to set each servo to 90 degrees. If your legs aren't in their middle position, you can adjust them by adjusting the position of the servo horns on each servo.

To access the serial port, you'll need to make sure that you have the Python serial library. If you don't, then type `apt-get install python-serial`. After you have installed the serial library, you can run your program by typing `python quad.py`.

Once you have the basic home position set, you can ask your robot to do something. Let's start by having your biped move its foot. Here is the Python code:



In this case, you are using your `setAngle` command to set your servos to manipulate your robot's right ankle.

Summary

You now have a robot than can move! In the next chapter, you'll learn how to make your robot do many amazing things. You'll learn how to make it walk forward and backward and how to make it dance and turn. With some basic knowledge, any number of movements is possible.

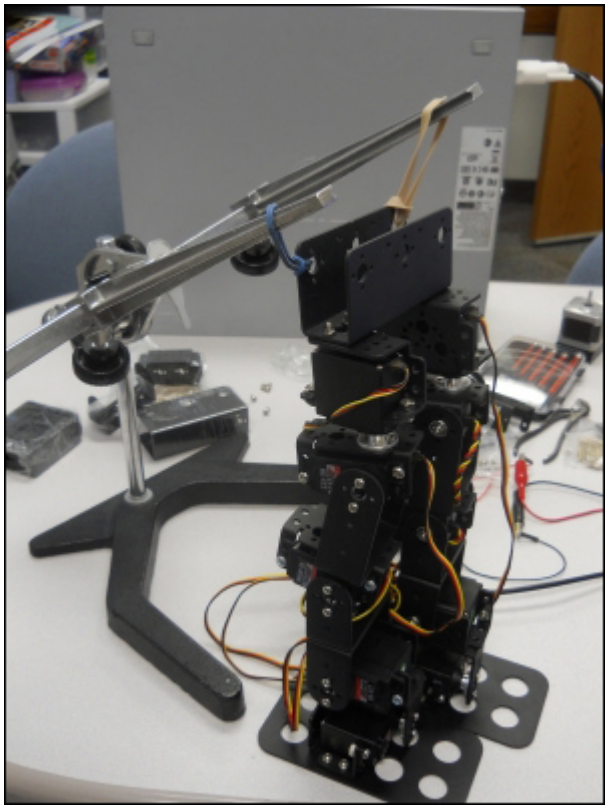
Chapter 3. Motion for the Biped

Now that you've got your biped all up and running, you can start developing interesting ways to make it move. In this chapter, you'll learn

- How to adjust the positions of your servos for the Tyrannosaurus Rex pose
- The basic walking gait for your robot
- The basic turn for your robot

Before you begin, however, it will be best if you create a harness for your biped. Your robot is going to be inherently unstable with only two legs, and, as you experiment, you're going to make some mistakes. With only two legs, these mistakes can, and probably will, result in your robot toppling over, which can damage the robot.

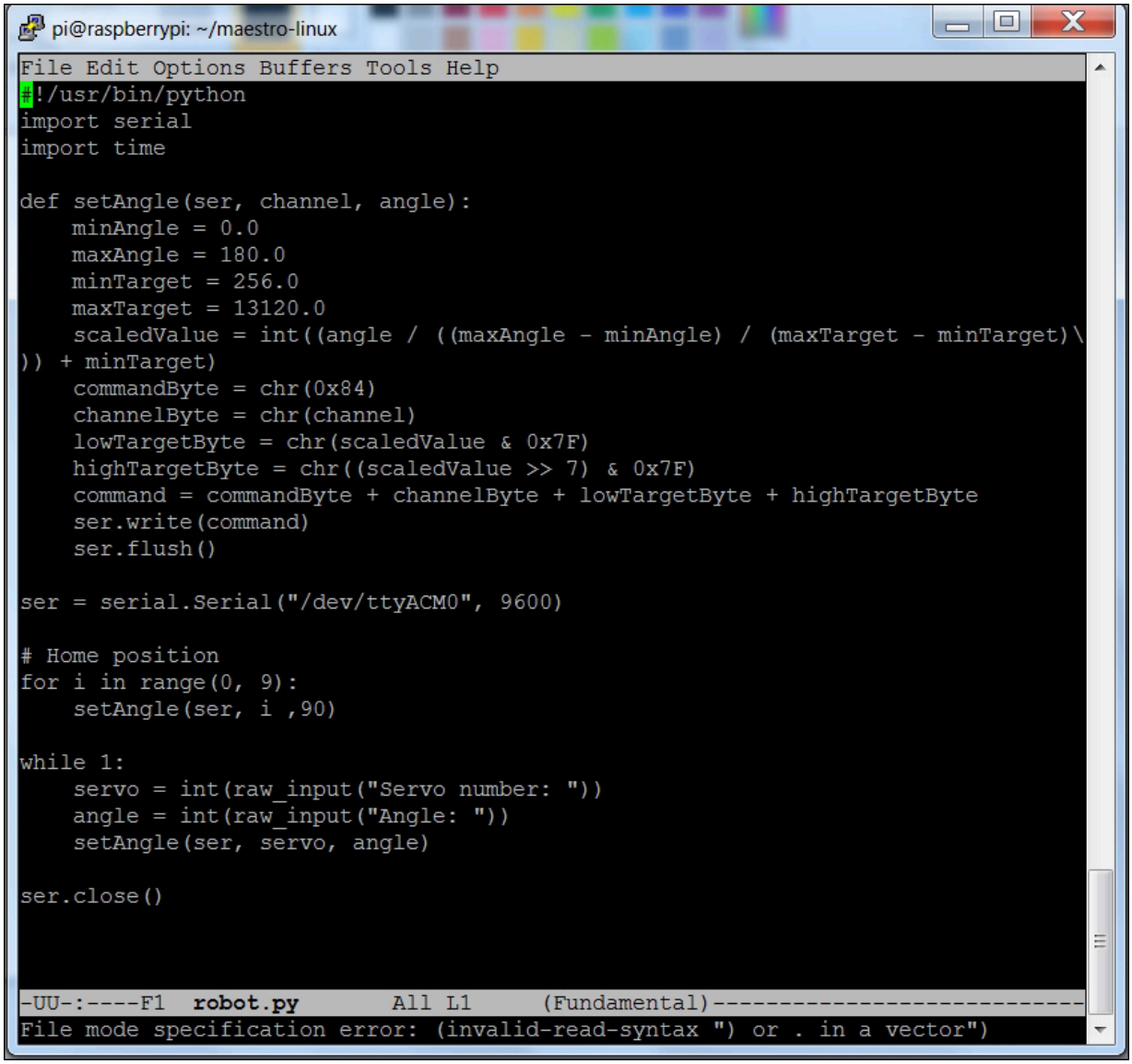
If you have an electronics board vise, or an "Extra Hands" device, they can be useful for this purpose. Here is a picture of how to use this device to create a harness:



If you don't, you can easily create this sort of overhead support using a PVC pipe or wood. Really, just something to keep your biped from crashing over during your experimentation.

A basic stable pose

Now that your biped is built and you know how to program the servos using Python, you can experiment with some basic poses. You'll first create a program that allows you to set individual servos so that you can experiment. Here is the program:



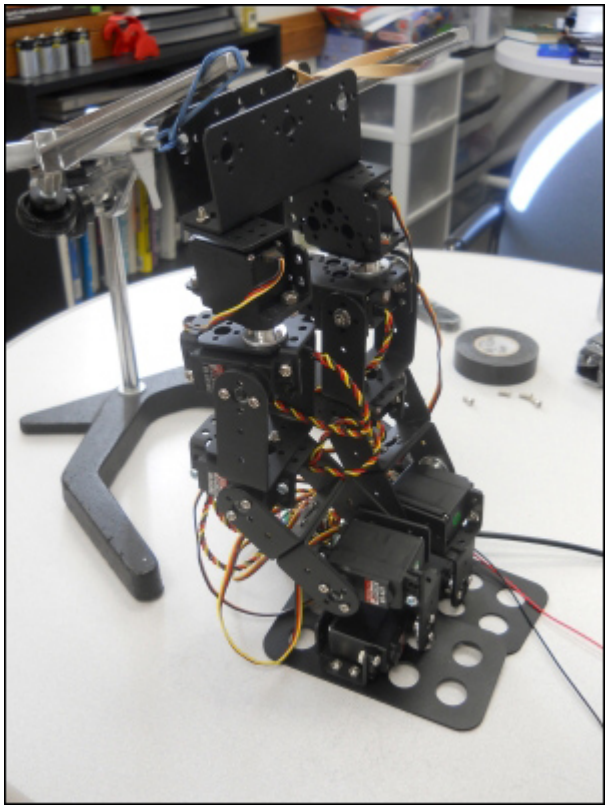
This code includes the Python `setAngle` function from Chapter 2, *Building the Biped*. The specifics were taken from the www.pololu.com website, but it simply allows you to set a specific servo to a specific angle.

The next part of the code sets all of the servos to their center location. The final piece of the code, the `while 1:` code set, simply asks the user for a servo and an angle, and then sends the command to the servo controller.

Once the program is run, you should see your biped standing straight up. If not, you may need to center your servos by adjusting the position of the horns. This is a useful pose, but there are others that are more stable. As an excellent first example, you can change the pose to be more like a Tyrannosaurus Rex pose, with knees pointing back. Here are the basic servo positions:

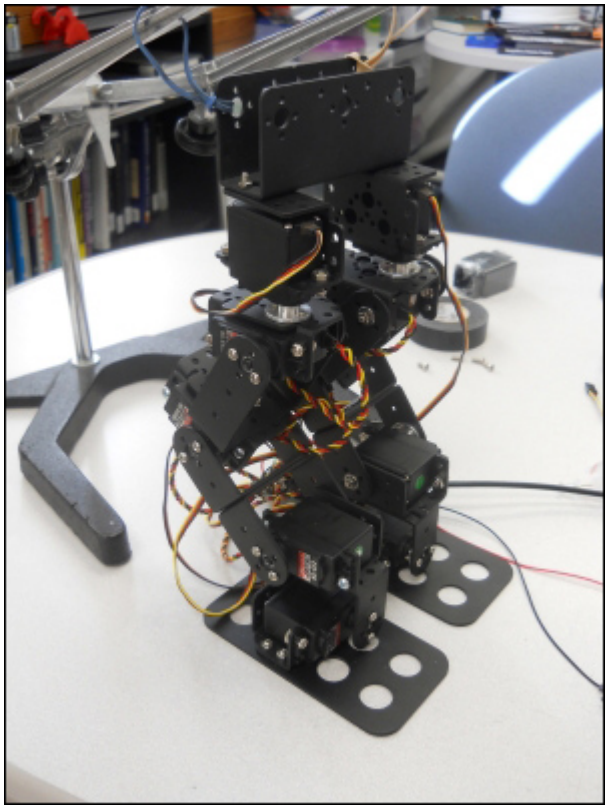
Servo	**Angle**
1	60
2	60
3	60
6	120
7	120
8	120

The robot pose should look like this:



You can use these angles to achieve this pose. However, this will leave you with some limited movement, as your servos will be toward the end of their ability to move in one direction. As this is going to be the starting pose for your robot, to achieve maximum flexibility, you'll want to center the servos at this position. To do this, run the default, `robot.py`, to set the legs to the center position. Now, adjust the servo horns to achieve this pose while the values of the servos are at a 90 degree angle.

It should look like this:

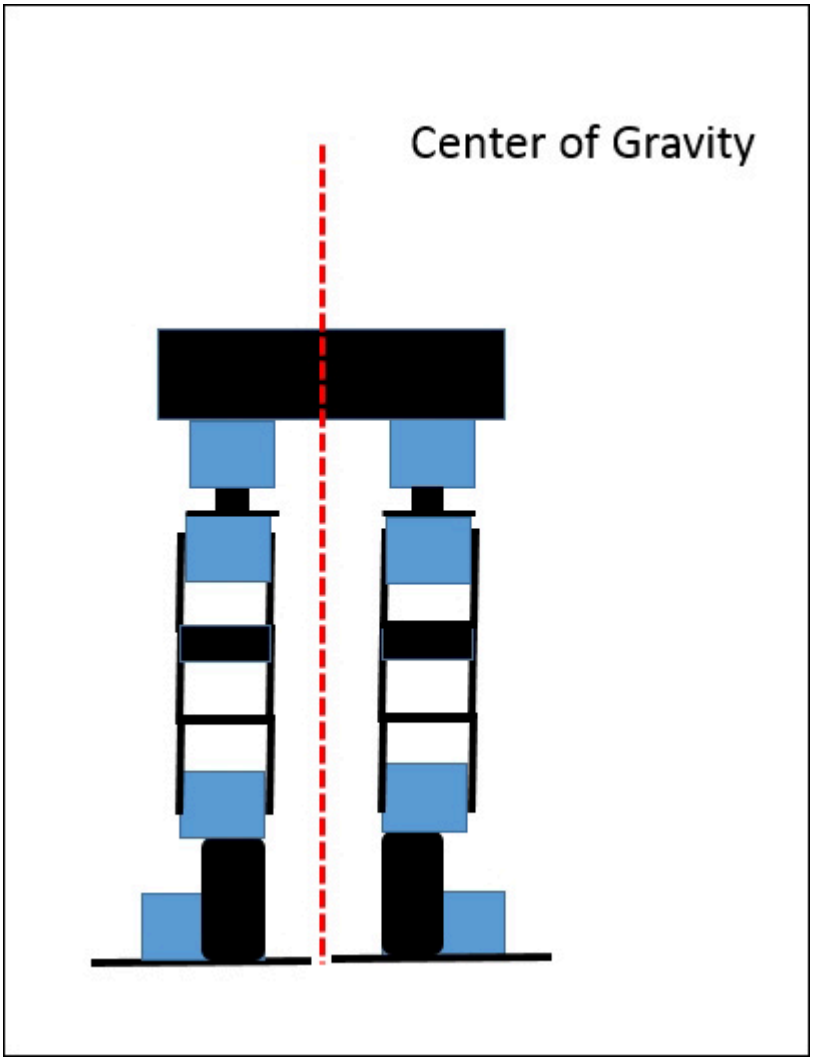


Now that you have a stable base to work from, you can start programming a simple walking motion.

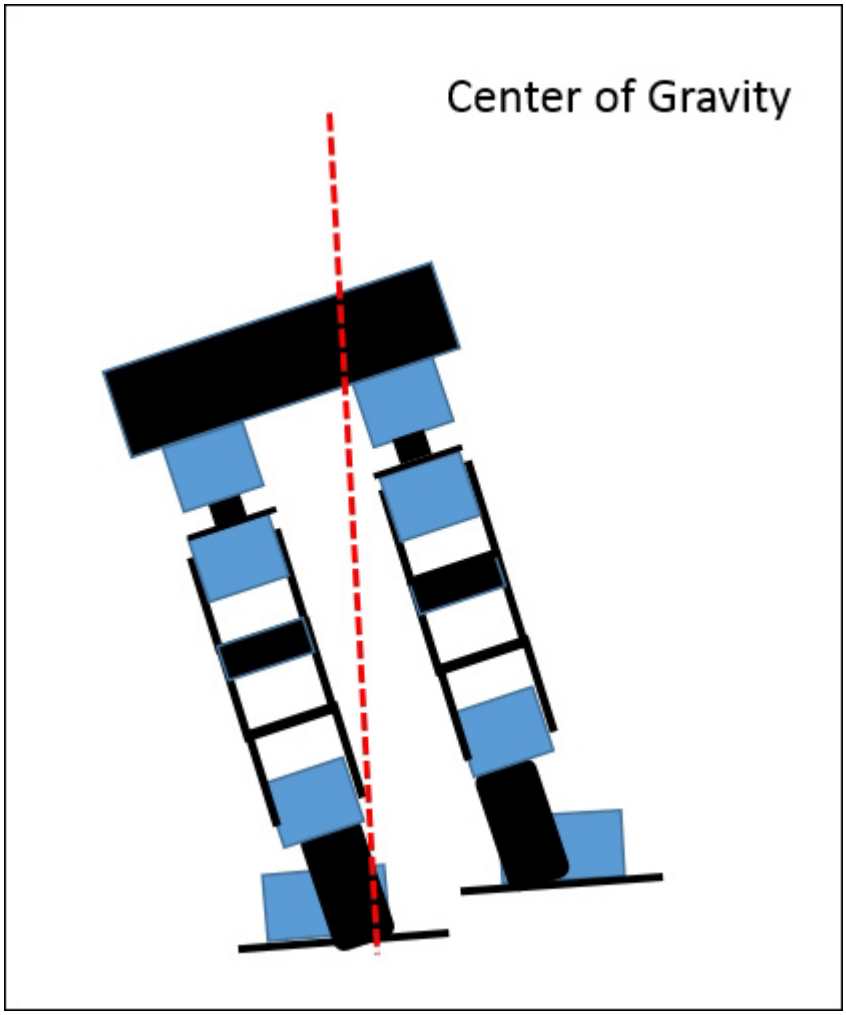
A basic walking motion

Your robot is poised to walk, however, you first have to get a leg off the ground. Of course, that is easy enough; if you simply lift the leg by changing the angle of the knee joint, your leg can get off the ground. You may also want to change the angle of the front to back ankle; this will allow you to lift the leg without raising it quite as high.

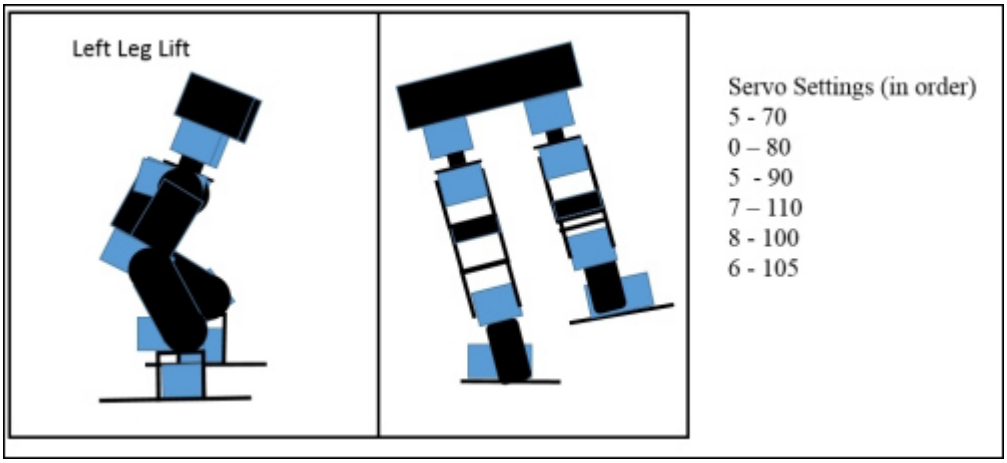
However, you'll have a problem if you change just these two servos; as you lift the leg, your robot will fall over. This is due to a simple principle called the center of gravity. When your robot is at rest, your center of gravity looks like this:



It is clear that if you lift a leg, the robot will fall over in the direction of the leg that has been lifted. What you need to do is to shift the center of gravity over the leg that will be left on the ground using the ankle servo that can tilt the robot left and right, so that it ends up like this:



You'll then want to set your servos to lift the left leg. Here is a side view of these servo settings:



Now, it's time for some Python code to make this happen. You'll start with your `robot.py` code and will add the following lines to a function called `liftLeftLeg`:

pi@raspberrypi: ~/maestro-linux

File Edit Options Buffers Tools Python Help

```
# Home position
def setHome(ser):
    for i in range(0, 9):
        setAngle(ser, i ,90)

def liftLeftLeg(ser):
    setAngle(ser, 5, 70)
    setAngle(ser, 0, 80)
    time.sleep(.5)
    setAngle(ser, 5, 90)
    setAngle(ser, 7, 110)
    setAngle(ser, 8, 80)
    setAngle(ser, 6, 105)

ser = serial.Serial("/dev/ttyACM0", 9600)

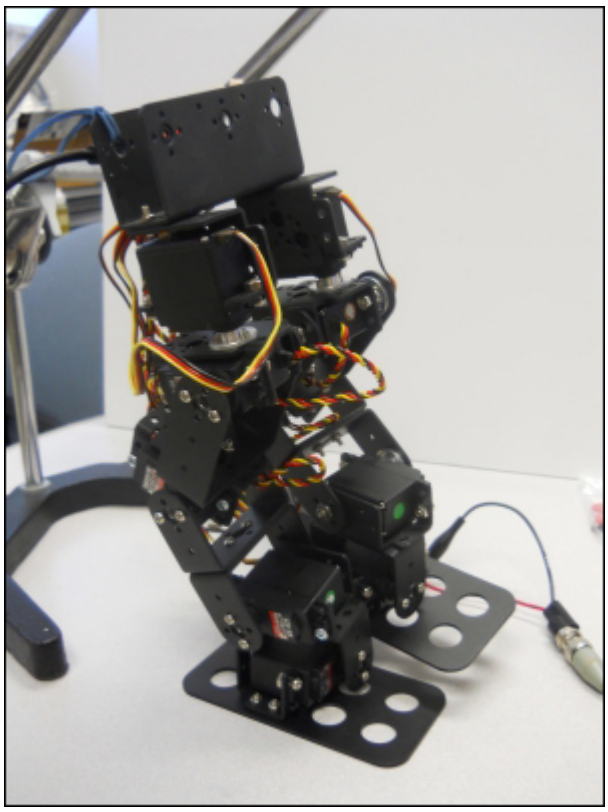
setHome(ser)
time.sleep(2)
liftLeftLeg(ser)
time.sleep(.5)

while 1:
    servo = int(raw_input("Servo number: "))
    angle = int(raw_input("Angle: "))
    setAngle(ser, servo, angle)

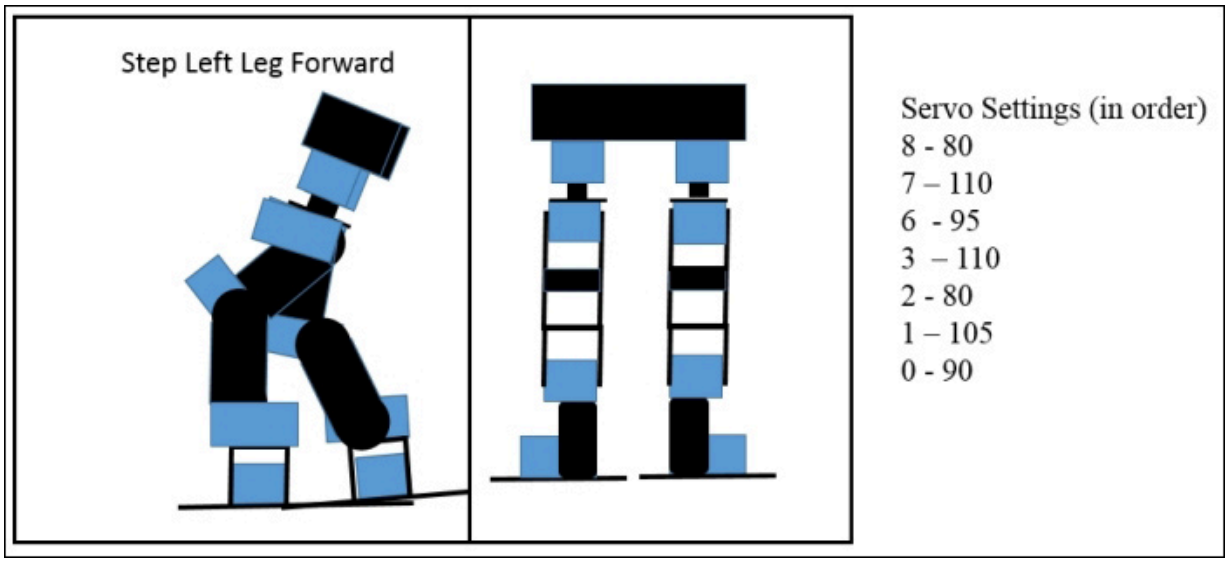
ser.close()
```

-UU-:**--F1 **robot.py** Bot L33 (Python)----------

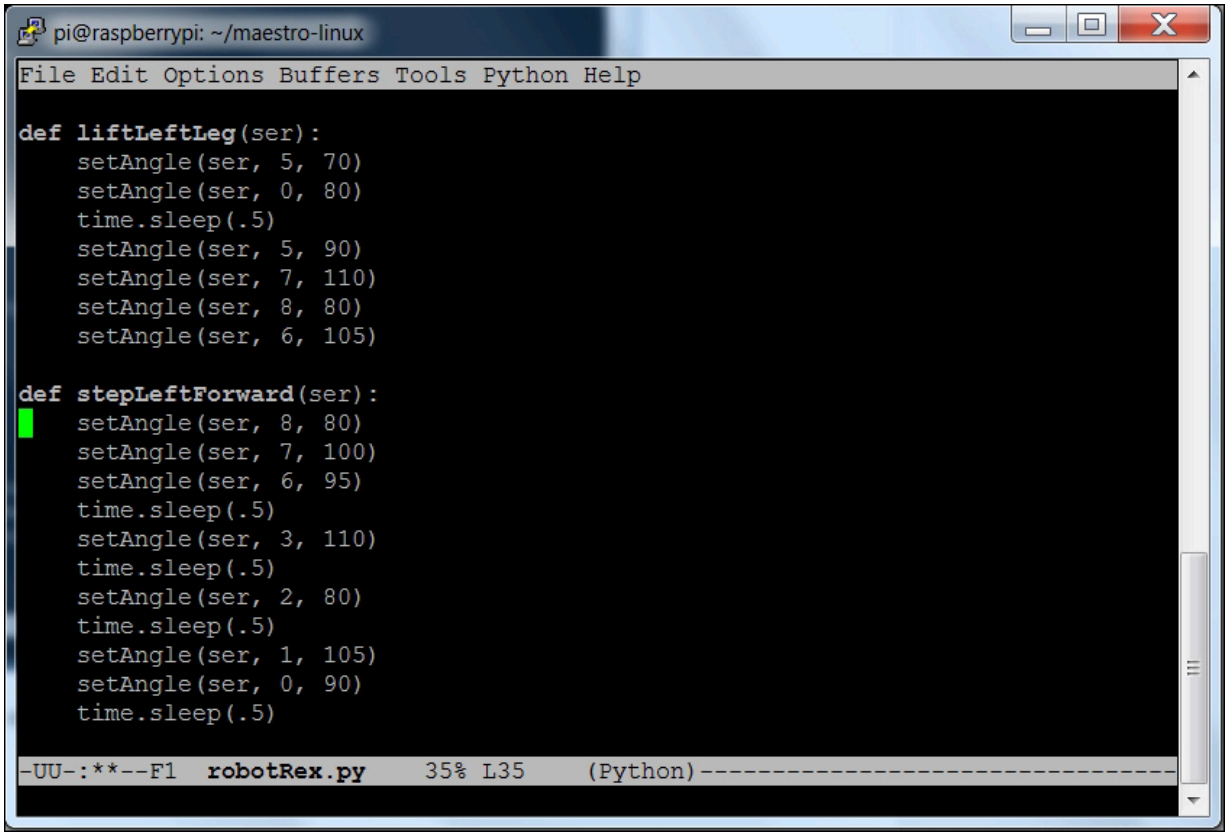
This will tip the robot onto its right leg, and then lift the left leg, like this:



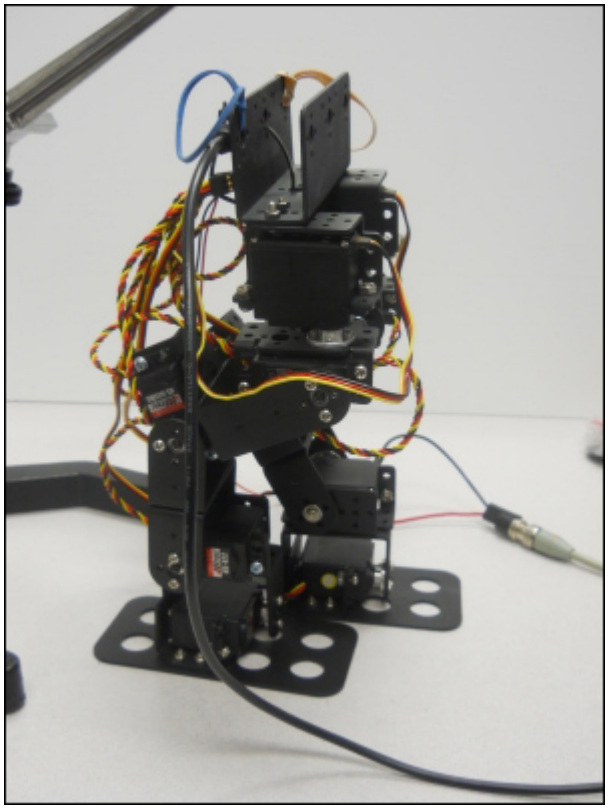
Now, it is fairly easy to step forward. Just move the hip joint on the left leg forward, and then move the ankle joint on the right leg to tip the entire robot forward. Here is the diagram and servo settings:



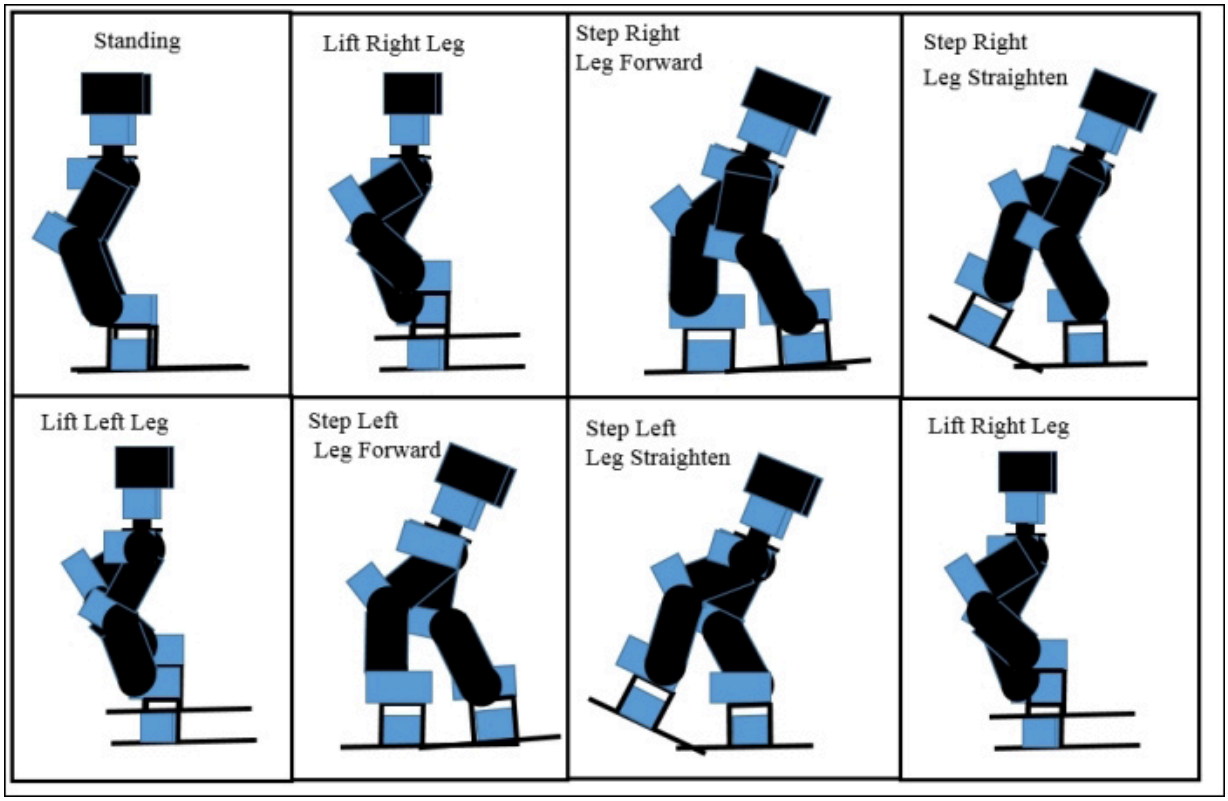
Here is the Python code:



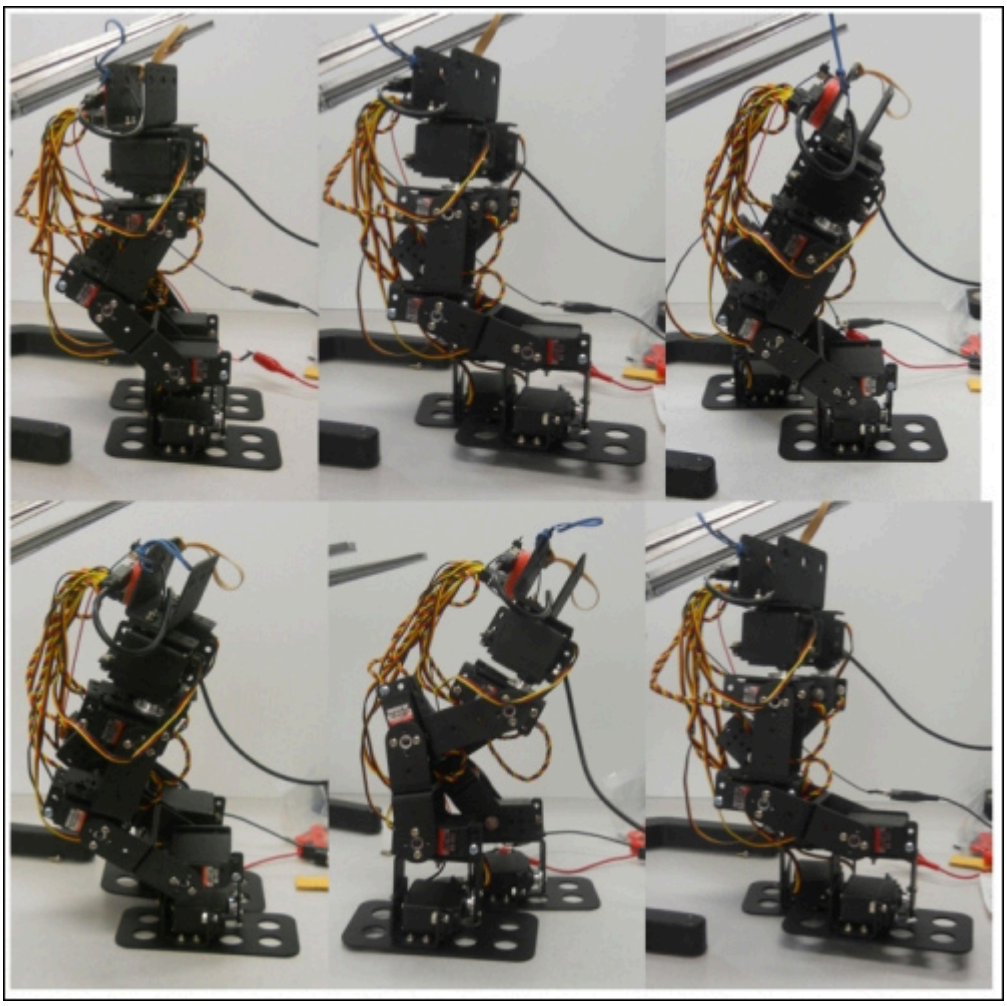
Here is a picture of the robot:



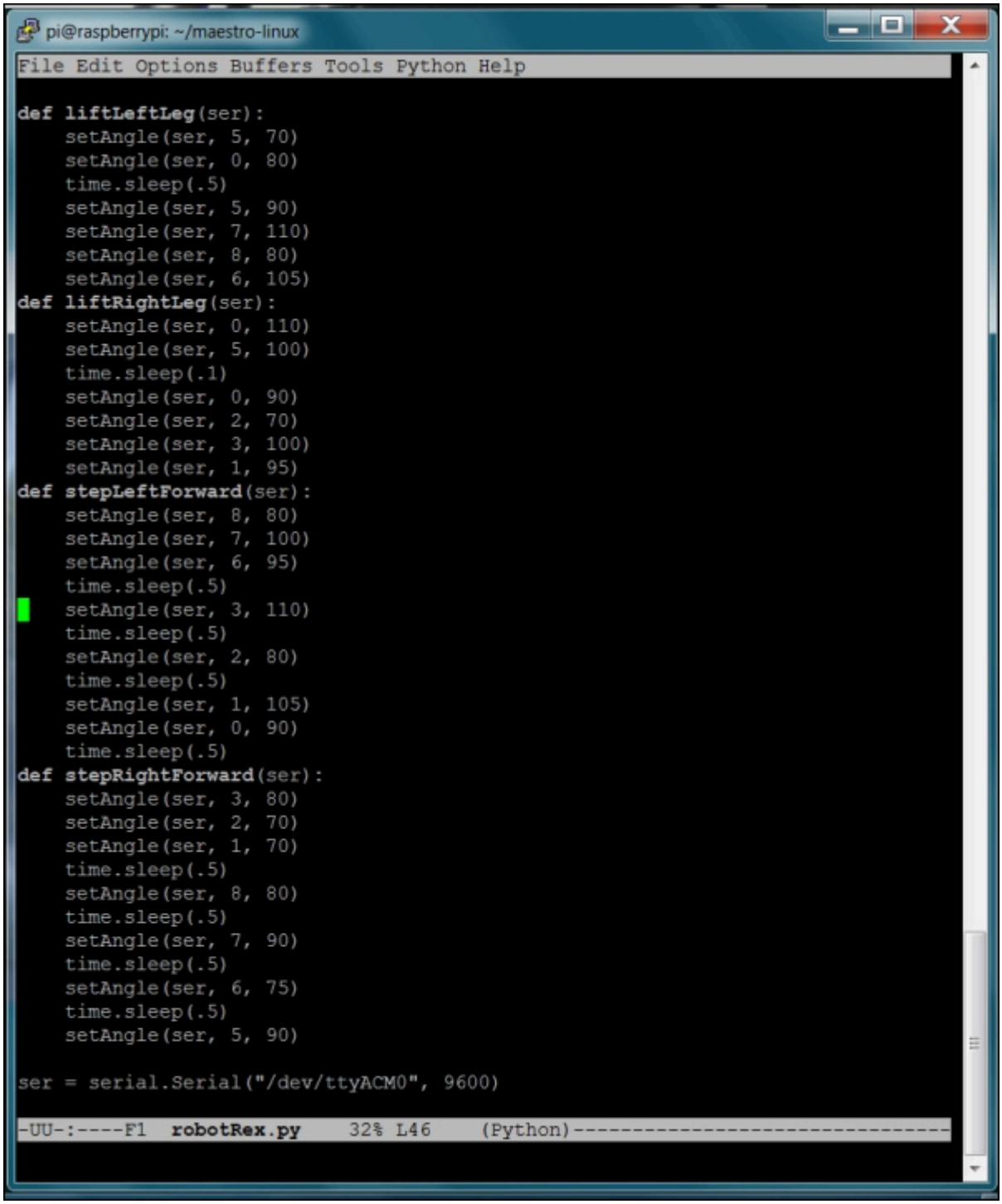
This is the first stage of a walking gait. So, let's detail all the motions you'll need in order to walk your robot forward. Here are the side view diagrams of the different states:



These are the pictures of the robot in each of the different states:

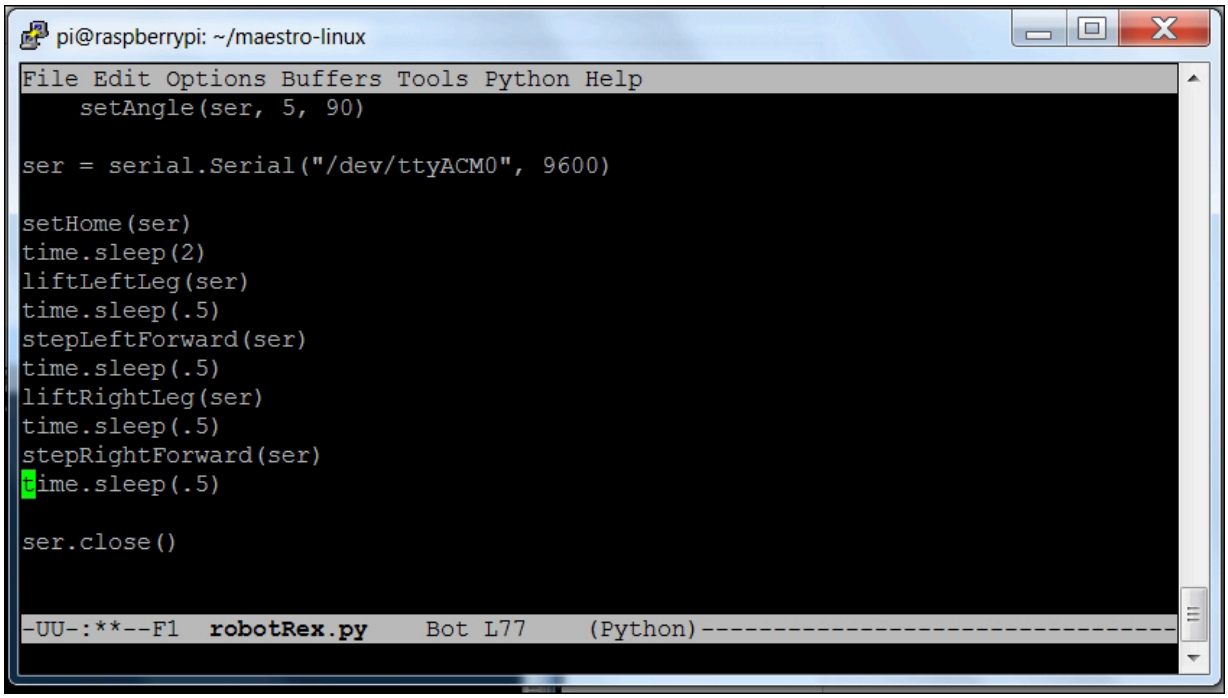


Here is the Python code for each of the functions for the different states:



You'll notice that each function has a number of different servo control statements; these must be performed in this order to get the desired result.

