



Community Experience Distilled

Raspberry Pi Robotic Projects

Create amazing robotic projects on a shoestring budget

Richard Grimmett

[PACKT]
PUBLISHING

Table of Contents

[Raspberry Pi Robotic Projects](#)

[Credits](#)

[About the Author](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Getting Started with Raspberry Pi](#)

[Getting started](#)

[The unveiling](#)

[Hooking up a keyboard, mouse, and display](#)

[Installing the operating system](#)

[Accessing the board remotely](#)

[Summary](#)

[2. Programming Raspberry Pi](#)

[Basic Linux commands on Raspberry Pi](#)

[Creating, editing, and saving files on Raspberry Pi](#)

[Creating and running Python programs on Raspberry Pi](#)

[Basic programming constructs on Raspberry Pi](#)

[The if statement](#)

[The while statement](#)

[Working with functions](#)

[Libraries/modules in Python](#)

[The object-oriented code](#)

[Introduction to the C/C++ programming language](#)

[Summary](#)

[3. Providing Speech Input and Output](#)

[Hooking up the hardware to make and input sound](#)

[Using Espeak to allow our projects to respond in a robot voice](#)

[Using PocketSphinx to accept your voice commands](#)

[Interpreting commands and initiating actions](#)

[Summary](#)

[4. Adding Vision to Raspberry Pi](#)

[Connecting the USB camera to Raspberry Pi and viewing the images](#)

[Downloading and installing OpenCV – a fully featured vision library](#)

[Using the vision library to detect colored objects](#)

[Summary](#)

[5. Creating Mobile Robots on Wheels](#)

[Gathering the required hardware](#)

[Using a motor controller to control the speed of your platform](#)

[Controlling your mobile platform programmatically using Raspberry Pi](#)

[Making your mobile platform truly mobile by issuing voice commands](#)

[Summary](#)

[6. Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs](#)

[Gathering the hardware](#)

[Connecting Raspberry Pi to the mobile platform using a servo controller](#)

[Connecting the hardware](#)

[Configuring the software](#)

[Creating a program in Linux to control the mobile platform](#)

[Making your mobile platform truly mobile by issuing voice commands](#)

[Summary](#)

[7. Avoiding Obstacles Using Sensors](#)

[Gathering the hardware](#)

[Connecting Raspberry Pi to an infrared sensor](#)

[Connecting Raspberry Pi to a USB sonar sensor](#)

[Connecting the hardware](#)

[Using a servo to move a single sensor](#)

[Summary](#)

[8. Going Truly Mobile – The Remote Control of Your Robot](#)

[Gathering the hardware](#)

[Connecting Raspberry Pi to a wireless USB keyboard](#)

[Using the keyboard to control your project](#)

[Working remotely with your Raspberry Pi through a wireless LAN](#)

[Working remotely with your Raspberry Pi through ZigBee](#)

[Summary](#)

[9. Using a GPS Receiver to Locate Your Robot](#)

[Connecting Raspberry Pi to a GPS device](#)

[Accessing the GPS programmatically](#)

[Summary](#)

[10. System Dynamics](#)

[Getting started](#)

[Creating a general control structure](#)

[Using the structure of the Robot Operating System to enable complex functionalities](#)

[Summary](#)

[11. By Land, Sea, and Air](#)

[Using Raspberry Pi to sail](#)

[Getting started](#)

[Using Raspberry Pi to fly robots](#)

[Using Raspberry Pi to make the robot swim underwater](#)

[Summary](#)

[Index](#)

Raspberry Pi Robotic Projects

Raspberry Pi Robotic Projects

Copyright © 2014 Packt Publishing All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, 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 book.

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

First published: February 2014

Production Reference: 1140214

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-432-2

www.packtpub.com

Cover Image by John Michael Harkness (<jtothem@gmail.com>)

Credits

Author

Richard Grimmett

Reviewers

Hector Cuesta

Lihang Li

Masumi Mutsuda Zapater

Acquisition Editors

Usha Iyer

Anthony Albuquerque

Nikhil Karkal

Content Development Editor

Ruchita Bhansali

Technical Editors

Novina Kewalramani

Rohit Kumar Singh

Pratish Soman

Copy Editors

Tanvi Gaitonde

Dipti Kapadia

Shambhavi Pai

Project Coordinator

Amey Sawant

Proofreaders

Paul Hindle

Ameesha Green

Indexer

Monica Ajmera Mehta

Graphics

Yuvraj Mannari

Abhinash Sahu

Production Coordinator

Kyle Albuquerque

Cover Work

Kyle Albuquerque

About the Author

Richard Grimmett has been fascinated by computers and electronics from his very first programming project, which used Fortran on punch cards. He has a bachelor's and master's degree in electrical engineering and a PhD in leadership studies. He also has 26 years of experience in the radar and telecommunications industries (he even has one of the original brick phones). He now teaches computer science and electrical engineering at Brigham Young University - Idaho, where his office is filled with his many robotic projects. He recently completed a book on using BeagleBone Black for robotic projects, and is now excited to release this title for those who prefer Raspberry Pi.

I would certainly like to thank my wife Jeanne and family for providing me with a wonderful, supportive environment that encourages me to take on projects such as this one. I would also like to thank my students; they showed me that amazing things can be accomplished by those who are unaware of all the barriers.

About the Reviewers

Hector Cuesta is the author of the book *Practical Data Analysis*, Packt Publishing, and holds a BA in Informatics and an MSc in Computer Science. He provides consulting services for software engineering and data analysis and has more than nine years of experience in various industries including financial services, social networking, e-learning, and human resources.

Hector is fluent in several programming languages and computational tools such as Java, Python, R, C-CUDA, ActiveMQ, Hadoop, C#, D3.js, MongoDB, SQL, Weka, Cassandra, OpenMP, and MPI.

He is a lecturer in the Department of Computer Science at the Autonomous University of Mexico State. His main research interests lie in computational epidemiology, machine learning, computer vision, high performance computing, big data, simulation, and data visualization.

He has done a research residency (Fall 2011) in Computational Epidemiology at University of North Texas in the Center of Computational Epidemiology and Response Analysis (CeCERA). He is currently working on modeling and simulation of infectious diseases using global stochastic cellular automata.

He has helped in the technical review of the books *Raspberry Pi Networking Cookbook*, Rick Golden, Packt Publishing, and *Hadoop Operations and Cluster Management Cookbook*, Shumin Guo, Packt Publishing. He is also a columnist with Software Guru and has published several scientific papers in international journals and conferences. He is an enthusiast of Lego Robotics and Raspberry Pi in his spare time.

You can follow him on Twitter at <https://twitter.com/hmCuesta>.

Lihang Li received his BE degree in Mechanical Engineering from Huazhong University of Science and Technology (HUST), China, in 2012 and is now pursuing his MS degree in Computer Vision from the National Laboratory of Pattern Recognition (NLPR) at the Institute of Automation, Chinese Academy of Sciences (IACAS).

As a graduate student, he is focusing on computer vision and specially on SLAM algorithms. In his free time, he likes to take part in open source activities and is now the President of Open Source Club at the Chinese Academy of Sciences. Also, building multicopter is his hobby, and he is with a team called OpenDrone from BLUG (Beijing Linux User Group).

Lihang has also reviewed *BeagleBone Robotic Projects*, *Richard Grimmett, Packt Publishing*, which has just been published. This is his second book as a reviewer.

His interests include Linux, open source, cloud computing, virtualization, computer vision algorithms, machine learning and data mining, and various programming languages.

You can know more about him on his personal website
<http://hustcalm.me>.

Many thanks to my girlfriend Jingjing Shao; it is her encouragement again and again that pushed me to be a reviewer for this book. I'm very appreciative of her kindness as sometimes I can't even spare time for her. Also, I would like to thank the entire team at Packt: Suraj, who is a very good project coordinator, and the other reviewers (though we haven't met); I'm really happy to have worked with you.

Lastly, to my parents: nothing would be possible without your kind support.

Masumi Mutsuda Zapater is a graduate of the Computer Science Engineering program at UPC University. He combines his artistic job as a voice actor with his technological job at itnig, an Internet startup accelerator. He is also a partner at Camaloon, an itnig accelerated startup that globally provides both custom designed and original products.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use

this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

The success of Arduino, Raspberry Pi, and other such processors has fueled a community of open source developers who now provide amazing capabilities at no cost. The continued support of the community now makes producing complex robotic projects something almost anyone with an interest in the area can do.

Raspberry Pi Robotic Projects is an attempt to organize that set of information and make it available to a wide audience of possible robotics developers. I can only hope it will help inspire a new generation that will be as comfortable with robots as this generation is with personal computers.

What this book covers

[Chapter 1](#), *Getting Started with Raspberry Pi*, will show you how to power up your Raspberry Pi, connect it to a keyboard, mouse, display, and a remote computer, and how to begin to access potential computing power.

[Chapter 2](#), *Programming Raspberry Pi*, will help you learn the basics of programming Raspberry Pi, both in Python and C programming languages.