Here is the Python code to sequence the functions for two steps: one with the left leg, and the other with the right leg:

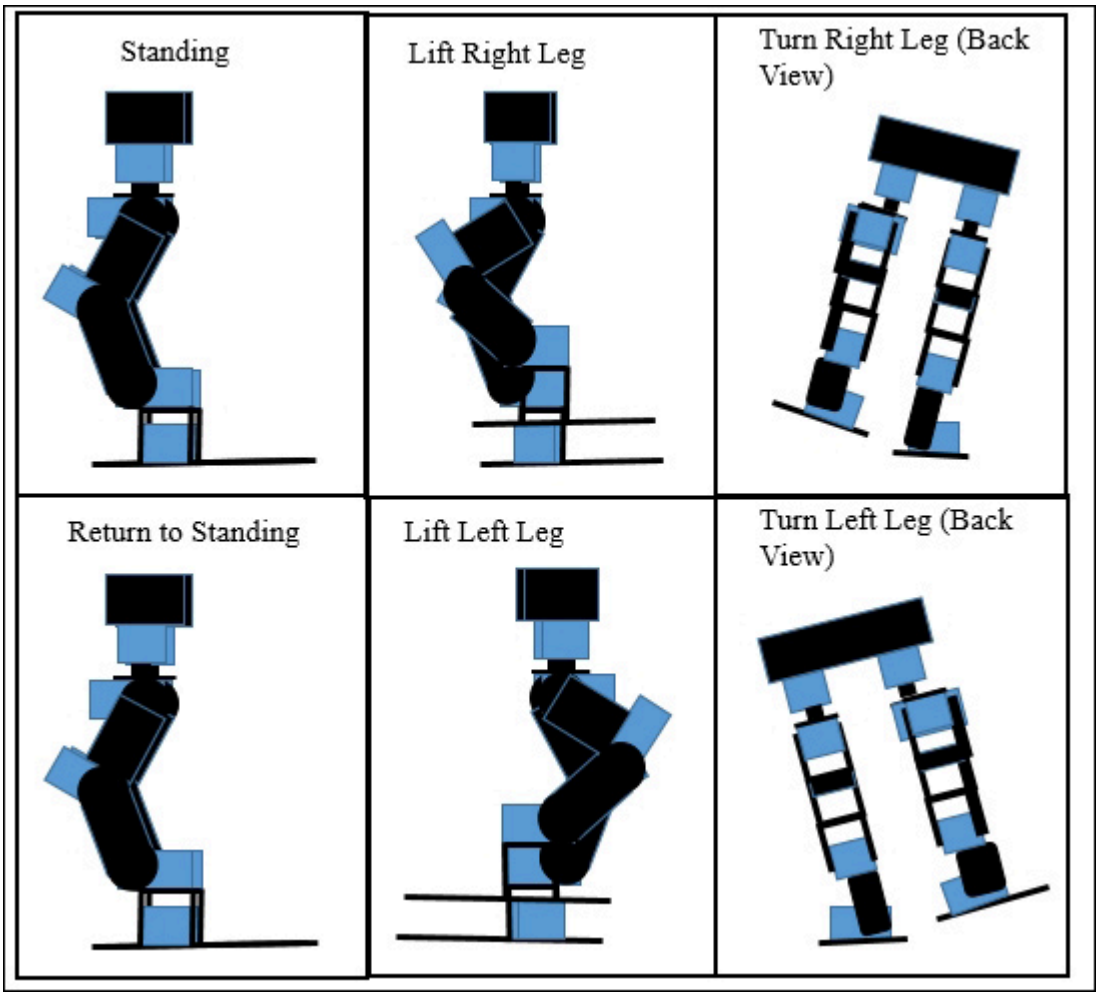


This is a very simple gait; it's not particularly elegant. You can see that each state is made up of many individual servo moves. You can certainly add more servo moves to make it smoother and more refined. Your exact servo angle settings will certainly vary from these; you'll need to do some experimentation to get your biped's legs positioned correctly.

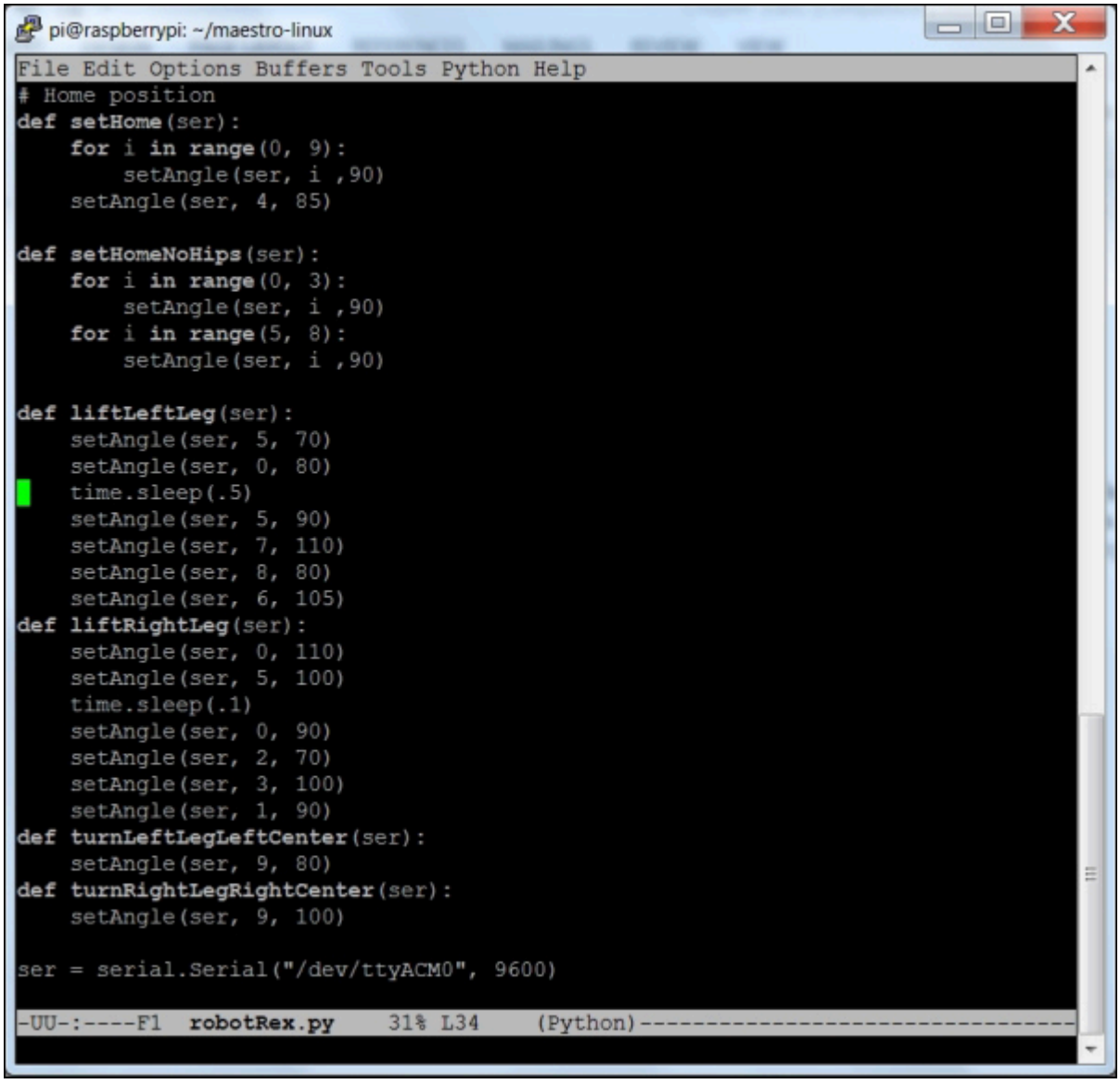
Now that you can walk, you will also need to teach your robot how to turn.

A basic turn for the robot

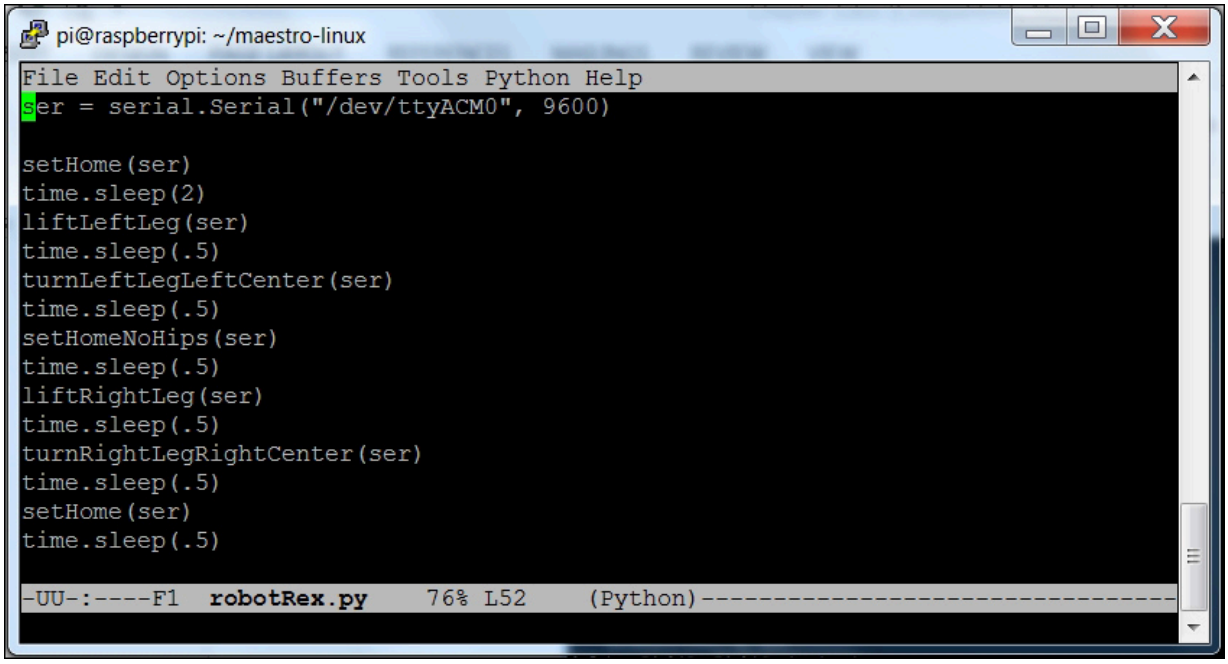
Your robot can walk forward, but you'll also want your robot to be able to turn. Your turning is limited to the amount you can turn the hip of your robot, which is around 20 degrees for this robot. So, to perform a full 90 degree turn, you'll need to take the turn in several steps. The big difference here is that when you return to the standing state, you do not want to reset your hip rotation servos to 90 degrees. Here are the diagrams, including several that are rear view, for a turn:



Here is the Python code for a basic turning operation:



And here is the Python code to chain these basic states together to step a turn:



Now your robot can walk and turn! Obviously, your robot could walk backward by reversing the order of servo control statements in each of the functions. There are many more types of motions you can program with your robot, following the planning method outlined in this chapter.

Summary

Now, your robot is mobile. The next step is to add some sensors so that your robot can avoid, or find, objects in its path.

Chapter 4. Avoiding Obstacles Using Sensors

You've constructed your biped robot. Now, your robot can move around. But what if you want the robot to sense the outside world, so you don't run into things? In this chapter, you'll discover how to add some sensors to help avoid barriers.

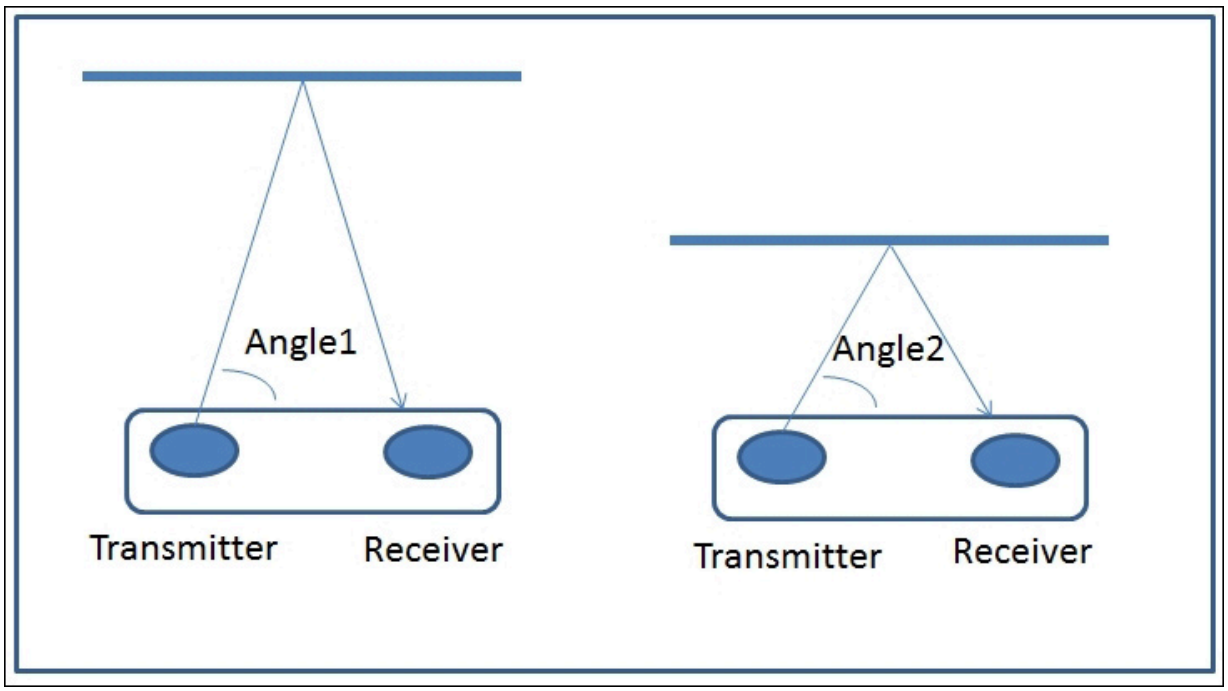
Specifically, you'll learn:

- How to connect Raspberry Pi to an **IR** (**infrared**) sensor
- How to connect Raspberry Pi to a USB **sonar sensor** to detect the world
- How to connect Raspberry Pi and its GPIO to a sonar sensor to detect the world

Connecting Raspberry Pi to an infrared sensor

Your robot can now move around, but you'll want to be able to sense a barrier or a target. One of the ways to do this is with an IR sensor. First, a tutorial on IR sensors is required. An IR sensor has both a transmitter and a sensor. The transmitter sends out a narrow beam of light, and the sensor receives this beam of light.

The difference in transit ends up as an angle measurement in the sensor, as shown in the following figure:



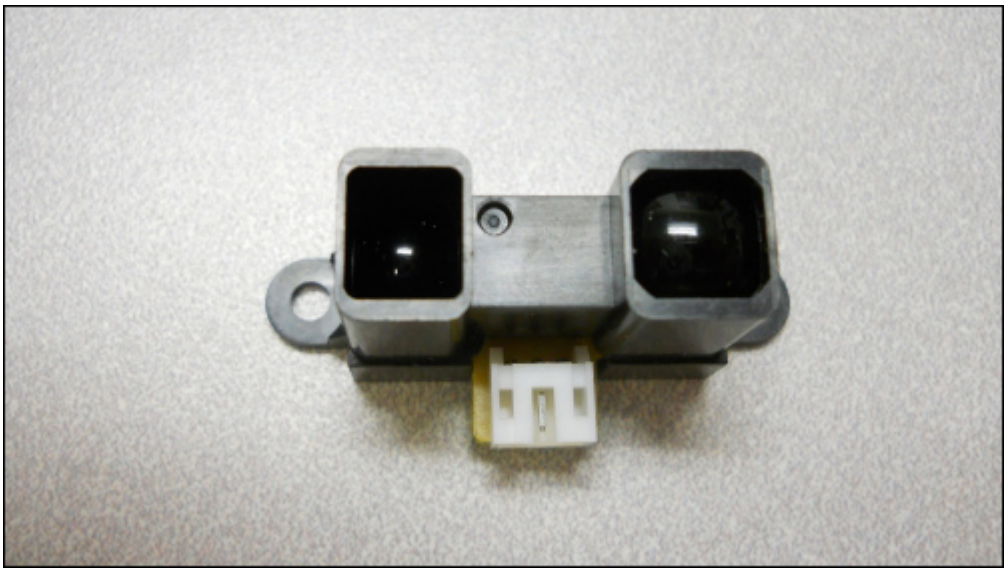
The different angles give you an indication of the distance from the object. The sensor turns these angle measurements into a voltage that you can sense to determine the distance. Unfortunately, the relationship between the output of the sensor and the distance is not linear, so you'll need to do some calibration in order to predict the actual distance and its relationship to the output of the sensor.

Note

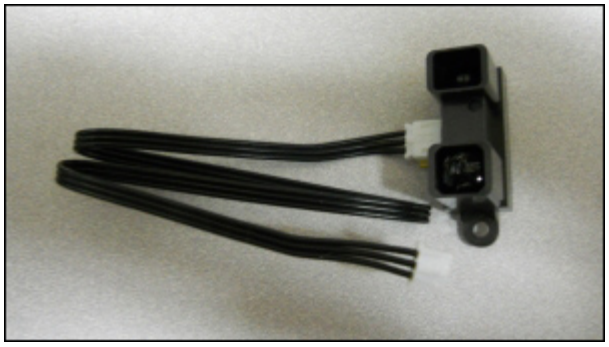
IR sensors are quite accurate, certainly with a low percentage of errors; however, they may not work well if the area is brightly lit. The accuracy is also affected by the reflective nature of the material being sensed. This can be a consideration when deciding which sensors to use.

Before you get started, you'll need to get a sensor. One of the more popular ones is an inexpensive IR sensor by Sharp. It is available at many online electronics stores, and it comes in models that sense various distances. You'll be using the **Sharp 2Y0A02** model, a unit

that provides sensing to a distance of 150 cm. Here is a picture of the sensor:



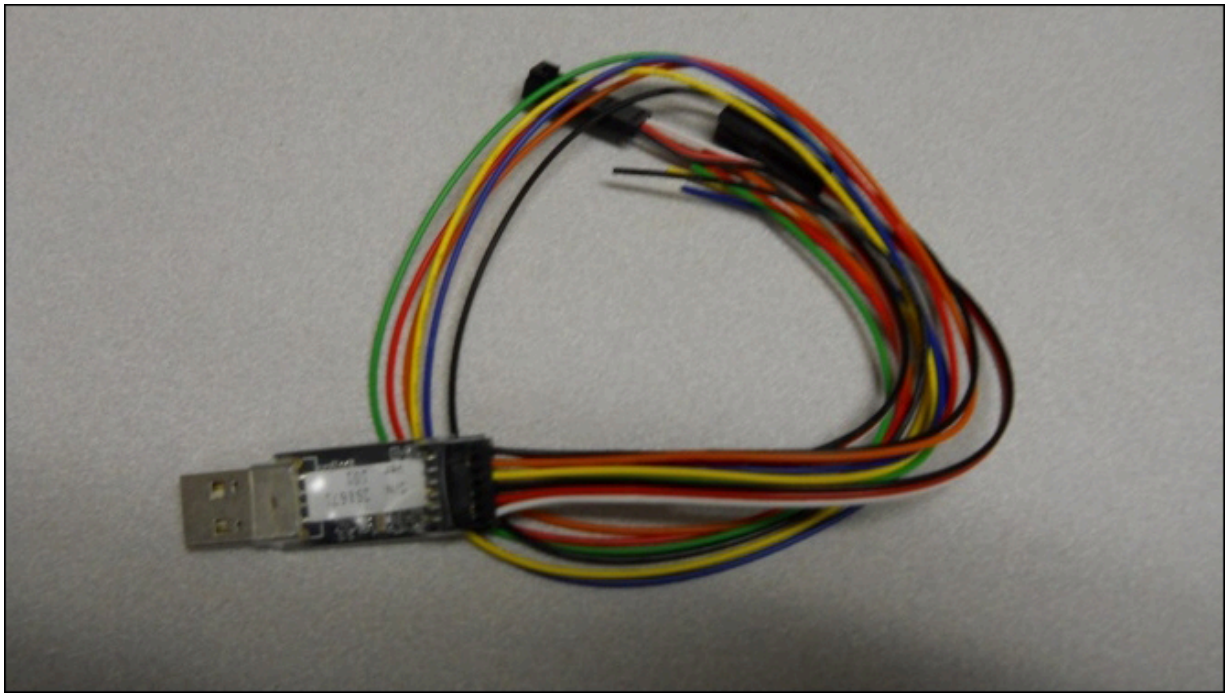
You'll also want to make sure you also get the connector cable for the device; it normally comes with the device. Here is a picture of the sensor with the cable attached:



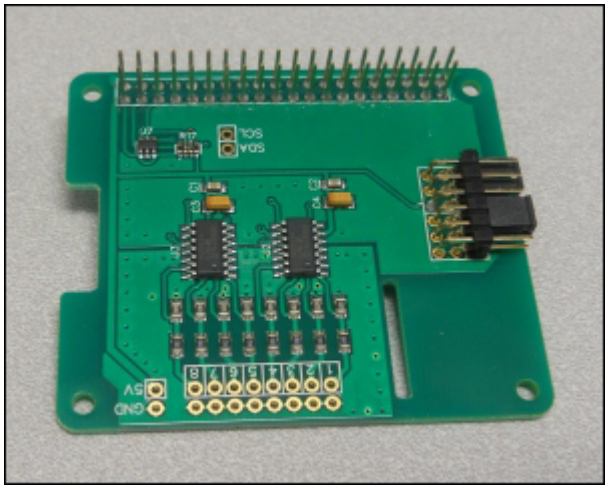
As noted in the tutorial, the voltage out of the sensor will be a voltage that will be an indication of the distance. However, this is an analog signal, and the Raspberry Pi doesn't have an analog-to-digital converter that can convert this analog voltage to a number that you can read in your program. You'll need to add an analog to digital converter to your project.

There are two choices. If you want an analog-to-digital converter that plugs directly into the USB interface, there is one offered by

www.phidgets.com. This board is really quite amazing; it takes the analog signals, turns them into digital numbers using an analog to digital converter, and then makes them available so that they can be read from the USB port. The model number of this part is **1011_0 - PhidgetInterfaceKit 2/2/2** and it is shown in the following:

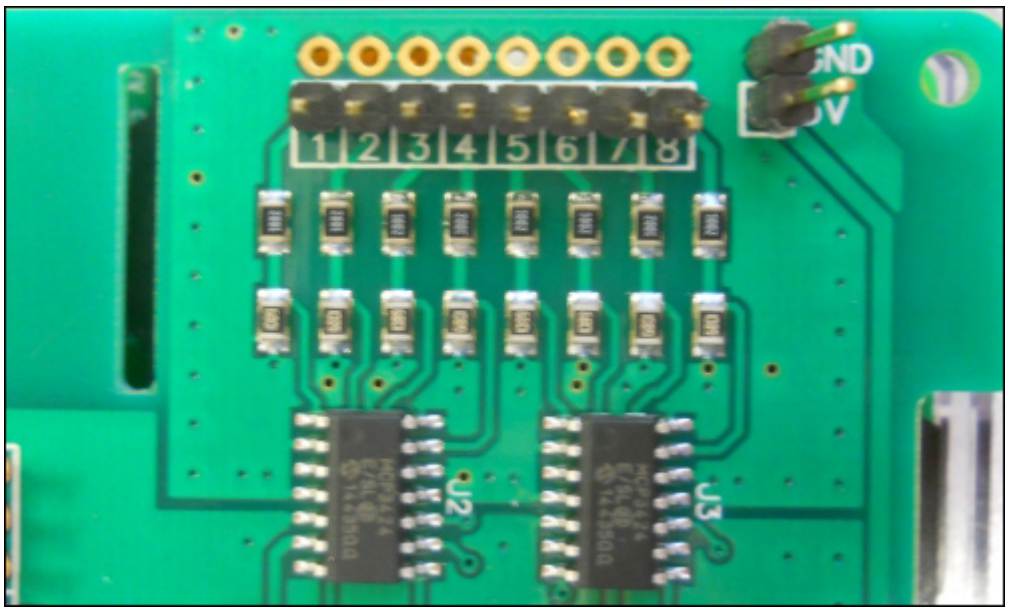


Unfortunately, it takes a bit of programming expertise to get it up and running. The other choice is to use an analog-to-digital converter that connects to the GPIO pins of the Raspberry Pi. There is a part, the **ADC pi+** from www.abelectronics.co.uk, that does this. It is pictured here:

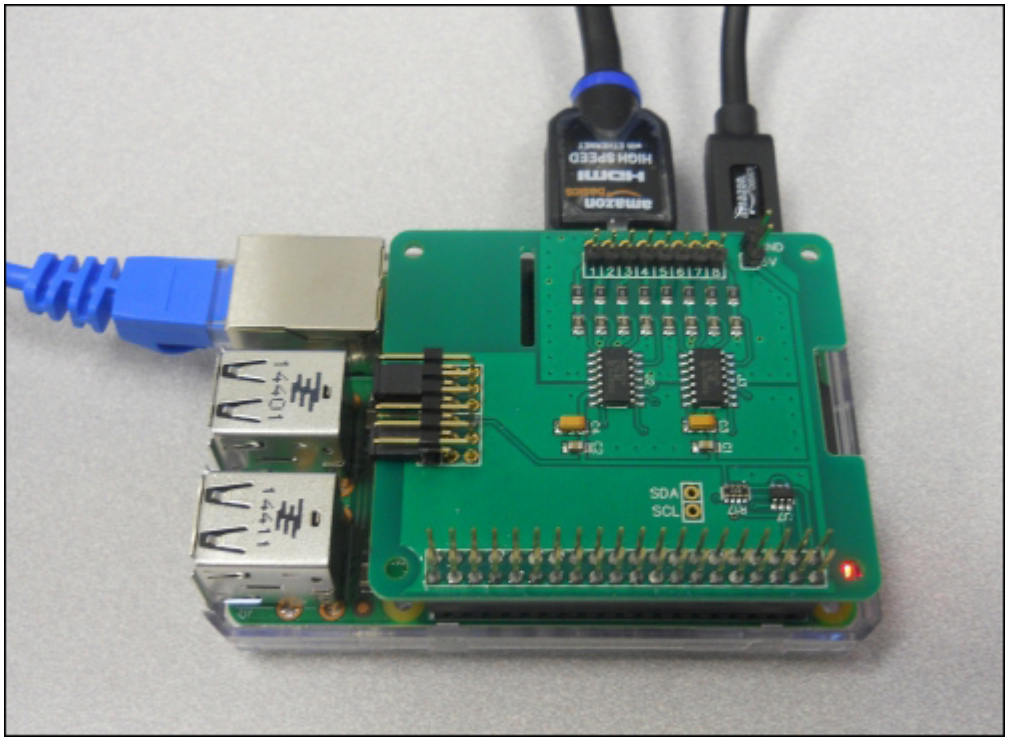


This device is easier to program, so this is what you'll use in this project. Now, let's connect the sensor:

1. Solder header pins to the ADC Pi+ board to connect it to the ADC, like this:



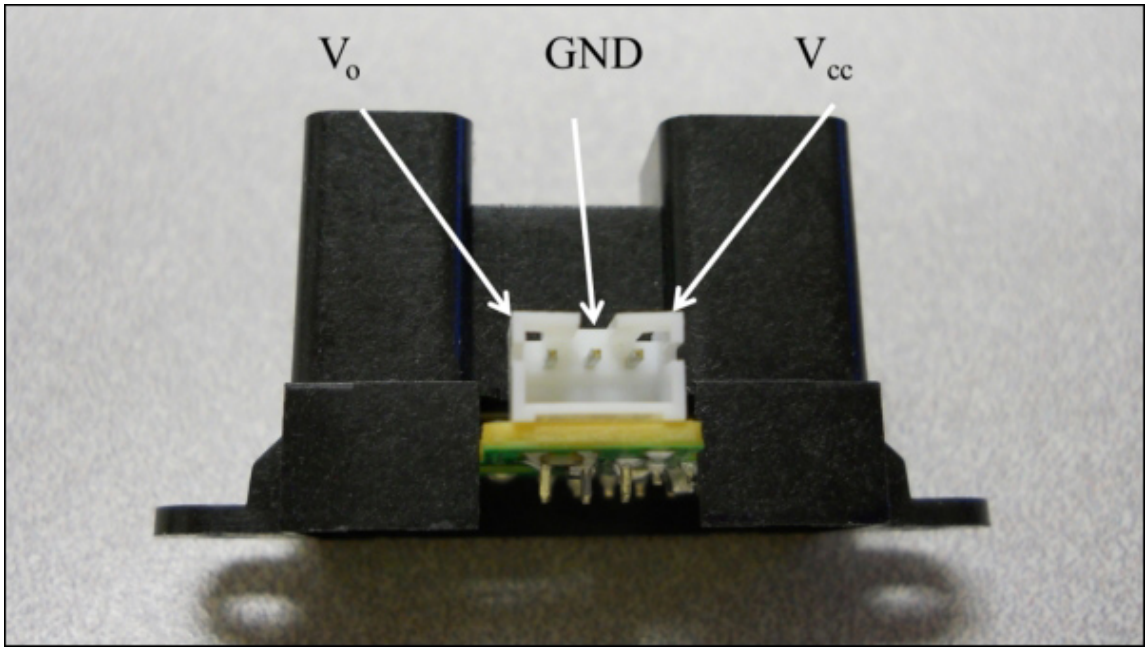
2. Now, plug the board into the Raspberry Pi B 2. Here is a picture of the combination:



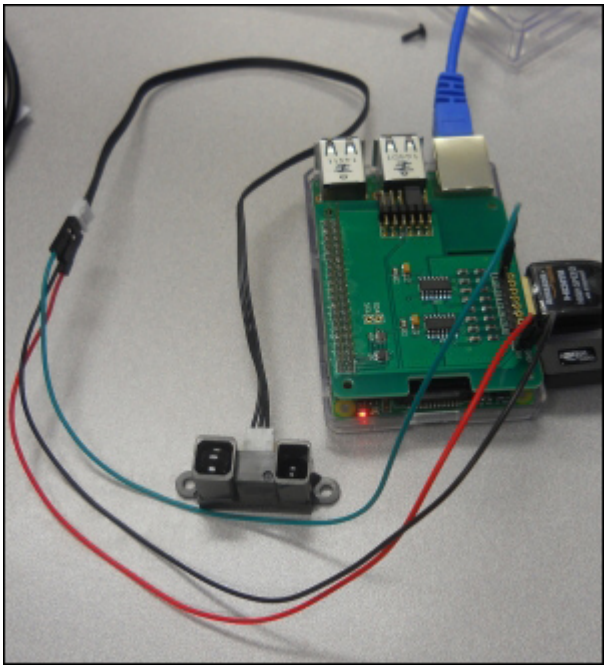
3. Now, you'll connect the IR sensor to the ADC. To connect this unit, you'll connect the three pins that are available at the bottom of the sensor. Here is the connection list:

ADC-DAC Board	**Sensor Pin**
5V	Vcc
GND	Gnd
In1	Vo

Unfortunately, there are no labels on the unit, but here are the pins you'll connect:

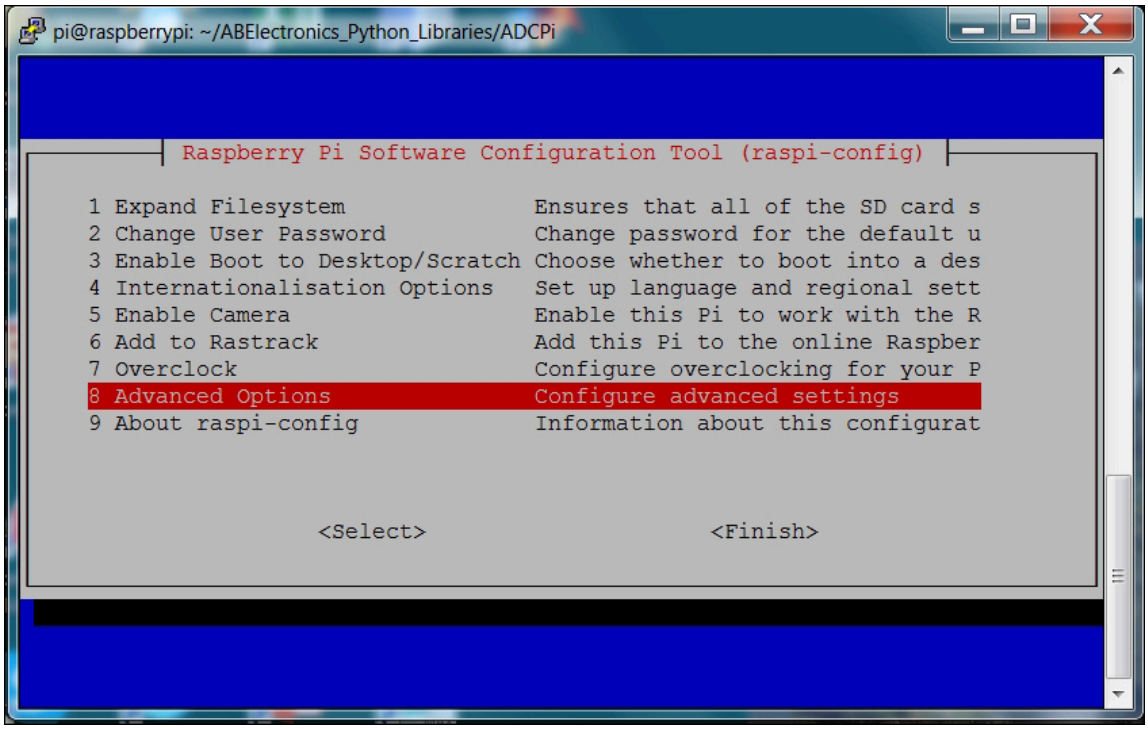


It's easiest to connect to the three-wire cable that normally comes with the sensor. Once the pins are connected, you are ready to access the data from the sensor via a Python program on the Raspberry Pi. The entire system looks like this:

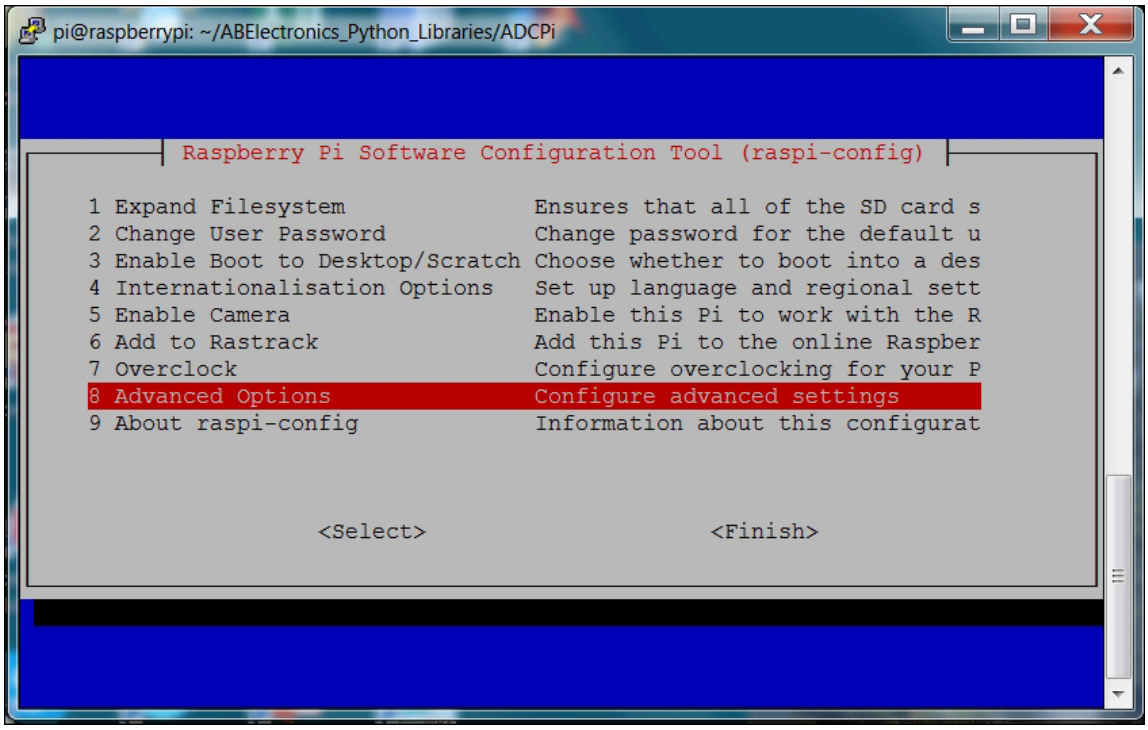


Now, you are ready to add some code to read the IR sensor. You'll need to follow these steps to talk to the ADC:

1. The first step in enabling the ADC is to enable the I2C interface. This is done by running `raspi-config` and selecting **8 Advanced Options** like this:

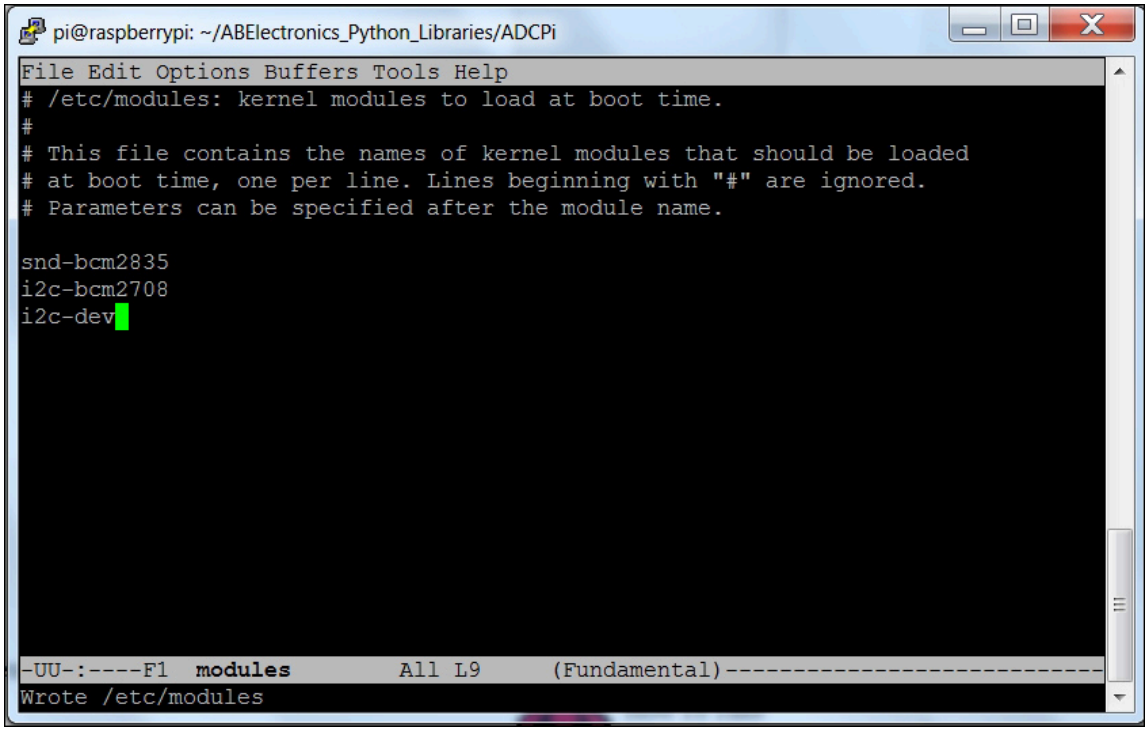


2. Once there, go to the **A7 I2C** selection to enable the I2C like this:

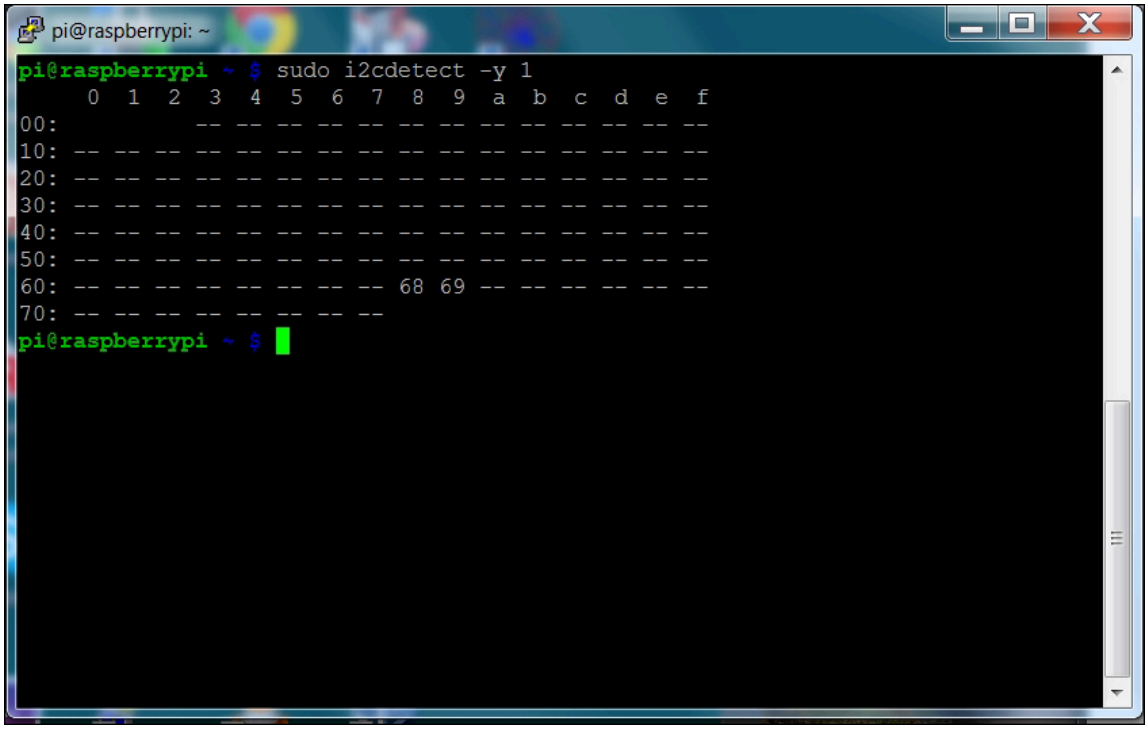


Perform all the selections to enable the I2C interface and load the library, and then reboot the Raspberry Pi.