[Chapter 3](#), *Providing Speech Input and Output*, will help you teach Raspberry Pi to both speak and listen.

[Chapter 4](#), *Adding Vision to Raspberry Pi*, will teach you to use a standard USB webcam to allow your robotic projects to see.

[Chapter 5](#), *Creating Mobile Robots on Wheels*, will show you how to connect Raspberry Pi to a mobile-wheeled platform and control its motors so that your robots can be mobile.

[Chapter 6](#), *Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs*, will show you how to make your robot able to walk.

[Chapter 7](#), *Avoiding Obstacles Using Sensors*, shows you how to sense the world around you. As your robot is now mobile, you'll want to avoid or find objects.

[Chapter 8](#), *Going Truly Mobile – The Remote Control of Your Robot*, shows you how to control your robot wirelessly as you'll want your robot to move around untethered by cables.

[Chapter 9](#), *Using a GPS Receiver to Locate Your Robot*, shows you how to use a GPS receiver so that your robot will know its location since your robot may get lost if it is mobile.

[Chapter 10](#), *System Dynamics*, focuses on how to bring together all the capabilities of the system to make complex robots.

[Chapter 11](#), *By Land, By Sea, and By Air*, teaches you how to add

capabilities to robots that sail, fly, and even go under water.

What you need for this book

Here is a partial list of the software you will need for the book:

- 7-zip: This software is a utility to archive and unarchive software
- Image Writer for Windows: This software is needed to write the images to an SD card
- WinSCP: This software provides the ability to transfer files to/from a PC
- PuTTY: This software allows the user remote access to Raspberry Pi
- VncServer/VNC Viewer: This software allows the user remote access to the graphical interface of the Raspberry Pi

Who this book is for

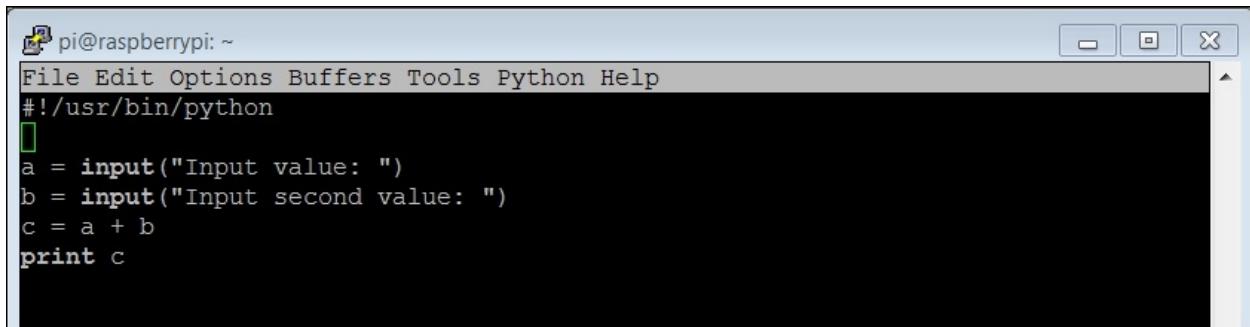
If you are curious about using Raspberry Pi to create robotics projects that previously have been the domain of research labs in major universities or defense departments, this is the book for you. Some programming background is useful, but if you know how to use a personal computer, with the aid of the step-by-step instructions in this book, you will be able to construct complex robotics projects that can move, talk, listen, see, swim, or fly.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive.

A block of code will be shown in screenshot as follows:

A screenshot of a terminal window titled "pi@raspberrypi: ~". The window has a standard title bar with icons for minimize, maximize, and close. The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "Python", and "Help". The main terminal area contains the following Python code:

```
#!/usr/bin/python
a = input("Input value: ")
b = input("Input second value: ")
c = a + b
print c
```

The code is written in black font on a dark background, with syntax highlighting for keywords like `input`, `print`, and variable names `a`, `b`, and `c`.