You'll also need to edit the `/etc/modules` file and add the following two lines:

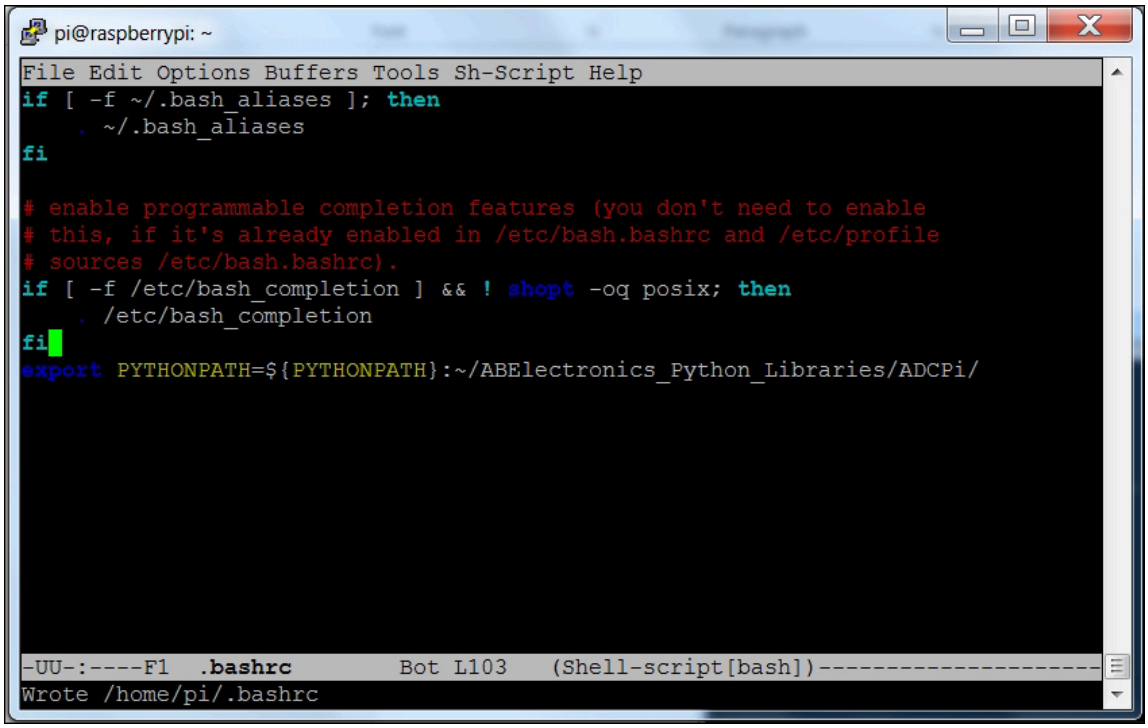


Reboot the Raspberry Pi. You can see whether the I2C is enabled by typing `sudo i2cdetect -y 1`, and you should see something like this:



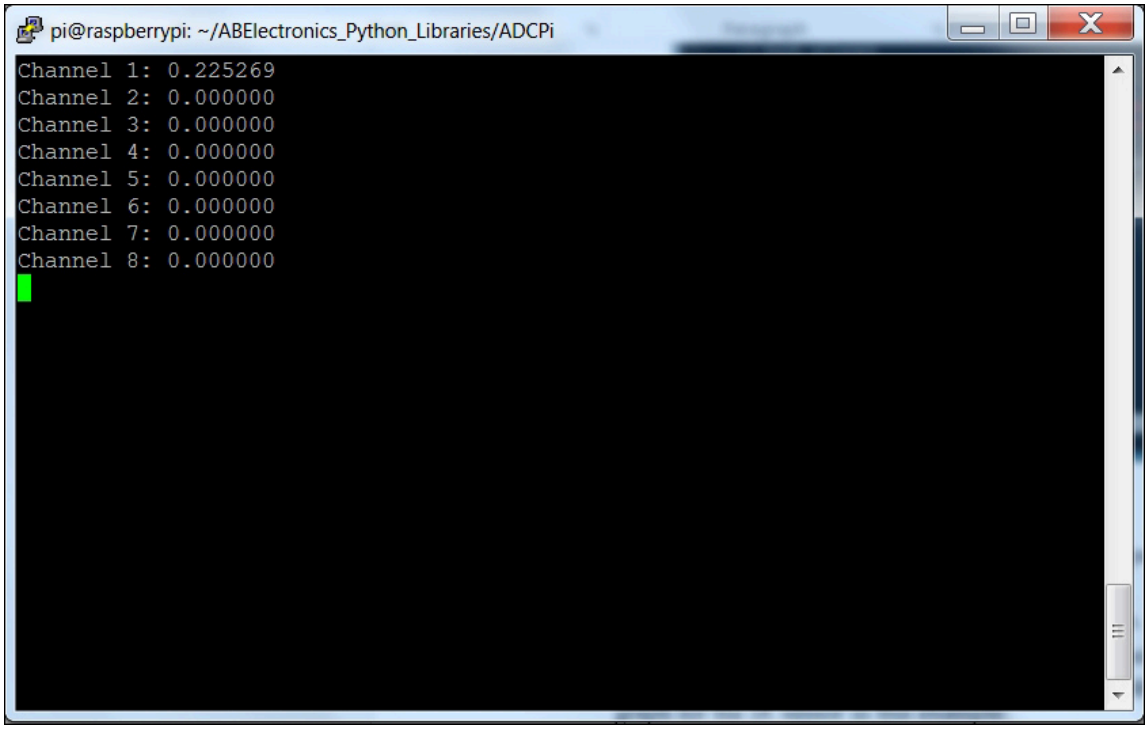
The I2C device, your ADC, is available at the **68** and **69** addresses.

3. Now, you can download the code. To do this, type `git clone https://github.com/abelectronicsuk/ABElectronics_Python_Libraries.git` from the home directory, and the Python libraries will be installed on your Raspberry Pi.
4. Go to the `./ABElecttronics_Python_Libraries/ADCPi` directory; here are the programs for your specific hardware. Following the instructions in the `README.md` file, type `sudo apt-get update`, and then type `sudo apt-get install python-smbus`. This will install the `smbus` library, which are required for the ADC to work. Also, type `sudo adduser pi i2c` to add `pi` to the group that can access i2c.
5. You'll need to edit your `.bashrc` file in your home directory, adding the following lines:

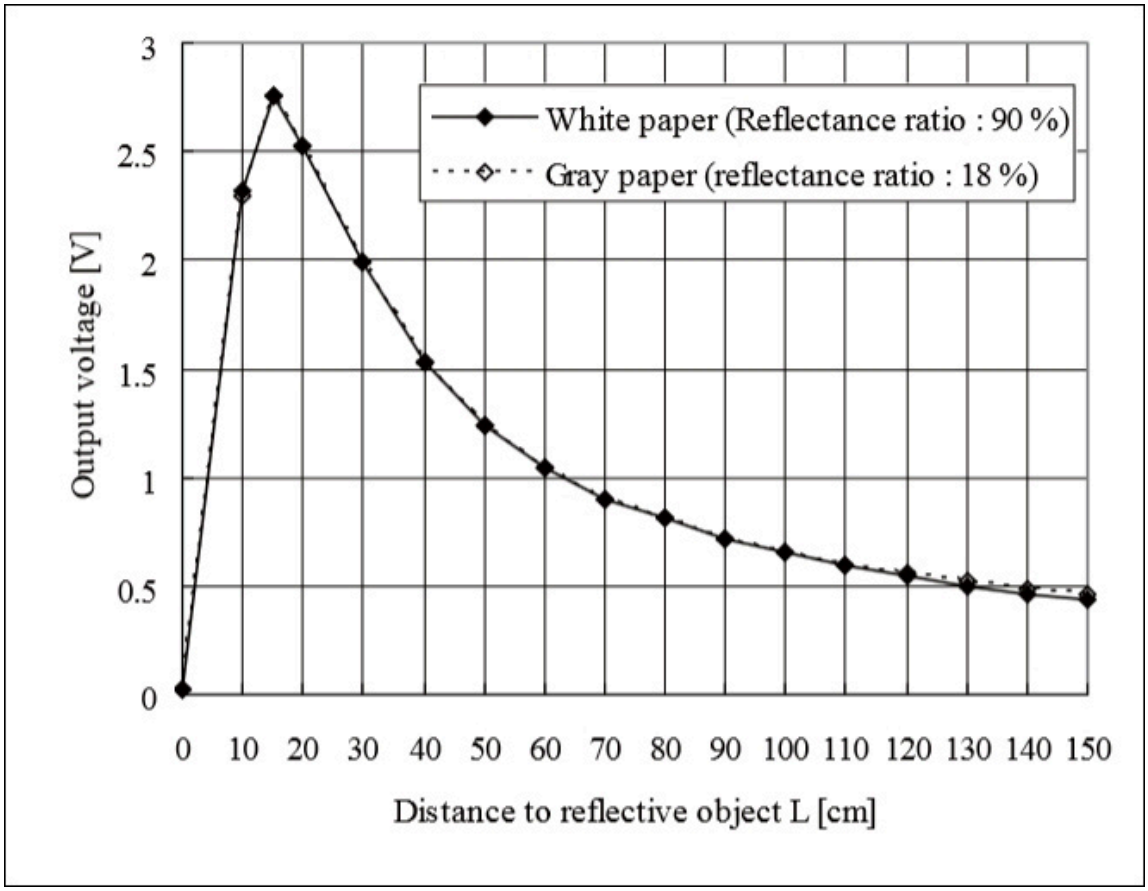


Adding this line will add this library to the path so that you can access the functionality. Reboot the Raspberry Pi.

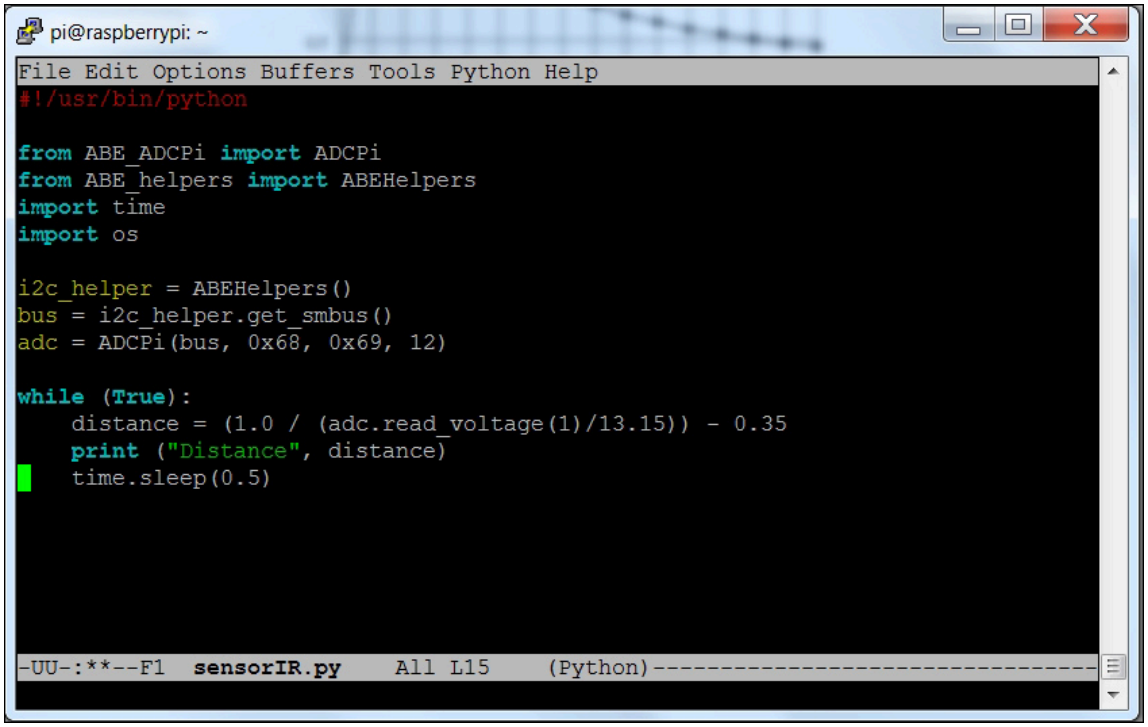
6. Now, you can run one of the demo programs. Type `python demo-readvoltage.py`, and you should see something like this:



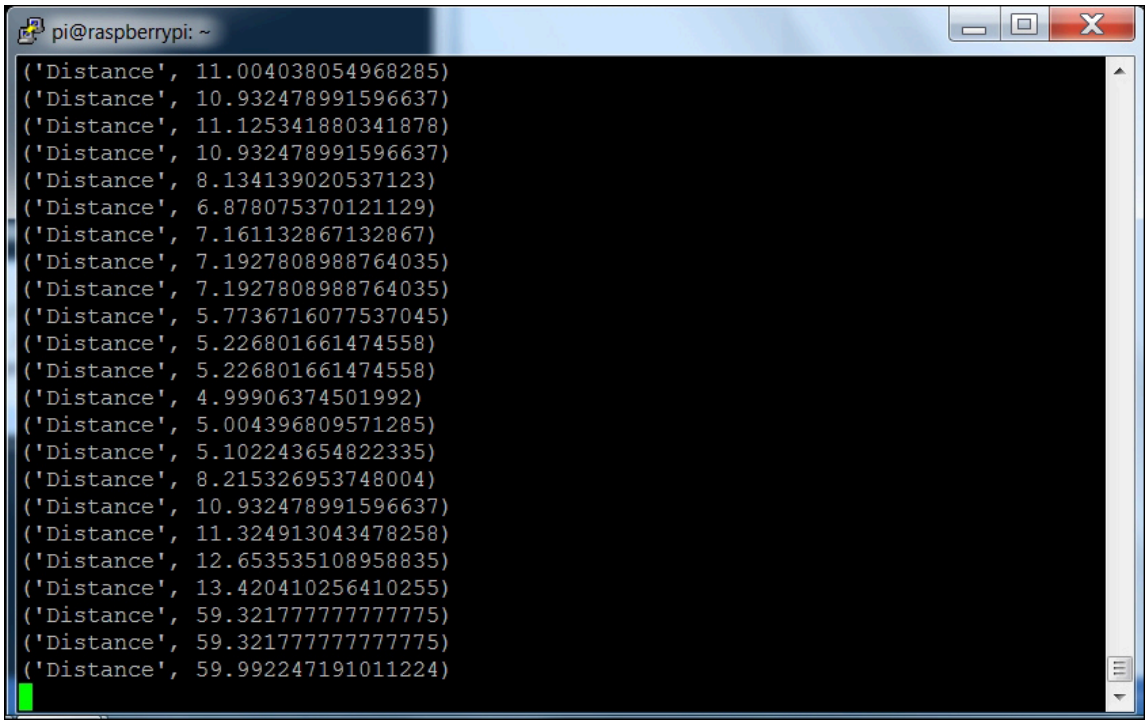
These raw readings are great, but now you'll want to build a program that takes the data from the first ADC and translates it to the distance. To do this, you'll need a graph of the voltage to distance readings for your sensor. Here is the graph for the IR sensor in this example:



There are really two parts to the curve; the first is the distance up to about 15 centimeters, and the second is the distance from 15 centimeters to 150 centimeters. It is easiest to build a simple mathematical model that ignores distances closer than 15 centimeters and models the distance from 15 centimeters. For more information on how to build this model, refer to http://davstott.me.uk/index.php/2013/06/02/raspberry-pi-sharp-infrared/. Here is the Python program using this model:



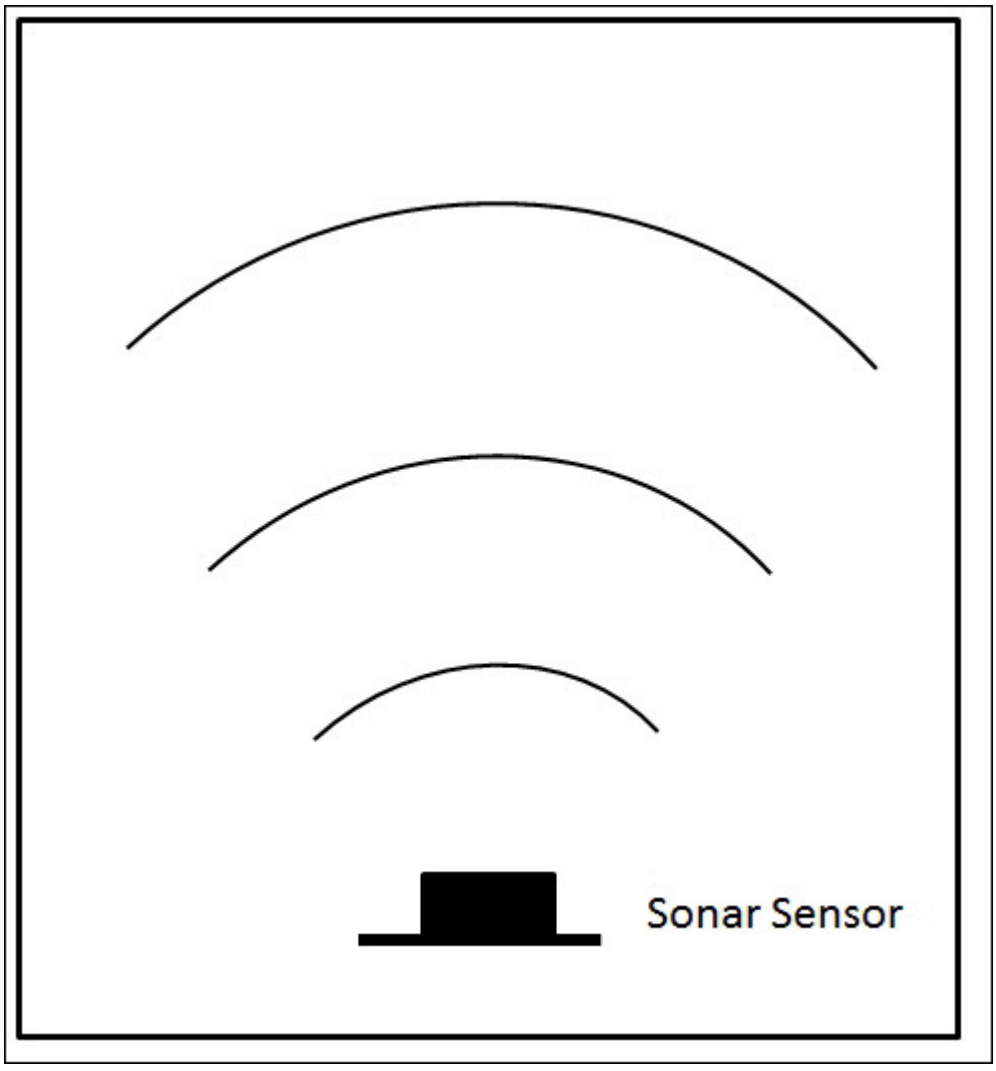
The only new line of code is the distance = (1.0 / (adc.read_adc_voltage(1) / 13.15)) - 0.35 line. It converts your voltage to distance. You can now run your program and you'll see the results in centimeters, like this:



Now, you can measure the distance to objects using your IR sensor!

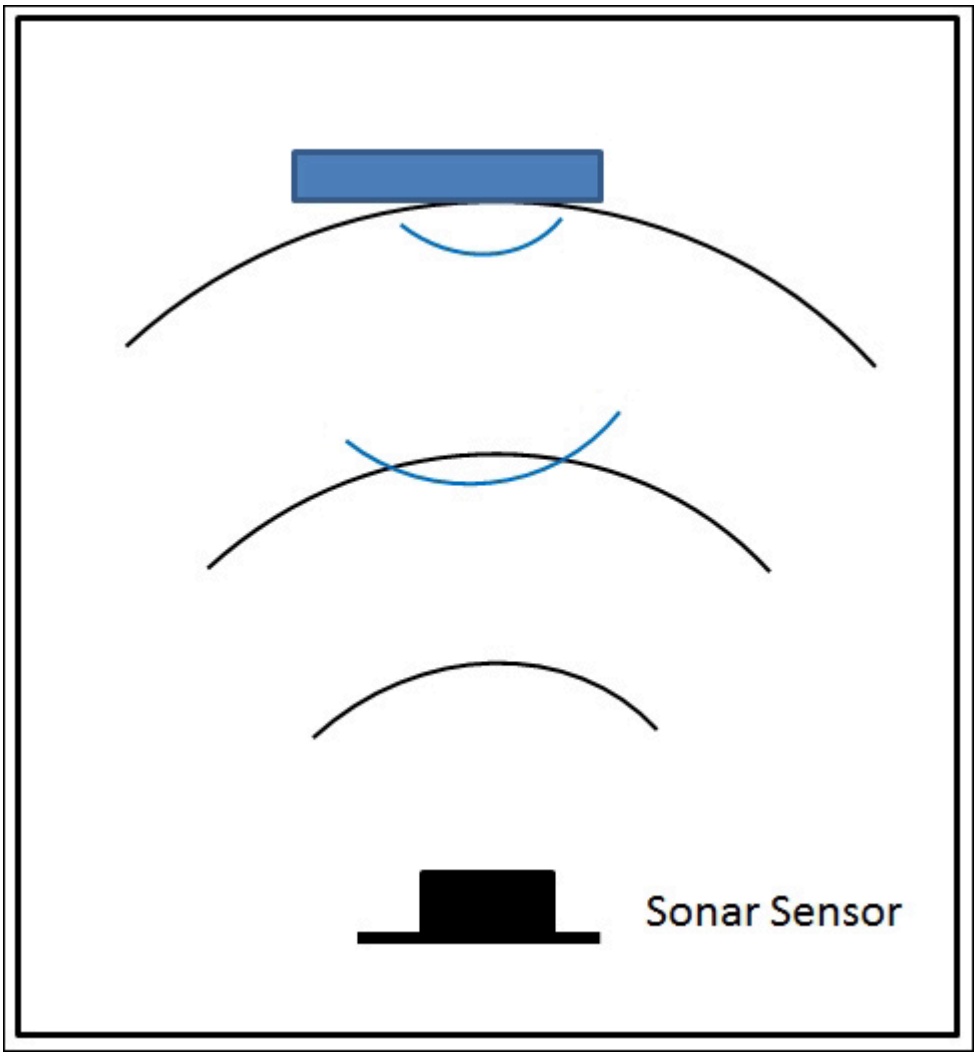
Connecting Raspberry Pi to a USB sonar sensor

There is yet another way to sense the presence of objects: using a sonar sensor. But before you add this capability to your system, here's a little tutorial on sonar sensors. This type of sensor uses ultrasonic sound to calculate the distance from an object. The sound wave travels out from the sensor, as illustrated in the following figure:



The device sends out a sound wave 10 times a second. If an object is in the path of these waves, then the waves reflect off the object,

sending waves that return to the sensor, as shown in the following figure:



The sensor then measures any return. It uses the time difference between when the sound wave was sent out and when it returned to measure the distance from the object.

Note

Sonar sensors are also quite accurate, normally with low percentage errors, and are not affected by the lighting or color in the environment.

There are several choices if you want to use a sonar sensor to sense the distance. The first is to use a sonar sensor that connects to the USB port. The following is an image of a USB sonar sensor:



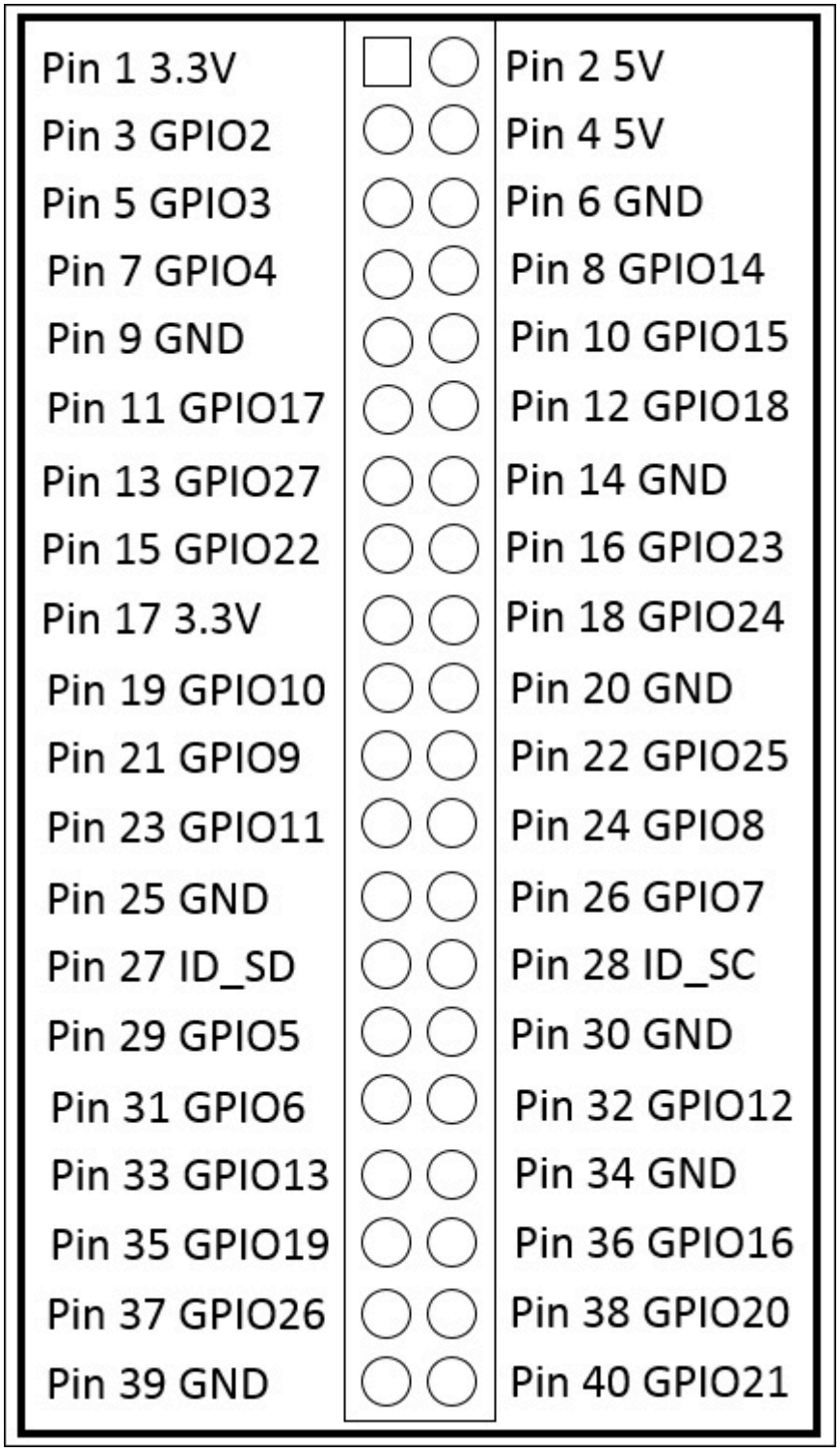
This is the **USB-ProxSonar-EZ** sensor, and can be purchased directly from MaxBotix or on Amazon. There are several models, each with a different distance specification; however, they all work in the same way.

You can also choose a sonar sensor that connects to the GPIO of the Raspberry Pi. Here is a picture of this sort of inexpensive sonar sensor:



This sensor is less expensive and easy to use; it takes a bit of processing power to coordinate the efforts of timing the send and receive signals, but the Raspberry Pi B 2 has the processing power needed. Here are the steps to set up this sonar sensor to sense the distance:

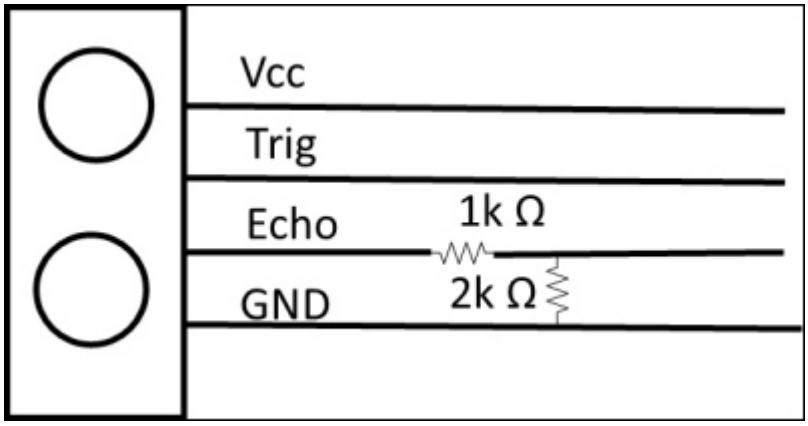
1. The first step is to understand the GPIO pins of the Raspberry Pi B 2. Here is a diagram of the layout of the pins:



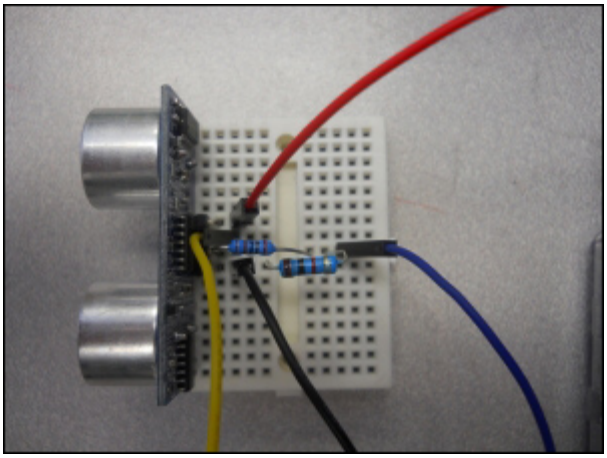
In this case, you'll need to connect to the 5 volt connection of the Raspberry Pi B2, which is pin 2. You'll also need to connect to the GND, which is pin 6. You'll use pin 16 as an output trigger

pin and pin 18 (GPIO24) as an input to time the echo from the sonar sensor.

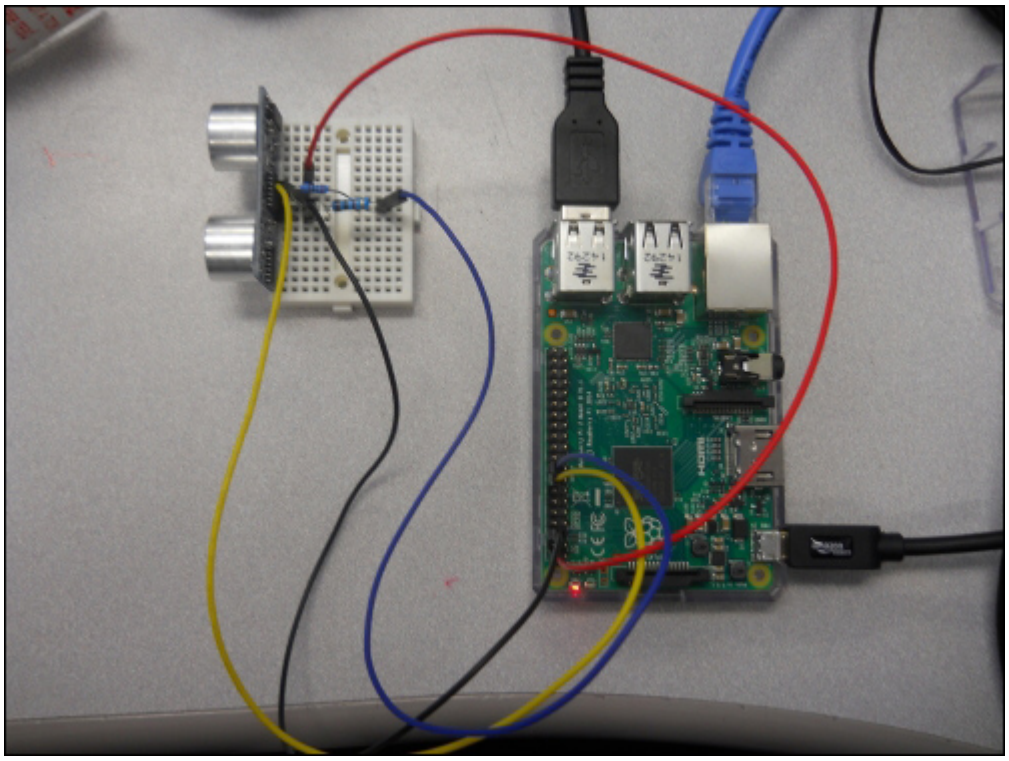
2. Now that you know the pins you'll connect to, you can connect the sonar sensor. There is a problem, as you can't connect the 5 volt return from the sonar sensor directly to the Raspberry Pi GPIO pins; they want 3.3 volts. You'll need to build a voltage divider that will reduce the 5 volts to 3.3 volts. This can be done with two resistors, which are connected as shown in this diagram:



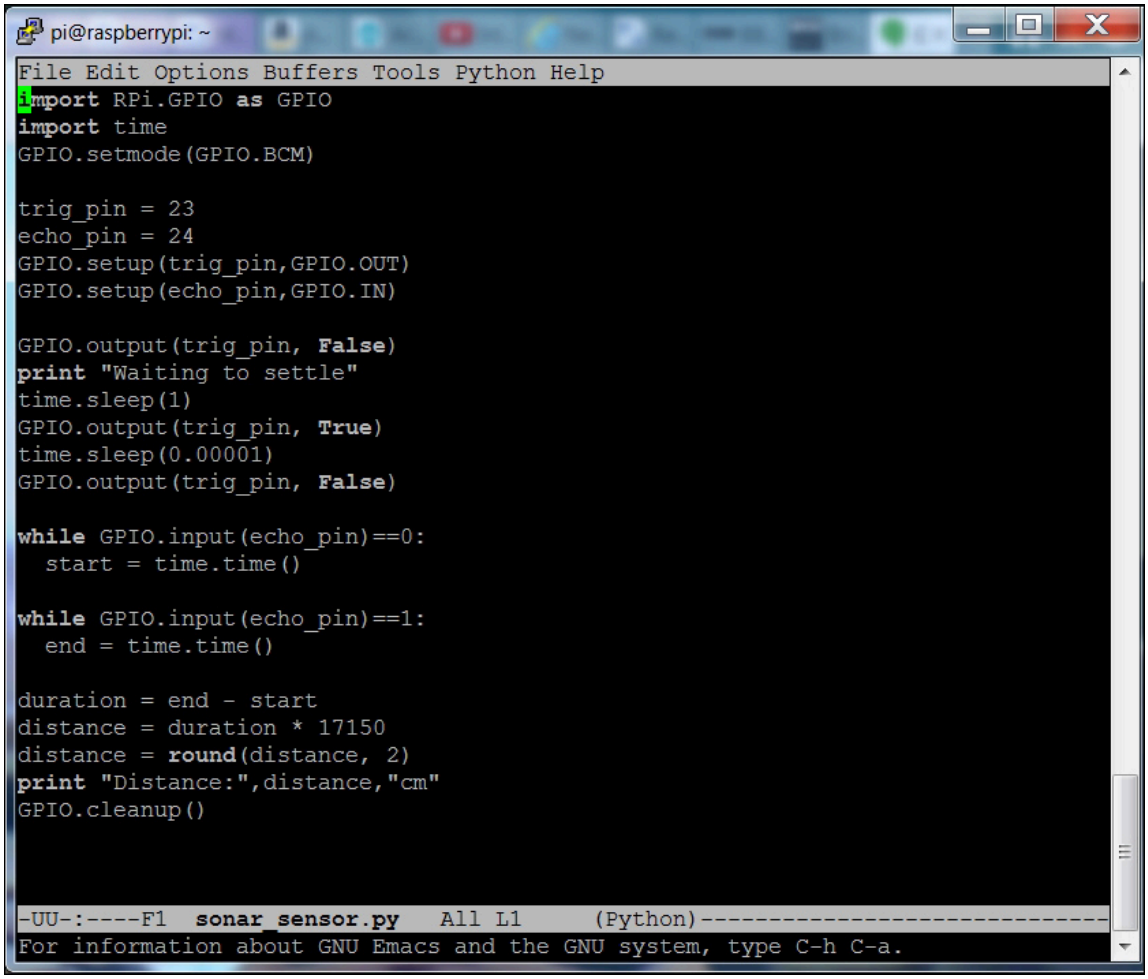
If you'd like more information on how the voltage divider works in this configuration, refer to http://www.modmypi.com/blog/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi. The combination of these two resistors will reduce the voltage to the desired levels. You may want to put all of this is in a small breadboard, as shown here:



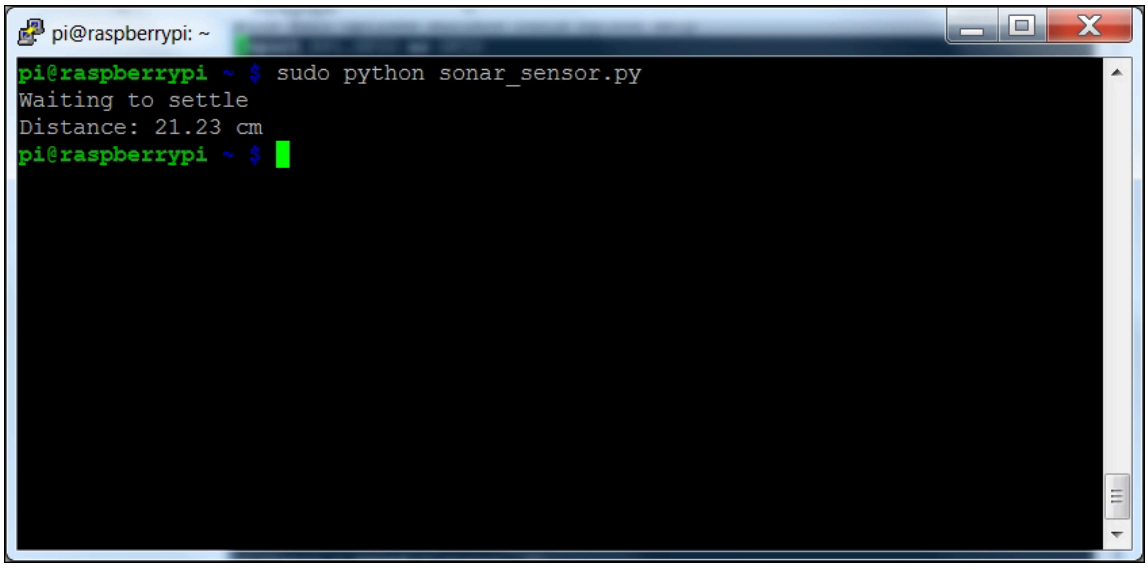
Finally, connect it to the Raspberry Pi, like this:



3. Now that the device is connected, you'll need a bit of code to read in the value, make sure it is settled (a stable measurement), and then convert it to distance. Here is the Python code for this program:



Now, you should be able to run the program and see a result, like this:



Now that you have your sensors up and working, you can avoid or find objects with your biped.

Summary

Congratulations! You can now detect and avoid walls and other barriers to your robot. You can also use these sensors to detect objects that you might want to find. In the next chapter, you'll learn how to perform path planning to move your robot from point A to point B and even give your robot intelligence as to what to do if it encounters a barrier in its path.

Chapter 5. Path Planning and Your Biped

Now that your biped is up and mobile and able to find barriers, you can now start to have it move around autonomously. However, you'll want to have your robot planed his path, that is, if it knows where it has started and the desired end point, it can move from the starting point to the end point.

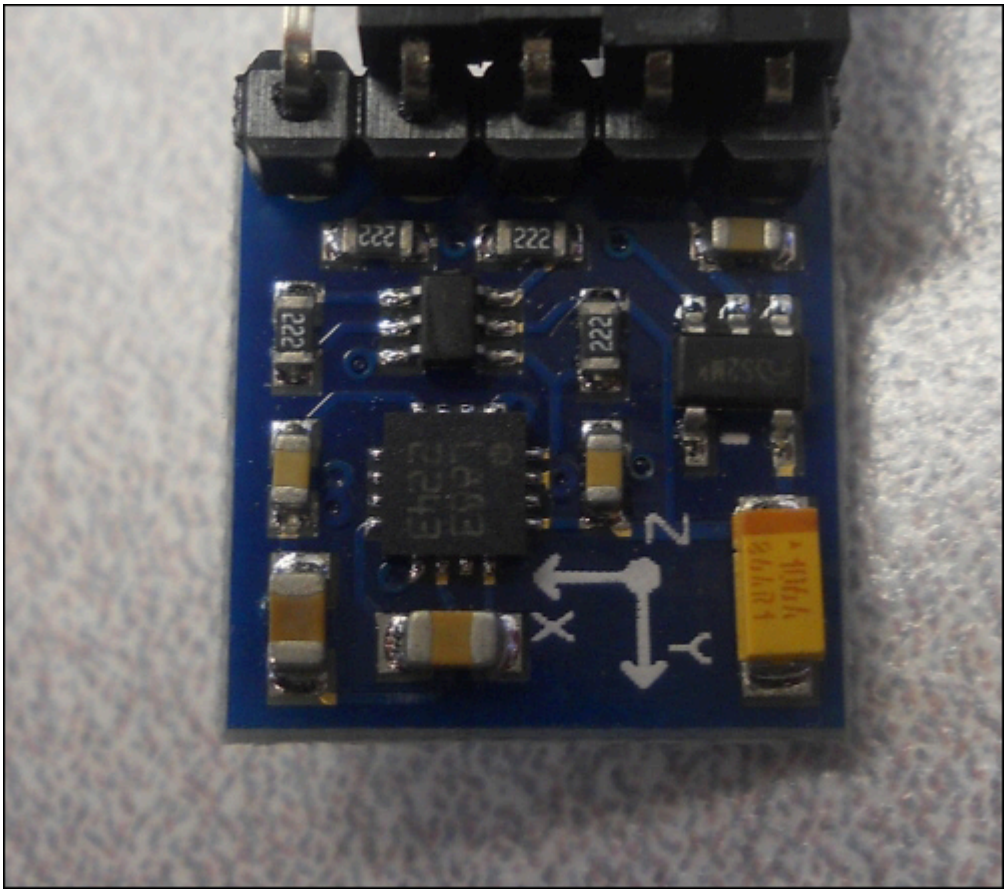
In this chapter, you will be learning about:

- How to add a compass to your biped, so you'll have a sense of direction
- Learning some basic path planning techniques for your robot

Connecting a digital compass to the Raspberry Pi

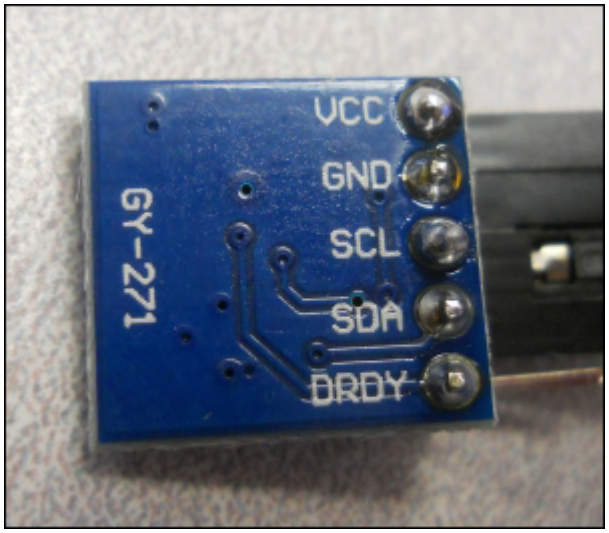
One of the important pieces of information that might be useful for your robot, it if is going to plan its own path, is its direction of travel. So, let's learn how to hook up a digital compass to the Raspberry Pi.