Any command-line input or output is written as follows:

```
sudodd if=2013-09-25-wheezy-raspbian.img of=/dev/sdX
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes appear in the text like this: "clicking the **Next** button moves you to the next screen."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via 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 on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, 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 all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

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 books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright 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

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Getting Started with Raspberry Pi

Raspberry Pi, with its low cost and amazing functionality package, has taken the robotic hobbyist community by storm. Unfortunately, many of them, especially those who are new to embedded systems and programming, can end up so discouraged that the board can end up on the shelf gathering dust next to floppy disks and chia pets.

Getting started

There is nothing as exciting as ordering and finally receiving a new piece of hardware. Yet things can go south quickly, even in the first few minutes. This chapter will hopefully help you avoid the pitfalls that normally accompany unpacking and configuring your Raspberry Pi. We'll step through the process, answer many of the different questions you might have, and help you understand what is going on. If you don't get through this chapter, then you'll not be successful with any of the others and your hardware will go unused, which would be a real tragedy. So let's get started!

One of the most challenging aspects of writing this guide was to decide the level at which I should describe each step. Some of you are beginners, others have some limited experience, and the rest will know significantly more in certain areas. I'll try to be brief but thorough, trying to detail the steps in order to be successful. So for this chapter, here are our objectives:

- Unbox and connect the board to power
- Connect a display, keyboard, and mouse
- Load and configure the operating system
- Access the board remotely

Note that Raspberry Pi comes in two flavors: A and B. Flavor B comes with additional input/output capability and will be the flavor we'll focus on in this book. That does not mean that many, if not most, of the projects

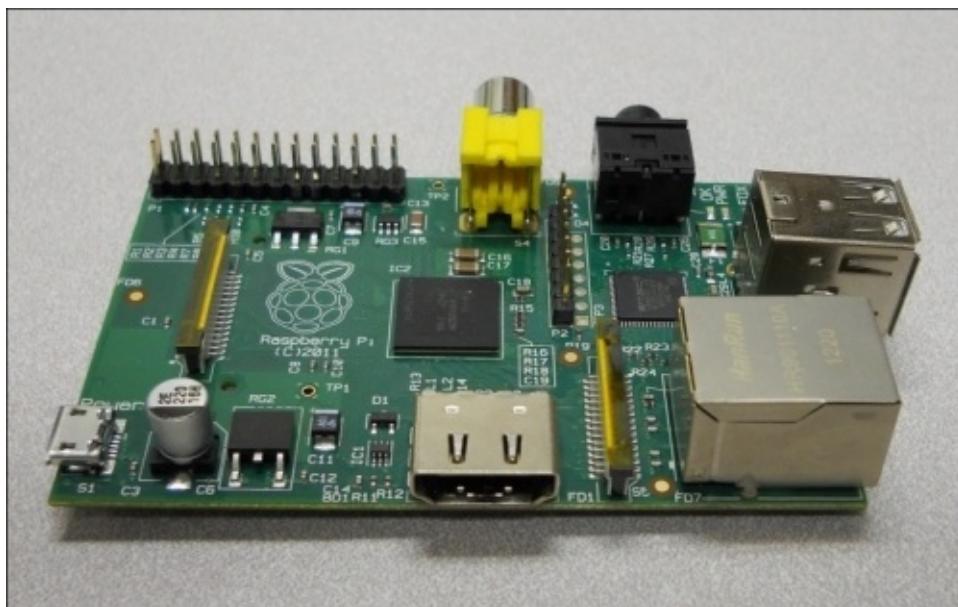
here require the extra capability of B. Rather, for the relative cost difference, flavor B provides enough benefits for me to assume that you have chosen this version. It will also make it simpler to explain how to use the board. Note that, the initial version of the Raspberry Pi model B had 256 MB of memory, and as of October 2012, the standard Raspberry Pi model B comes with 512 MB of memory.

Here are the items you'll need for this chapter's projects:

- A Raspberry Pi model B
- A USB cable to provide power to the board
- A display with proper video input
- A keyboard, mouse, and a powered USB hub
- An SD card with minimum 4 GB memory
- An SD card writer
- Another computer that is connected to the Internet
- An Internet connection for the board
- A LAN cable

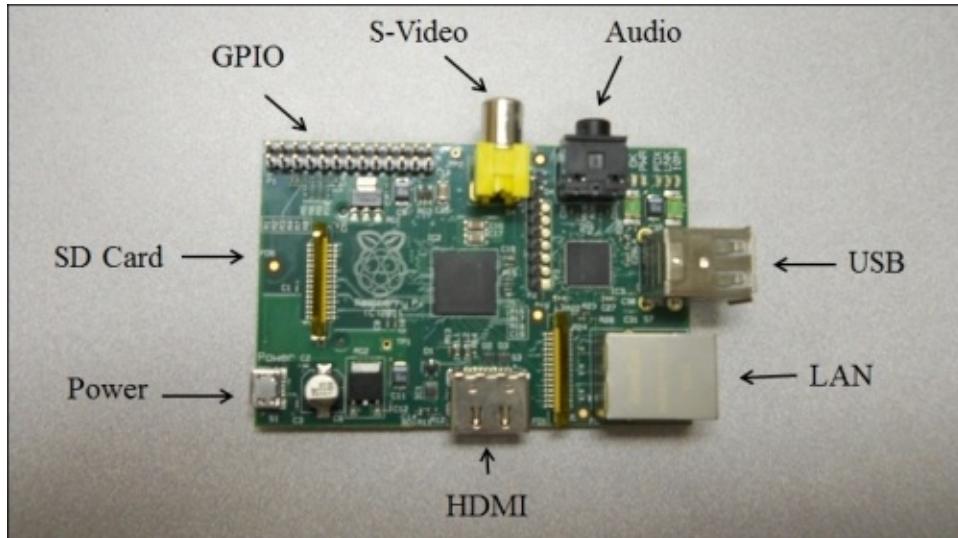
The unveiling

This is what the board should look like:



Before plugging in anything, inspect the board for any issues that may

have occurred during shipping. This is normally not a problem, but it is always good to do a quick visual inspection. You should also acquaint yourself with the different connections on the board; they are labelled for your information in the following image:

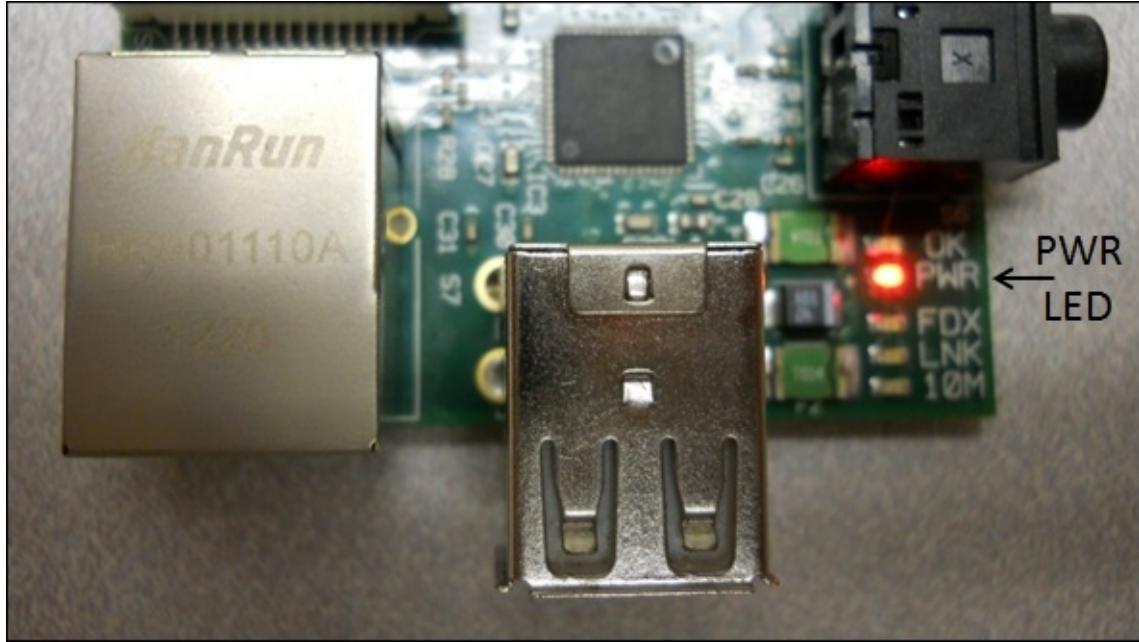


Let's first power the board. To do this, you'll need to go through the USB client connection. This is done by performing the following steps:

1. Connect the micro USB connector end of the cable to the board.
2. Connect the standard-sized USB connector either to a PC or to a compatible DC power source that has a USB connection.

If you are going to use a DC power source at the standard USB connector end, make sure that the unit can supply enough current. You'll need a power supply that can provide at least 1000 mA at 5 volts.

When you plug the board in, the **PWR** LED should be red. The following image is a close-up of the LED locations, so you're certain of which one to look for:



The **OK** LED will flash green when you install a card and boot up an operating system. The other three indicators are indications of a valid LAN connection. They will show/flash green or yellow, based on the availability of a valid LAN connection.

If you've reached this point, congratulations! You're now ready for the next step.

Hooking up a keyboard, mouse, and display

Now that your board works, you're going to add peripherals so that it can operate as a standalone computer system. This step is optional, as in future, your projects will often be in systems where you won't connect directly to the board with a keyboard, mouse, and display. However, this can be a great learning step, and is especially useful if you need to do some debugging on the system.

You'll need the following peripherals:

- A USB mouse
- A USB keyboard (this can be wireless and can contain a built-in mouse pad)
- A display that accepts HDMI, DVI Video, or SVideo inputs
- A powered USB hub (this is optional for this instance, but you most certainly will need it for future projects)

You may have most of this stuff already, but if you don't, there are some things to consider before buying additional equipment. Let's start with the keyboard and mouse. Most mice and keyboards have separate USB connectors. You'll notice, however, that your Raspberry Pi has only two USB ports. If you want to connect other devices, you may want to choose a keyboard that has a built-in mouse pad. That way, you only have one USB connection for both the devices.

You may also want to consider purchasing a powered USB hub. Before deciding on the hub to connect to your board, we need to understand the difference between a powered USB hub and one that gets its power from the USB port itself. Almost all USB hubs are not powered; that is, we don't plug in the USB hub separately. The reason for this is that almost all of these hubs are hooked up to computers with very large power supplies, and powering USB devices from the computer is not a problem. This is not the case for our board. The USB port on our board has very limited power capabilities, so if we are going to hook up devices that require significant power (for instance, a WLAN adapter or a webcam),

we're going to need a powered USB hub; one that provides power to the devices through a separate power source. The following is an image of such a device:



Notice that there are two connections on this hub. The one to the far right is a power connection, and it will be plugged into a battery with a USB port. The connection to the left is the USB connection, which will be plugged into Raspberry Pi.

Now, you'll also need a display. Fortunately, your Raspberry Pi offers lots of choices here. There are a number of different video standards; the following image is a reference to some of the most prevalent ones:

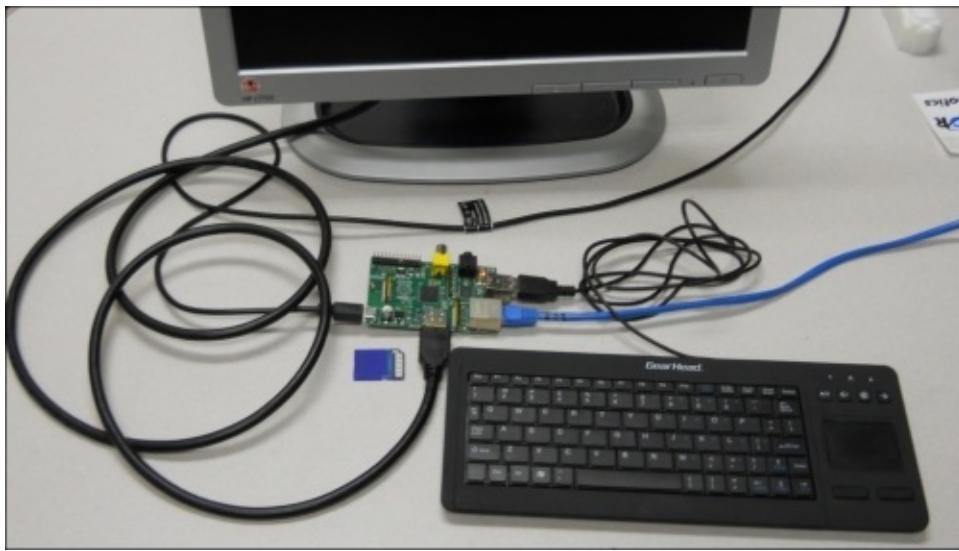


There is an SVideo output and an HDMI connector on Raspberry Pi. The easiest method to create a connection is to connect the board directly to a monitor or TV that has an SVideo or HDMI input; however, you'll need to buy a cable to go between the two. Check the video input on the monitor or TV; it will normally either have a set of RCA jack inputs, normally colored yellow (video), white (left-audio), and red (right-audio), or a multipin SVideo connector. If you are connecting using the RCA jack

inputs, connect the output of the Raspberry Pi video to the yellow input. The output of Raspberry Pi is an RCA jack output, so make sure you get a cable, with or without adapters, so that you can make the proper connection.

If you want to use the HDMI output, simply connect your cable with regular HDMI connections to Raspberry Pi and your TV or monitor that has an HDMI input connector. HDMI monitors are relatively new, but if you have a monitor that has a DVI input, you can buy adapters relatively inexpensively that provide an interface between DVI and HDMI. The display I use has a DVI input.

Don't be fooled by adapters that claim that they go from HDMI or DVI to VGA, or HDMI or DVI to -video. These are two different kinds of signals: HDMI and DVI are digital standards, and VGA and SVideo are analog standards. There are adapters that can do this, but they must contain circuitry and require power, so are significantly more expensive than any simple adapter. Now that you have all the bits and bobs, connect the USB hub to the standard USB port, the keyboard and mouse to the standard USB port, and the display to the proper connector as shown in the following image:



Once all these are connected, you are ready to plug in Raspberry Pi. I am using a standard USB 5 volt power supply. Make sure you connect all your devices before you switch on the unit. Most operating systems support the hot swap of devices, which means you are able to connect a

device after the system has been powered; but this is a bit shaky in the embedded environment. You should always cycle power when you connect new hardware.

Unfortunately, even though your hardware configuration is complete, you'll need to complete the next section to switch on the device. So let's figure out how to install an operating system.

Installing the operating system

Now that your hardware is ready, you need to install an operating system. You are going to install Linux, an open source version of Unix, on your Raspberry Pi. Now Linux, unlike Windows, Android, and IOS, is not tightly controlled by a single company. It is a group effort, mostly open source, and while it is available for free, it grows and develops a bit more chaotically.

Thus, a number of distributions have emerged, each built on a similar kernel or core set of capabilities. These core capabilities are all based on the Linux specification. However, they are packaged slightly differently and developed, supported, and packaged by different organizations; Debian, Arch, and Fedora are names of some of the versions. There are others as well, but these are the main choices for the distribution that you might put on your card.

I choose to use Raspbian, a Debian distribution of Linux, on my Raspberry Pi projects for a couple of reasons. First, the Debian distribution is used as the basis for another distribution, Ubuntu, and Ubuntu is arguably the most popular distribution of Linux, which makes it a good choice because of the community support it offers. Also, I personally use Ubuntu when I need to run Linux on my own personal computer. It provides a complete set of features, is well organized, and generally supports the latest hardware and software. Having roughly the same version on both my personal computer and my Raspberry Pi makes it easier for me to use both, as they operate, at least to a certain degree, in the same way. I can also try some things on my computer before trying them on Raspberry Pi. I've also found that Ubuntu/Debian has excellent support for new hardware, and this can be very important for your projects.

So, we are going to install and run a version of Debian, Raspbian, on our Raspberry Pi.

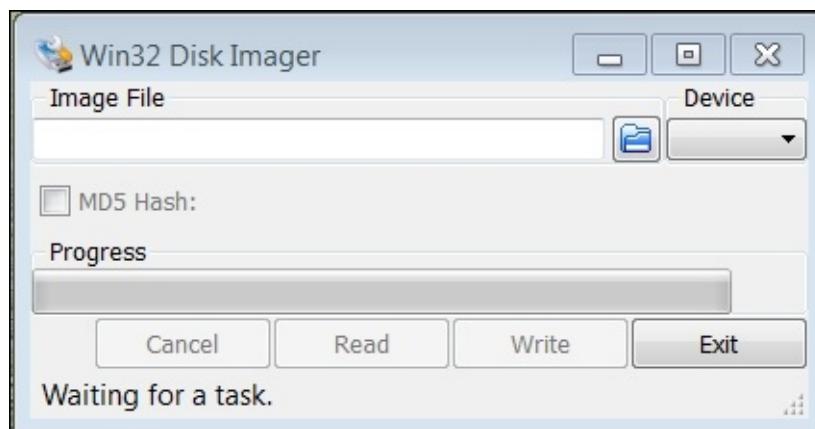
There are two approaches to getting Raspbian on our board. The board is getting popular enough for us to buy an SD card that already has Raspbian installed, or you can download it onto your personal computer

and then install it on the card. I'll assume you don't need any directions if you want to purchase a card—simply do an Internet search for companies selling such a product.

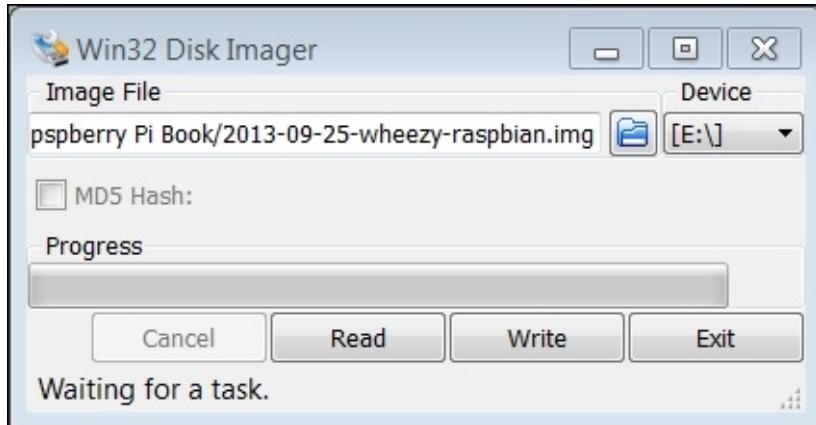
If you are going to download a distribution, you need to decide if you are going to use a Windows computer to download and create an SD card, or a Linux machine. I'll give brief directions for both here.

First, we'll need to download an image. This part of the process is similar for either Windows or Linux. Open a browser window, go to the Raspberry Pi organization's site www.raspberrypi.org, and select the **Downloads** selection at 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 user name and password; you'll need them later.

If you're using Windows, you'll need to unzip the file using an archiving program such as 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. I use the Image Writer for Windows program. You can find a link to this program at the top of this section on the www.raspberrypi.org website. Plug your card into the PC, run this program, and you should see the following screenshot:



Select the correct card and image; it should look something like the following screenshot:



Then select **Write**. This will take some time, but when 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 screenshot:

A screenshot of a terminal window on a Linux system. The prompt is "richard@vicki-automated:~". The user runs the command "ls -la /dev/sd*". The output shows five entries for drives sda through sda5. The last entry, "/dev/sda5", is highlighted in green.