There are several chips that provide digital compass capability; one of the most common is the **HMC5883L 3-Axis Digital Compass chip**. This chip is packaged onto a module by several companies, but almost all of them result in a similar interface. The following is a picture of one the **GY-271 HMC5883L Triple Axis Compass Magnetometer Sensor Module**, which is available from a number of online retailers:

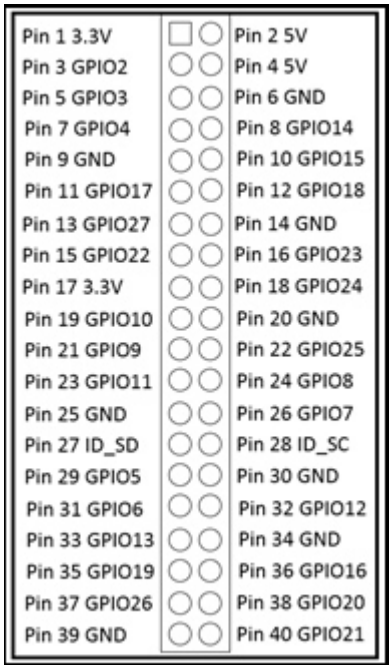


This type of digital compass uses magnetic sensors to discover the earth's magnetic field. The output of these sensors is then made accessible to the outside world through a set of registers that allow the user to set things such as the sample rate, and continuous or single sampling. The *x*, *y*, and *z* directions are output-using registers as well.

The connections to this chip are straightforward and the device communicates with the Raspberry Pi by using the I2C bus, a standard serial communications bus. The I2C interface is a synchronous serial interface and provides more performance than an asynchronous Rx/Tx serial interface. The SCL data line provides a clock, while the data flows on the SDA line. The bus also provides addressing so that more than one device can be connected to the master device at the same time. On the back of the module, the connections are labeled, as shown in the following image:



You'll then connect the device to the GPIO pins on Raspberry Pi. The following is the pin out of Raspberry Pi:

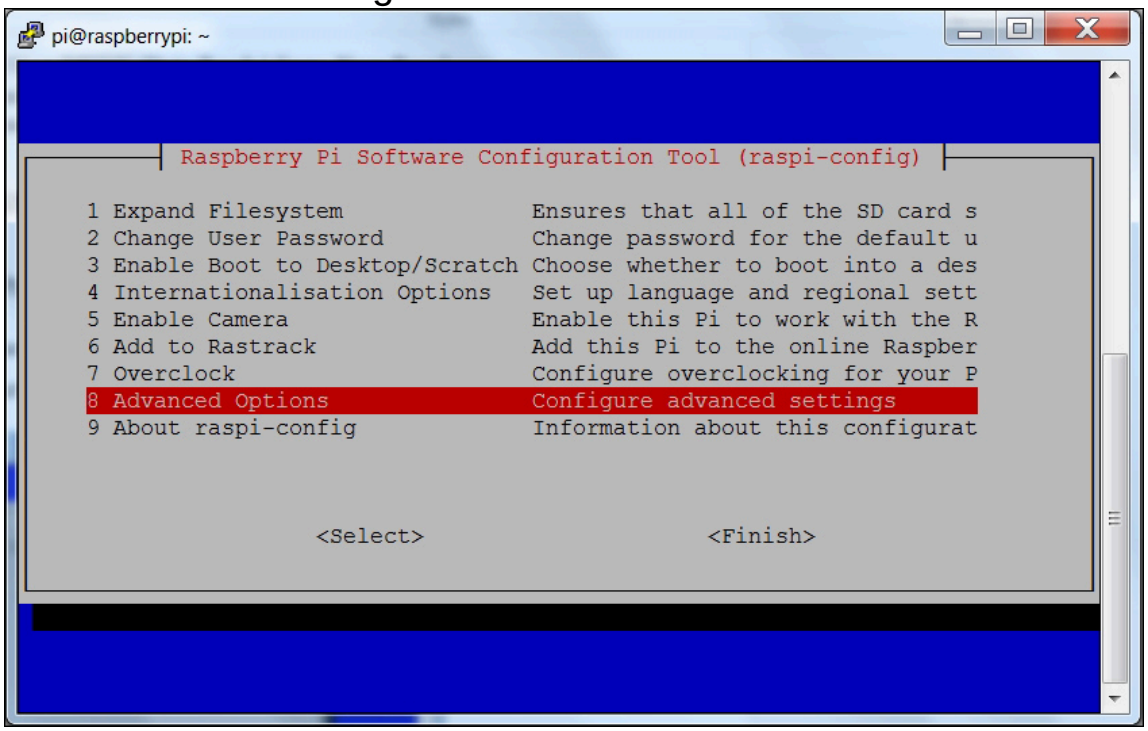


Connect your device to the VCC on the device to Pin 1 (3.3 V) on Raspberry Pi. Connect GND to Pin 9 (GND). Connect SCL on the device to Pin 5 (GPIO 3) and SDA to Pin 3 (GPIO 2) on the device. Notice that you will not connect the **Data Ready** (**DRDY**) line. Now, you are ready to communicate with the device.

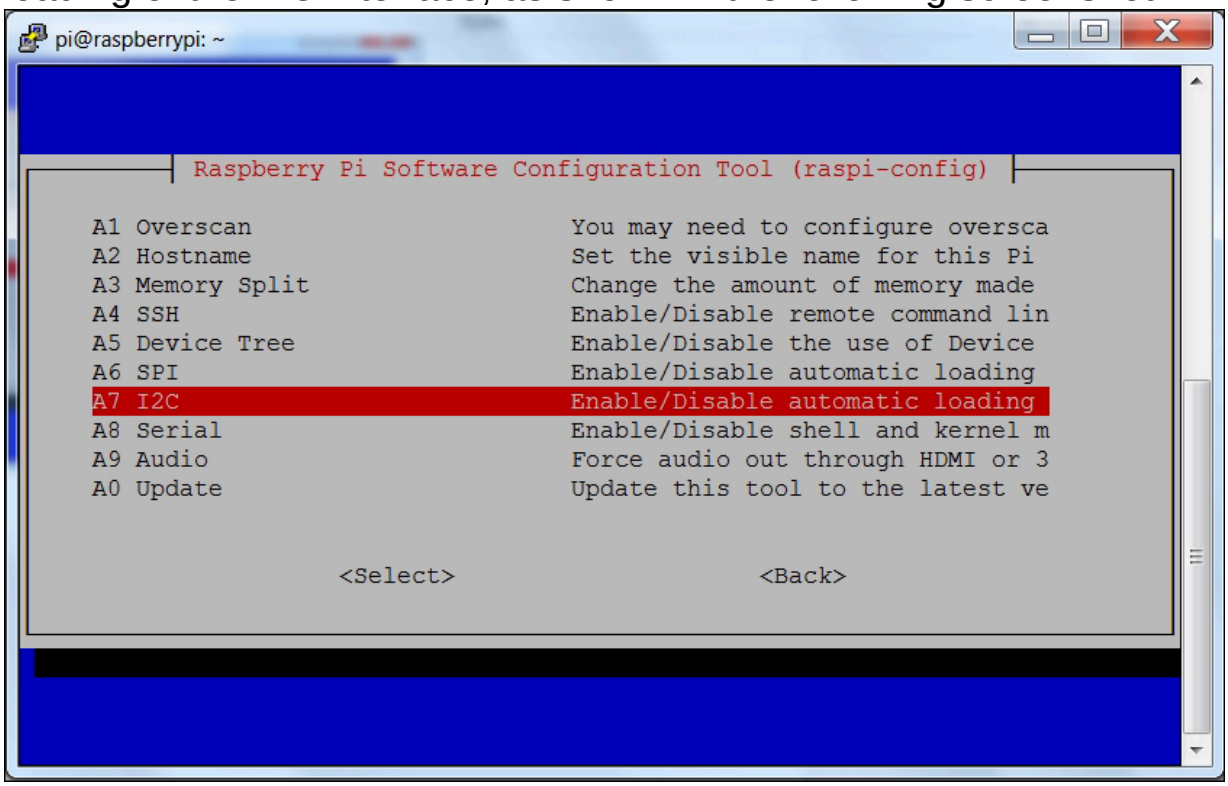
Accessing the compass programmatically

In order to access the compass capability, you'll need to enable the I2C library on Raspberry. If you used the IR sensor and ADC additional hardware in Chapter 4, *Avoiding Obstacles Using Sensors*, you will have already done this. If not, follow these instructions to enable the I2C interface:

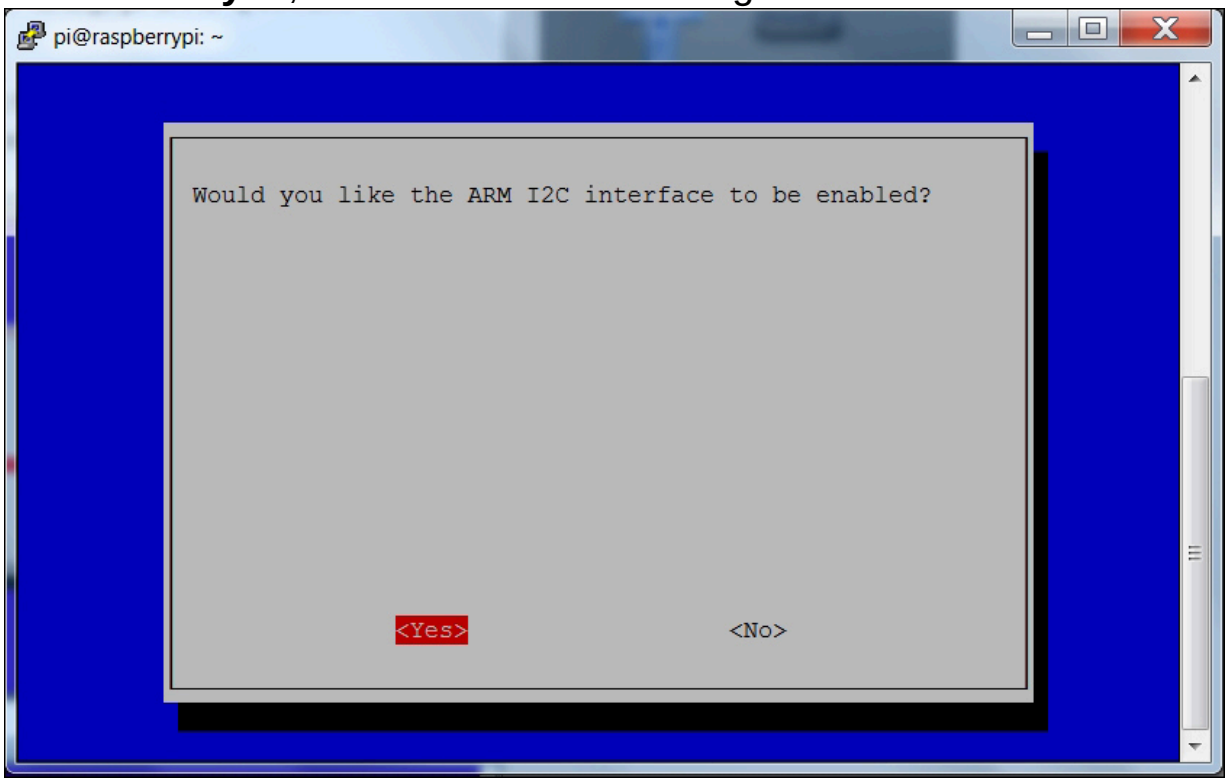
1. Run `raspi-config`. Select the **Configure advanced settings**, as shown in the following screenshot:



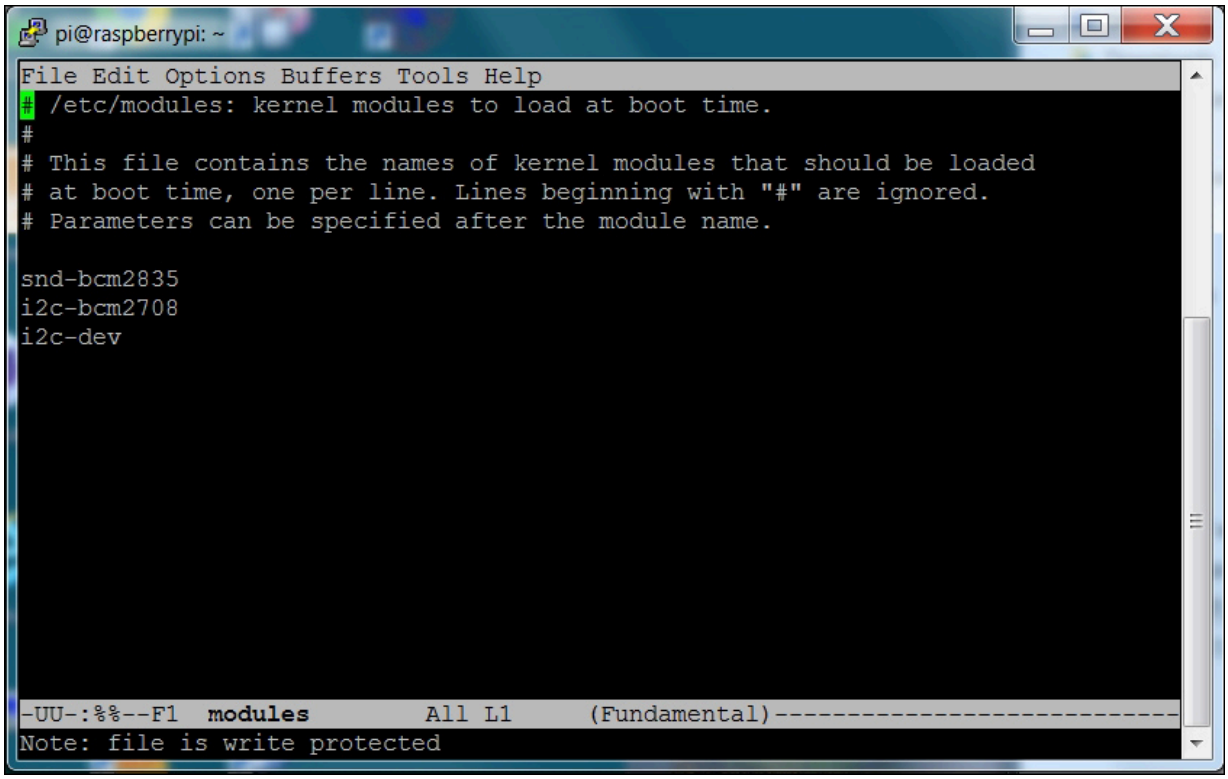
From the next selection page, select the Enable/disable automatic loading of the I2C interface, as shown in the following screenshot:



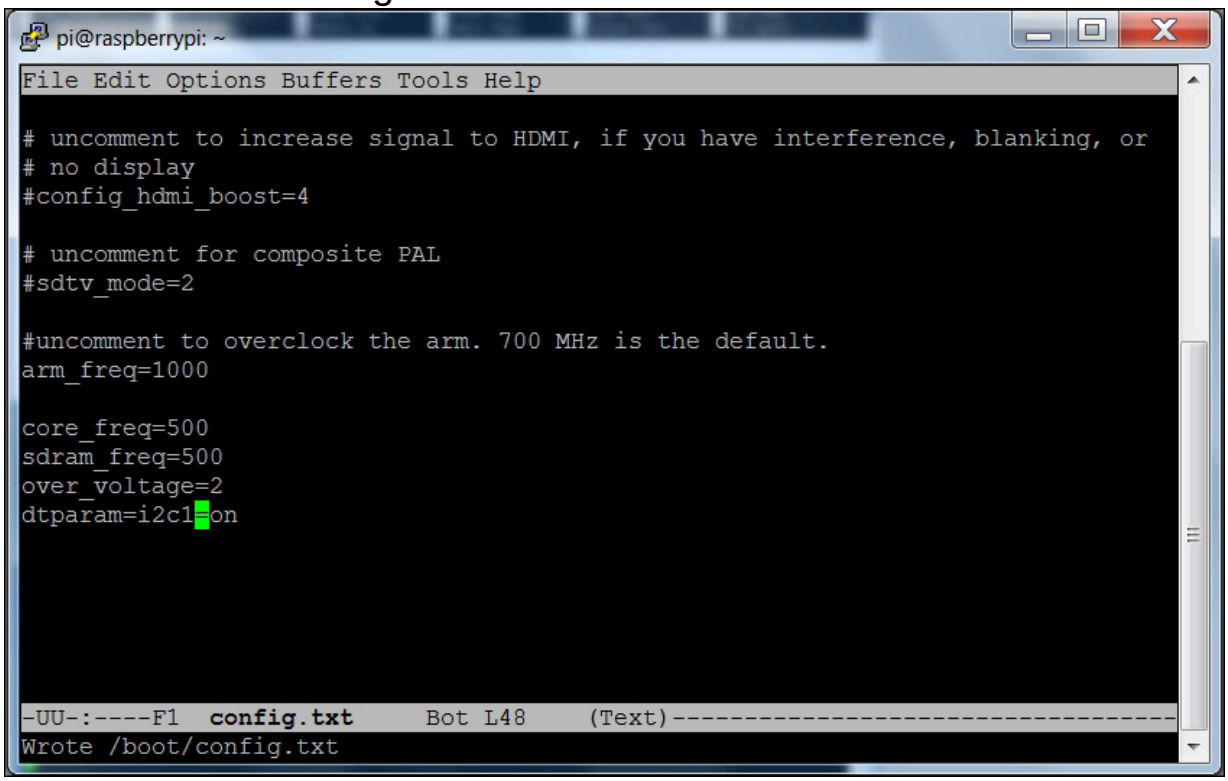
Then select **yes**, as shown in the following screenshot:



You'll also want to edit the file `/etc/modules` and add the lines `i2c-bcm2708` and `i2c-dev`, as shown in the following screenshot:

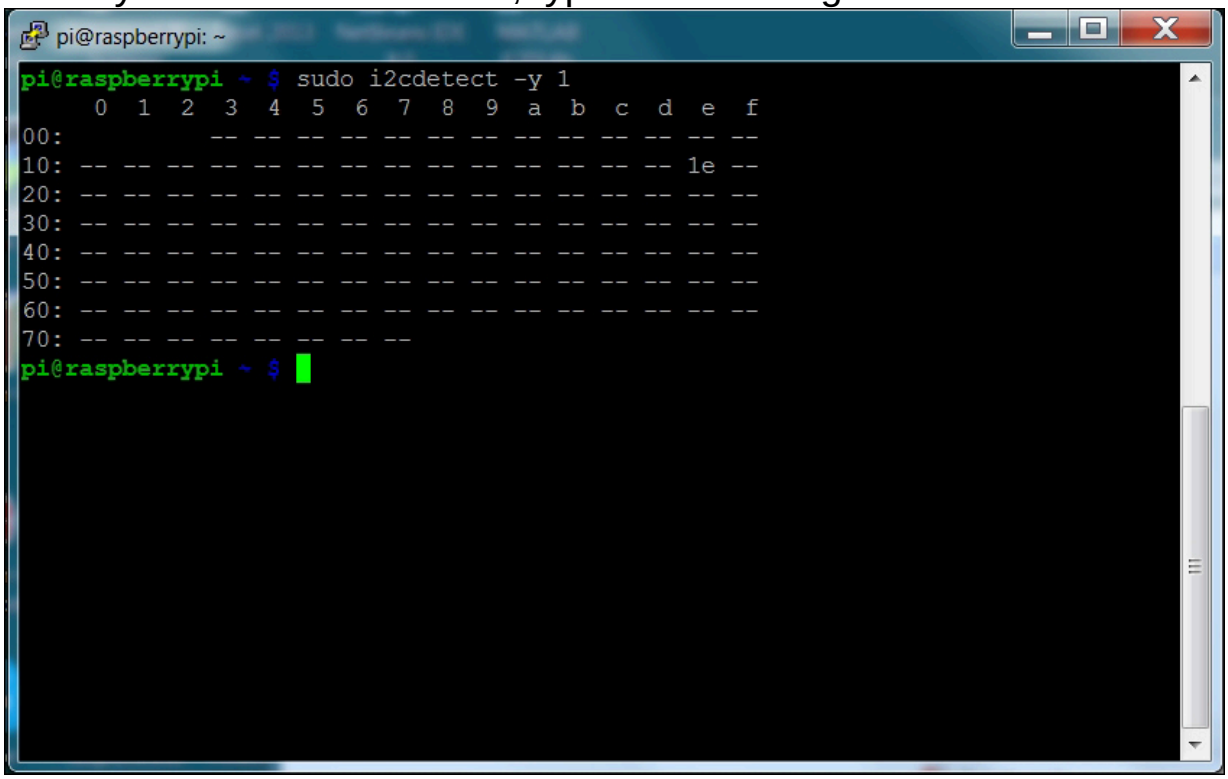


And one final edit, change the last line in `/boot/config.txt`, as shown in the following screenshot:



Now, reboot Raspberry Pi.

- With the device connected, you can see if the system knows about your device. To do this, type the following command:

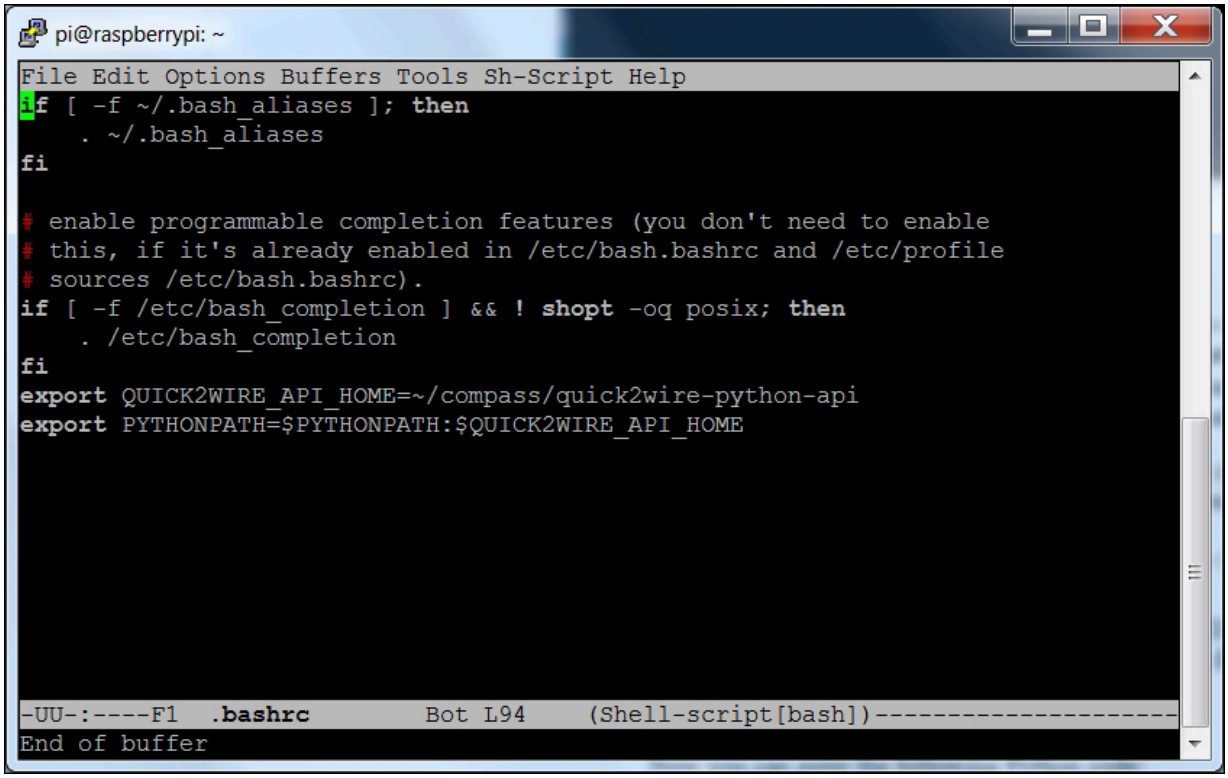


You can see the device at **1e**.

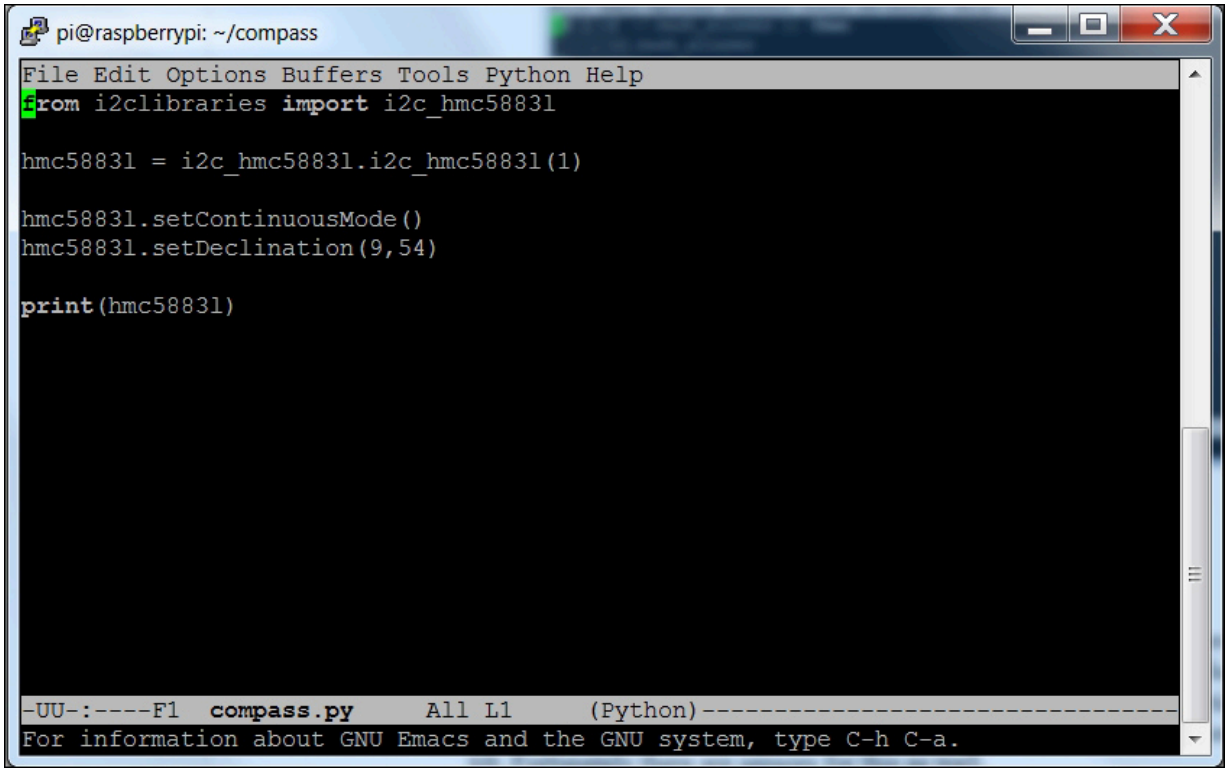
- Now you communicate with your digital compass. To do this, you'll need to create a Python program. But before you create your Python code, you'll want to download a library that will make this all much easier. To do this, first create a directory a directory called `compass` and `cd` to that directory. Then, type `git clone https://github.com/quick2wire/quick2wire-python-api.git` to download the `quick2wire-python-api` library. Finally, type `git clone https://bitbucket.org/thinkbowl/i2clibraries.git` to get the i2clibraries.

You'll also need to set some environment variables. Do this by going to your home directory and editing the `.bashrc` file, adding these two

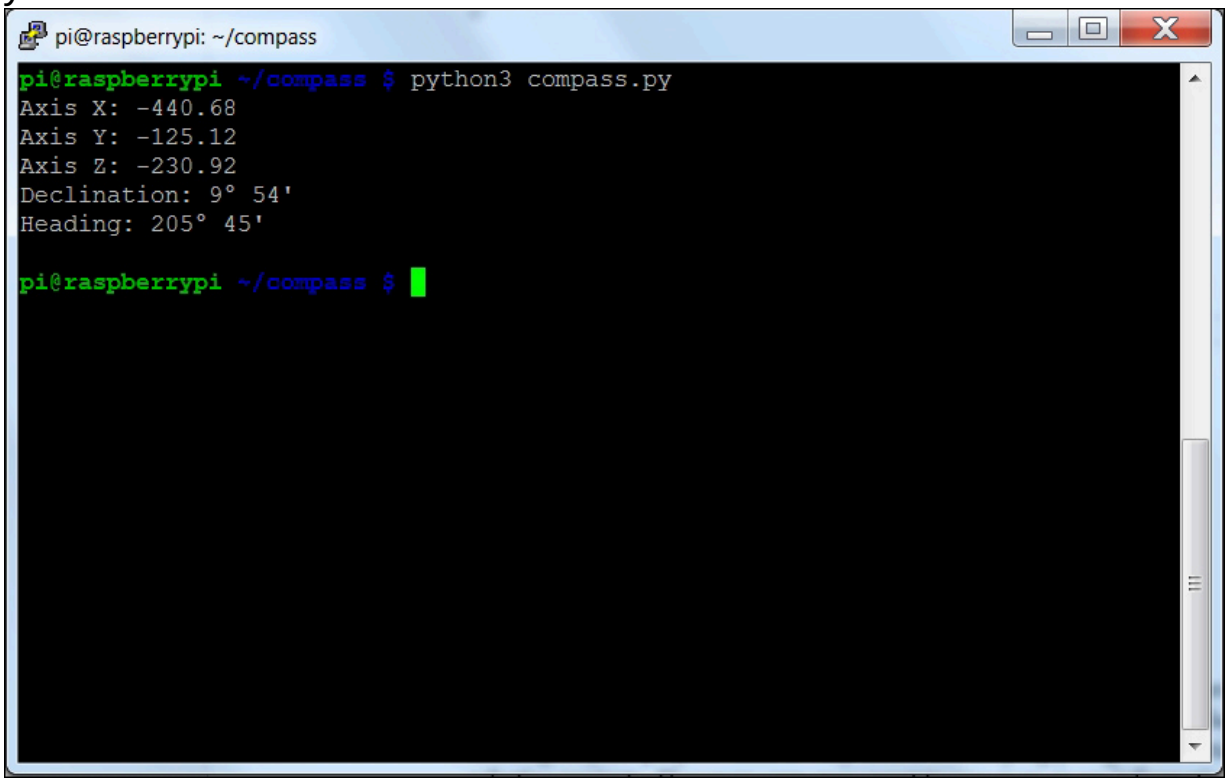
lines at the end:



- Now, you can create the following Python code:



- Now, run the code by typing `python3 compass.py` command and you should see:



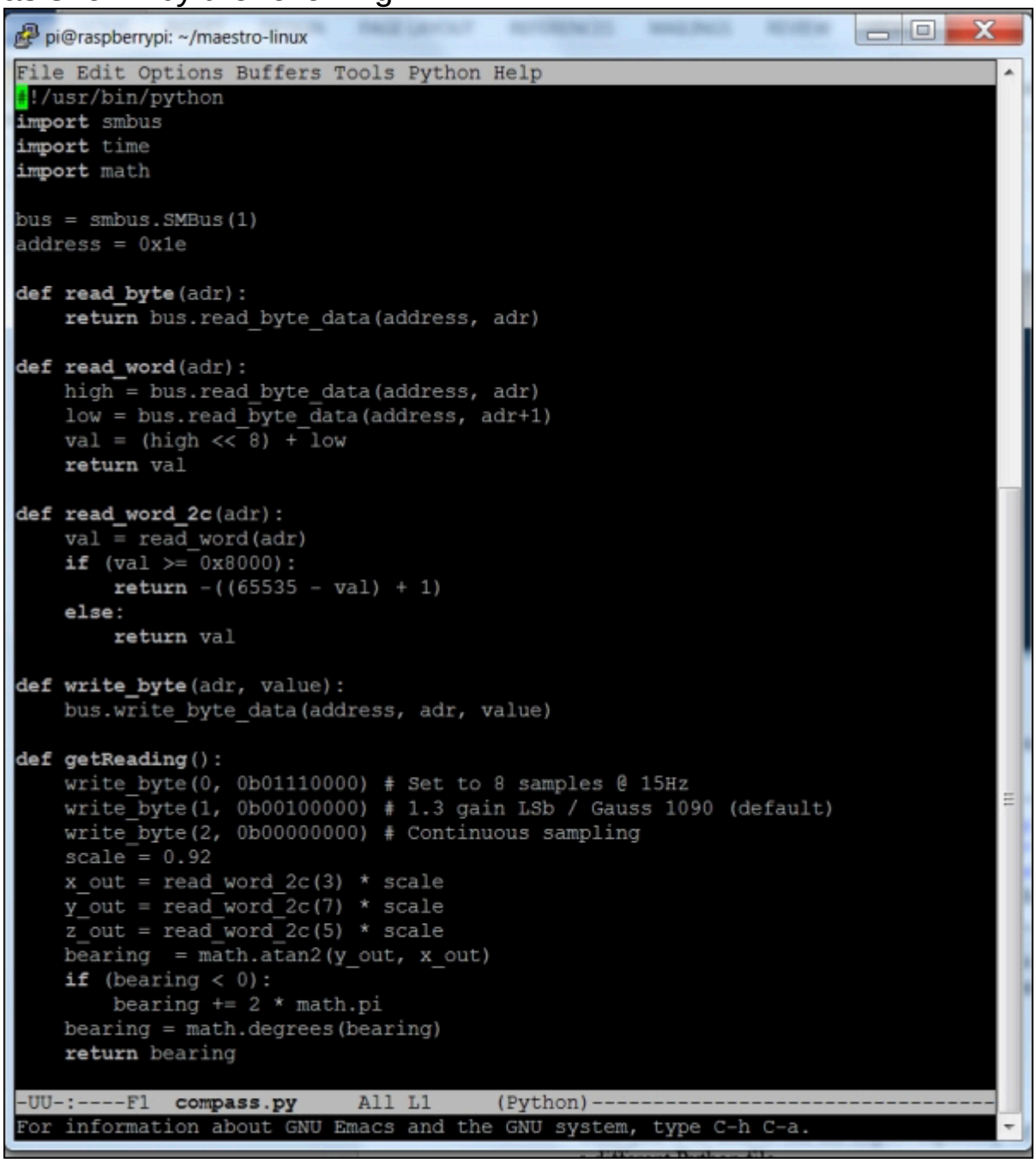
Now, you can add direction to your project! As you move the device around, you should see the **Heading** value change.

Note

This is a basic program; you can find out more about other features that are available with this library at http://think-bowl.com/raspberry-pi/i2c-python-library-3-axis-digital-compass-hmc5883l-with-the-raspberry-pi/.

One final step in developing your compass code is to make it a file where the functions can then be imported to a different Python program. To do this, edit the file so that all of the code is in functions,

as shown by the following:



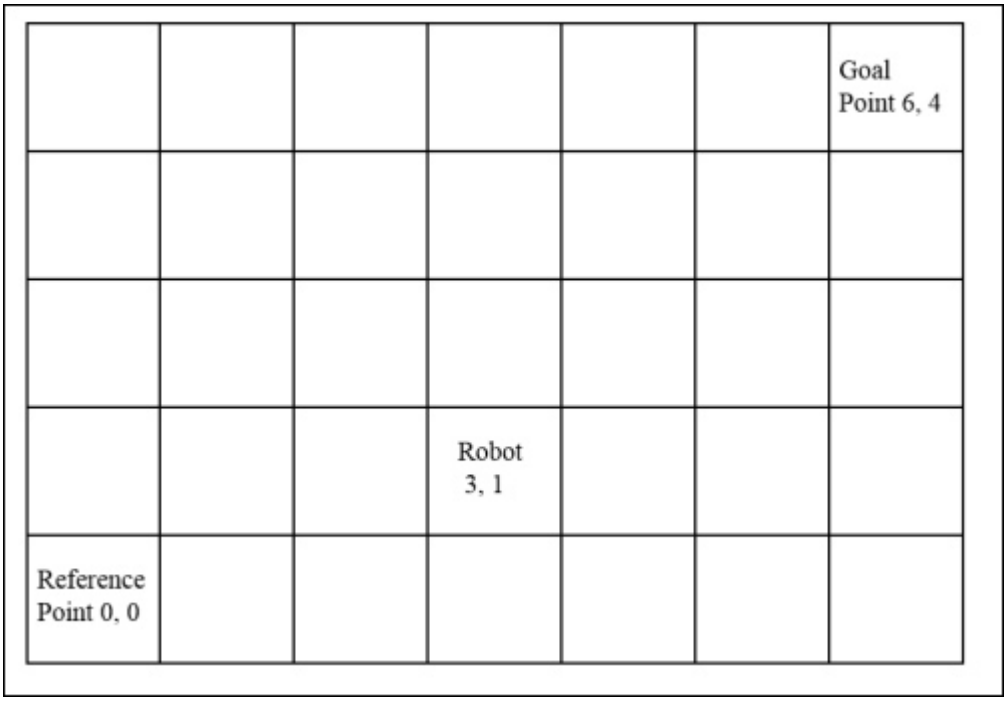
Then you'll be able to use the import capability of Python to import this functionality into a different Python file.

Dynamic path planning for your robot

Now that you can see barriers and also know direction, you'll want to do dynamic path planning. Dynamic path planning simply means that you don't have a knowledge of the entire world with all the possible barriers before you encounter them. Your robot will have to decide how to proceed while it is in the middle and actually moving. This can be a complex topic, but there are some basics that you can start to understand and apply as you ask your robot to move around its environment. Let's first address the problem of knowing where you want to go and needing to execute a path without barriers, and then adding in barriers.

Basic path planning

In order to talk about dynamic path planning, that is, planning a path where you don't know what barriers you might encounter, you'll need a framework to understand where your robot is as well as to determine the location of the goal. One common framework is an *x-y* grid. The following is a drawing of such a grid:



There are three key points:

- The lower left point is a fixed reference position. The directions *x* and *y* are also fixed, and all other positions will be measured with respect to this position and these directions.
- Another important point is the starting location of your robot. Your robot will then keep track of its location by using its *x* coordinate, or position itself with respect to some fixed reference position in the *x* direction, and its *y* coordinate, its position with respect to some fixed reference position in the *y* direction to the goal. It will use the compass to keep track of these directions.
- The third important point is the position of the goal, also given in *x* and *y* coordinates with respect to the fixed reference position. If you know the starting location and the starting angle of your robot ,you can plan an optimum (shortest distance) path to this goal. To do this, you can use the goal location and the robot location, and some fairly simple math to calculate the distance and angle from the robot to the goal.

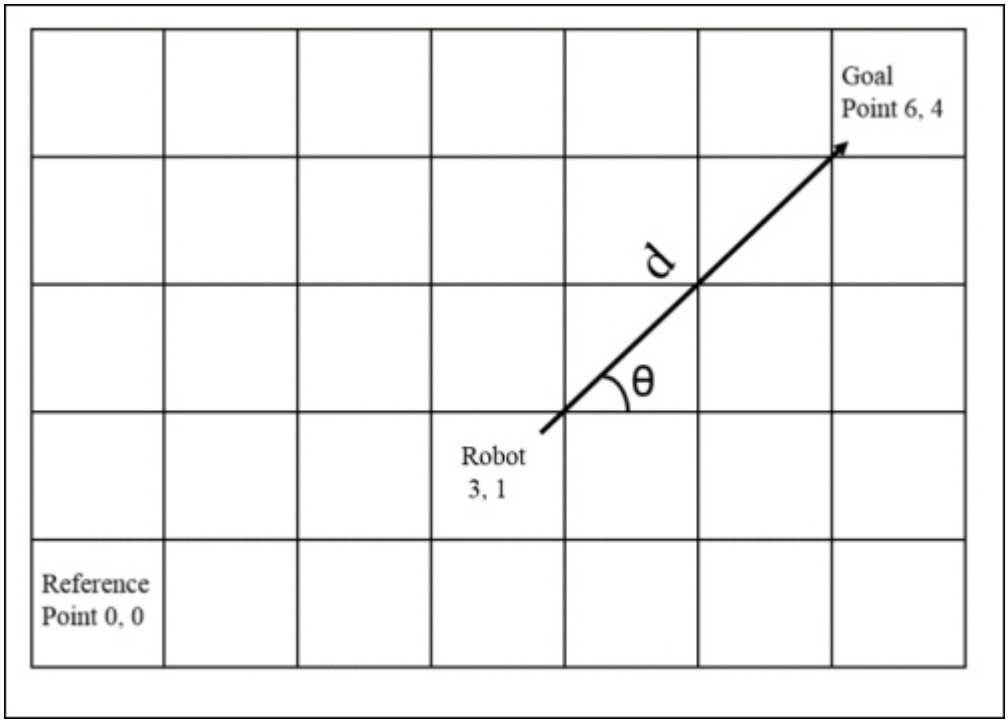
  To calculate the distance, use the following equation:

$$d = \sqrt{\left(\left(X\ goal - X\ goal \right)^2 + \left(Y\ goal - Y\ robot \right)^2 \right)}$$

Use the following equation to tell your robot how far to travel to the goal. A second equation will tell your robot the angle it needs to travel:

$$\theta = \arctan\left(\frac{Y\ goal - Y\ robot}{X\ goal - X\ goal} \right)$$

The following is a graphical representation of these two pieces of information:



Now that you have a goal angle and distance, you can program your robot to move. To do this, you will write a program to do path planning and call the movement functions that you created in Chapter 3, *Motion for the Biped*. You will, however, need to know the distance that your robot travels in a step so that you can tell your robot how far to travel in steps, not distance units.

You'll also need to be able to translate the distance that might be covered by your robot in a turn; however, this distance may be too small to be of any importance. If you then know the angle and the distance, you can move your robot to the goal.