```
richard@vicki-automated:~$ ls -la /dev/sd*
brw-rw---- 1 root disk 8, 0 Jul 4 10:34 /dev/sda
brw-rw---- 1 root disk 8, 1 Jul 4 10:34 /dev/sda1
brw-rw---- 1 root disk 8, 2 Jul 4 10:34 /dev/sda2
brw-rw---- 1 root disk 8, 5 Jul 4 10:34 /dev/sda5
richard@vicki-automated:~$
```

After plugging in your card, you might see something like the following screenshot:

A screenshot of a terminal window on a Linux system. The prompt is "richard@vicki-automated:~". The user runs the command "ls -la /dev/sd*". The output shows nine entries for drives sda through sdb2. The new drives, "/dev/sdb", "/dev/sdb1", and "/dev/sdb2", are highlighted in green.

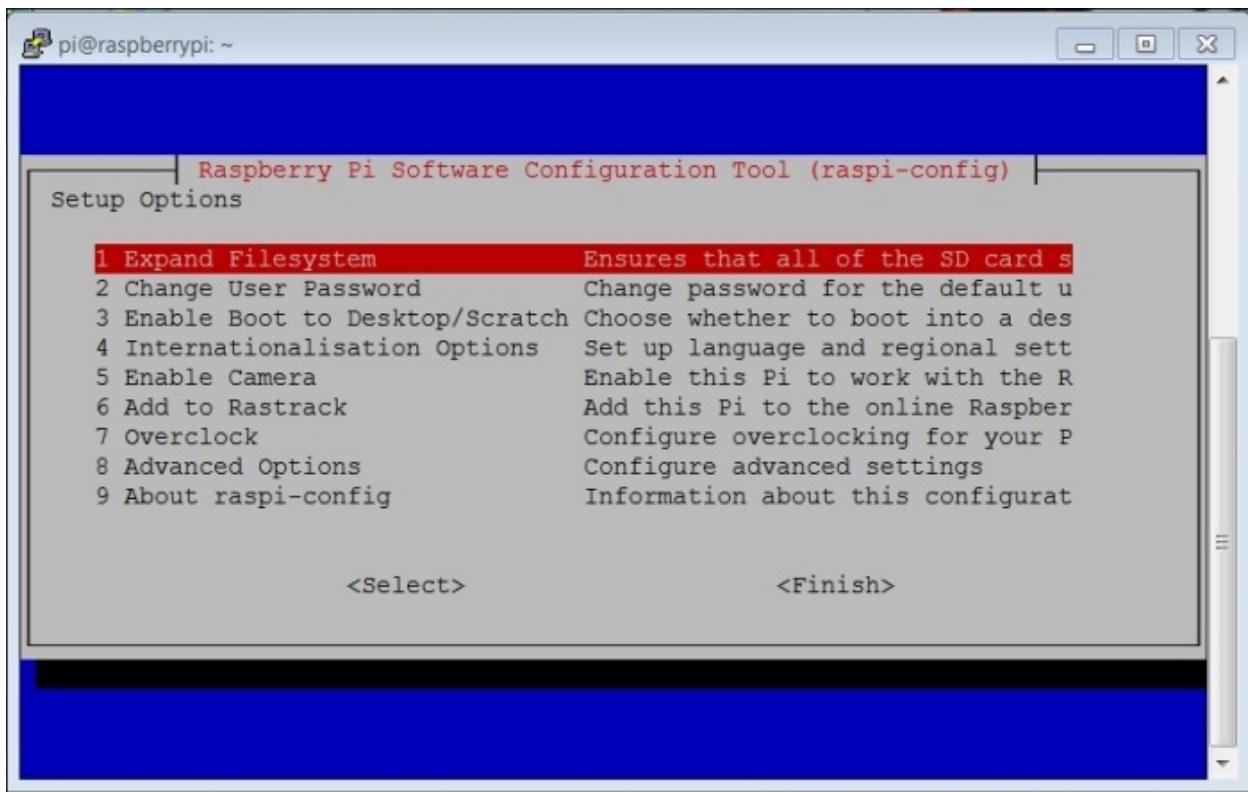
```
richard@vicki-automated:~$ ls -la /dev/sd*
brw-rw---- 1 root disk 8, 0 Jul 4 10:34 /dev/sda
brw-rw---- 1 root disk 8, 1 Jul 4 10:34 /dev/sda1
brw-rw---- 1 root disk 8, 2 Jul 4 10:34 /dev/sda2
brw-rw---- 1 root disk 8, 5 Jul 4 10:34 /dev/sda5
brw-rw---- 1 root disk 8, 16 Jul 11 09:50 /dev/sdb
brw-rw---- 1 root disk 8, 17 Jul 11 09:50 /dev/sdb1
brw-rw---- 1 root disk 8, 18 Jul 11 09:50 /dev/sdb2
richard@vicki-automated:~$
```

Notice that your card is at `sdb`. Now, go to the directory where you downloaded the archived image file and issue the following command:

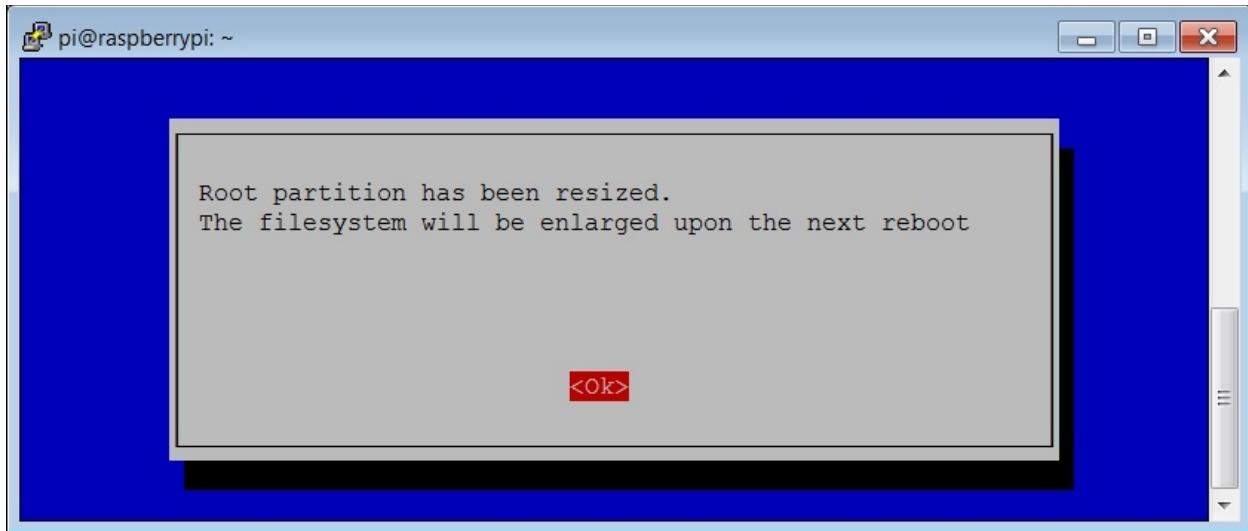
```
sudodd if=2013-09-25-wheezy-raspbian.img of=devsdx
```

The `2013-09-25-wheezy-raspbian.img` command will be replaced by the image file that you downloaded, and the `devsdx` command will be replaced by your card ID, in this example, `devsdb.Eject`. Once the file is written, eject the card and you are ready to plug it into the board and boot.

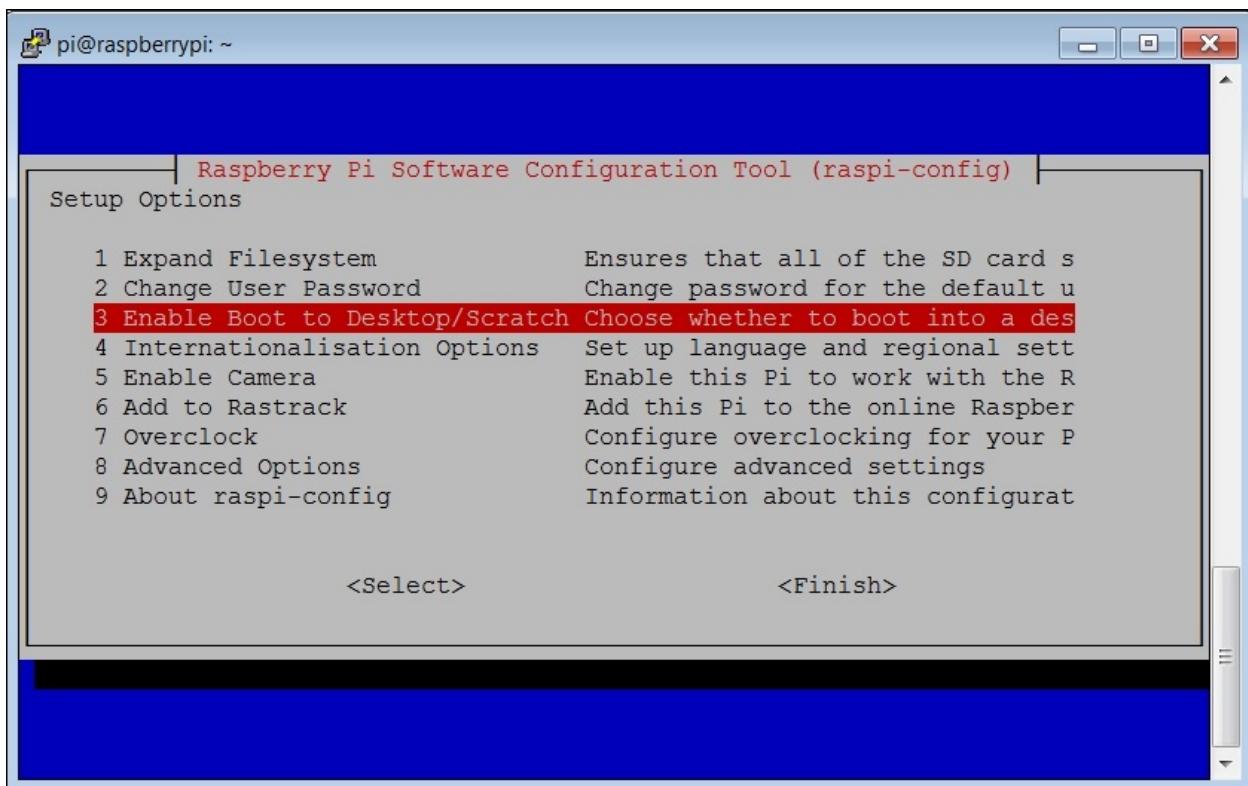
Make sure your Raspberry Pi is unplugged and insert the SD card into the slot. Then switch on the device. After the device boots, you should get a screen that looks like the following screenshot:



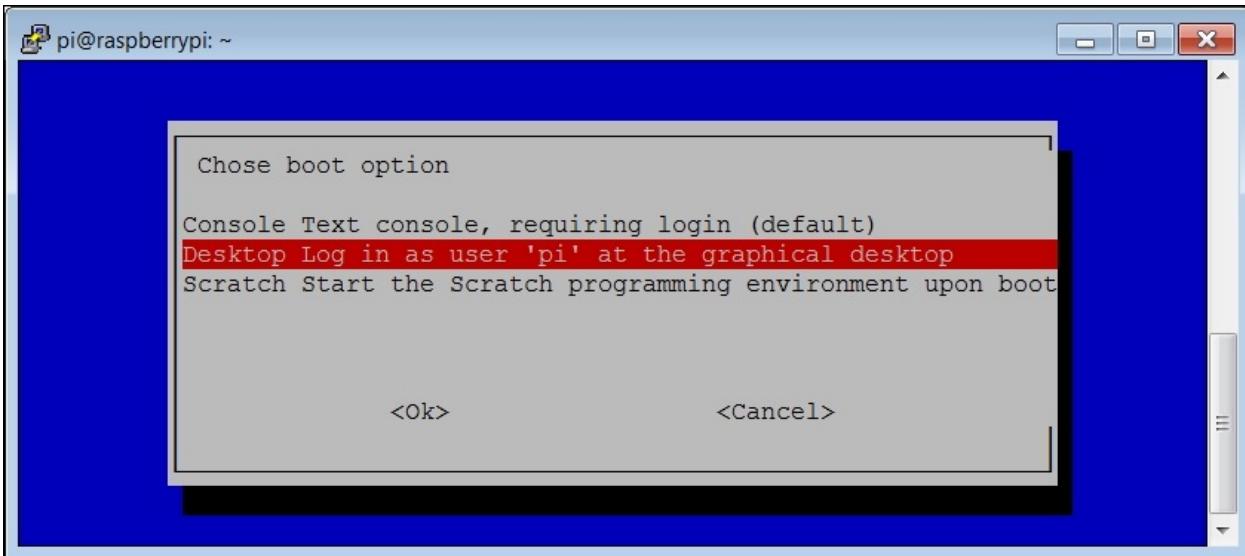
You are going to do two things, and perhaps something else as well, based on your personal preference. First, you'll want to expand the filesystem to take up the entire card. So hit the *Enter* key, and you'll see the following screenshot:



Hit *Enter* once again and you'll go back to the main configuration screen. Now select the **Enable Boot to Desktop/Scratch** option.

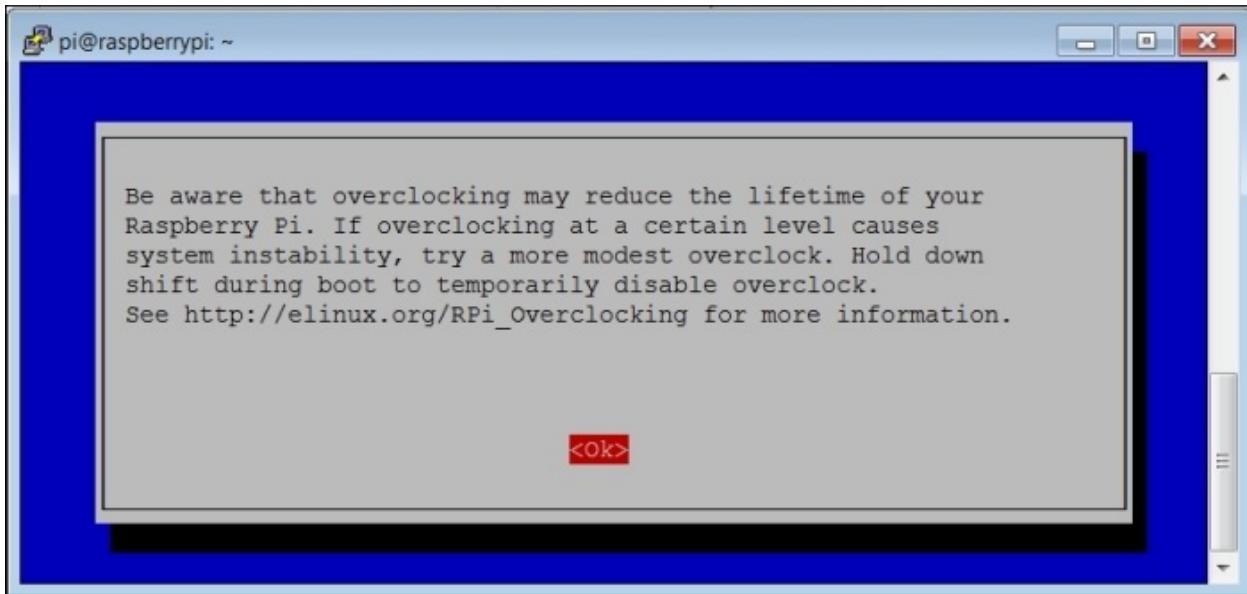


When you hit *Enter*, you'll see the following screenshot:

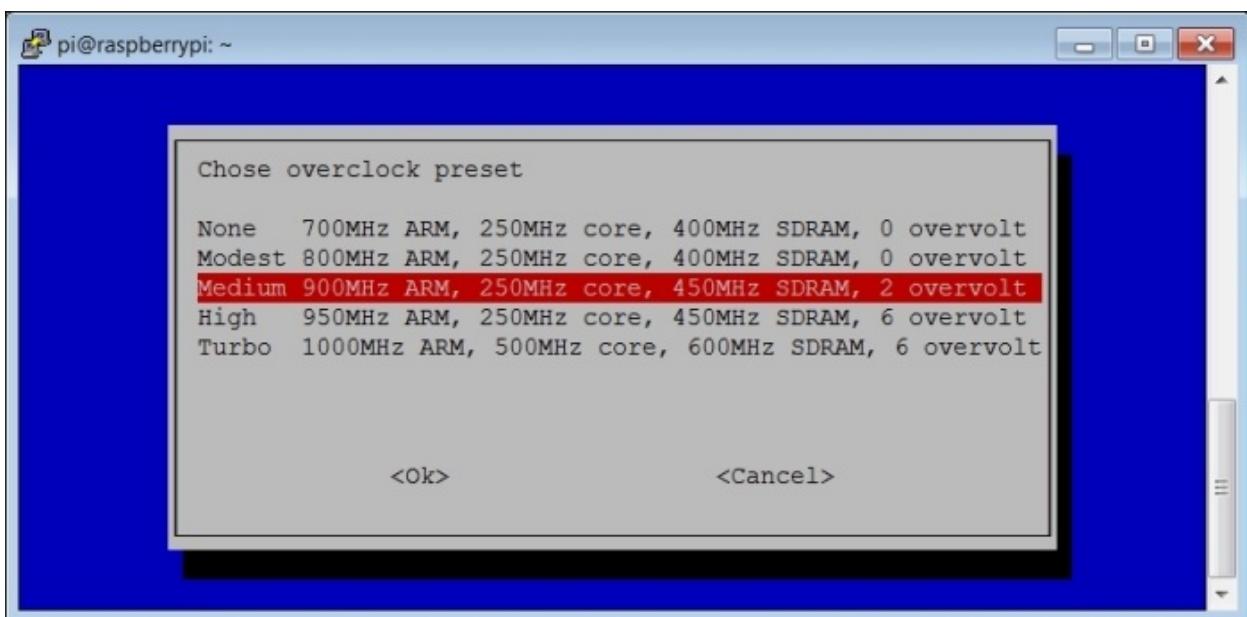


I prefer to select the middle selection, namely, **Desktop Log in as user 'pi' at the graphical desktop**. It normally sets up the system the way I like to have it boot up. You could also choose **Console Text console, requiring login (default)**; however, you will need to log in whenever you want to access the graphical environment, for example, while using the vncserver, which we will cover later.