The following are the steps you will program:

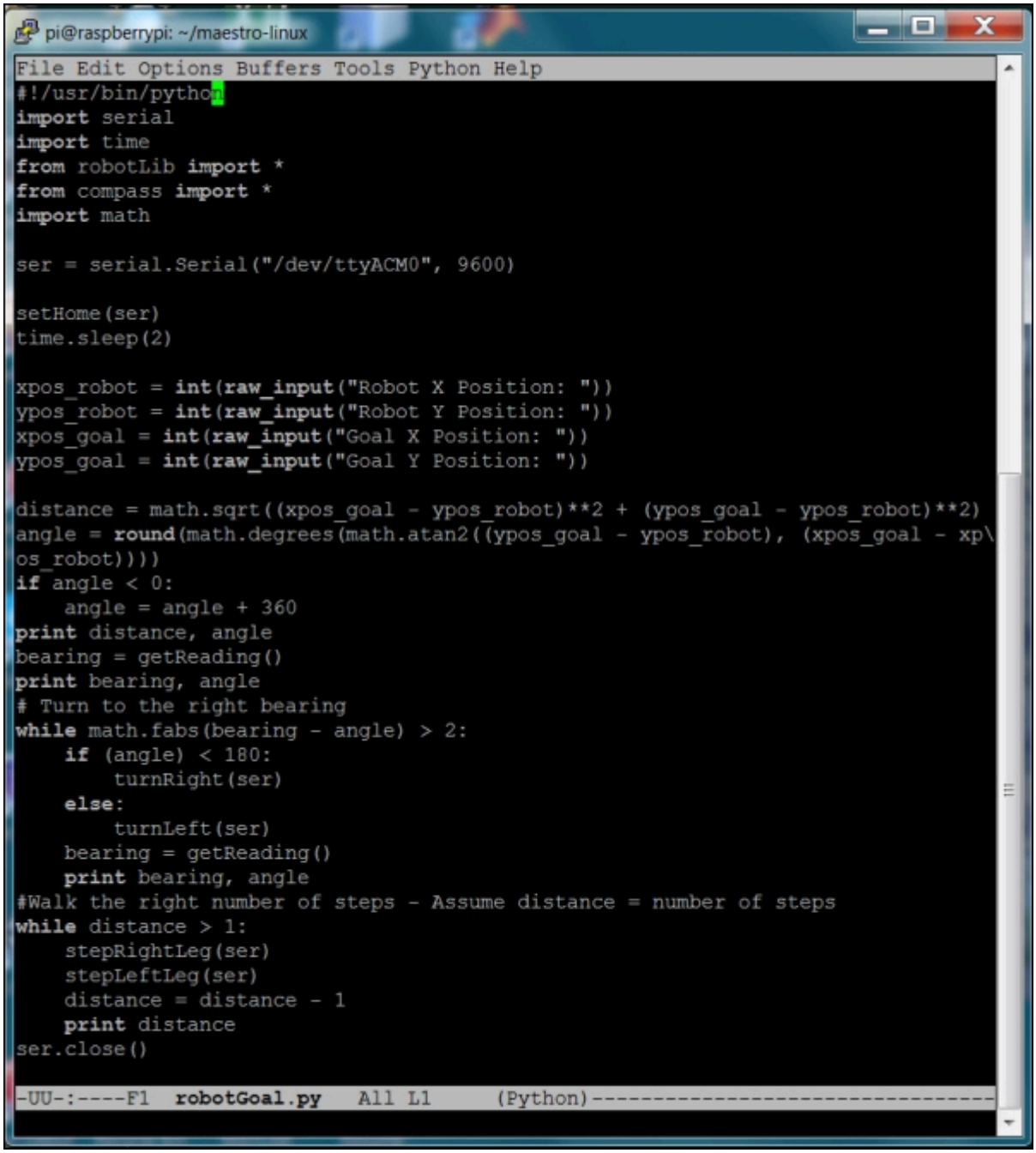
1. Calculate the distance in units that your robot will need to travel in order to reach the goal. Convert this to number of steps to achieve this distance.
2. Calculate the angle that your robot will need to travel to reach the goal. You'll use the compass and your robot turn functions in order to achieve this angle.
3. Now, call the step functions the proper number of times required to move your robot the correct distance.

That's it. Now, we will use some very simple Python code that executes this by using functions to move the robot forward and turn the robot. In this case, it makes sense to create a file called `robotLib.py` with all of the functions that do the actual servo settings to step the biped robot forward and turn the robot. You'll then import these functions using the `from robotLib import *` statement, and your Python program can call these functions. This makes the path planning Python program much smaller and more manageable. You'll do the same thing with the `compass` program, using the `from compass import *` command.

Note

For more information on how to import the functions from one Python file to another, refer to http://www.tutorialspoint.com/python/python_modules.htm.

The following is a listing of the program:

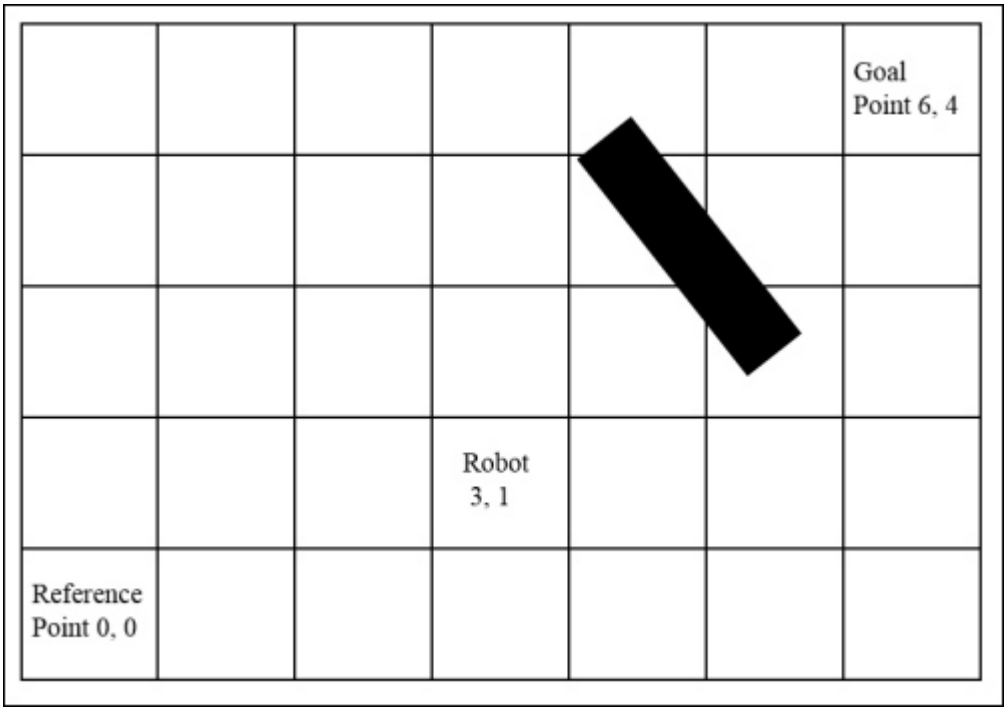


In this program, the user enters the goal location and the robot first decides the shortest direction to the desired angle by reading the angle. To make it simple, the robot is placed in the grid with it heading in the direction of angle 0. If the goal angle is less than 180 degrees, the robot will turn right. If it is greater than 180 degrees, the robot will turn left. The robot turns until the desired angle and its

measured angle are within a few degrees. Then, the robot takes the number of steps to reach the goal.

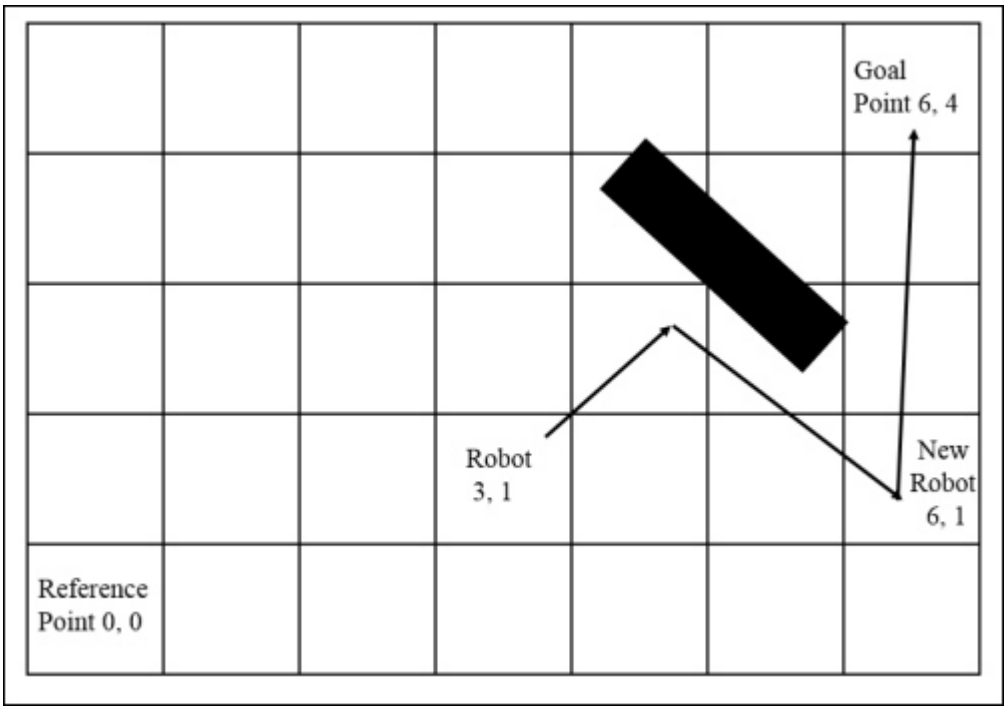
Avoiding obstacles

Planning paths without obstacles, as has been shown, is quite easy. However, it becomes a bit more challenging when your robot needs to walk around obstacles. Let's look at an instance where there is an obstacle in the path that you calculated previously. It might look like the following:

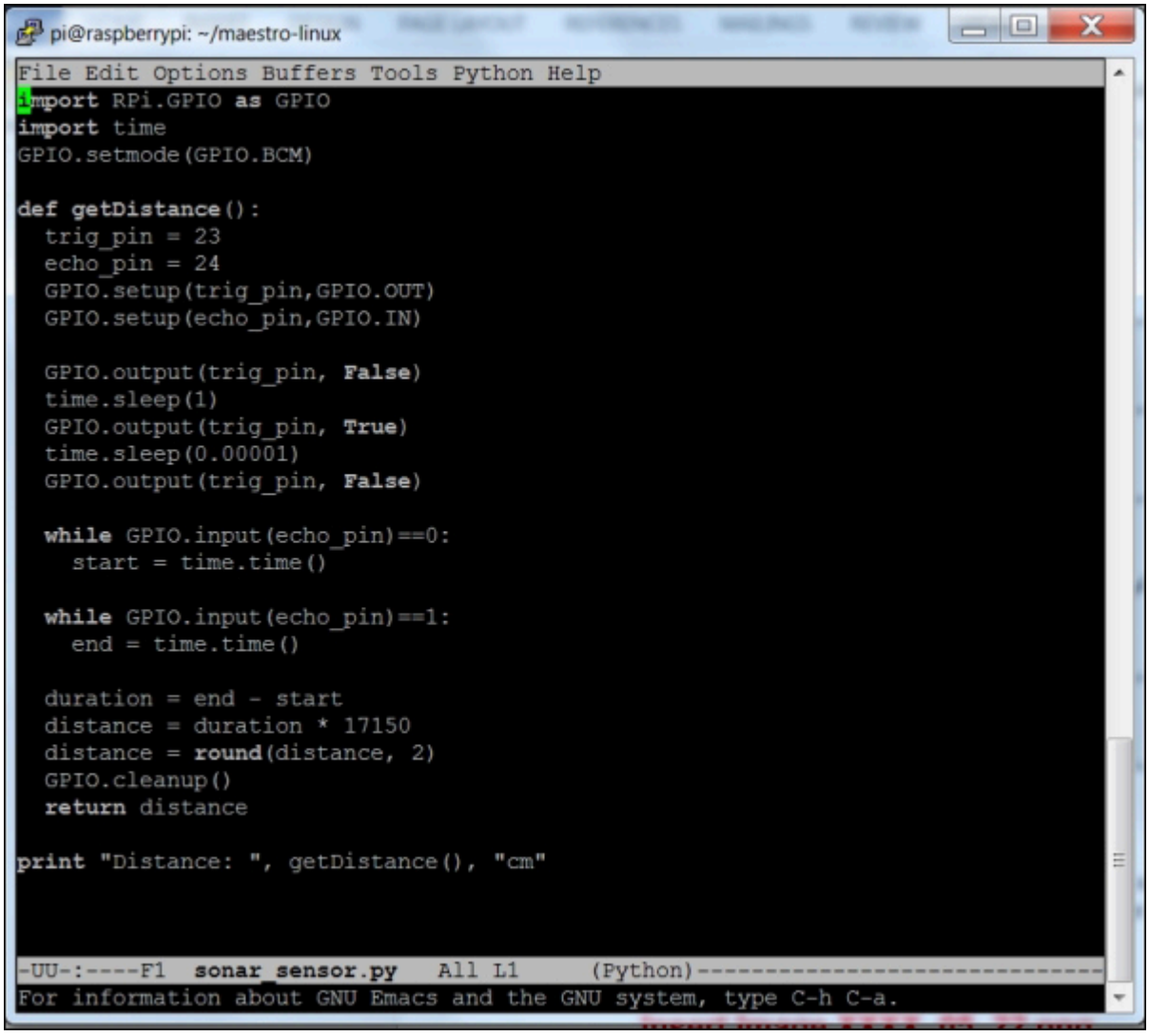


You can still use the same path planning algorithm to find the starting angle; however, you'll now need to use your sonar sensor to detect the obstacle. When your sonar sensor detects the obstacle, you'll need to stop and recalculate a path to avoid the barrier, and then recalculate the desired path to the goal. One very simple way to do this, when your robot senses a barrier, is to turn right 90 degrees, go a fixed distance, and then recalculate the optimum path. When you turn back to move toward the target, if you sense no barrier, you will be able to move along the optimum path.

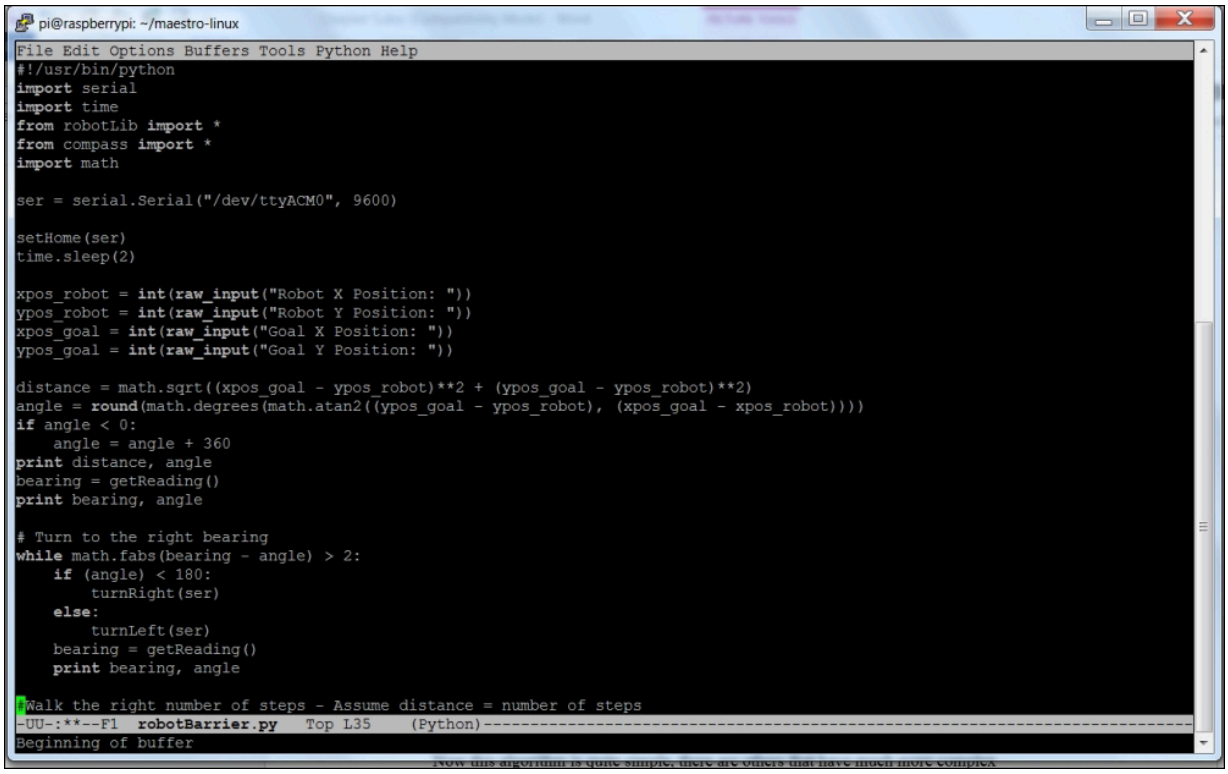
However, if your robot encounters the obstacle again, it will repeat the process, until it reaches the goal. In this case, using these rules, the robot will travel the following path:



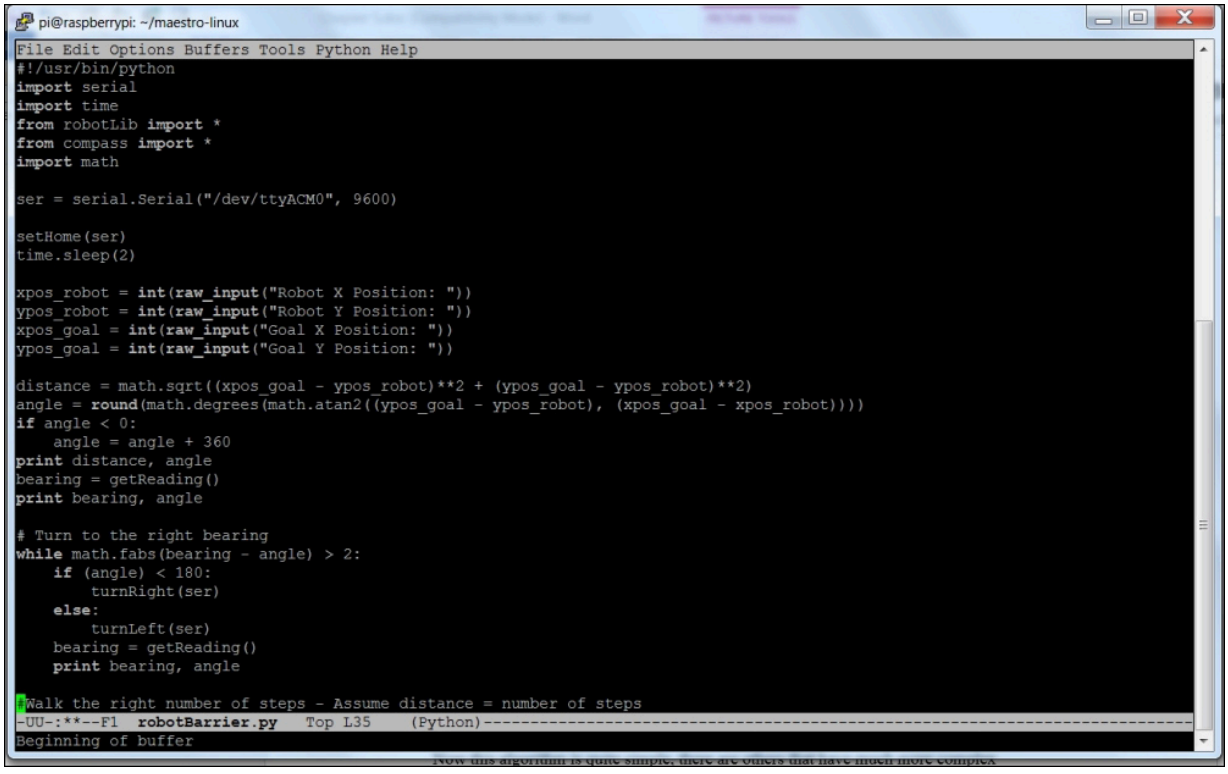
There is one more step you'll need to take before adding the sonar sensor's capability to your robot. You'll need to change the sonar sensor code so that it can be added to the Python code as a library. The following is that code:



You'll also import this code using the `from compass import *` statement. You'll also be using the time library and the `time.sleep` command to add delay between different statements in the code. And the following is the first part of the code that uses all of this to detect the barrier, turn to the right, then first part of the Python code that utilizes the sonar sensor:



And the following is the final piece of the code:



Now, this algorithm is quite simple; there are others that have much more complex responses to barriers. You can also see that by adding sonar sensors to the sides your robot could actually sense when the barrier has ended. You could also provide more complex decision processes about which way to turn to avoid an object. Again, there are many different path finding algorithms. See http://www.academia.edu/837604/A_Simple_Local_Path_Planning_Algorithm_for_Autonomous_Mobile_Robots for an example of this. These more complex algorithms can be explored by using the basic functionality that you have built in this chapter.

Summary

You've now added path planning to your robot's capability. Your robot can now not only move from point A to point B, but can also avoid barriers that might be in the way. In the next chapter, you'll learn how to add a webcam to your biped. This will introduce a whole new set of ways for your robot to experience the world around it.

Chapter 6. Adding Vision to Your Biped

Now that your biped is up and mobile, is able to find barriers, and knows how to plan its path, you can now start to have it move around autonomously. However, you may want your robot to follow a color or motion.

In this chapter, you will be learning:

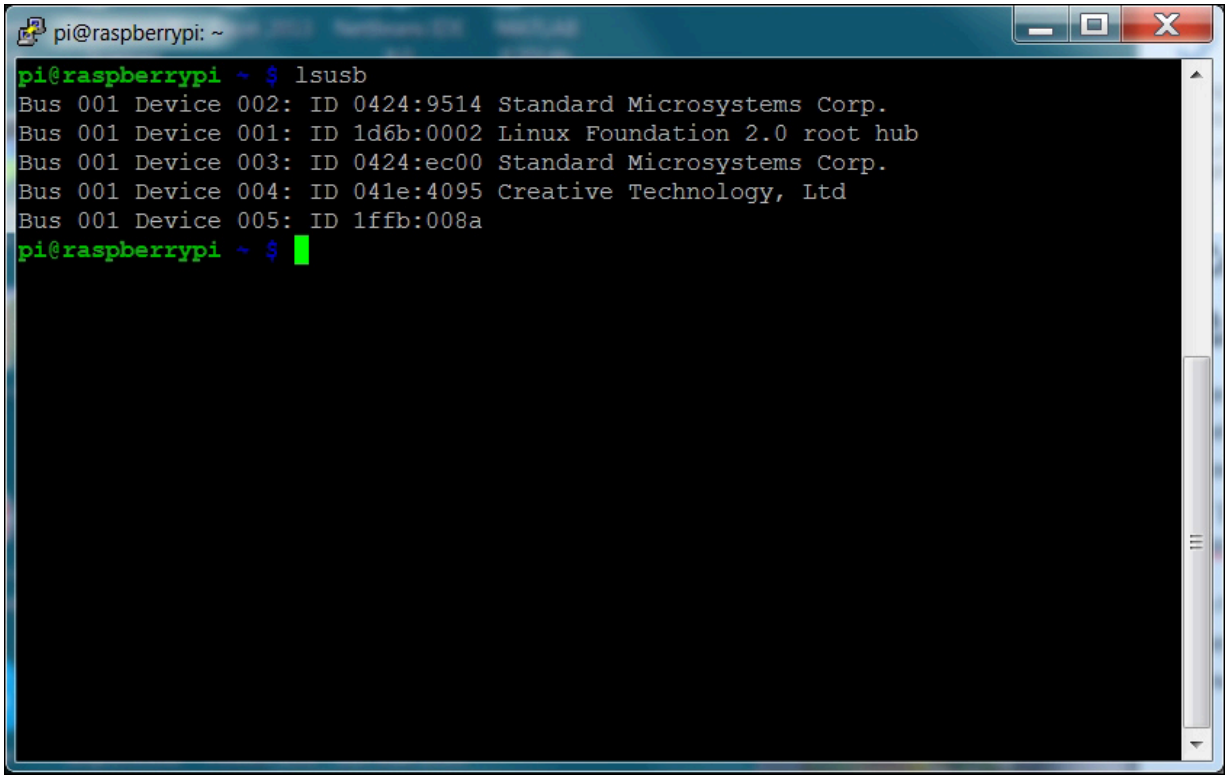
- How to add a webcam to your biped robot
- How to add RaspiCam to your biped robot
- How to install and use OpenCV, an open source vision package
- How to follow motion with your biped robot

Installing a camera on your biped robot

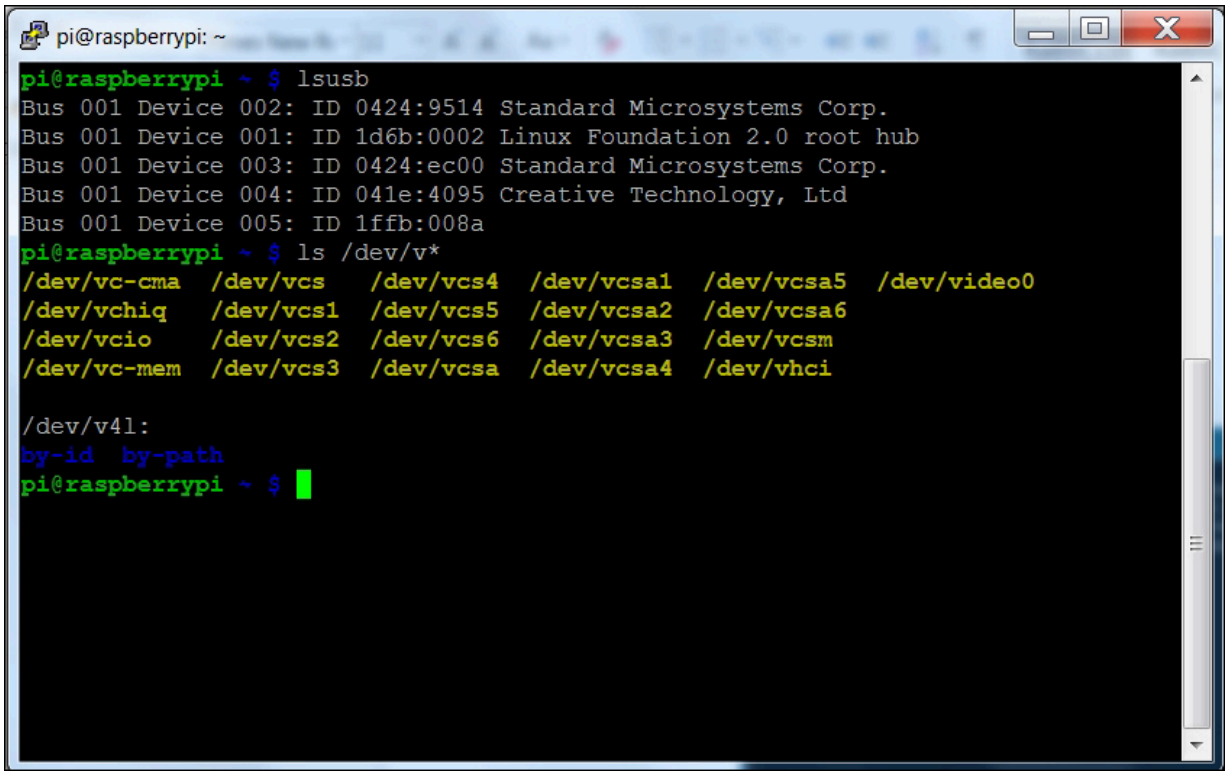
Having vision capability is a real advantage for your biped robot; you'll use this functionality in lots of different applications. Fortunately, adding hardware and software for vision is both easy and inexpensive. There are two choices as far as vision hardware is concerned. You can add a USB webcam to your system, or you can add RaspiCam, a camera designed specifically for Raspberry Pi.

Installing a USB camera on Raspberry Pi

Connecting a USB camera is very easy. Just plug it into the USB slot. To make sure that your device is connected, type `lsusb`. You should see the following:

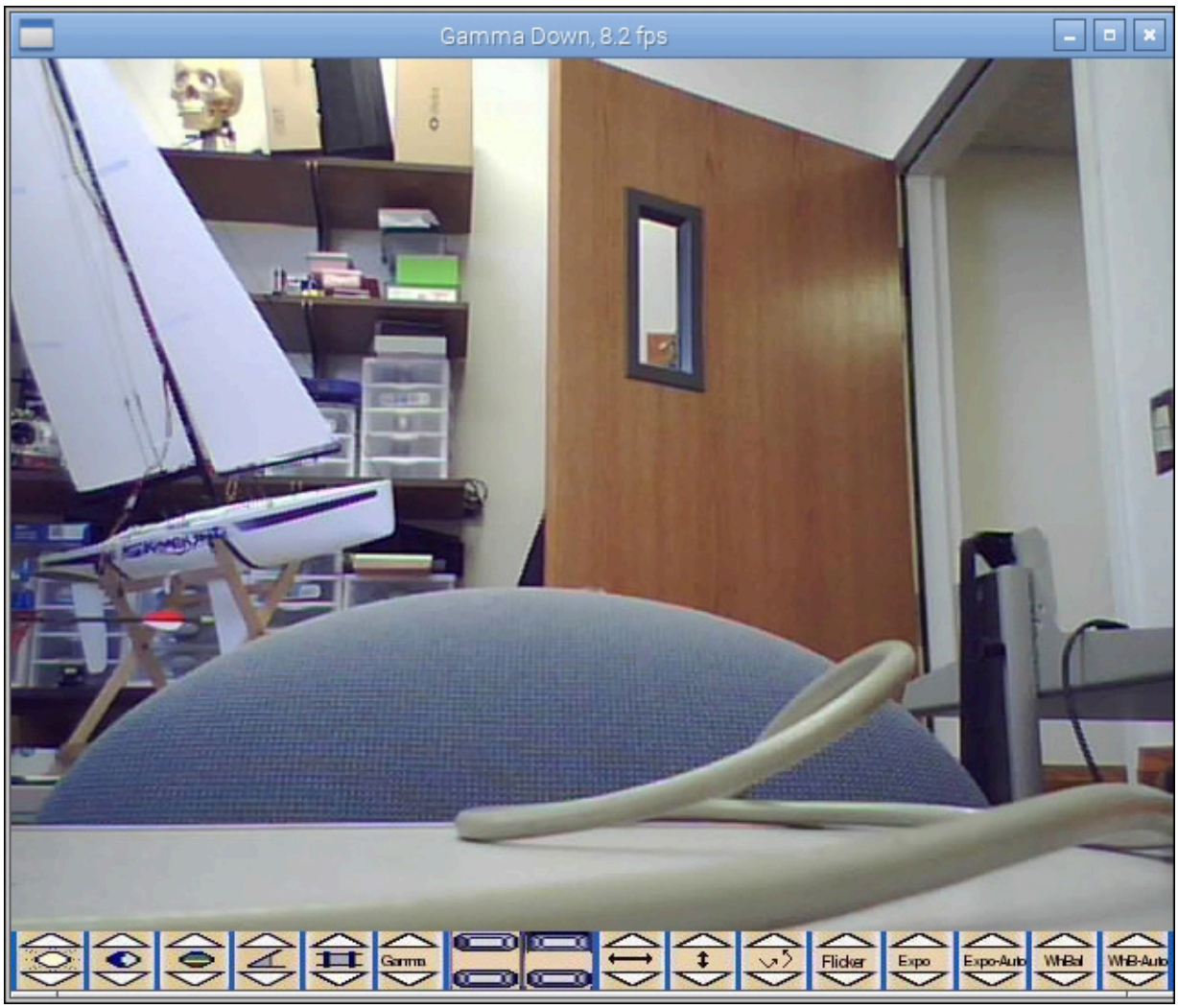


This shows a Creative Webcam located at Bus 001 Device 004: ID 041e:4095. To make sure that the system sees this as a video device, type `ls /dev/v*` command and you should see something like the following:



The `/dev/video0` is the webcam device. Now that your device is connected, let's actually see if you can capture images and video. There are several tools that can allow you to access the webcam, but a simple program with video controls is called luvcview. To install this, type `sudo apt-get install luvcview`. Once the application is installed, you'll want to run it. To do this, you'll either need to be connected directly to a display or able to access Raspberry Pi via a remote VNC connection, such as vncserver, as displaying images will require a graphical interface.

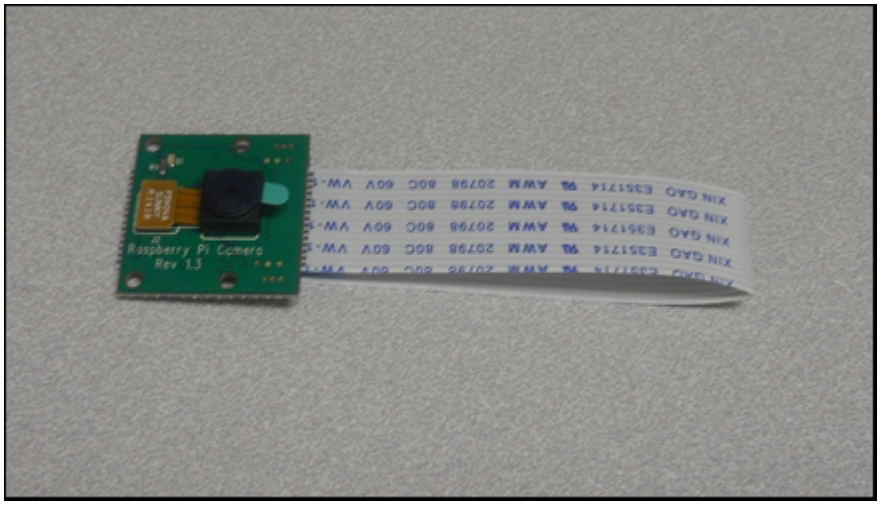
Once you are connected in this manner, open a terminal window on Raspberry Pi and run `luvcview`. You should see something like the following:



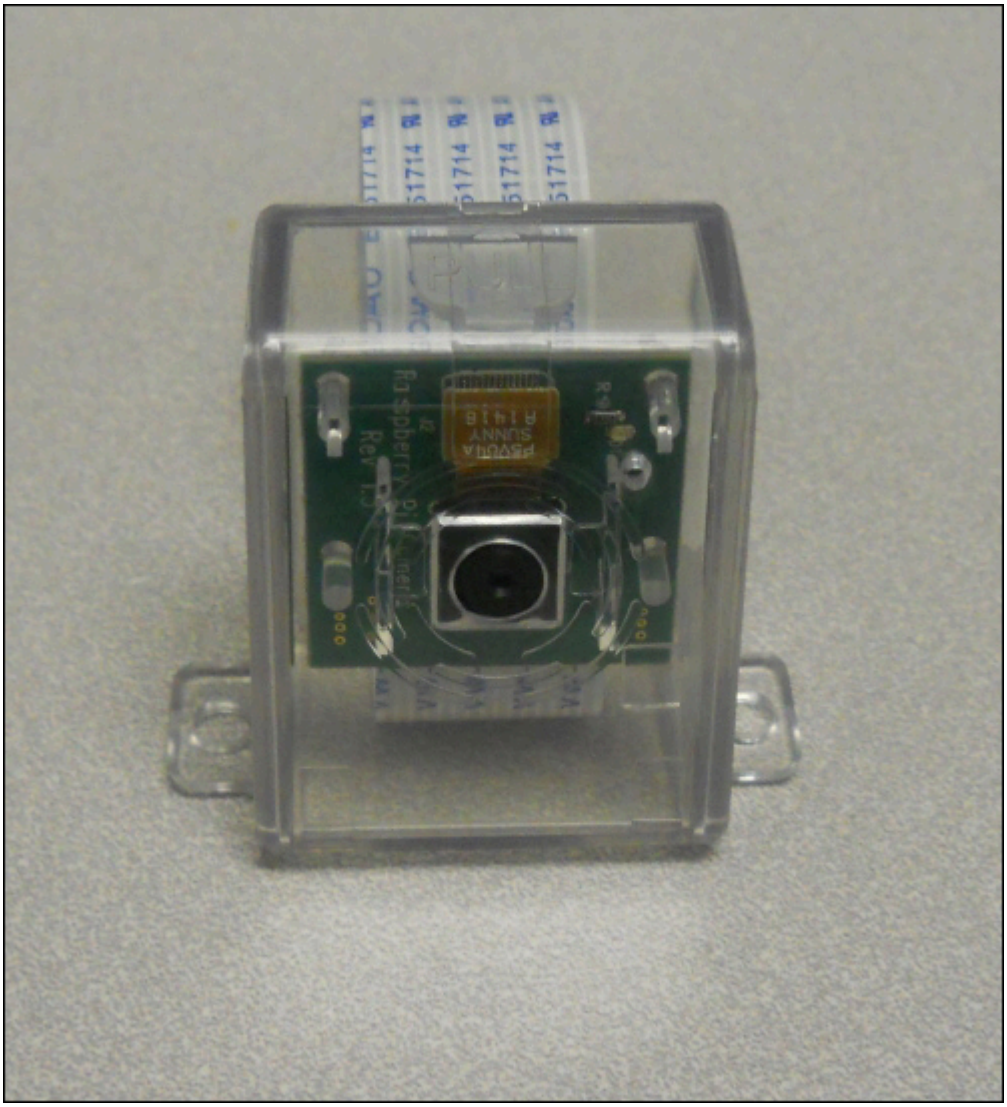
Don't worry about the quality of the image, you'll be capturing and processing your images inside of OpenCV, a vision framework.

Installing RaspiCam on Raspberry Pi

The other choice for seeing the outside world on Raspberry Pi is to use the RaspiCam. Installing this camera is a bit more involved; you are going to connect it to a special connector on the Raspberry Pi. The following is a picture of the camera with its special connector:

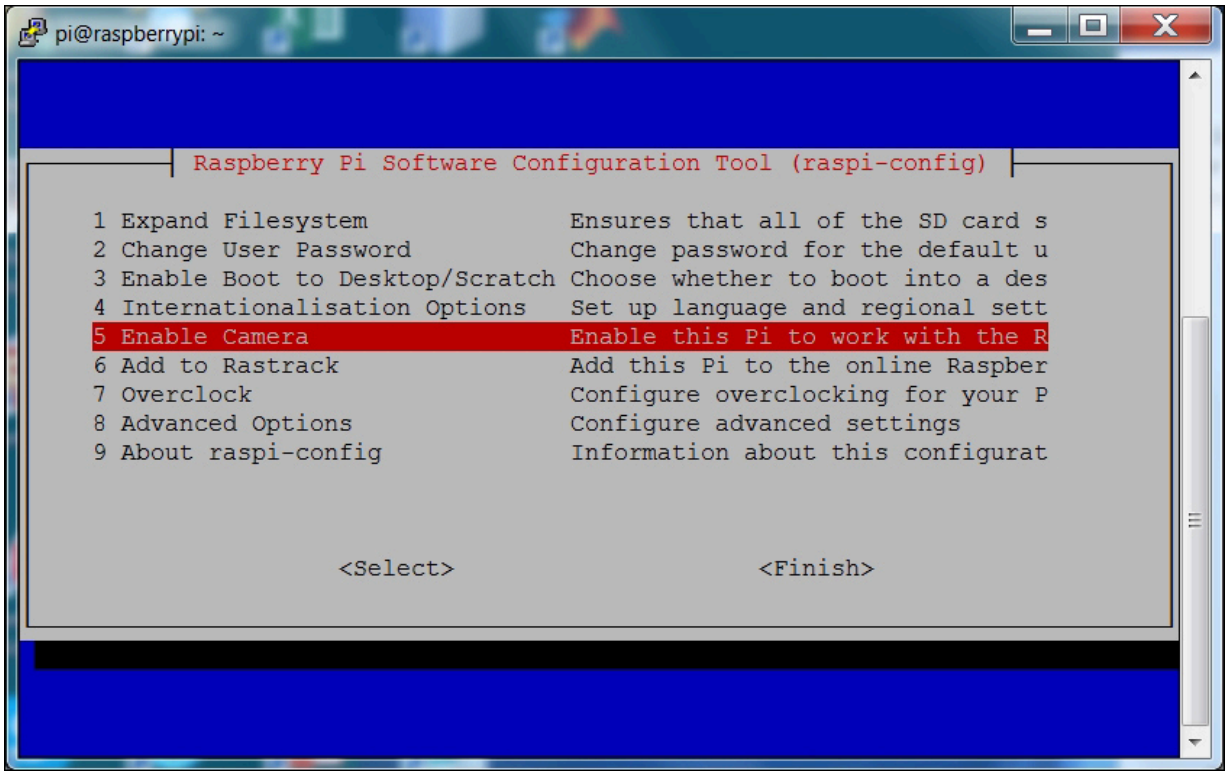


You may also want to add the protective cover for the camera; assembling it looks like the following:

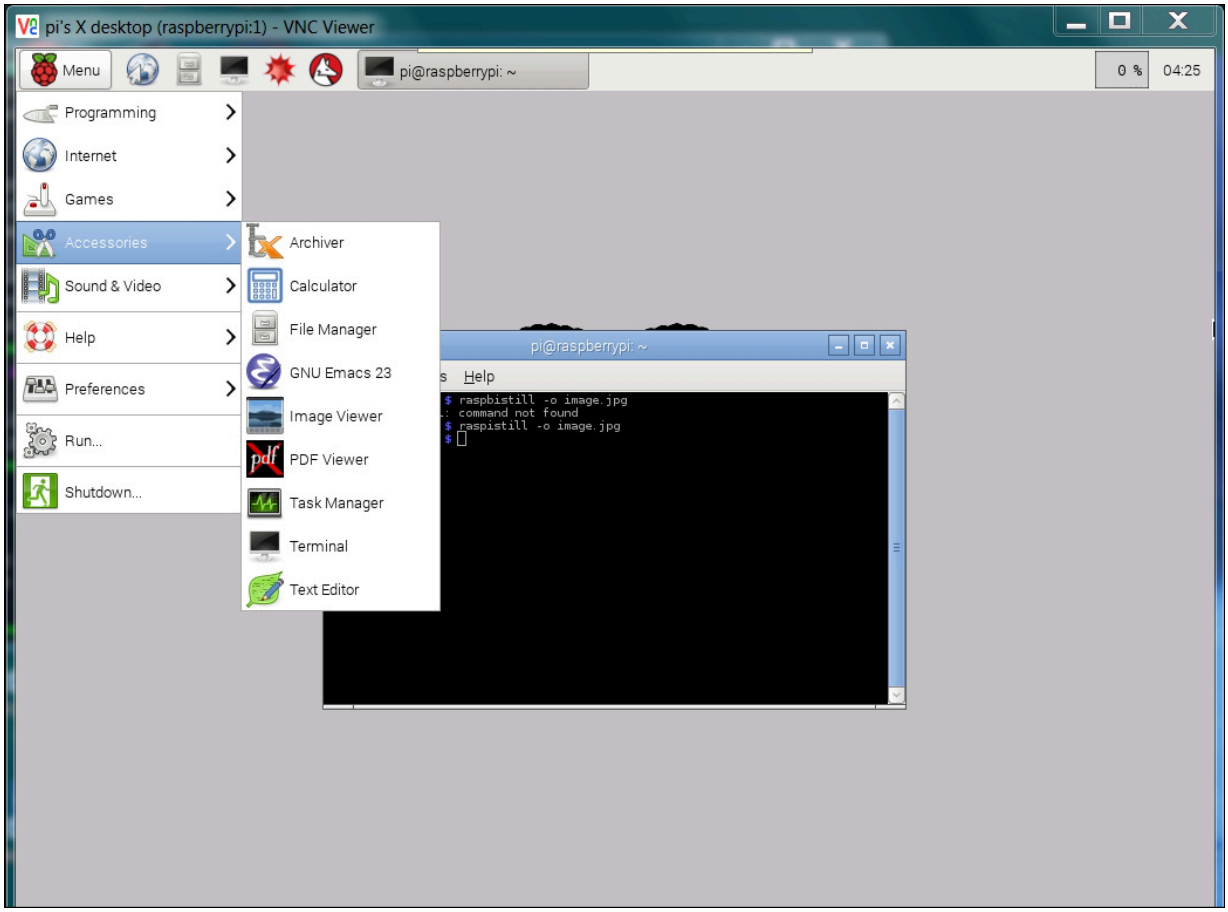


Now you are ready to connect the camera to Raspberry Pi. The camera connects to the Raspberry Pi by installing it into the connector marked Camera on the Raspberry Pi. To see how this is done, see the video at http://www.raspberrypi.org/help/camera-module-setup/.

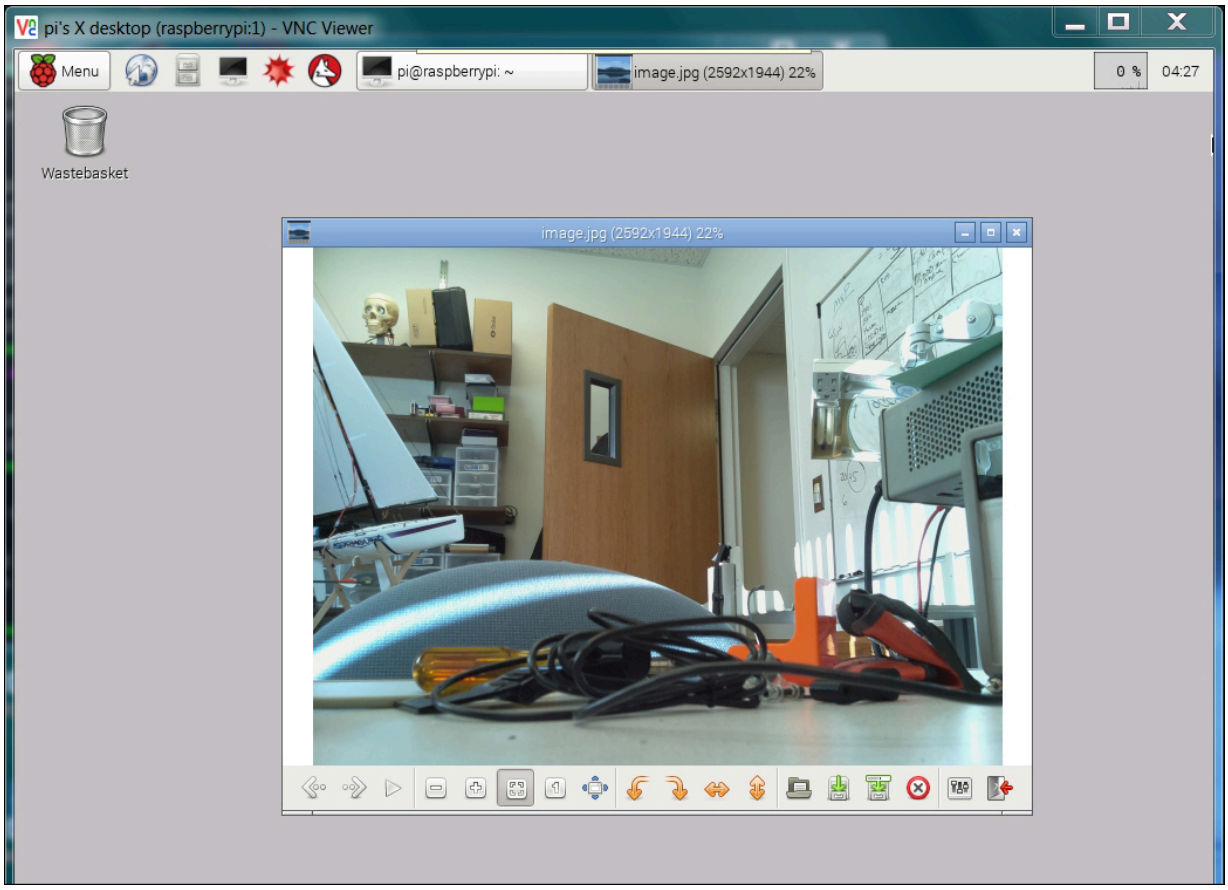
Now that the camera is connected, you'll want to enable the camera using the `raspi-config` utility. Type `sudo raspi-config`, then select the **Enable Camera**, as shown in the following screenshot:



Now reboot Raspberry Pi. If you are developing from a remote computer and want to see your images, you will want to open a vncserver connection between your computer and the Raspberry Pi. For details, see Chapter 1, *Configuring and Programming Raspberry Pi*. To take a picture with the camera, simply type `raspistill -o image.jpg`. This will take a picture with the camera, and then store the image in the `image.jpg` file. Once you have the picture, you can view it by opening the Raspberry Pi image viewer by selecting the lower left icon for **Menu**, then **Accessories**, and then **Image Viewer**, as shown in the following screenshot:



Open the **image.jpg** file, and you should see the results of your picture:



Before you can access OpenCV with the Raspberry Pi camera, you'll need to do two things. First, you'll need to add a Python library; it is called `picamera`. To get this, and the required libraries, type `sudo apt-get install python-picamera python3-picamera python-rpi.gpio`. Second, you'll need to type `sudo modprobe bcm2835-v4l2`. The Raspberry Pi camera can now be used in the OpenCV examples in the next section.

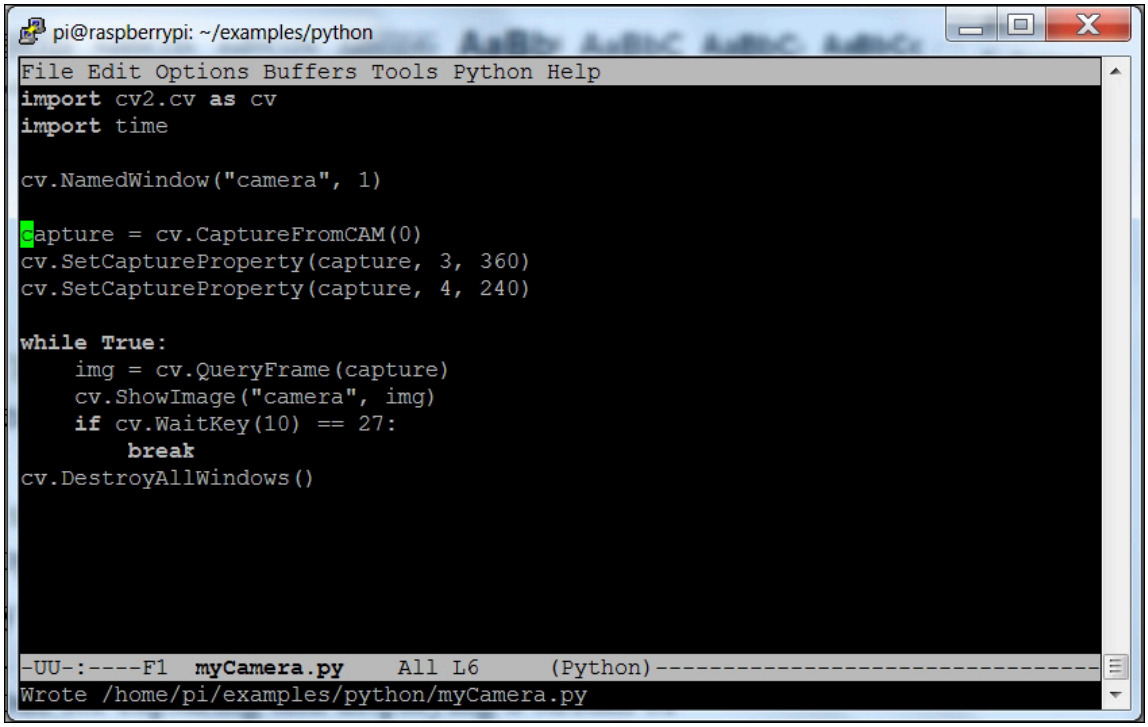
Downloading and installing OpenCV – a fully featured vision library

Now that you have your camera connected, you can begin to access some amazing capabilities that have been provided by the open source community. Open a terminal window and type the following commands:

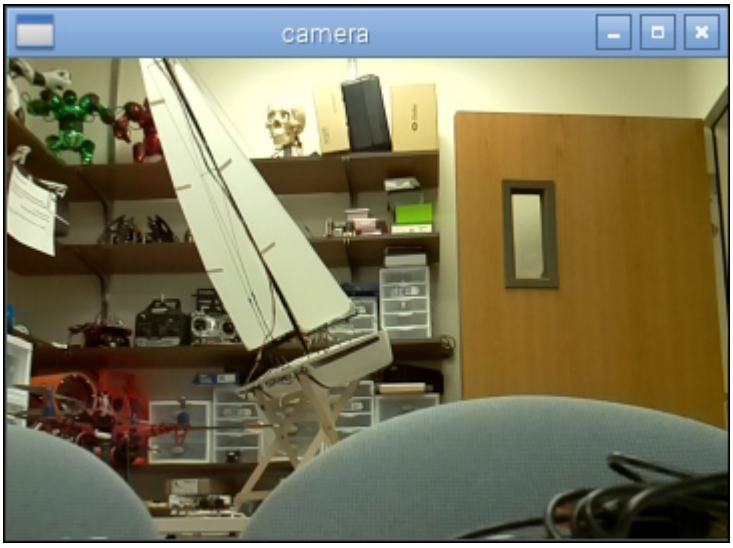
1. `sudo apt-get update`: You're going to download a number of new software packages, so it is good to make sure that everything is up to date.
2. `sudo apt-get install build-essential`: Although you may have done this earlier, this library is essential to build OpenCV.
3. `sudo apt-get install libavformat-dev`: This library provides a way to code and decode audio and video streams.
4. `sudo apt-get install ffmpeg`: This library provides a way to transcode audio and video streams.
5. `sudo apt-get install libcv2.4 libcvaux2.4 libhighgui2.4`: This command shows the basic OpenCV libraries. Note the number in the command. This will almost certainly change as new versions of OpenCV become available. If 2.4 does not work, either try 3.0 or search on Google for the latest version of OpenCV.
6. `sudo apt-get install python-opencv`: This is the Python development kit needed for OpenCV, as you are going to use Python.
7. `sudo apt-get install opencv-doc`: This command will show the documentation for OpenCV just in case you need it.
8. `sudo apt-get install libcv-dev`: This command shows the header file and static libraries to compile OpenCV.
9. `sudo apt-get install libcvaux-dev`: This command shows more development tools for compiling OpenCV.

10. `sudo apt-get install libhighgui-dev`: This is another package that provides header files and static libraries to compile OpenCV.
11. Now type `cp -r /usr/share/doc/opencv-doc/examples /home/pi/`. This will copy all the examples to your home directory.

    Now that OpenCV is installed, you can try one of the examples. Go to the `/home/pi/examples/python` directory. If you do an `ls`, you'll see a file named `camera.py`. This file has the most basic code for capturing and displaying a stream of picture images. Before you run the code, make a copy of it using `cp camera.py myCamera.py`. Then, edit the file to look like the following:



    The two lines that you'll add are the two with the `cv.SetCaptureProperty`; they will set the resolution of the image to 360 by 240. To run this program, you'll need to either have a display and keyboard connected to Raspberry Pi or use vncviewer. When you run the code, you should see the window displayed, as shown in the following image:

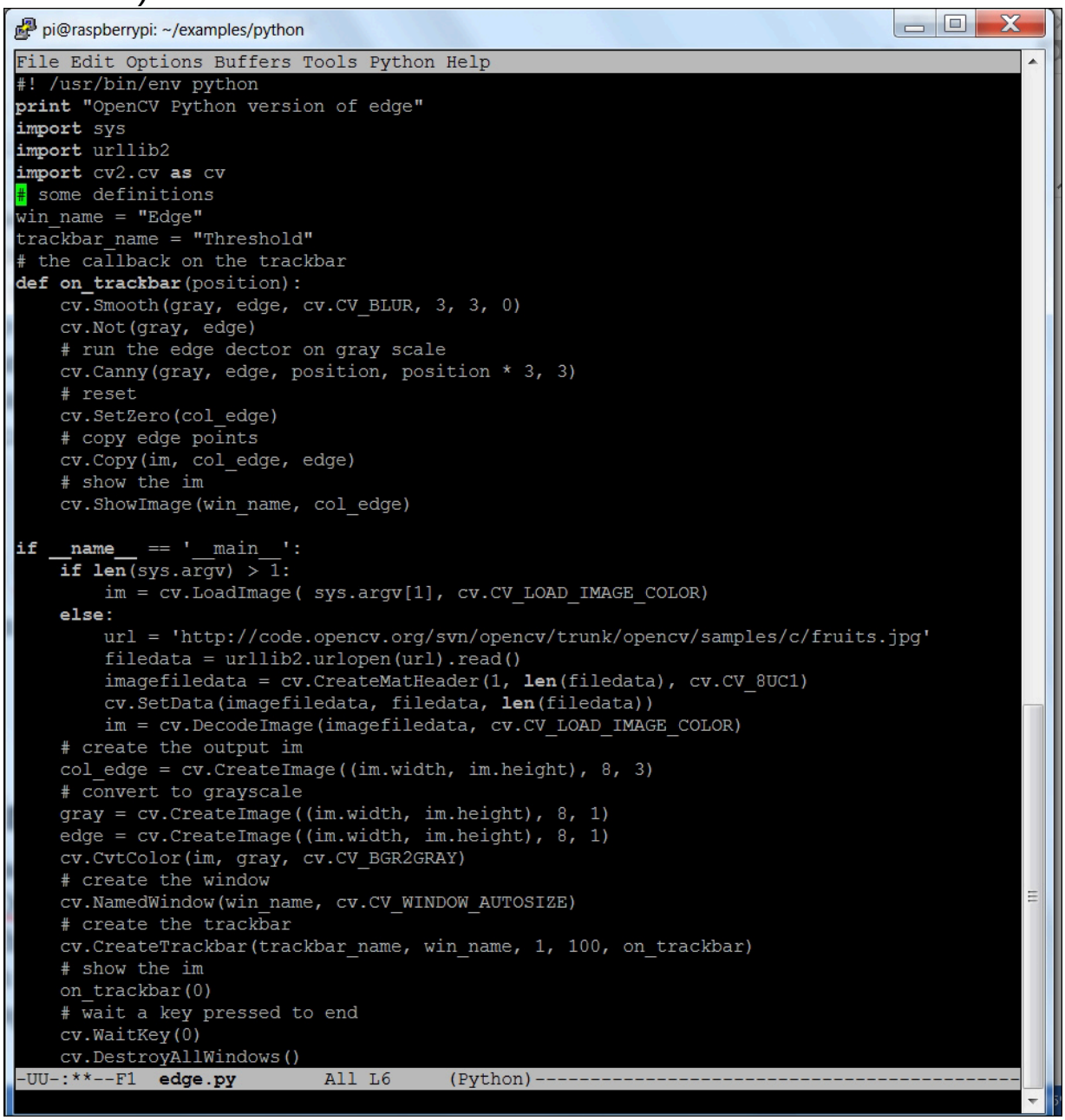


If you are using RaspiCam and don't see an image, you will need to run the `sudo modprobe bcm2835-v4l2` command. Now you can see the outside world!

You may want to play with the resolution to find the optimum settings for your application. Bigger images are great—they give you a more detailed view on the world—but they also take up significantly more processing power. You'll play with this more as you actually ask your system to do some real image processing. Be careful if you are going to use vncserver to understand your system performance, as this will significantly slow down the update rate. An image that is twice the size (width/height) will involve four times more processing. You can now use this capability to do a number of impressive tasks.

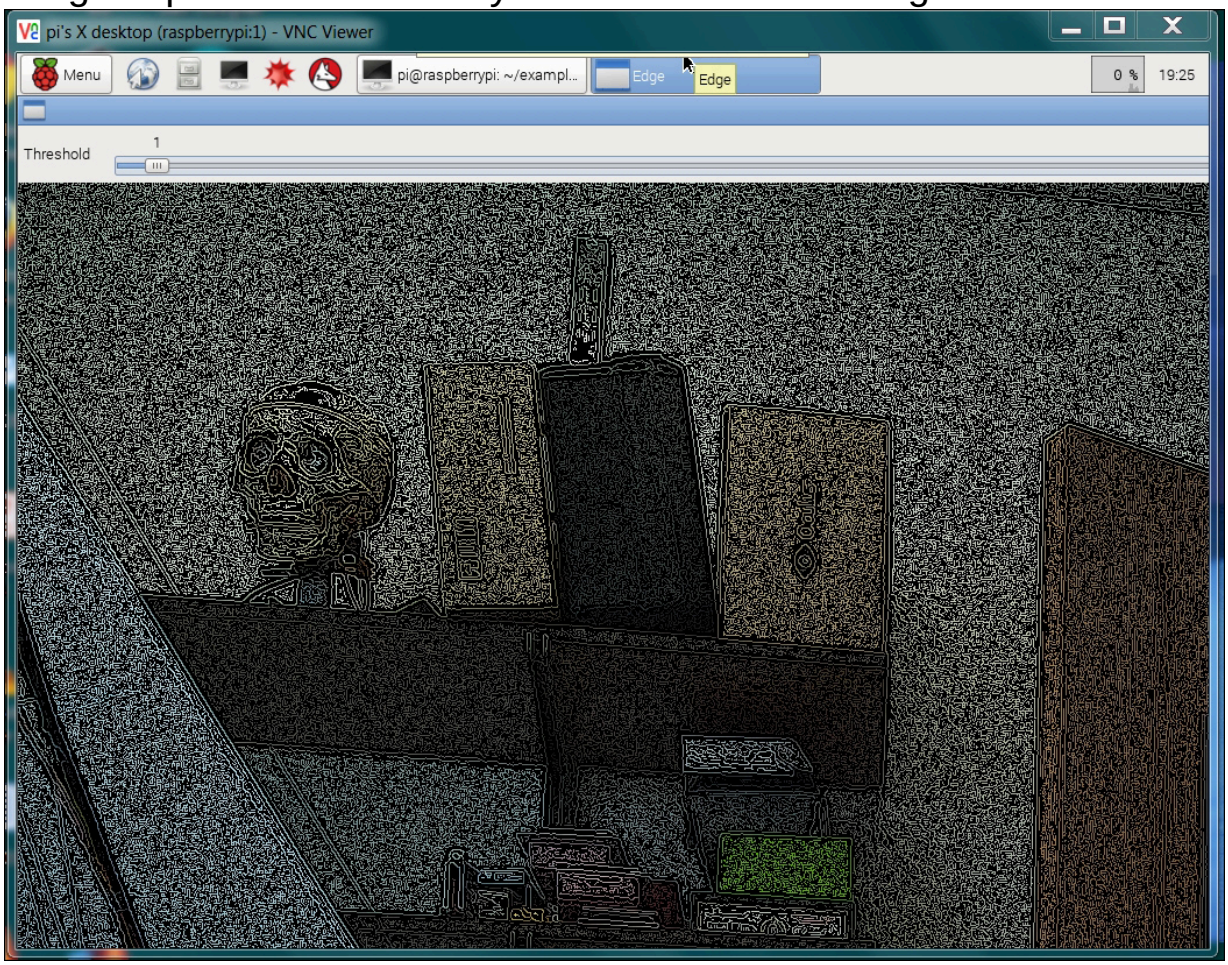
Edge Detection and OpenCv

Fortunately, one of the examples in the OpenCV Python set is a program named `edge.py`. The following is that file (with blank lines removed):



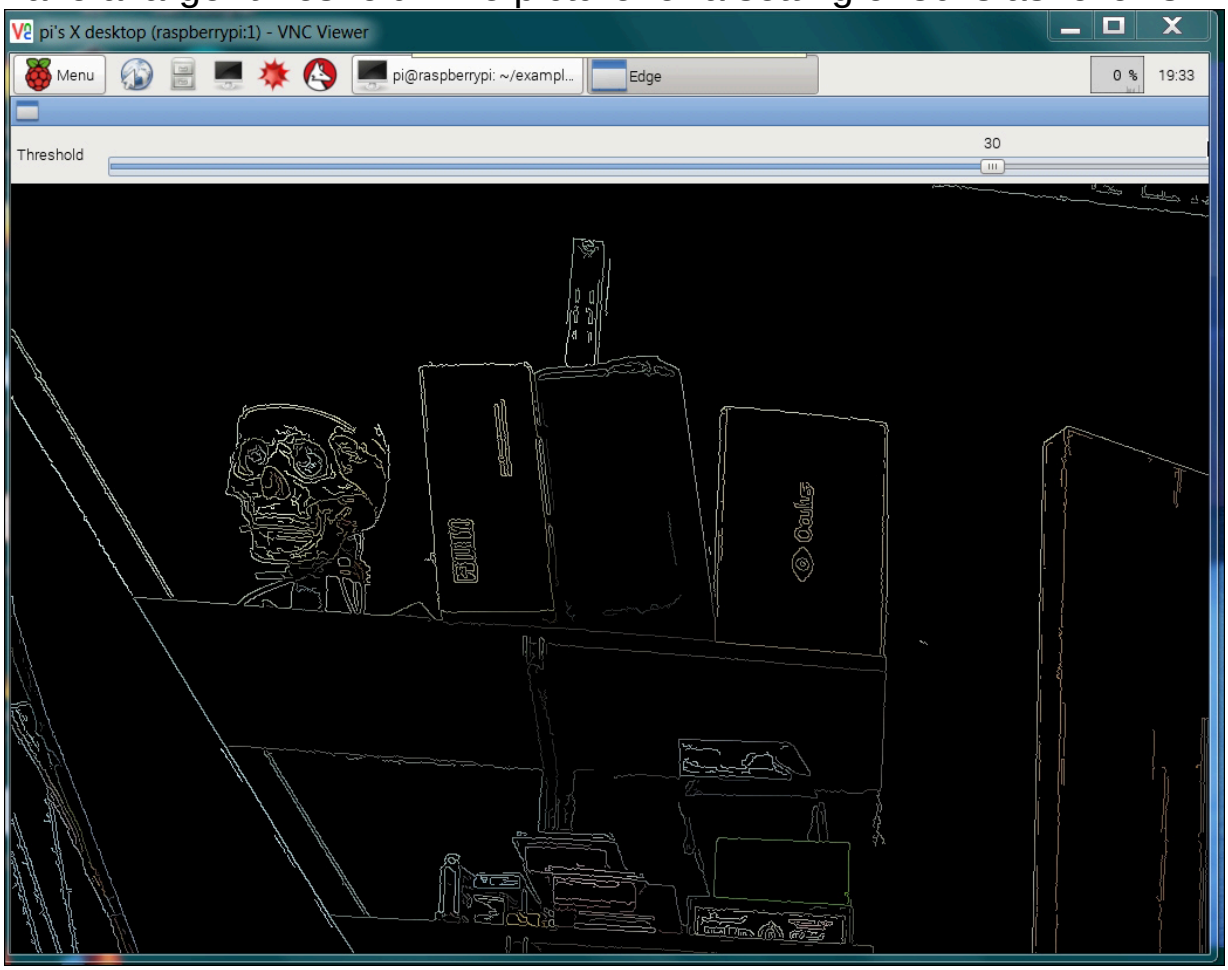
This program uses the Canny image detection algorithm implemented by OpenCV to find the edges in any image. For more

on the Canny edge algorithm, refer to http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html or http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html. You captured an image earlier; you can use this program to look at the edges and to also see how setting a different threshold can show more/less edges. Run the program with the image captured earlier and you will see the following:



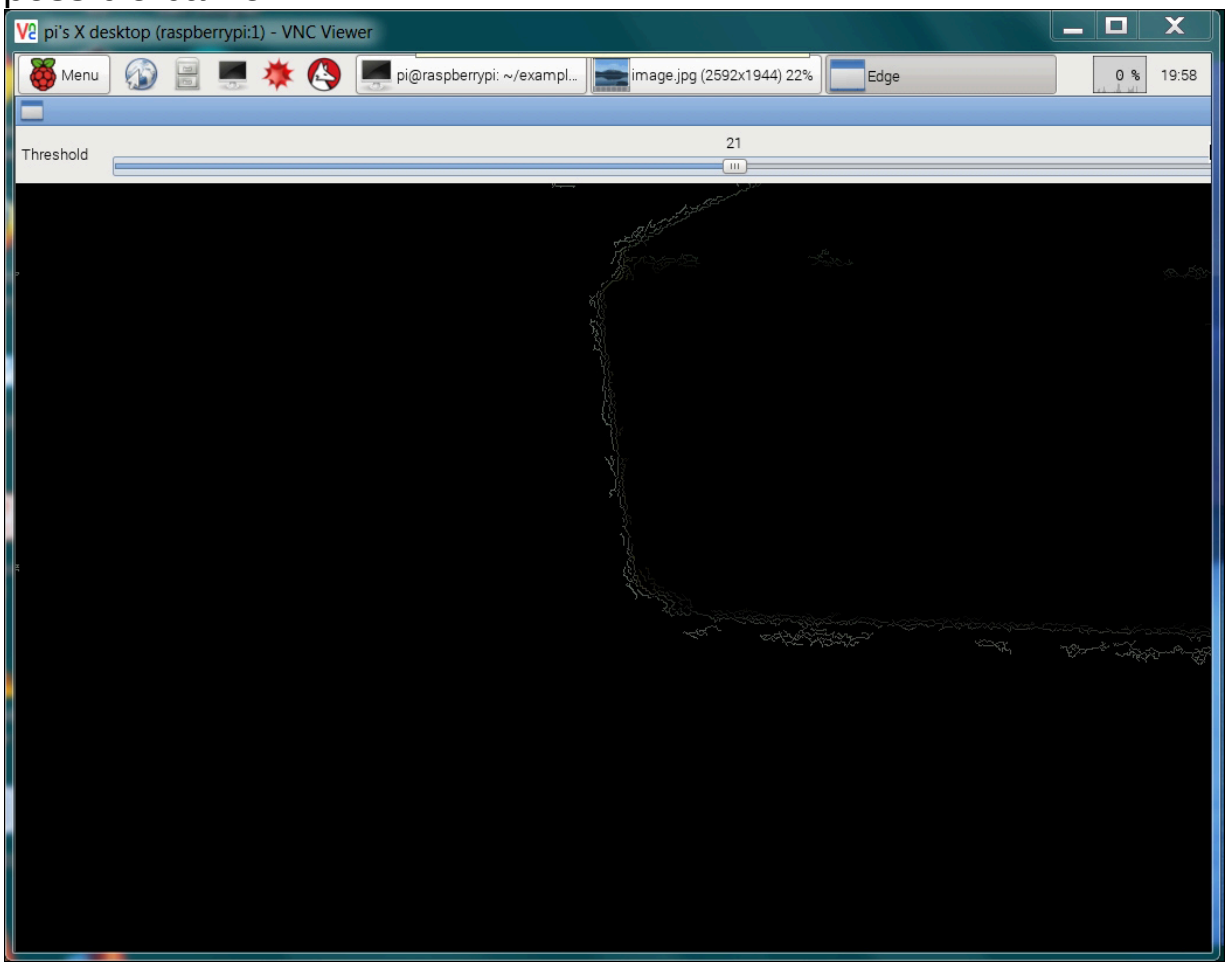
You will notice that there is a threshold slide bar setting at the top. If you adjust this threshold up, it will find fewer edges—the edges that

have a larger threshold. The picture for a setting of 30 is as follows:



Now you can see how this process could be translated to an image of a blank floor and a barrier. The following is such an image with a

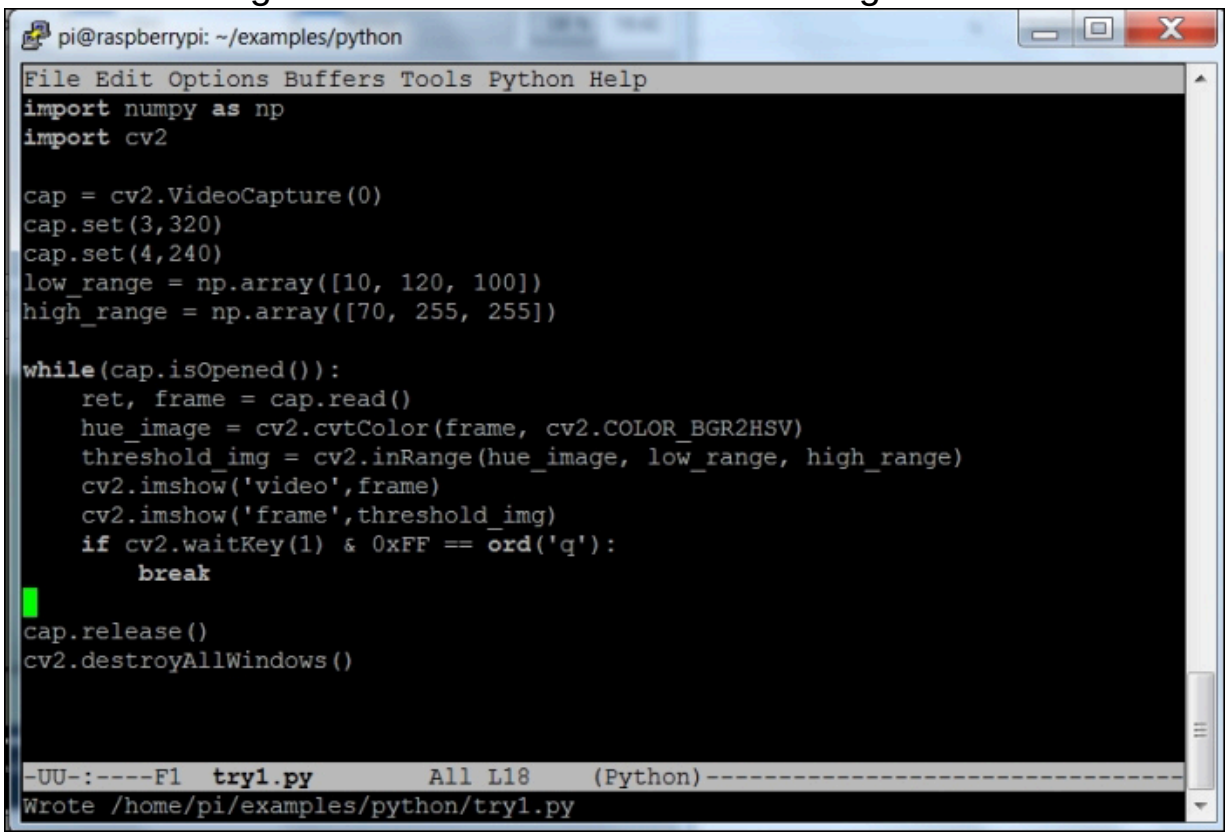
possible barrier:



You can calibrate the distance to the object based on the pixels and the position of the camera.

Color and motion finding

OpenCV and your webcam can also track colored objects. This will be useful if you want your biped to follow a colored object. OpenCV makes this amazingly simple by providing some high-level libraries that can help us with this task. To accomplish this, you'll edit a file to look something like what is shown in the following screenshot:

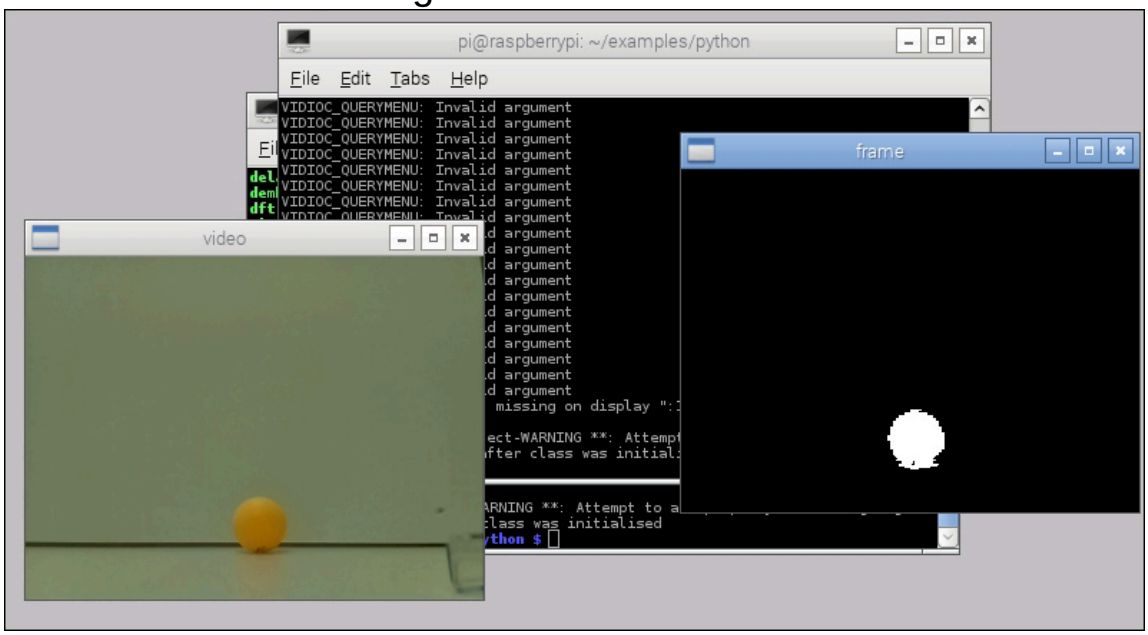


Let's look specifically at the code that makes it possible to isolate the colored ball:

- `hue_img = cv.CvtColor(frame, cv.CV_BGR2HSV):` This line creates a new image that stores the image as per the values of **hue** (color), **saturation**, and **value** (**HSV**), instead of the **red**, **green**, and **blue** (**RGB**) pixel values of the original image. Converting to HSV focuses our processing more on the color, as opposed to the amount of light hitting it.

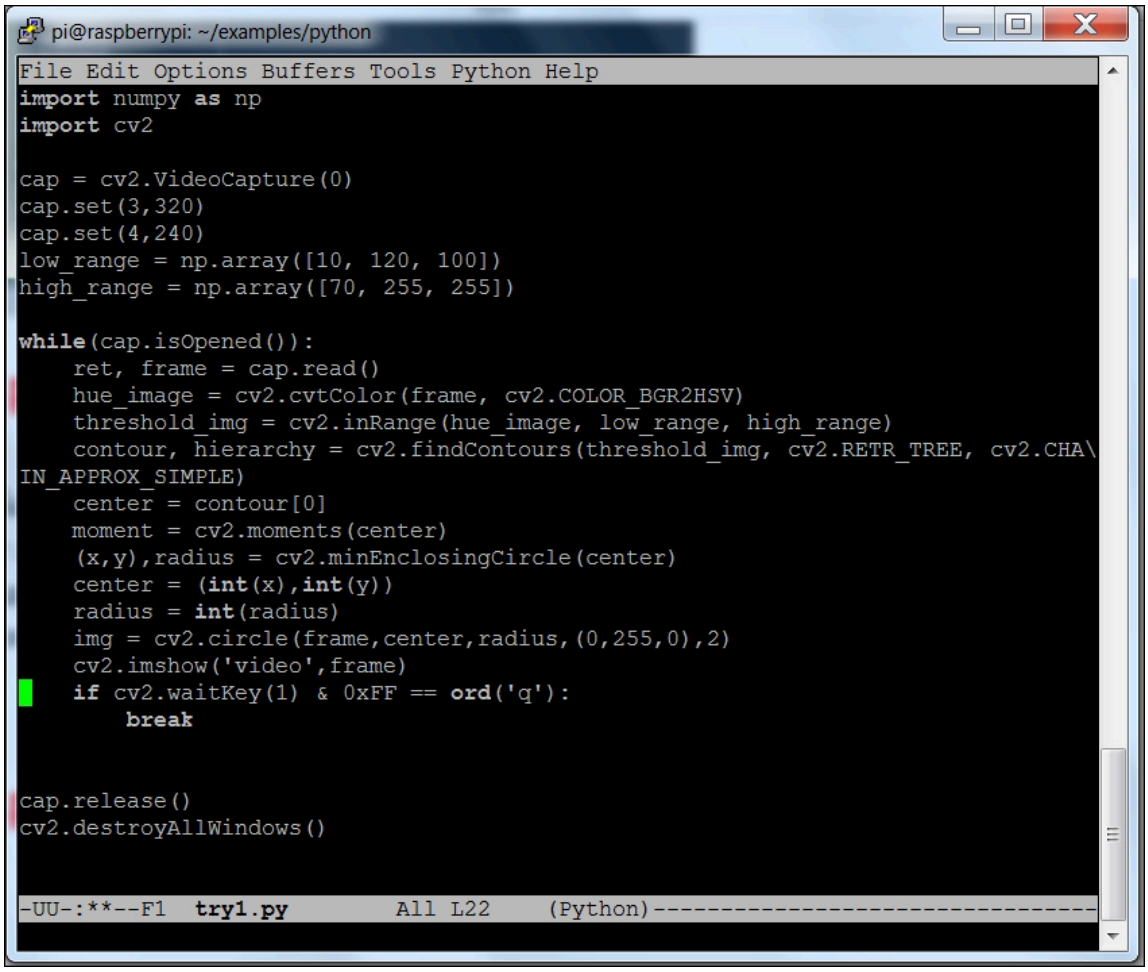
- `threshold_img = cv.InRangeS(hue_img, low_range, high_range)`: The `low_range, high_range` parameters determine the color range. In this case, it is an orange ball, so you want to detect the color orange. For a good tutorial on using hue to specify color, refer to http://www.tomjewett.com/colors/hsb.html. Also, http://www.shervinemami.info/colorConversion.html includes a program that you can use to determine your values by selecting a specific color.

  Run the program. If you see a single black image, move this window, and you will expose the original image window as well. Now, take your target (in this case, an orange ping-pong ball) and move it into the frame. You should see something like what is shown in the following screenshot:



  Notice the white pixels in our threshold image showing where the ball is located. You can add more OpenCV code that gives the actual location of the ball. In our original image file of the ball's location, you can actually draw a rectangle around the ball

as an indicator. Edit the file to look as follows:



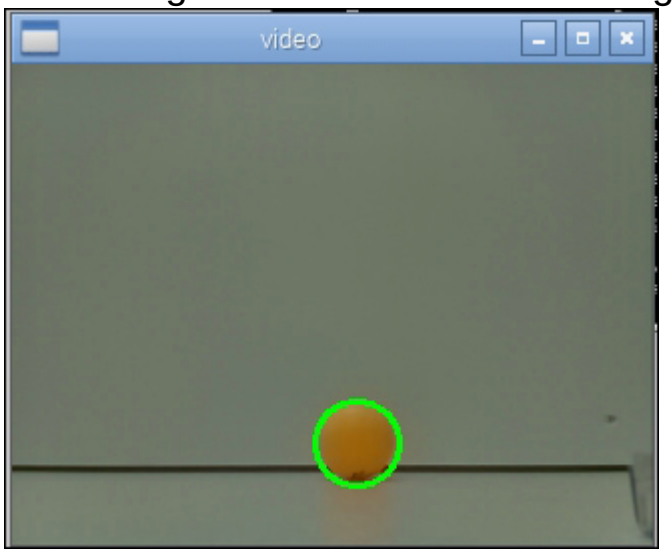
The added lines look like the following:

- `hue_image = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV):` This line creates a hue image out of the RGB image that was captured. Hue is easier to deal with when trying to capture real world images; for details, refer to http://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_Changing_ColorSpaces_RGB_HSV_HLS.php.
- `threshold_img = cv2.inRange(hue_image, low_range, high_range):` This creates a new image that contains only those pixels that occur between the `low_range` and `high_range` n-tuples.
- `contour, hierarchy = cv2.findContours(threshold_img, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE):` This finds the

contours, or groups of like pixels, in the `threshold_img` image.

- `center = contour[0]:` This identifies the first contour.
- `moment = cv2.moments(center):` This finds the moment of this group of pixels.
- `(x,y),radius = cv2.minEnclosingCircle(center):` This gives the *x* and *y* locations and the radius of the minimum circle that will enclose this group of pixels.
- `center = (int(x),int(y)):` Find the center of the *x* and *y* locations.
- `radius = int(radius):` The integer radius of the circle.
- `img = cv2.circle(frame,center,radius,(0,255,0),2):` Draw a circle on the image.

Now that the code is ready, you can run it. You should see something that looks like the following screenshot:



You can now track your object. You can modify the color by changing the low_range and high_range n-tuples. You also have the location of your object, so you can use the location to do path planning for your robot.

Summary

Your biped robot can walk, use sensors to avoid barriers, plans its path, and even see barriers or target. In the final chapter, you'll learn to connect your biped robot remotely so that you can control it and monitor it, without the wires.

Chapter 7. Accessing Your Biped Remotely

Now that your biped is up and running, you'll want to able to send it on its way into the world, but still be able to monitor and control it remotely. This will help you in development as well as deployment and will open up all sorts of new scenarios and applications.