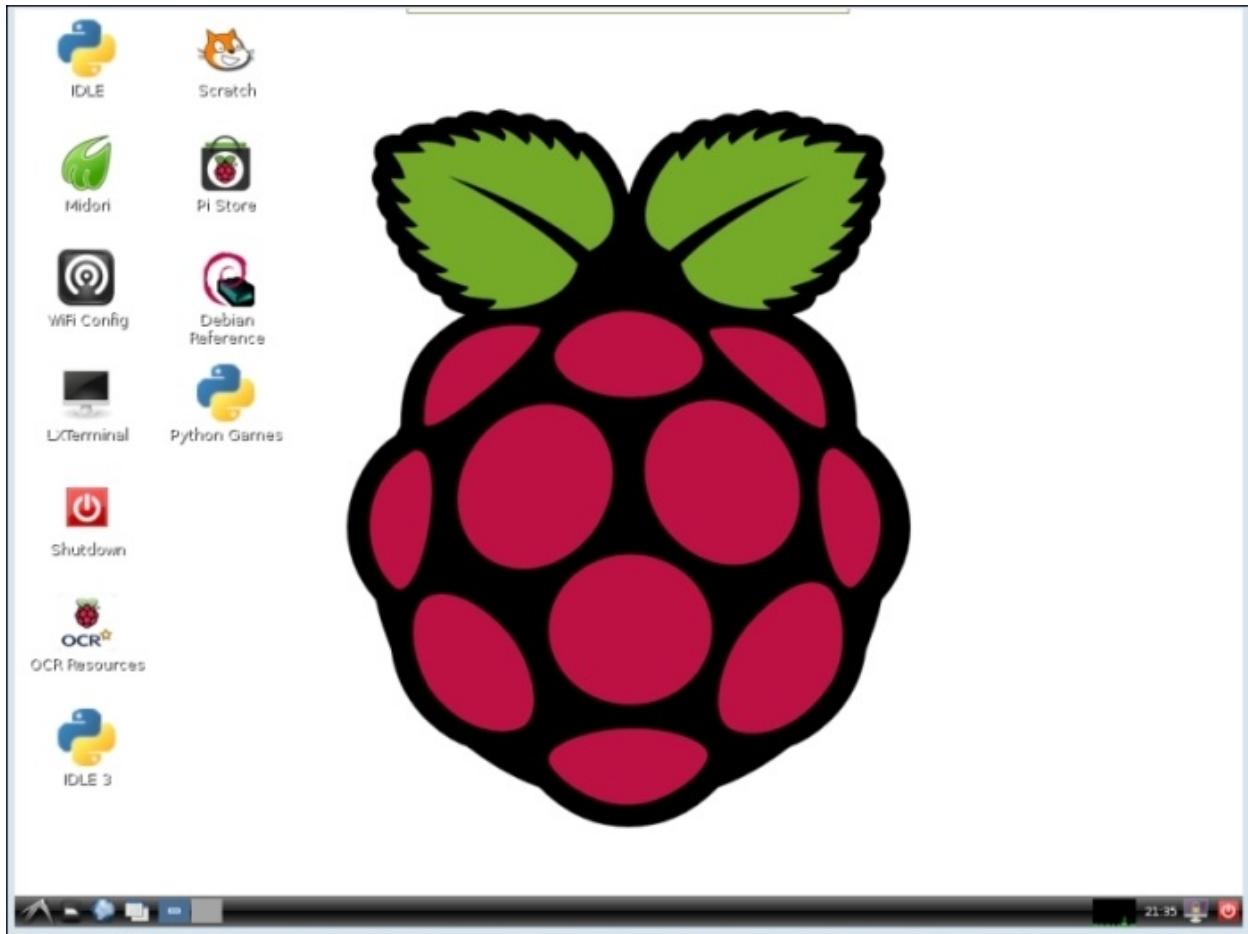
The final choice you can make is to change the overclocking on Raspberry Pi. This is a way for you to get higher performance from your system; however, there is a risk that you can end up with a system that has reliability problems. Here is the warning that comes up when you make this selection:



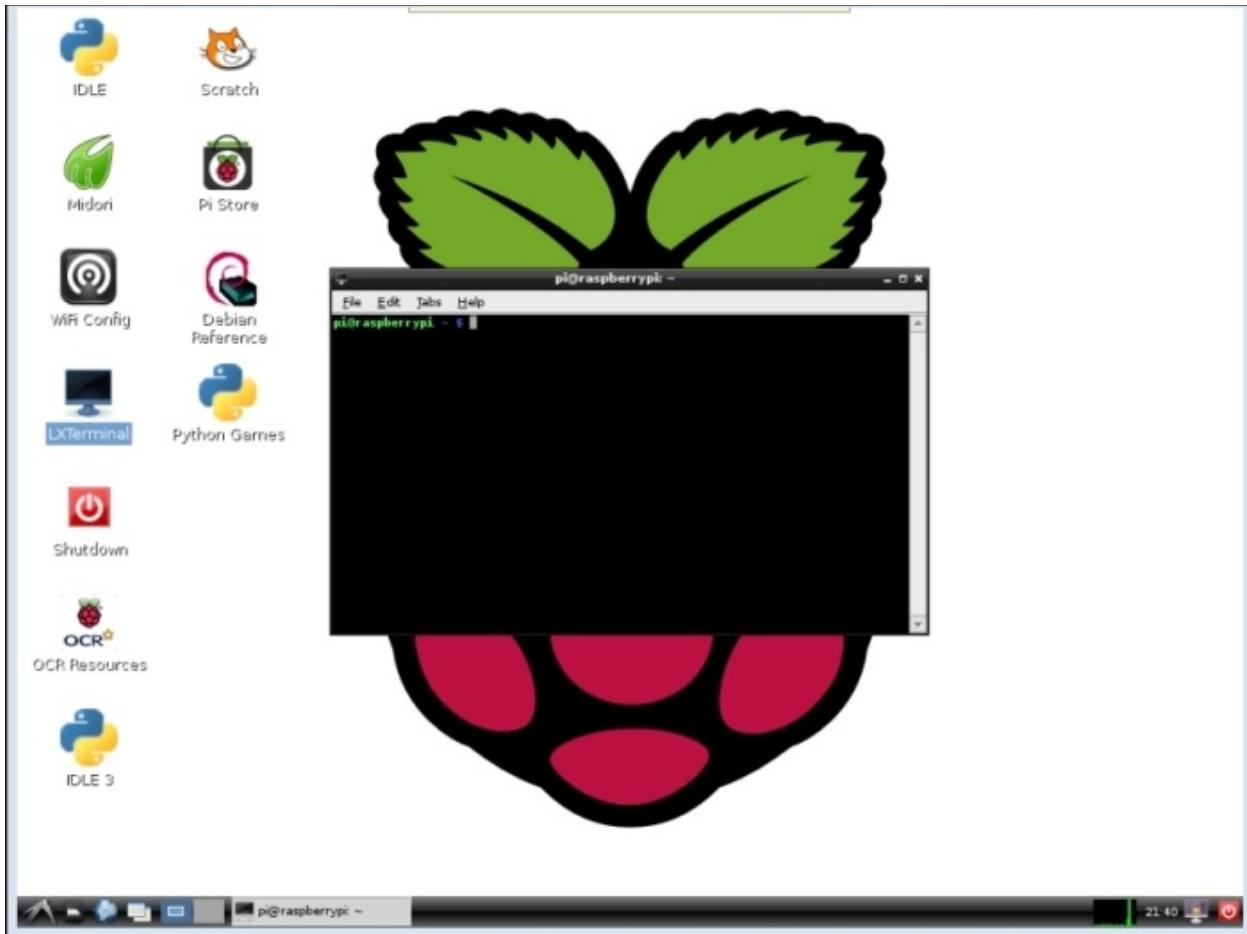
I normally do a bit of overclocking; I'll select the **Medium** setting, as shown in the following screenshot:



Once you are done and are back to 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. Now when you boot, your system will take you all the way into the Windows screen. Raspberry Pi uses the LXDE Windows system, and should look like the following screenshot:



Now when the Windows system is up and running, you can bring up a terminal by double-clicking on the **LXTerminal** selection on the screen. You should end up with a terminal window that looks like the following screenshot:



You are now ready to start interacting with the system!

Two questions arise: do we need an external computer during the creation of our projects? And what sort of computer do we need? The answer to the first question is a resounding yes. Most of our projects are going to be self-contained robots with very limited connections and display space; we will be using an external computer to issue commands and to see what is going on inside our robotic projects. The answer to the second question is a bit more difficult. Because your Raspberry Pi is working in Linux, most notably a version of Debian that is very closely related to Ubuntu, there are some advantages of having an Ubuntu system available as your remote system. You can then try some things on your computer before trying them in your embedded system. You'll also be working with similar commands for both, and this will help your learning curve.

However, the bulk of personal computers today run some sort of

Windows operating system, so that is what will normally be available. Even with a Windows machine, you can issue commands and display information, so either way will work. I'll try to give practical examples for both.

There is one more choice, and it is the choice that I actually prefer. I have access to both systems on my PC. Previously, this was done by a process called dual booting, where both systems were installed on the computer and the user chose which system they wanted to run on boot up. Changing systems in this kind of a configuration was time consuming, and used up a lot of disk space. However, there is a better way available now.