In this chapter, you will learn:

- How to add a wireless LAN dongle to your biped robot and set it up as a wireless access point
- How to control your biped robot using this access and a joystick
- How to use the wireless LAN connection to get **First Person Video** (**FPV**) back so that you can see what your biped robot is seeing

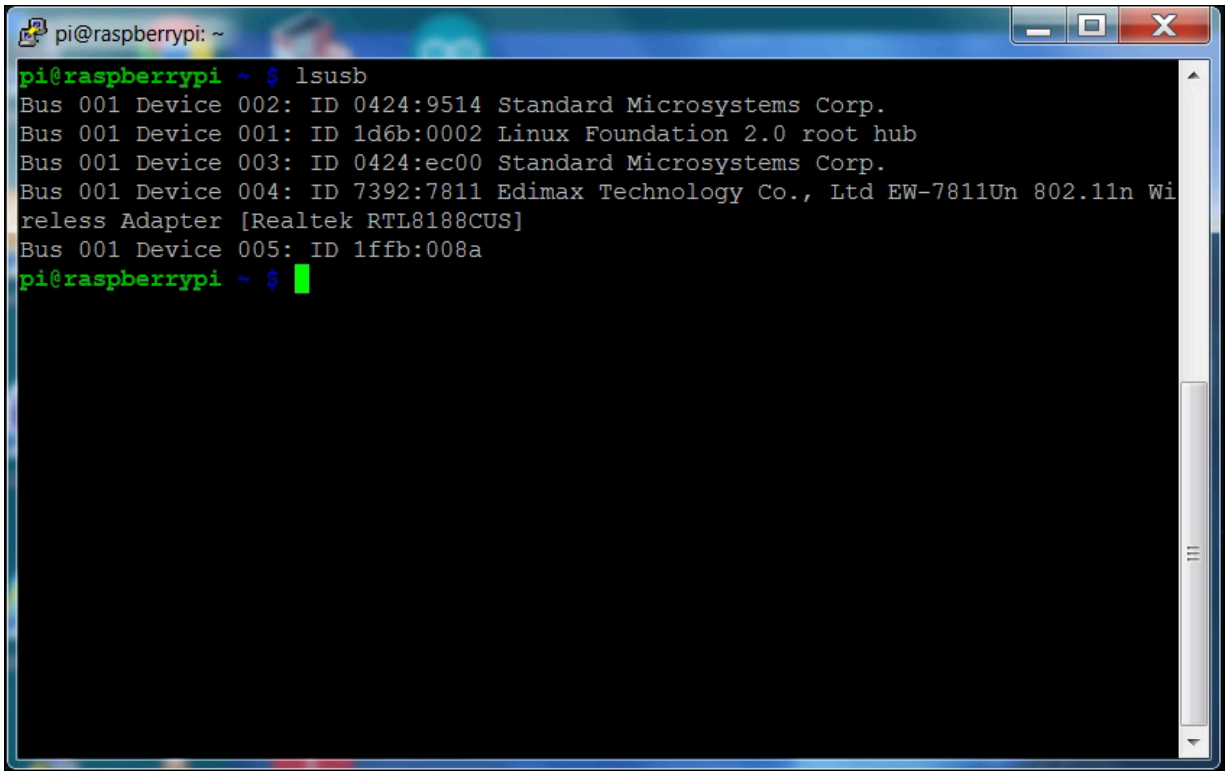
Adding a wireless dongle and creating an access point

In Chapter 1, *Configuring and Programming Raspberry Pi*, you learned how to add a wireless dongle and have the Raspberry Pi connect to your wireless network. This is a useful way to access the Raspberry Pi, but if you want to take your robot outside the coverage of your wireless LAN, you'll want to set it up as an access point.

The first step in doing this is to install the wireless LAN device. One device that is inexpensive and easy to configure is the Edimax Wifi Adapter device (the product information is available at http://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/global/wireless_adapters_n150/ew-7811un). It is available at most online electronic outlets:

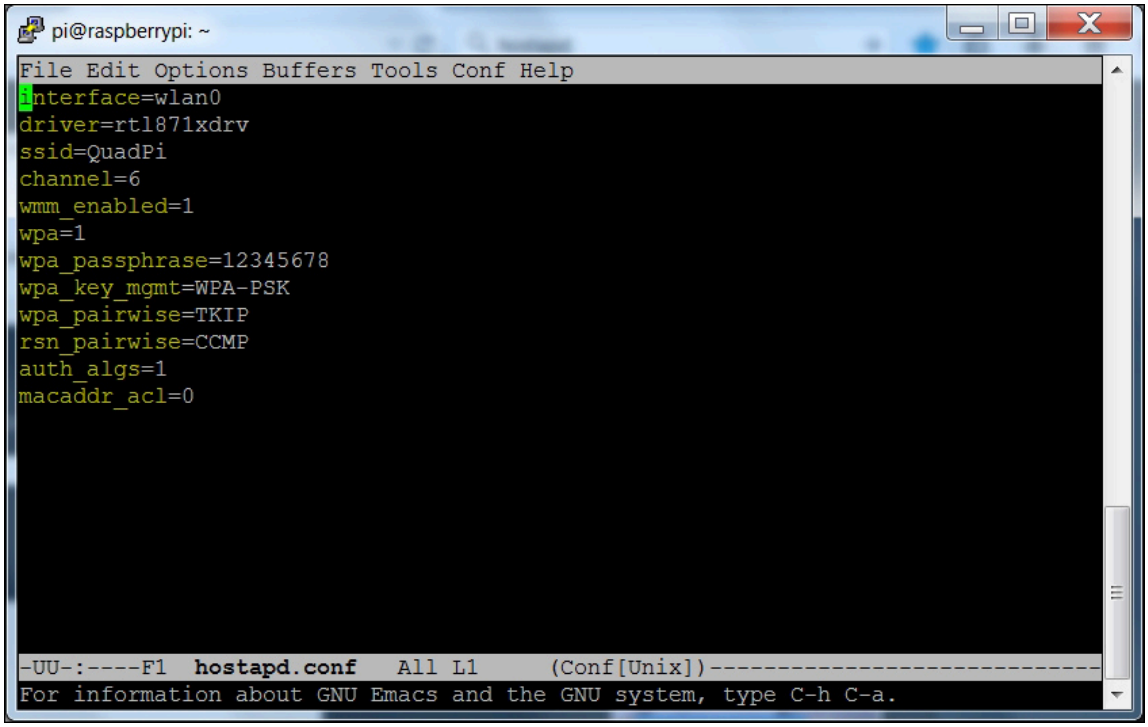


Once you have installed the device and booted Raspberry Pi, type `lsusb` command. This should display something like the following screenshot:

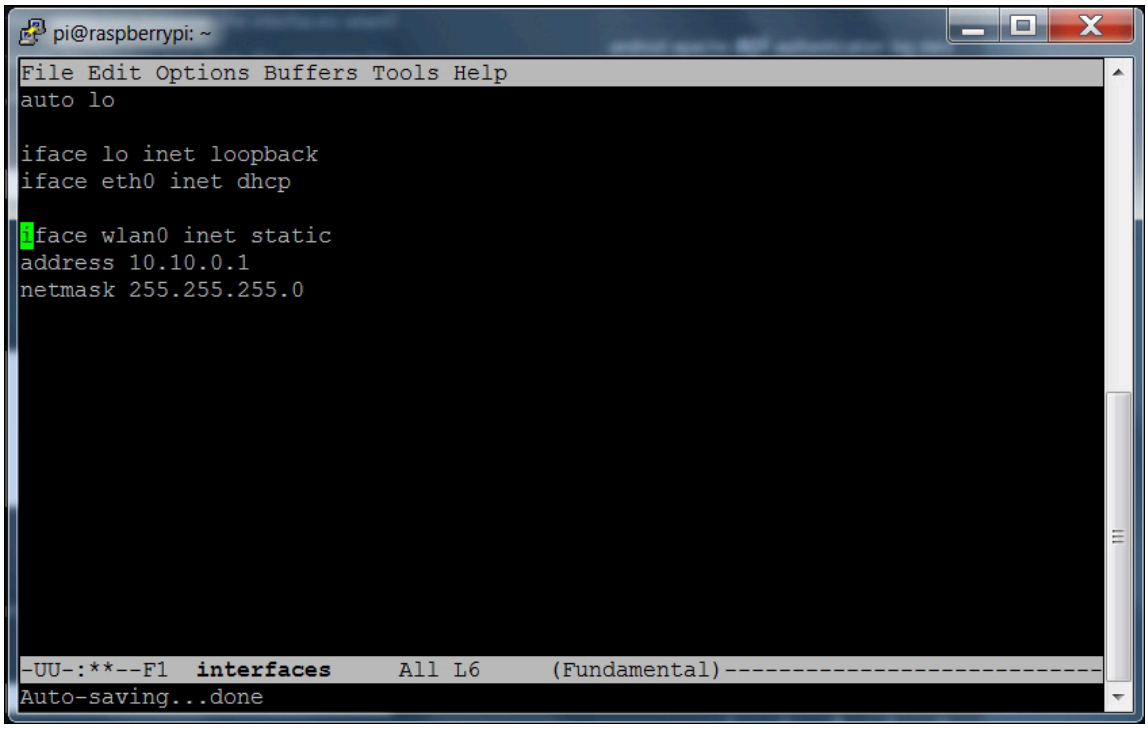


The Edimax device is listed in the set of devices connected to the USB port. Now, execute the following steps:

1. Make sure that you have `hostapd` installed by typing `sudo apt-get install hostapd`. This application is a background application that controls the configuration of wireless on Raspberry Pi.
2. The default version of `hostapd` unfortunately does not support the Edimax chipset by default. So, you'll need to download a version that does by typing `wget http://www.daveconroy.com/wp3/wp-content/uploads/2013/07/hostapd.zip`.
3. Now, unzip this file by typing `unzip hostapd.zip`.
4. Make a backup of your original `hostapd` application by typing `sudo mv /usr/sbin/hostapd /usr/sbin/hostapd.bak`. This way, you'll have it if you want to restore it later.
5. Now, move the new version of `hostapd` to the proper directory by typing `sudo mv hostapd /usr/sbin/hostapd.edimax`.
6. For the next step, type `sudo ln -sf /usr/sbin/hostapd.edimax /usr/sbin/hostapd`; this will create a soft link to the new file so that it will be executed as the `hostapd` application.
7. Type `sudo chown root.root /usr/sbin/hostapd`, and this will change the owner and the group of this file to `root`.
8. Type `sudo chmod 755 /usr/sbin/hostapd` to make this file executable to the owner.
9. Now you will want to configure your wireless access point. Edit the file by typing `sudo emacs /etc/hostapd/hostapd.conf` so that it looks like the following screenshot:

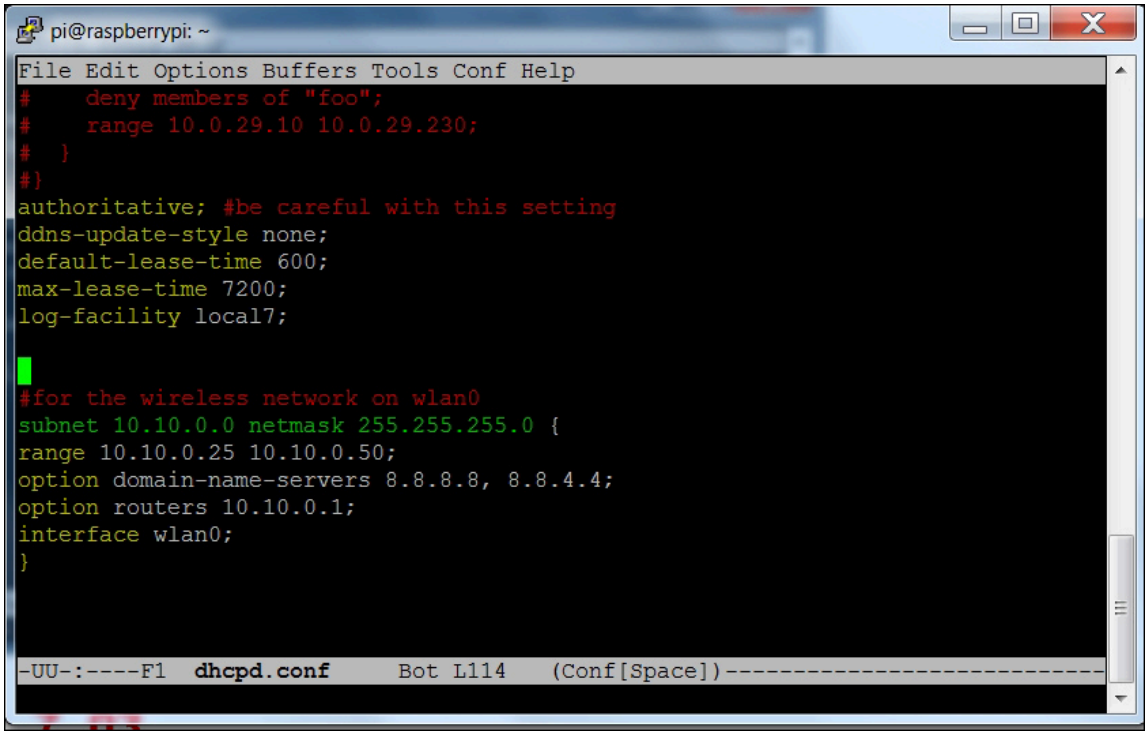


10. You'll now want to edit the `/etc/network/interfaces` file, as shown in the following screenshot:

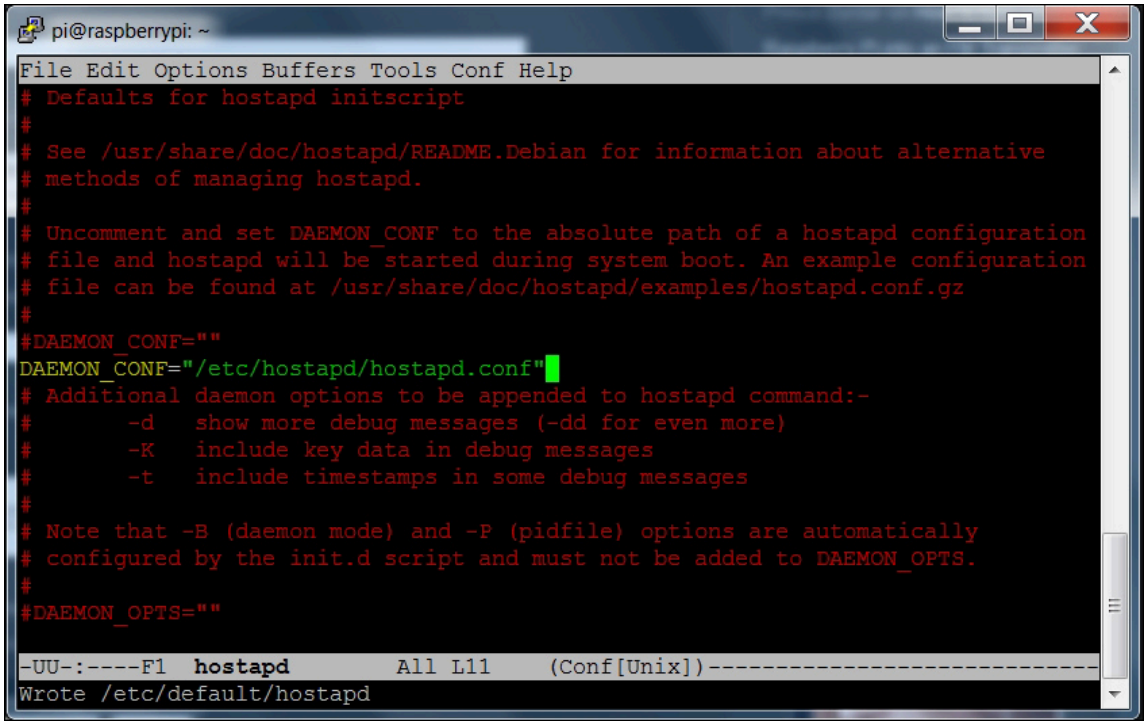


11. This will set the address of the access point to **10.10.0.1**.

12. Now, type `sudo apt-get install isc-dhcp-server` to install a `dhcp` server so that devices that connect to it will be able to get a dynamic address.
13. Now, edit the `/etc/dhcp/dhcpd.conf` and add these lines:



14. The next step is to edit the `/etc/default/hostapd` so that this will all start at power up by adding this line:



15. Now type the following two commands; `sudo update-rc.d hostapd enable` and `sudo update-rc.d isc-dhcp-server enable` and then reboot the Raspberry Pi.

You should now be able to connect to your Raspberry Pi as a wireless access point.

Adding a joystick remote control

Now that you can access your Raspberry Pi from a remote computer, you can SSH, just like you may have been doing with a wired connection, issue commands, and even control the biped using the remote computer. This introduces a number of different possibilities, one of which is to control your project with a joystick connected to the remote computer.

To add the game controller, you'll need to first find a game controller that can connect to your computer. If you are using Microsoft Windows as the OS on the host computer, pretty much any USB controller that can connect to a PC will work. The same type of controller also works if you are using Linux for the remote computer. In fact, you can use another Raspberry Pi as the remote computer.

Since the joystick will be connected to the remote computer, you'll need to run two programs: one on the remote computer and one on the Raspberry Pi on the biped robot. You'll also need a way to communicate between them. In the following example, you'll use the wireless LAN interface and a client-server model of communication. You'll run the server program on the remote computer, and the client program on the Raspberry Pi on the biped robot.

Note

For an excellent tutorial of this type of model and how it is used in a gaming application, see http://www.raywenderlich.com/38732/multiplayer-game-programming-for-teens-with-python.

The first step is to simply plug in your USB game controller to the remote computer. Once you have the controller connected to the remote computer, you'll need to create a Python program on the Raspberry Pi that will take the signals sent from the remote computer client and send the control to the server running on Raspberry Pi so that you can send the correct signals to the servos on your biped.

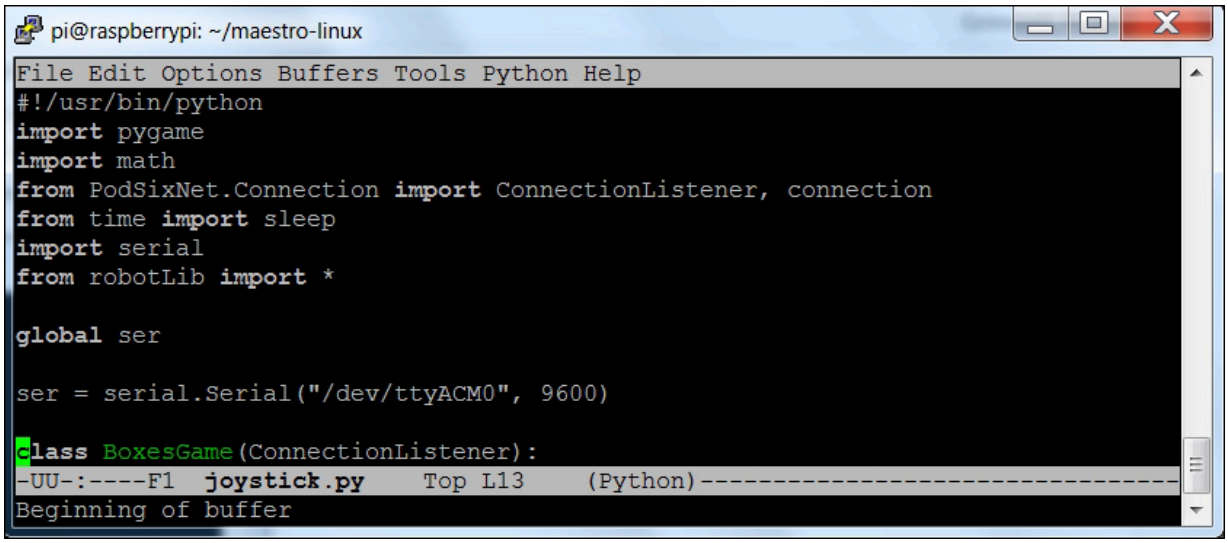
Before you do this, you'll need to install the libraries on Raspberry Pi that will allow this to work. The first is a library called `pygame`. Install this by typing `sudo apt-get install python-pygame`. You'll also need to install a set of Python install tools by typing `sudo apt-get install python-setuptools`. Then, you'll need a LAN communication layer library called `PodSixNet`. This will allow the two applications, the client on the remote computer and the server running on Raspberry Pi, to communicate. To install this, follow the instructions at http://mccormick.cx/projects/PodSixNet/. Now you are ready to create the program on Raspberry Pi on the biped. The first part of the program is the Python functions from the program you created in Chapter 3, *Motion for the Biped*. In this section, you'll create a class called `QuadGame`. This class will take the inputs from the game controller connected to the server and turn them into commands that will be sent to the servo controller for your biped robot.

The following is a table of those controls:

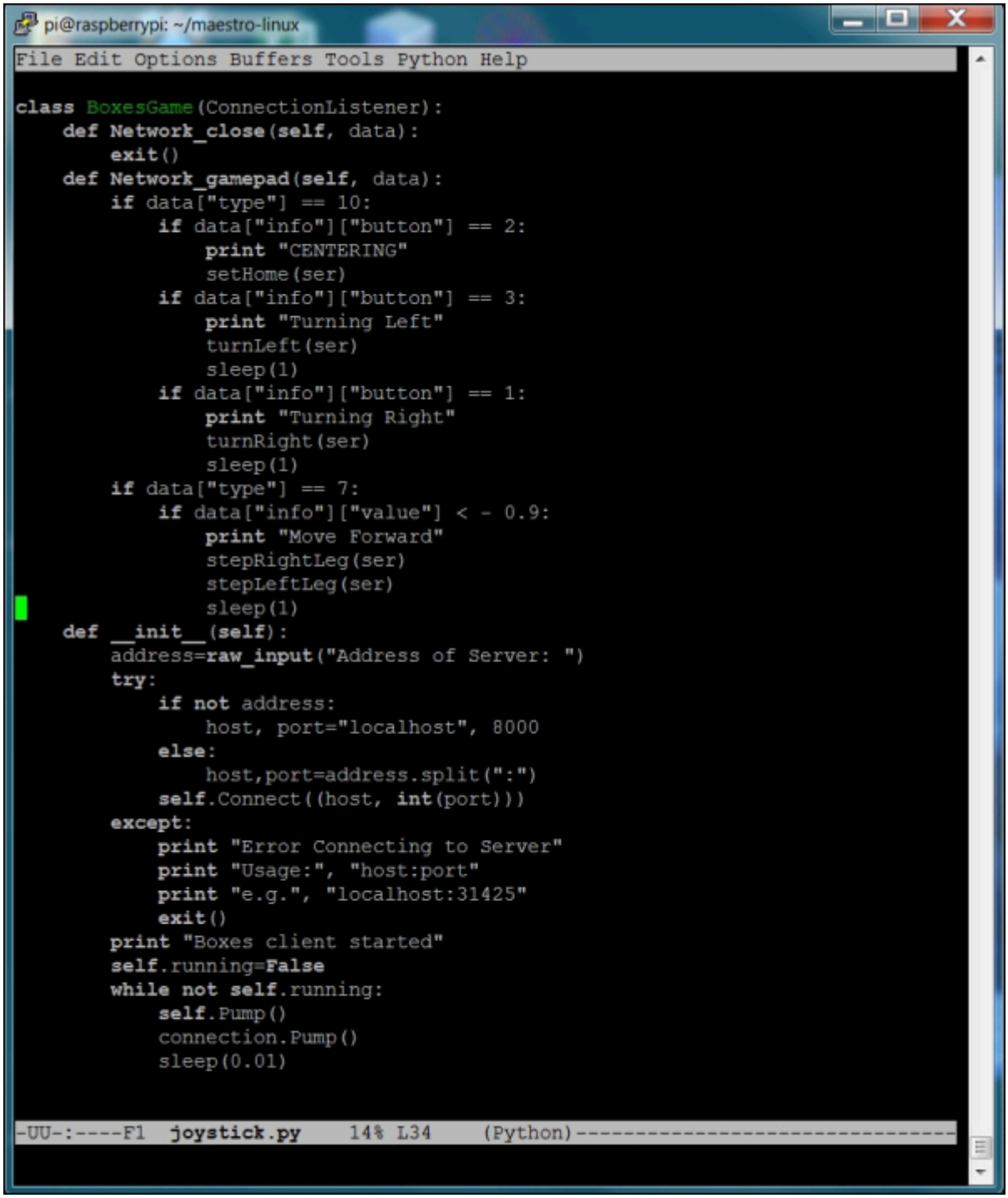
Joystick control	**Biped control**
Button 2	Robot home position
Button 1	Robot turn right

Joystick control	Biped control
Button 3	Robot turn left
Joystick Up	Robot walk forward

Now, the following is the initial part of the code, the Python import statements:



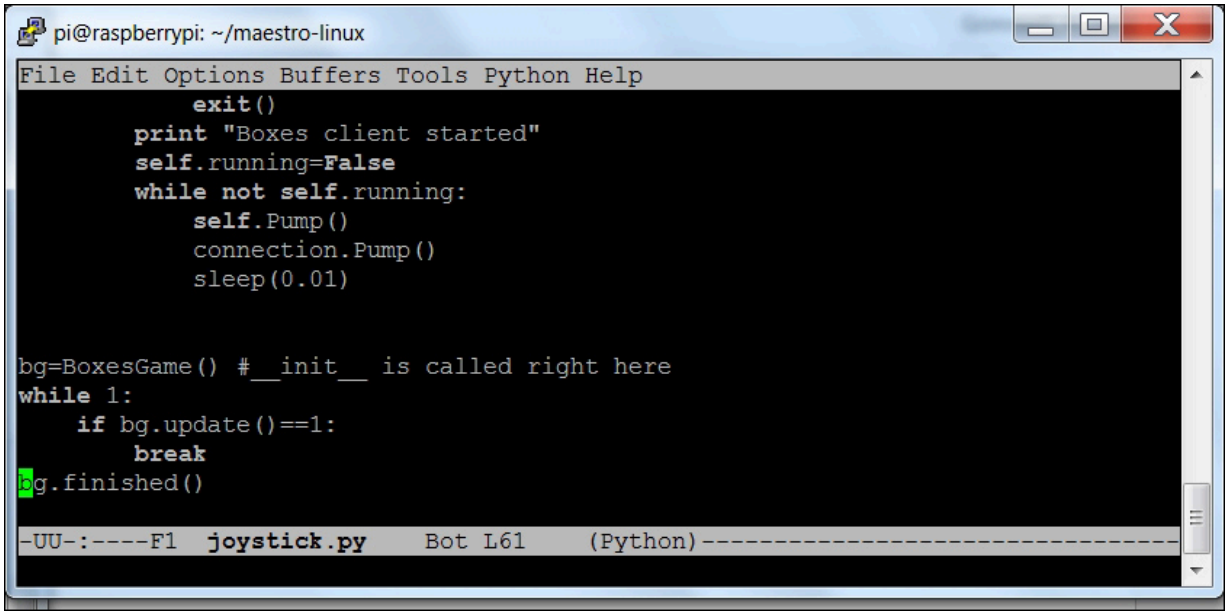
And the following is the `BoxesGame` class, the code that will respond to the joystick:



This is the interesting part of the code. This code takes the input from the remote computer and translates it into action. The first `if` statement determines what type of data is being sent from the remote computer with the joystick attached. It can be a button press, where `data["type"] == 10`, and then the statement `data["info"]`

`["button"] == 2` determines that **button 2** has been pressed. In this case, this will send commands that will cause the robot to go to the home position. If the `if data["type"] == 7:`, then this is a joystick event, and the `if data["info"]["value"] < - 0.9`, then this will determine that the joystick is in the up position and the robot should move forward.

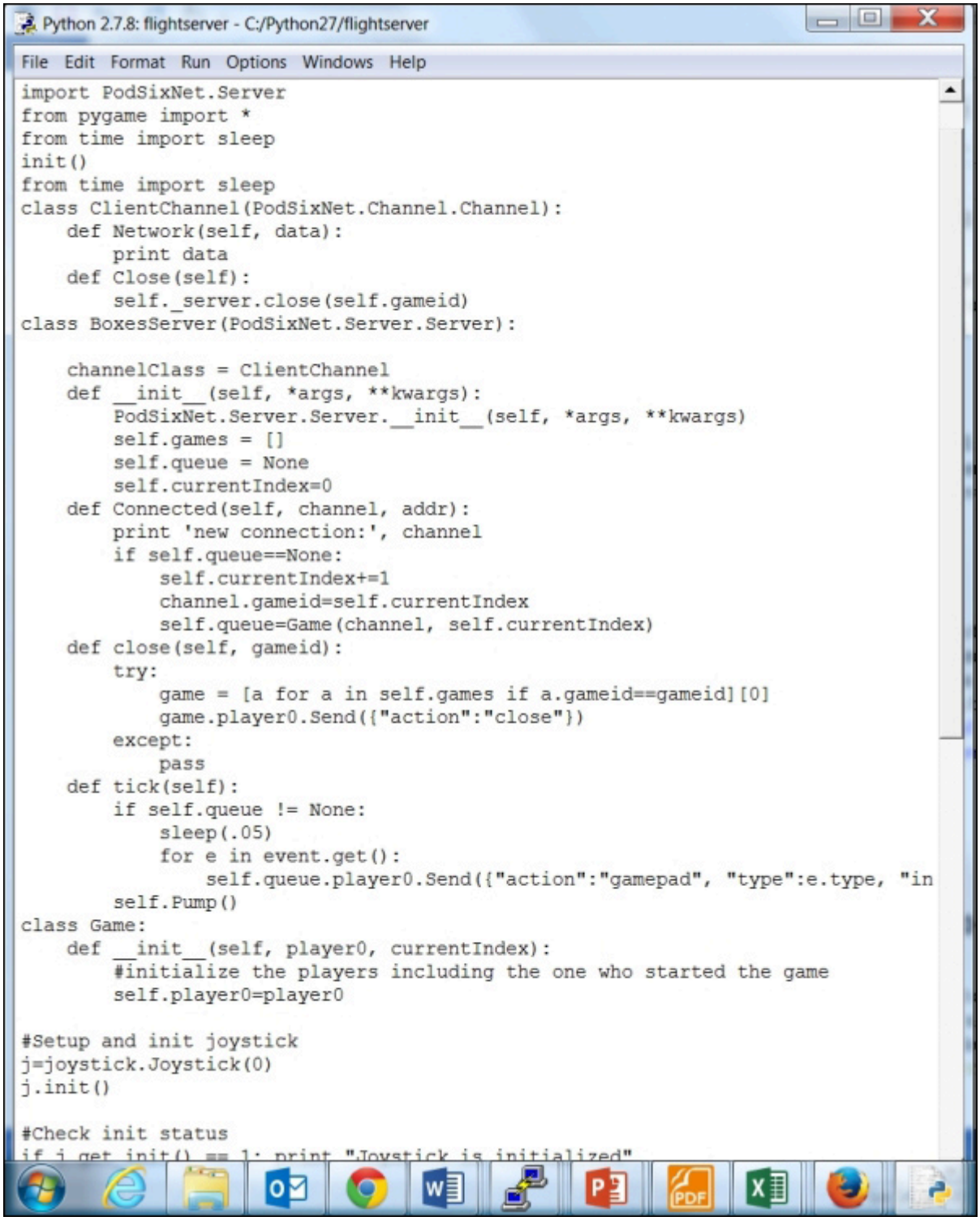
The following is the final part of the joystick controller aspect of the client program for completeness:



This final piece of code initializes the game loop, which loops while taking the inputs, sends them to the servo controller, and on to the flight controller.

You'll also need a server program running on the remote computer that will take the signals from the game controller and send them to the client. You'll be writing this code in Python using Python version 2.7, which can be installed from here. Additionally, you'll need to install the `pygame` library. If you are using Linux on the remote computer, then type `sudo apt-get install python-pygame`. If you are using Microsoft Windows on the remote machine, then follow the instructions at http://www.pygame.org/download.shtml.

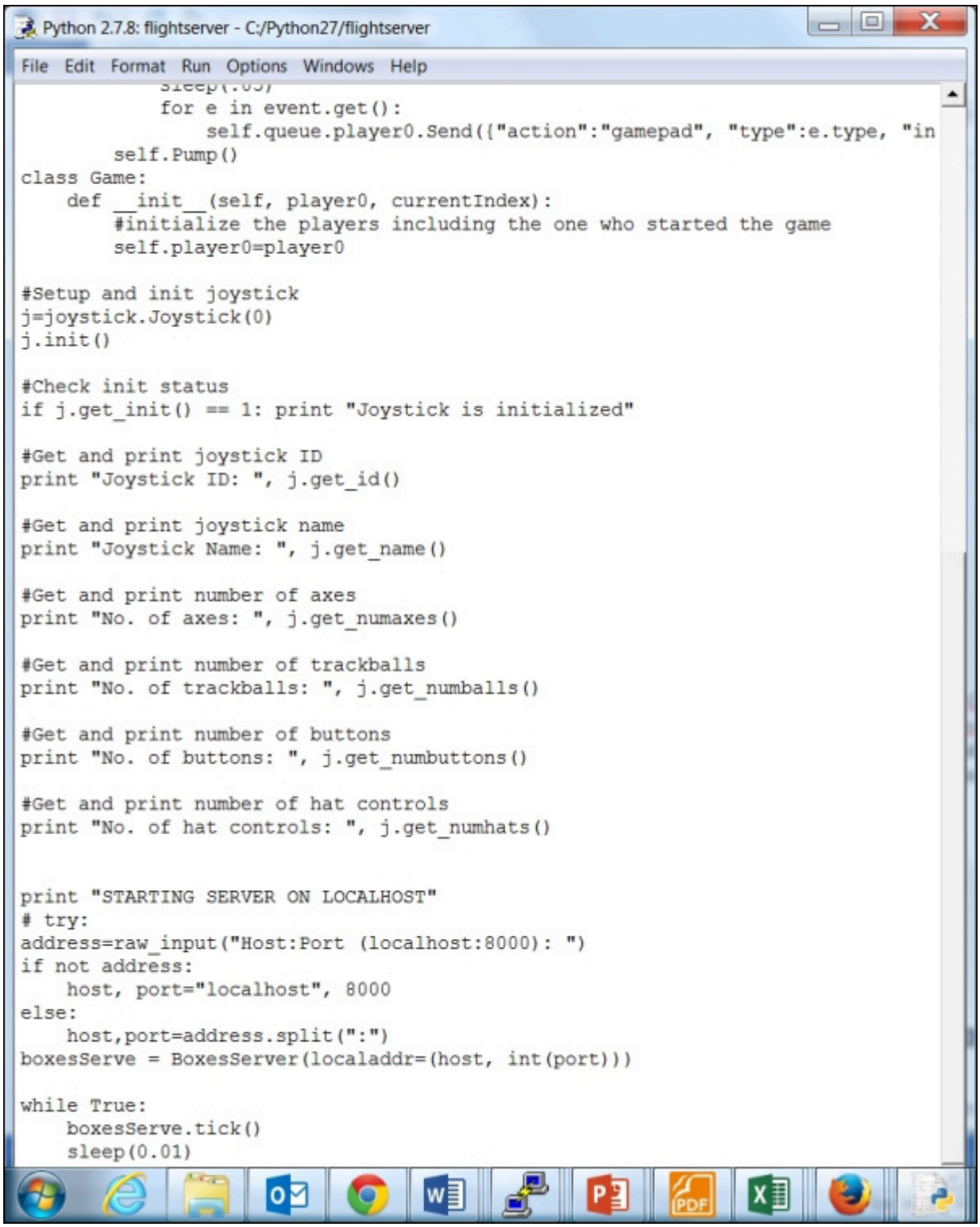
You'll also need the LAN communication layer described previously. You can find a version that will run on Microsoft Windows or Linux at http://mccormick.cx/projects/PodSixNet/. The following is a listing of the server code in two parts:



This first part creates three classes:

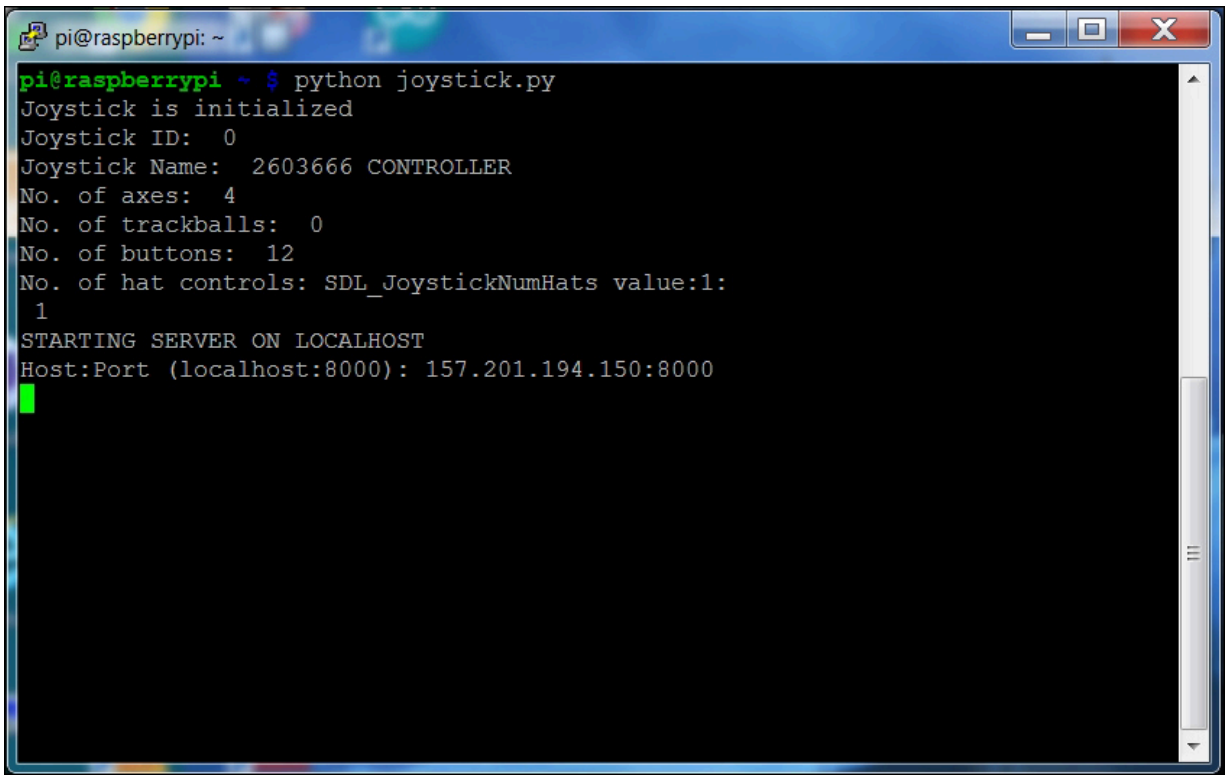
1. The first, class `ClientChannel`, establishes a communication channel for your project.
2. The second, class `BoxServer`, sets up a server so that you can communicate the joystick action to the Raspberry Pi on the biped.
3. Finally, the third class, `Game` , just initializes a game that contains everything you'll need.

The following is the latter part of the code:

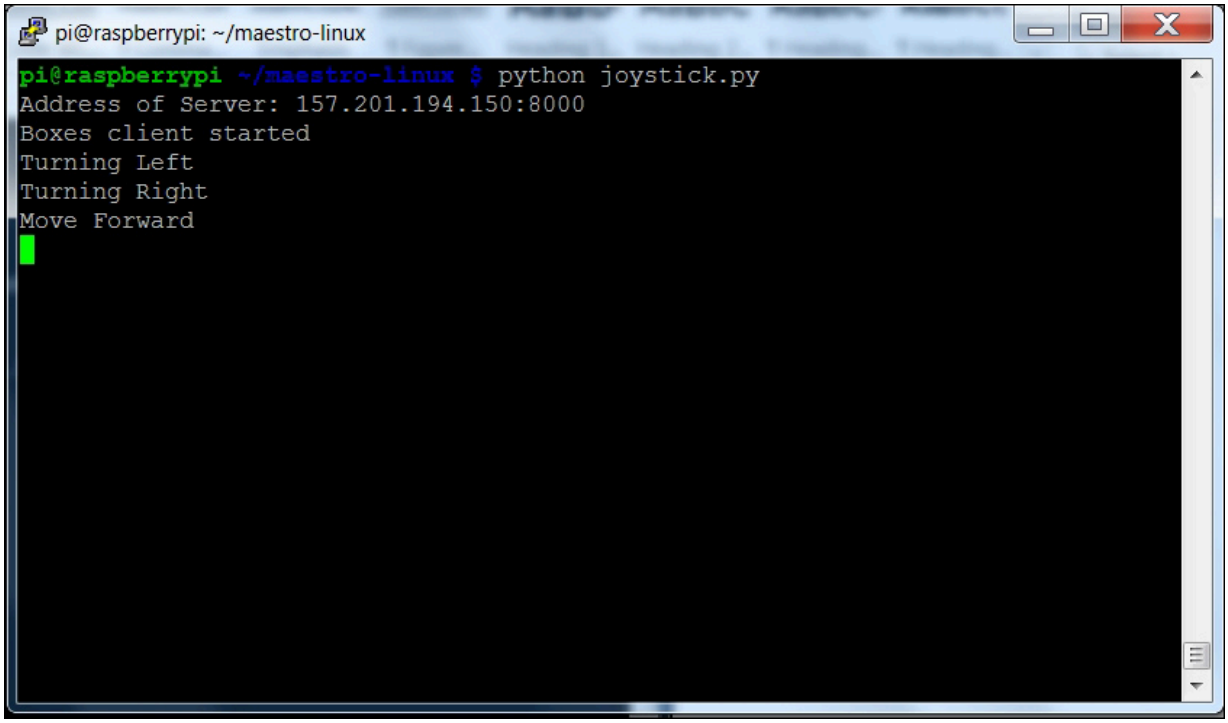


This part of the code initializes the joystick so that all the controls can be sent to the biped's Raspberry Pi.