On my Windows PC, I have a virtual Ubuntu machine running under a free program from Oracle called VirtualBox. This program lets me run a virtual Ubuntu machine hosted by my Windows operating system. That way, I can try things in Ubuntu yet keep all the functionalities of my Windows machine. I'm not going to explain how to install this; there is plenty of help on the Web; just search for Ubuntu and VirtualBox. There are several websites that offer easy, step-by-step instructions. One of my favorites is <http://www.psychocats.net/ubuntu/>.

Accessing the board remotely

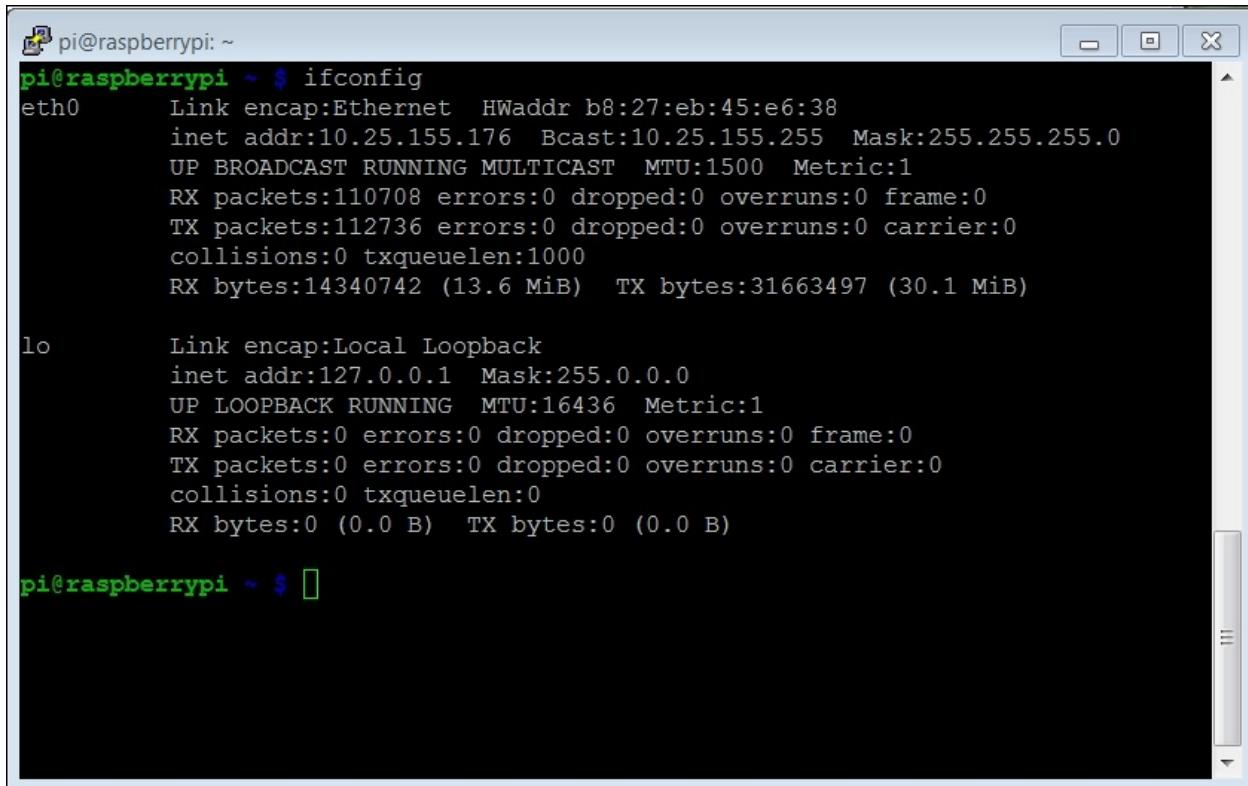
You now have a very usable Debian computer system. You can use it to access the Internet, write riveting novels, balance your accounts—just about anything you could do with a standard personal computer.

However, this is not your purpose; you want to use your embedded system to power our delightfully inventive projects. In most cases, you wouldn't want to connect a keyboard, mouse, and display to your projects. However, you still need to communicate with your device, program it, and have it tell you what is going on and when things don't work right. We'll spend some time in this section establishing remote access to our device.

The following are three ways by which we are going to access our system from our external PC:

- Through a terminal interface called SSH.
- Using a program called vncserver. This allows you to open a graphical user interface, which mirrors the graphical user interface on Raspberry Pi remotely.
- If you are using Microsoft Windows on your remote computer, I'll show how you can transfer files via a program called WinSCP, which is custom-made for this purpose. You can also use WinSCP to transfer a file in Linux; I'll show you how to do this as well.

So, first make sure your basic system is up and working. Open a terminal window and check the IP address of your unit. You're going to need this no matter how you want to communicate with the system. Do this by issuing the `ifconfig` command. It should look like the following screenshot:



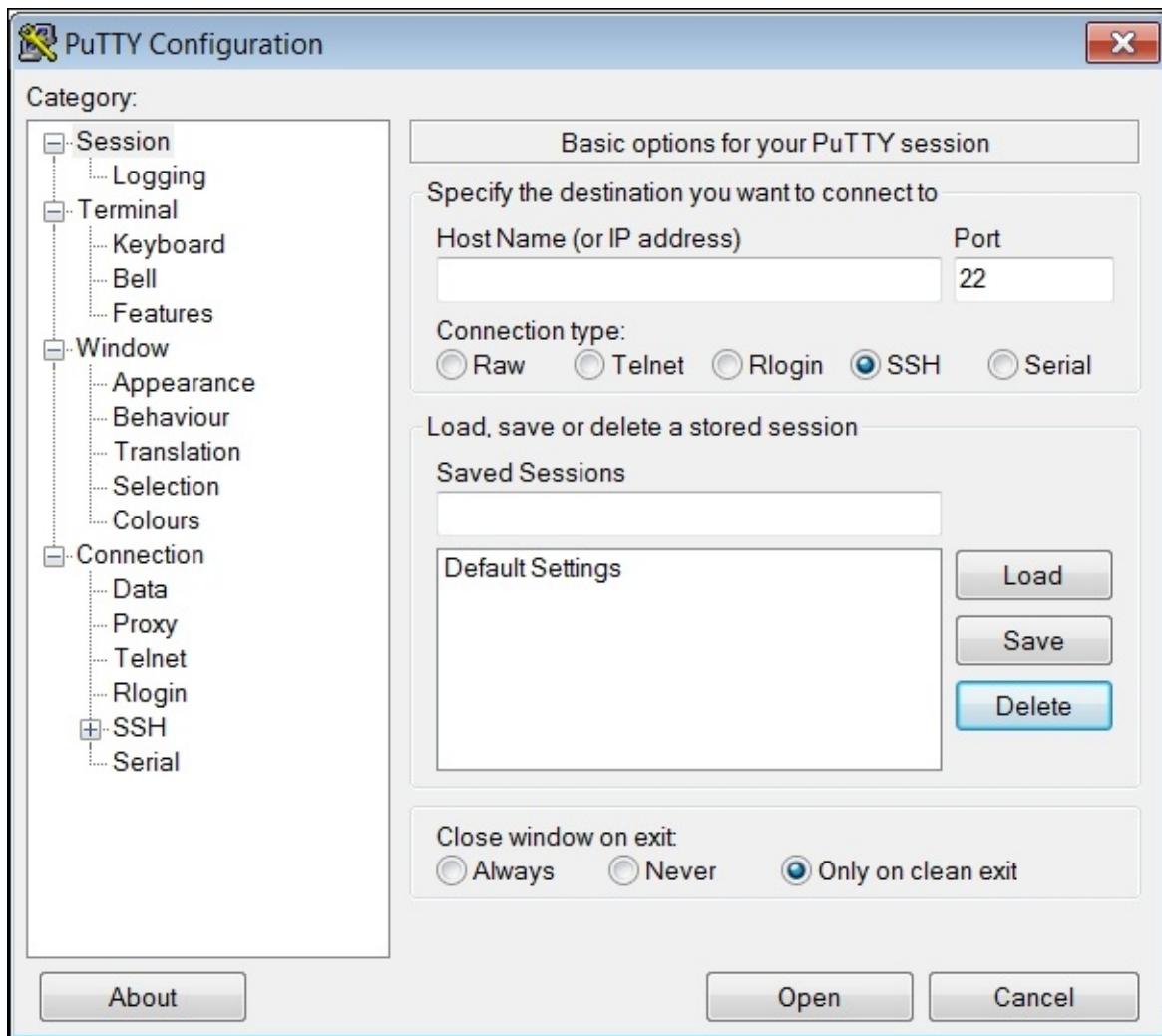
```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ifconfig
eth0      Link encap:Ethernet HWaddr b8:27:eb:45:e6:38
          inet addr:10.25.155.176 Bcast:10.25.155.255 Mask:255.255.255.0
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:110708 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:112736 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:14340742 (13.6 MiB) TX bytes:31663497 (30.1 MiB)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
                  UP LOOPBACK RUNNING MTU:16436 Metric:1
                  RX packets:0 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