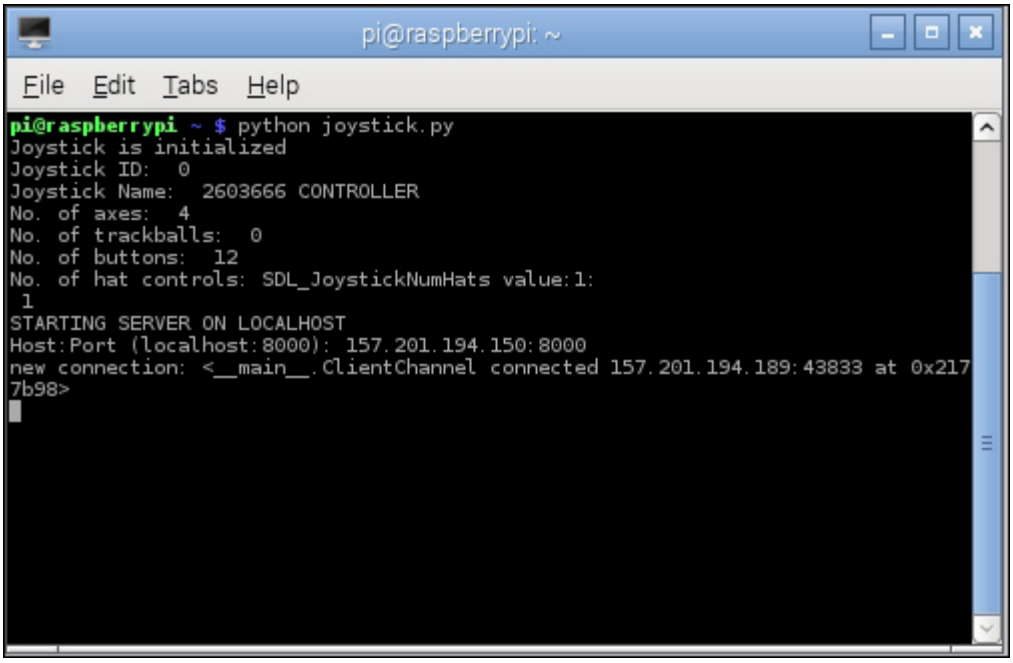
You'll need to run these programs on both computers, entering the Internet address of the remote computer connected to the joystick. The following is what running the program on that computer will look like, before running the program on the remote computer:



And the following is what the program will look like when run on the Raspberry Pi and connected to the robot:



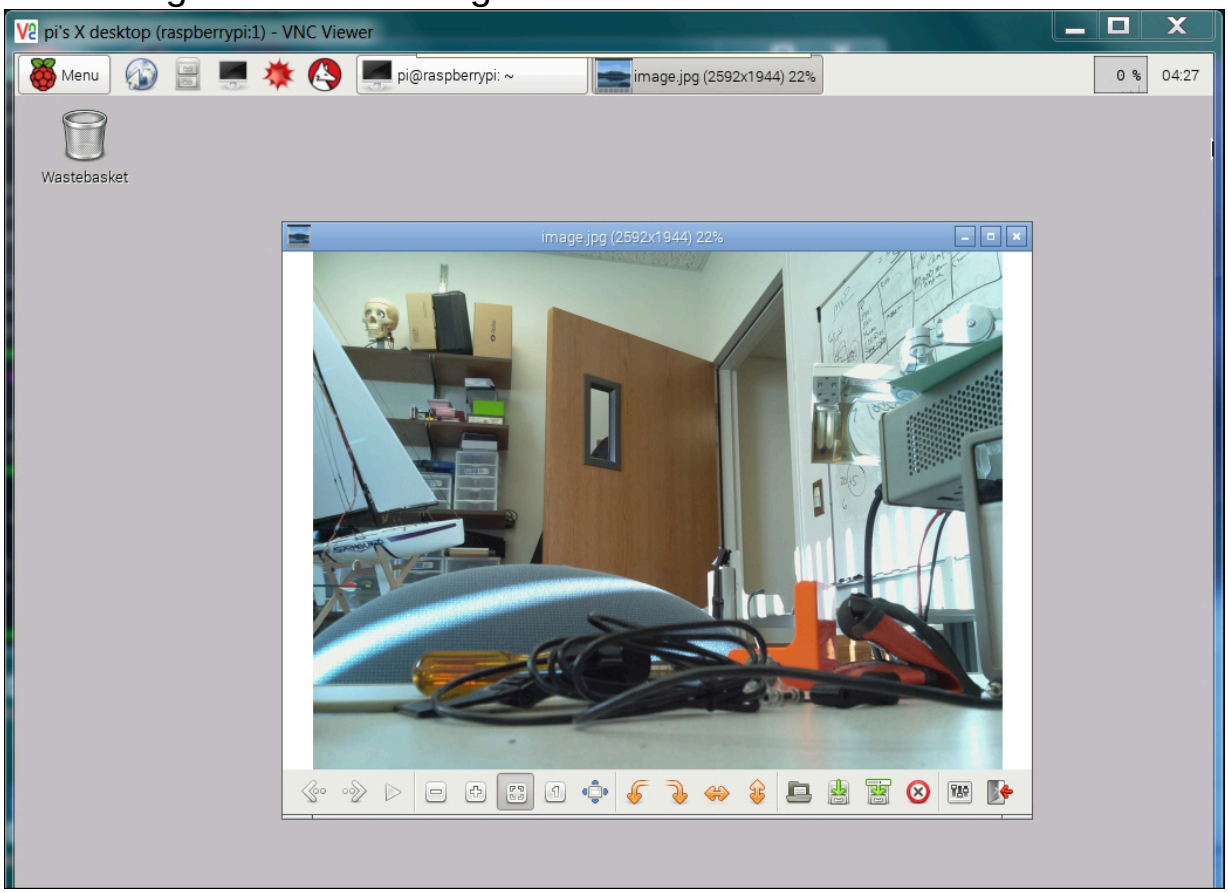
Finally, the following is what the program will look like on the remote computer when the robot's Raspberry Pi is up and connected:



Now you can control you robot remotely using the joystick!

Adding the capability to see remotely

Your biped can now get information from your remote computer and respond to joystick key presses, but you may want to be able to see what the biped sees from its webcam. This is straightforward to configure with a webcam, vncserver, and the capability you used in Chapter 6, *Adding Vision to Your Biped*. Using this method, you can easily get a picture of what your biped is seeing, and it should be something like the following:



Now you can both see where your robot is going and control it via a joystick.

Summary

That's it, but really it is only the beginning. Your robot has some basic motions and some basic control capability, but now you should also have the knowledge and skills to take your biped robot much further. You can teach it how to dance, follow gestures, and almost anything that you can imagine.

Appendix A. Bibliography

This Learning Path is packaged keeping your journey in mind. It includes content from the following Packt products:

- *Raspberry Pi By Example, Ashwin Pajankar and Arush Kakkar*
- *Building a Home Security System with Raspberry Pi, Matthew Poole*
- *Raspberry Pi Robotics Essentials, Richard Grimmett*

Index

A

- access point
  - creating / Adding a wireless dongle and creating an access point
- Adafruit
  - references / The Raspberry Pi Zero
- ADC pi+
  - URL / Connecting Raspberry Pi to an infrared sensor
- Advanced IP Scanner
  - URL / Installing the operating system
- alarm system
  - about / Alarm system diagram
  - diagram / Alarm system diagram
  - elements / Overview of the system elements
  - starting automatically / Automatically starting the system
  - building, tips / Tips for building systems
- Allied Electronics
  - URL / Raspberry Pi
- anti-tamper circuits
  - about / Anti-tamper circuits

- anti-tamper loop / Anti-tamper circuits
- Apache
    - installing, on Raspbian / Introducing and installing Apache on Raspbian
- API (Application Programming Interface)
    - about / Installing the Twitter API for Python, The basics of sockets
- Arduino / Gertboard
- arm
    - about / Arming yourself
    - documenting / Arming yourself
- arm device / Connecting a 433-MHz receiver
- ASHRAE
    - URL / A carbon monoxide detector
- ATmega microcontroller / Gertboard
- attachments
    - sending / Sending attachments

B

- backlog
    - about / Listening for incoming connections
- Bash command-line calculator (bc tool)
    - used for converting hx values to binary / Building the control script

- Bash script / I'm someone else
- basic poses
    - defining / A basic stable pose
- battery-backed / A +12V power supply
- benevolent dictator for life (BDFL)
    - about / Getting started with Python
- Berkeley Software Distribution (BSD)
    - about / Introducing OpenCV
- binary clock
    - building / Building a binary clock
- binary fractal tree
    - drawing / Drawing a binary fractal tree
- biped
    - assembling / Building the biped platform
- biped platform
    - building / Building the biped platform
- bit banging / The alternative approach (because we have no choice)
- Bluetooth Low Energy (BLE)
    - about / The Raspberry Pi 3
- Buzzer
    - URL / Introducing the GrovePi

C

- C++ programming language
    - defining / An introduction to the C/C++ programming language
- C/C++ code example
    - defining / An introduction to the C/C++ programming language
- camera
    - installing, on biped robot / Installing a camera on your biped robot
- camera, adding to security systems
    - prerequisites / Prerequisites
    - Raspberry Pi camera module / The Raspberry Pi camera module
    - Pi NoIR camera module / Night vision
    - USB webcam / Using USB cameras
- camera modules
    - URL / Working with the Pi Camera and NoIR Camera modules
- camera serial interface (CSI) / The Raspberry Pi camera module
- Canny edge algorithm
    - URL / Edge Detection and OpenCv
- capability

    - adding, for remote vision / Adding the capability to see remotely
- carbon monoxide (CO) detector
    - about / A carbon monoxide detector
- cascading style-sheet (CSS) / Creating the web page
- center of gravity
    - about / A basic walking motion
- chat program
    - about / A chat program
    - chat server / The chat server
    - chat client / The chat client
- cluster
    - performance benchmarking / Performance benchmarking of the cluster
- code
    - running, in parallel / Running code in parallel
- colored objects
    - tracking / Color and motion finding
- color range
    - tracking / Tracking in real time based on color
- colorspaces
    - about / Colorspaces and conversions
- commands, OpenCV

    - sudo apt-get update / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install build-essential / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install libavformat-dev / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install ffmpeg / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install libcv2.4 libcvaux2.4 libhighgui2.4 / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install python-opencv / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install opencv-doc / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install libcv-dev / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install libcvaux-dev / Downloading and installing OpenCV – a fully featured vision library
    - sudo apt-get install libhighgui-dev / Downloading and installing OpenCV – a fully featured vision library
    - cp -r /usr/share/doc/opencv-doc/examples /home/pi/ / Downloading and installing OpenCV – a fully featured vision library
- compass capability
    - accessing / Accessing the compass programmatically
- Compute Module
    - about / Raspberry Pi models
- Compute Module and Compute Module development kit
    - references / Raspberry Pi models
- Computer Vision

    - defining / Introducing Computer Vision
- control panel
    - remote access, setting up / Remote access to our control panel
- control panel HTML template / The control panel HTML template
- controls
    - Joystick control / Adding a joystick remote control
    - about / Adding a joystick remote control
    - Biped control / Adding a joystick remote control
- control scripts
    - designing / Designing the control scripts
    - building / Building the control script
    - code, exploring / Exploring the script code
- control scripts code
    - exploring / Exploring the script code
    - declarations / Declarations
    - config settings, updating / Updating config settings
    - GPIO, setting up / Setting up the GPIO
    - I2C port expander, setting up / Setting up the I2C port expander
    - zone inputs status, decoding / Decoding the zone inputs status
    - initialization / Initialization
    - system monitoring loop / The system monitoring loop
    - system, arming / Arming the system
    - zones, monitoring / Monitoring the zones
    - entry delay / Entry delay
    - main alarm, sounding / Sounding the main alarm
    - system, disarming / Disarming and resetting the system
    - system, resetting / Disarming and resetting the system
    - features, implementing / We're done (almost)–

- control variables / Declarations
- conversions
  - about / Colorspaces and conversions
- C programming language
  - defining / An introduction to the C/C++ programming language
- cron
  - about / Crontab
- crontab
  - URL / Crontab
- cv2.destroyWindow() function
  - about / Working with images
- cv2.imread() function
  - about / Working with images
- cv2.imshow() function
  - about / Working with images
- cv2.imwrite() function
  - about / Working with images
- cv2.waitKey() function
  - about / Working with images

D

- data logging
    - detecting / Logging detection data
- Data Ready (DRDY) / Connecting a digital compass to the Raspberry Pi
- ddclient updater / The Raspberry Pi dynamic DNS client
- DHCP
    - about / Introducing DHCP
- DHCP (Dynamic Host Configuration Protocol)
    - about / Introducing DHCP
- DHCP server
    - Raspberry Pi, configuring to act as / Configuring a Raspberry Pi to act as a DHCP server
- digital compass
    - connecting, to Raspberry Pi / Connecting a digital compass to the Raspberry Pi
- digital I/O pins / Digital I/O pins
- disarm
    - about / Arming yourself
- disarm device / Connecting a 433-MHz receiver
- distance
    - calculating / Basic path planning
- DnsOMatic
    - URL / Setting up a dynamic DNS account

- DNS server
  - setting up, on Pi / Setting up a DNS server on the Pi
- Domain Name System (DNS)
  - about / Introducing Domain Naming System (DNS)
- Domain Naming System (DNS)
  - defining / Introducing Domain Naming System (DNS)
- doors / The working of magnetic contact sensors
- dyna -micro
  - about / Single-board computers
- dynamic DNS account
  - setting up / Setting up a dynamic DNS account
  - Raspberry Pi dynamic DNS client / The Raspberry Pi dynamic DNS client
- dynamic path planning, for robot
  - defining / Dynamic path planning for your robot
  - path, planning / Basic path planning
  - obstacles, avoiding / Avoiding obstacles
- DynDNS
  - URL / Setting up a dynamic DNS account

E

- Edge Detection
  - defining / Edge Detection and OpenCv

- Edimax Wifi Adapter device
    - URL / Adding a wireless dongle and creating an access point
- Egoman
    - URL / Raspberry Pi
- elaborate light switch
    - about / The most elaborate light switch in the world, The Elaborate light switch re-visited
    - breadboard / The most elaborate light switch in the world
    - circuit diagram / The most elaborate light switch in the world
    - illuminating script / The illuminating script
- electronic keypad / Arming and disarming the system
- elements, alarm system
    - overview / Overview of the system elements
    - +12V power supply / A +12V power supply
    - +3.3V power supply / A +3.3V power supply
    - opto-isolator input module / The opto-isolator input module
    - port expander / The port expander
    - arm/disarm switch / An arm/disarm switch
    - outputs / Alarm outputs
- email
    - notification, setting / You have new mail
    - sender client, setting up / Setting up the e-mail sender client
    - attachments, sending / Sending attachments
    - information, obtaining / Where was that taken?
- entry / exit purposes / Arming yourself

F

- fake-hwclock package / fake-hwclock
- ffmpeg tool / Be a video star
- file system, SD Card
    - raspi-config utility, using / Booting your Pi, Using the raspi-config utility
    - expanding / Expanding the file system
- fixed IP address / Setting up a dynamic DNS account
- flow-chart / Designing the control scripts
- FourCC
    - URL / Saving a video using OpenCV
- fractal tree
    - about / Drawing a binary fractal tree
- fritzing
    - URL / Let there be light
- fswebcam
    - URL / Installing the webcam
- fswebcam utility
    - installing / Working with webcams
    - about / Working with webcams
- functions, importing
    - URL / Basic path planning

G

- GalaxSee

    - about / Introducing N-Body simulations
    - URL / Introducing N-Body simulations
    - installing / Installing and running GalaxSee
- GalaxSee program
    - executing / Installing and running GalaxSee
- Gertboard
    - about / Gertboard
    - URL / Gertboard
- GPAC package / Be a video star
- GPIO connector
    - prerequisites, for connecting to / Prerequisites
    - about / Say hello to the GPIO
    - access, providing / Say hello to the GPIO
    - digital I/O pins / Digital I/O pins
    - Inter-Integrated Circuit (I2C) bus / The I2C bus
    - Serial Peripheral Interface (SPI) bus / The SPI bus
    - Universal Asynchronous Receiver and Transmitter (UART) bus / The UART serial bus
    - Universal Serial Bus (USB) ports / USB ports
    - power connections / Power connections
    - using / Getting acquainted with the GPIO
    - connecting, to LED / Let there be light, Getting flashy–
    - switch, adding / Adding a switch
    - floating state / Pulling yourself together
    - detection script / The detection script
- GPIO pin
    - export / Setting up the GPIO
- GPIO pins
    - defining / Introducing GPIO pins

- GPU (Graphics Processor Unit)
  - about / Booting up our Pi for the first time
- grid
  - drawing / Basic path planning
- GrovePi
  - defining / Introducing the GrovePi
  - URL / Introducing the GrovePi
  - setting up / Setting up the GrovePi
- GrovePi+
  - URL / Introducing the GrovePi
- GY-271 HMC5883L Triple Axis Compass Magnetometer Sensor Module / Connecting a digital compass to the Raspberry Pi

H

- H.264 video stream / Be a video star
- hall-effect sensors / Getting into the zone
- high-level process / Designing the control scripts
- Hitec servos
  - about / Building the biped platform
- HMC5883L 3-Axis Digital Compass chip / Connecting a digital compass to the Raspberry Pi
- Hobbytronics MCP23017 expander port kit
  - about / Hobbytronics MCP23017 expander port kit
  - URL / Hobbytronics MCP23017 expander port kit
- hue

    - URL / Color and motion finding
- hue (color), saturation, and value (HSV) / Color and motion finding
- humidity and temperature sensor
    - URL / Introducing the GrovePi
    - using / Displaying the weather

- I/O pins / Power connections
- I2C bus
    - about / The I2C bus
    - Serial Data Line (SDA) wire / Just 2 wires
    - Serial Clock Line (SCL) wire / Just 2 wires
    - address, identifying / What's your address?
    - data, transmitting / There is a parallel universe
    - serial-to-parallel conversion / Serial-to-parallel conversion
    - enabling / Enabling the I2C Bus
    - tools package, installing / Installing the I2C tools package
- I2C interface
    - enabling / Accessing the compass programmatically
- I2C port expander
    - building / Building an I2C expander
    - board, building / Building your expansion board
    - setting up / Setting up the I2C port expander, Setting up the port expander
    - I2C bus, enabling / Enabling the I2C Bus
    - I2C tools package, installing / Installing the I2C tools package
    - devices, searching / Finding our devices

- I2C port expander circuit
    - about / The I2C port expander circuit, Let's walk through the circuit
    - components / The I2C port expander circuit
    - MCP23017 port expander chip / Let's walk through the circuit
    - high / Let's walk through the circuit
    - low / Let's walk through the circuit
    - floating / Let's walk through the circuit
- i2cset tool / Setting up the port expander
- image color channels
    - splitting / Splitting and merging image color channels
    - merging / Splitting and merging image color channels
    - image, negating / Negating an image
- image properties
    - retrieving / Retrieving image properties
- images
    - working with / Working with images
    - matplotlib, using / Using matplotlib
    - arithmetic operations / Arithmetic operations on images
    - logical operations / Logical operations on images
    - thresholding / Thresholding images
- inductive loads
    - driving / Driving inductive loads
    - bells / Driving inductive loads
    - lamps / Driving inductive loads
- infra-red lighting / Is that a badger?
- infrared sensor

    - Raspberry Pi, connecting to / Connecting Raspberry Pi to an infrared sensor
- integrated circuits (ICs) / Serial-to-parallel conversion
- Inter-Integrated Circuit (I2C) bus / The I2C bus
- Internet of Things
    - defining / Introducing the Internet of Things
- Internet Protocol (IP)
    - about / The basics of sockets
- Intruder detection system
    - about / Intruder detection system
- intrusion / Beyond intrusion
- IP address / Setting up your Pi
- IR sensor, building
    - URL / Connecting Raspberry Pi to an infrared sensor
- Itseez
    - URL / Introducing OpenCV

J

- Jessie version / Booting your Pi
- joystick remote control
    - adding / Adding a joystick remote control
    - URL / Adding a joystick remote control

K

- key switch / Arming and disarming the system

- kit, biped platform
    - URL / Building the biped platform

L

- LAN communication layer
    - URL / Adding a joystick remote control
- Last In First Out (LIFO)
    - about / Drawing a binary fractal tree
- LED
    - URL / Introducing the GrovePi
- LED Blinker
    - building / Building an LED Blinker
    - button, connecting / Connecting a button
- LED Resistor Values
    - calculating / Let there be light
- lighttpd web server / Installing the web server
- Linux
    - reference / Preparing MicroSD card manually
    - using / Using Linux
- logic operators
    - used, in Python / Building a binary clock

M

- 433-MHz receiver
    - connecting / Connecting a 433-MHz receiver
    - connecting, alternative approach / The alternative approach (because we have no choice)
- 433-MHz wireless alarm systems
    - about / 433-MHz wireless alarm systems
    - advantages / 433-MHz wireless alarm systems
    - disadvantages / 433-MHz wireless alarm systems
- Mac OS
    - reference / Preparing MicroSD card manually
- magnetic contact sensor
    - adding, prerequisites / Prerequisites
    - working / The working of magnetic contact sensors
    - connecting / Connecting our magnetic contact sensor
    - monitoring / Monitoring the sensor
- magnetic field / The working of magnetic contact sensors
- master configuration file
    - using / The master configuration file
- matplotlib
    - URL / Using matplotlib
- MCP23017 / Building an I2C expander
- Microsoft Windows
    - using / Using Microsoft Windows
- Minecraft

    - URL / Introduction to Minecraft Pi
- Minecraft Pi
    - defining / Introduction to Minecraft Pi
    - playing / Playing Minecraft Pi
    - URL / Playing Minecraft Pi
    - Movement control / Movement control in Minecraft Pi
    - Action control / Action control in Minecraft Pi
    - other controls, defining / Other controls in Minecraft Pi
    - Python programming, used for / Python programming for Minecraft Pi
- Minecraft Pi Python API
    - URL / Python programming for Minecraft Pi
- MMD-1 (Mini Micro Designer 1)
    - about / Single-board computers
- mobile optimized / Creating the web page
- Model 2 / Raspberry Pi Model B+ and Model 2
- Model 2 board / So which one?
- Model A / Raspberry Pi Model A
- Model B / Raspberry Pi Model B
- Model B+ / Raspberry Pi Model B+ and Model 2
- Modules file / Enabling the I2C Bus
- morphological transformations
    - on images / Morphological transformations on images
- motion detection
    - about / Motion detection and tracking
- MP4Box / Be a video star
- MPI4Py

    - configuring / Installing and configuring MPICH2 and MPI4PY
    - installing / Installing and configuring MPICH2 and MPI4PY
- MPI4PY
    - installing / Installing MPI4PY
- mpi4py library
    - URL / Installing MPI4PY
- MPICH2
    - installing / Installing and configuring MPICH2 and MPI4PY
    - configuring / Installing and configuring MPICH2 and MPI4PY
- MPICH library
    - installing / Installing the MPICH library
    - URL / Installing the MPICH library
- multicamera set-up
    - about / The multicamera setup
    - Slave driver / The Slave driver
- MySQL
    - installing / Installing PHP and MySQL

N

- N-Body simulation
    - defining / Introducing N-Body simulations
- nano

    - URL / Getting started with Python

  / Getting flashy–
- Nano / Getting a bit of help first
- NCAT
    - and UDP servers / UDP servers and NCAT
- NCAT utility
    - about / UDP servers and NCAT
- networking concepts
    - about / A few networking concepts
- Network Time Protocol (NTP) / Getting the right time, ntp
- Newark element14
    - URL / Raspberry Pi
- night time shots
    - capturing / Night vision
    - TIP120 Darlington transistor, using / An illuminating experience
    - elaborate light switch / The Elaborate light switch re-visited
    - infra-red lighting, using / Is that a badger?
    - badger, watching / Is that a badger?
- Nmap
    - URL / Automating node discovery in a network
    - installing / Installing the operating system
- No-IP
    - URL / Setting up a dynamic DNS account

- node discovery
    - automating, in network / Automating node discovery in a network
- NoIR Camera module
    - working with / Working with the Pi Camera and NoIR Camera modules
- NoIR camera module / Night vision
- noise
    - defining / Noise
    - removing, kernels used / Kernels for noise removal
    - 2D convolution filtering / 2D convolution filtering
    - low pass filtering / Low pass filtering
- normally closed (NC) / Connecting our PIR motion sensor
- normally open (NO) / Connecting our PIR motion sensor
- NumPy
    - defining / Introducing NumPy
    - array creation / Array creation
    - array operations / Basic operations on arrays
    - linear algebra examples / Linear algebra
    - URL / Linear algebra

O

- object location
    - tracking / Color and motion finding
- OLED display
    - URL / Introducing the GrovePi

- on-board real-time clock hardware
  - obtaining / Getting the right time, fake-hwclock
  - fake-hwclock package / fake-hwclock
  - Network Time Protocol (NTP) / ntp
- OpenCV
  - defining / Introducing OpenCV, Edge Detection and OpenCv
  - URL / Introducing OpenCV
  - Pi, setting up for / Setting up Pi for Computer Vision
  - used, for working with webcam / Working with Webcam using OpenCV
  - used, for saving video / Saving a video using OpenCV
  - and Picamera / Pi Camera and OpenCV
  - downloading / Downloading and installing OpenCV – a fully featured vision library
  - installing / Downloading and installing OpenCV – a fully featured vision library
- OpenCV, with Raspberry Pi
  - URL / Motion detection and tracking
- OpenCV homepage
  - URL / Introducing OpenCV
- OpenCV installation
  - testing, with Python / Testing the OpenCV installation with Python
- Open Source Computer Vision (OpenCV)
  - about / Introducing OpenCV
- operating system (OS)

    - about / The basics of sockets
- operating systems
    - about / Operating systems
    - Raspbian / Raspbian
- Otsu's method
    - defining / Otsu's method
- outputs
    - allocating / Being in control
    - documenting / Being in control

P

- parallel-to-serial conversion / Serial-to-parallel conversion
- path finding algorithms
    - URL / Avoiding obstacles
- PCF8591
    - URL / A simple temperature sensor
- PHP
    - installing / Installing PHP and MySQL
- PHP5 installation
    - testing / Testing the PHP5 installation
- PHP support / Installing the web server
- Pi
    - DNS server, setting up / Setting up a DNS server on the Pi

    - types / Which flavor of Pi?
    - Model A / Raspberry Pi Model A
    - Model B / Raspberry Pi Model B
    - Model B+ / Raspberry Pi Model B+ and Model 2
    - Model 2 / Raspberry Pi Model B+ and Model 2
    - models, comparing / Model comparison table
    - version, selecting / So which one?
    - URL / So which one?
    - booting / Booting your Pi
    - file system, expanding / Expanding the file system
    - setting up / Setting up your Pi
    - operating system, updating / Getting up to date
    - on-board real-time clock hardware, obtaining / Getting the right time
    - security / Talking of security–
    - sudo / What is this sudo thing anyway?
- Pi based supercomputer
    - defining / Introducing a Pi-based supercomputer
- Picamera
    - and OpenCV / Pi Camera and OpenCV
- Pi Camera
    - working with / Working with the Pi Camera and NoIR Camera modules
- picamera
    - about / Using picamera in Python with the Pi Camera module, Installing RaspiCam on Raspberry Pi
- Pi Camera and Pi NoIR camera
    - references / Working with the Pi Camera and NoIR Camera modules

- picamera module
    - URL / Using picamera in Python with the Pi Camera module
- PiFace Digital I/O expansion board
    - about / PiFace Digital I/O expansion board
    - URL / PiFace Digital I/O expansion board
- PiGlow
    - installing / Installing PiGlow
    - using / Using PiGlow
    - URL / Using PiGlow
- pins, GPIO / Connecting a digital compass to the Raspberry Pi
- PIR sensors
    - prerequisites / Prerequisites
    - about / Passive infrared sensors explained, Setting up your PIR sensor
    - setting up / Setting up your PIR sensor
    - guidelines / Setting up your PIR sensor
    - power supply / Give me power (again)
    - connecting / Connecting our PIR motion sensor
- PodSixNet
    - about / Adding a joystick remote control
    - URL / Adding a joystick remote control
- Polulu
    - URL / A basic stable pose
- Polulu SW
    - URL / Communicating with the servo controller with a PC

- port-forwarding / Port-forwarding
- positions
    - adjusting, of servos / A basic stable pose
- power supply
    - about / Give me power
    - part requisites / Give me power
- prerequisites, expansion board / Prerequisites
- prerequisites, PIR sensors / Prerequisites
- prerequisites, Raspberry Pi
    - setting / Setting up the prerequisites
    - camera, setting up / Setting up and testing the camera
    - camera, testing / Setting up and testing the camera
    - hardware button, adding / Adding the hardware button
- pressure mat sensors / Getting into the zone
- program
    - creating, for controlling biped / Creating a program to control your biped
- programming, on Raspberry Pi
    - defining / Programming on Raspberry Pi
- programming concepts
    - URL / Creating and running Python programs on the Raspberry Pi
- Pulse-Width-Modulated (PWM)
    - about / How servo motors work
- Putty

    - URL / Installing the operating system
    - downloading / Installing the operating system
- PuTTY
    - URL, for download / Setting up your Pi
- PyGame
    - defining / Introducing PyGame
    - about / Introducing PyGame
    - modules, defining / Introducing PyGame
    - URL / Introducing PyGame
    - installing / Installing PyGame
- pygame library
    - URL / Adding a joystick remote control
- Python
    - defining / Getting started with Python
    - URL / Getting started with Python
    - turtle programming, using with / Turtle programming with Python
    - OpenCV installation, testing with / Testing the OpenCV installation with Python
    - Twitter API, installing for / Installing the Twitter API for Python
    - SQLite database, setting up / Setting up a SQLite database in Python
    - Telnet client / A Telnet client in Python
- python library
    - URL / Accessing the compass programmatically
- Python OpenCV API documentation
    - URL / Logical operations on images

- Python UDP sockets
  - used, for echo server / An echo server using Python UDP sockets

R

- RAM-based file system
  - creating / Creating a RAM-based file system
- Raspberry Pi
  - defining / Raspberry Pi
  - URL / Raspberry Pi, Preparing MicroSD card manually, Working with the Pi Camera and NoIR Camera modules, Installing the operating system
  - references / Raspberry Pi, Raspberry Pi models, Setting up the Raspberry Pi
  - models / Raspberry Pi models
  - setting up / Setting up the Raspberry Pi
  - MicroSD card, preparing manually / Preparing MicroSD card manually
  - instructions, for Windows users / Preparing MicroSD card manually
  - booting / Booting up our Pi for the first time
  - shutting down / Shutting down and rebooting Pi safely
  - rebooting / Shutting down and rebooting Pi safely
  - updating / Updating the Pi
  - configuring, for acting as DHCP server / Configuring a Raspberry Pi to act as a DHCP server
  - static IP, setting up / Setting up a static IP on your Raspberry Pi
  - remote administration / Remote administration for our Raspberry Pi
  - about / Conclusion

    - configuring / Configuring Raspberry Pi – the brain of your robot
    - Python programs, creating on / Creating and running Python programs on the Raspberry Pi
    - Python programs, running on / Creating and running Python programs on the Raspberry Pi
    - servo controller, connecting to / Connecting the servo controller to the Raspberry Pi
    - connecting, to infrared sensor / Connecting Raspberry Pi to an infrared sensor
    - connecting, to USB sonar sensor / Connecting Raspberry Pi to a USB sonar sensor
    - digital compass, connecting to / Connecting a digital compass to the Raspberry Pi
    - USB camera, installing on / Installing a USB camera on Raspberry Pi
    - RaspiCam, installing on / Installing RaspiCam on Raspberry Pi
- Raspberry Pi 3
    - defining / The Raspberry Pi 3
    - references / The Raspberry Pi 3
- Raspberry Pi camera module
    - raspistill, using / Using raspistill and raspivid
    - raspivid, using / Using raspistill and raspivid
    - picamera used, in Python with / Using picamera in Python with the Pi Camera module
    - versus webcam / The Pi camera versus the webcam
    - about / The Raspberry Pi camera module
    - connecting / Connecting the camera module
    - setting up / Setting up the camera module
    - testing / Testing the camera module
- Raspberry Pi cluster
    - setting up / Setting up the Raspberry Pi cluster

- Raspberry Pi Models
  - URL / Working with the Pi Camera and NoIR Camera modules
- Raspberry Pi Networking Cookbook
  - URL / Connecting via Wi-Fi
- Raspberry Pi Zero
  - defining / The Raspberry Pi Zero
  - references / The Raspberry Pi Zero
- Raspbian
  - about / Raspbian
  - URL / Raspbian
  - Apache, installing on / Introducing and installing Apache on Raspbian
- Raspbian image
  - downloading, URL / Downloading the Raspbian image
- Raspbian operating system
  - installing / Installing the operating system
- Raspbian OS
  - URL / Downloading the Raspbian image
- RaspiCam
  - installing, on Raspberry Pi / Installing RaspiCam on Raspberry Pi
- RaspiCam, on Raspberry Pi

    - URL / Installing RaspiCam on Raspberry Pi
- raspistill options / Where was that taken?
- ready-made expansion boards
    - about / Using ready-made expansion boards
    - using / Using ready-made expansion boards
    - Hobbytronics MCP23017 expander port kit / Hobbytronics MCP23017 expander port kit
    - PiFace Digital I/O expansion board / PiFace Digital I/O expansion board
    - Gertboard / Gertboard
- Real VNC
    - URL / Adding a remote graphical user interface
- receiver
    - wiring diagram / The receiver wiring diagram
- rectifier diode / Driving inductive loads
- red, green, and blue (RGB) / Color and motion finding
- reed switch
    - about / The working of magnetic contact sensors
    - normally open (NO) / The working of magnetic contact sensors
    - normally closed (NC) / The working of magnetic contact sensors
- references
    - defining / References
- remote access, granting to control panel
    - about / Remote access to our control panel
    - dynamic DNS account, setting up / Setting up a dynamic DNS account

    - static IP, setting up on Raspberry Pi / Setting up a static IP on your Raspberry Pi
    - port-forwarding / Port-forwarding
- remote administration, Raspberry Pi
    - about / Remote administration for our Raspberry Pi
    - Webmin / Getting Webmin
- remote graphical user interface
    - adding / Adding a remote graphical user interface
    - wireless access, establishing / Establishing wireless access
- root user / What is this sudo thing anyway?
- RS Components
    - URL / Raspberry Pi

S

- SD Card
    - preparing / Preparing the SD card
    - Raspbian OS image, downloading / Downloading the Raspbian image
    - Microsoft Windows, using / Using Microsoft Windows
    - Linux, using / Using Linux
- SD card
    - preserving / Preserving the SD card
    - RAM-based file system, creating / Creating a RAM-based file system
- security / Port-forwarding
- security keypad / Arming and disarming the system
- Sensor Pin

    - about / Connecting Raspberry Pi to an infrared sensor
- sensors
    - hall-effect sensors / Getting into the zone
    - pressure mat sensors / Getting into the zone
- serial-to-parallel conversions / Serial-to-parallel conversion
- Serial Peripheral Interface (SPI) bus / The SPI bus
- servo controller
    - used, for controlling servos / Using a servo controller to control the servos
    - communicating, with PC / Communicating with the servo controller with a PC
    - connecting, to Raspberry Pi / Connecting the servo controller to the Raspberry Pi
    - URL / Creating a program to control your biped
- Servo motors
    - working / How servo motors work
- servo positions
    - defining / A basic stable pose
- servos
    - controlling, servo controller used / Using a servo controller to control the servos
- servo settings
    - about / A basic walking motion
- setup, for web server
    - configuring / Configuring the setup for a web server

- Sharp 2Y0A02 model
  - about / Connecting Raspberry Pi to an infrared sensor
- shift registers / Serial-to-parallel conversion
- Simple DirectMedia (SDL)
  - about / Introducing PyGame
- simple temperature sensor
  - about / A simple temperature sensor
  - working / How it works
- simple water detector
  - about / A simple water detector
  - working / How it works
- single-board computers
  - defining / Single-board computers
- snake game
  - building / Building a snake game
- socket programming
  - in Python / Looking back
- sockets
  - defining / The basics of sockets
- socket servers
  - programming / Programming socket servers
  - socket, binding / Binding a socket

    - incoming connections, listening / Listening for incoming connections
    - multiple connections, handling / Handling multiple connections
- soft-switch function / An arm/disarm switch
- sonar sensors
    - about / Connecting Raspberry Pi to a USB sonar sensor
    - setting up / Connecting Raspberry Pi to a USB sonar sensor
- speech capabilities
    - adding, to weather station / Adding speech capabilities to our weather station
- SQLite (pronounced Sequel Lite)
    - about / Setting up a SQLite database in Python
- SQLite3 module
    - URL / Setting up a SQLite database in Python
- SQLite database
    - setting up, in Python / Setting up a SQLite database in Python
- SSH access
    - setting up, from host to client / Setting up SSH access from the host to the client
- static IP address
    - setting, on Raspberry Pi / Setting up a static IP on your Raspberry Pi

- stop-motion animation
    - defining / Introducing stop-motion animation
- sudoer / I'm someone else
- sudo rights / Accessing Webmin locally
- system
    - arming / Arming and disarming the system
    - disarming / Arming and disarming the system

T

- TCP
    - and UDP, comparing / The difference between TCP and UDP
- TCP sockets
    - architecture / The architecture of TCP sockets
    - TCP socket, creating / Creating a TCP socket
    - server, connecting with / Connecting to a server with a TCP socket
    - data, receiving from server / Receiving data from the server
- thresholding
    - about / Thresholding images
- Tightvncserver
    - about / Adding a remote graphical user interface
- Timelapse photography
    - about / Creating a timelapse sequence using fswebcam

- TIP120 Darlington transistor / An illuminating experience
- torque
  - about / Building the biped platform
- tracking system
  - about / Motion detection and tracking
- Transmission Control Protocol (TCP)
  - about / The difference between TCP and UDP
- turn, for robot
  - defining / A basic turn for the robot
- turtle API
  - URL / Turtle programming with Python
- tutorials, C programming
  - URL / An introduction to the C/C++ programming language
- Tweepy
  - about / Installing the Twitter API for Python
  - URL / Installing the Twitter API for Python
  - references / Installing the Twitter API for Python
  - using / Using Tweepy
- tweeting weather station
  - building / Building a tweeting weather station
- Twitter
  - URL / Using Tweepy

- Twitter API
  - installing, for Python / Installing the Twitter API for Python

U

- 433-Util software project
  - URL / The alternative approach (because we have no choice)
- UDP
  - and TCP, comparing / The difference between TCP and UDP
  - data, sending with / Sending and receiving data with UDP
  - data, receiving with / Sending and receiving data with UDP
- UDP client
  - defining / A UDP client
- UDP servers
  - and NCAT / UDP servers and NCAT
- UDP sockets
  - architecture / The architecture and programming of UDP sockets
  - programming / The architecture and programming of UDP sockets
- Ultrasonic ranger
  - URL / Introducing the GrovePi
- Un-used Modules section / Accessing Webmin locally

- Universal Asynchronous Receiver and Transmitter (UART) bus / The UART serial bus
- Universal Serial Bus (USB) ports / USB ports
- USB-ProxSonar-EZ sensor
    - about / Connecting Raspberry Pi to a USB sonar sensor
- USB camera
    - installing, on Raspberry Pi / Installing a USB camera on Raspberry Pi
- USB connection
    - URL / Connecting the servo controller to the Raspberry Pi
- USB interface
    - URL / Connecting Raspberry Pi to an infrared sensor
- USB sonar sensor
    - Raspberry Pi, connecting to / Connecting Raspberry Pi to a USB sonar sensor
- USB webcam
    - about / Using USB cameras
    - using / Using USB cameras
    - installing / Installing the webcam
    - URL / Installing the webcam
    - snap, capturing / Taking a snap
    - fswebcam, setting up / Snap snap snap
- USB webcams
    - URL / Working with webcams
    - reference / Working with webcams

- User Datagram Protocol (UDP)
    - about / The difference between TCP and UDP

V

- 3.3V voltage regulator / Give me power
- 12V alarm zone circuits
    - using / 12V alarm zone circuits
    - protecting / Alarm circuit protection
    - working / How it works
- video
    - rendering / Rendering the video
    - capturing / Be a video star
    - using, in security system / Caught on camera
- VideoLAN
    - URL / Be a video star
- vim
    - URL / Getting started with Python
- vision hardware
    - USB webcam, adding / Installing a camera on your biped robot
    - RaspiCam, adding / Installing a camera on your biped robot
- visudo utility / I'm someone else
- Vncserver
    - URL / Adding a remote graphical user interface
    - about / Adding a remote graphical user interface

- voltage divider
    - URL / Connecting Raspberry Pi to a USB sonar sensor

W

- walking motion, robot
    - defining / A basic walking motion
- walking robots
    - building / Building robots that can walk
- webcam
    - versus Raspberry Pi camera module / The Pi camera versus the webcam
- webcams
    - working with / Working with webcams
    - crontab / Crontab
    - timelapse sequence creating, fswebcam used / Creating a timelapse sequence using fswebcam
    - video recording and playback / Webcam video recording and playback
- Webmin
    - URL / Getting Webmin, Remotely accessing Webmin
    - about / Getting Webmin
    - repository sources, updating / Updating the repository sources
    - signing key, importing / Importing the signing key
    - accessing locally / Accessing Webmin locally
    - root user account / Accessing Webmin locally
    - pi user / Accessing Webmin locally

    - remotely accessing / Remotely accessing Webmin
- web page
    - creating / Creating the web page
    - control panel HTML template / The control panel HTML template
    - styling / Giving it some style
    - modifying dynamically / Making it dynamic
    - PHP configuration file setting, modifying / Getting a bit of help first
    - main PHP code / The main PHP code
    - running, as different user / I'm someone else
- web server
    - installing / Installing the web server
    - PHP5 installation, testing / Testing the PHP5 installation
- Wi-Fi
    - connecting via / Connecting via Wi-Fi
- Win32 Disk Imager utility
    - URL / Using Microsoft Windows
- Win32DiskImager installer
    - URL / Preparing MicroSD card manually
- windows / The working of magnetic contact sensors
- WinSCP / Getting flashy–
- WinZip
    - URL / Preparing MicroSD card manually
- wireless devices
    - URL / Establishing wireless access

- wireless dongle
    - adding / Adding a wireless dongle and creating an access point
- wireless PIR sensors
    - about / Wireless PIR motion sensors
    - 433-MHz wireless alarm systems / 433-MHz wireless alarm systems
    - 433-MHz receiver, connecting / Connecting a 433-MHz receiver
    - receiver wiring diagram / The receiver wiring diagram
- WordPress
    - installing / Installing WordPress
    - references / Installing WordPress
    - URL / Configuring the WordPress installation
- WordPress installation
    - configuring / Configuring the WordPress installation

Z

- zones
    - creating / Getting into the zone