pi@raspberrypi ~ $
```

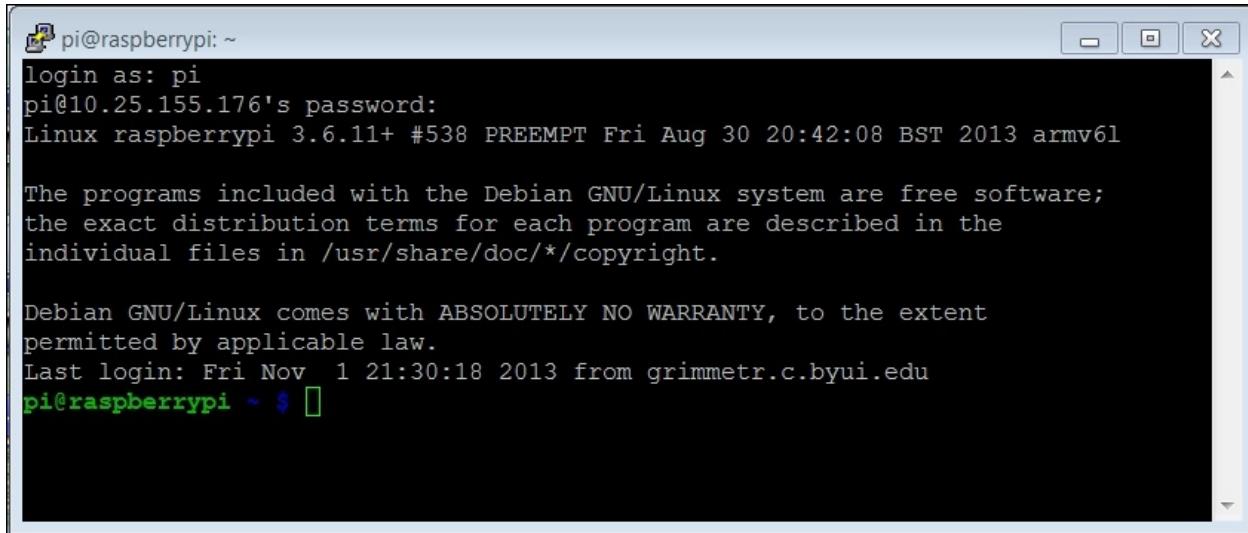
You'll need `inet addr`, shown on the second line, to contact your board via LAN. You'll also need an SSH terminal program running on your remote computer. An SSH terminal is a **Secure Shell Hypterminal (SSH)** connection, which simply means you'll be able to access your board and type in commands at the prompt, just like you have done previously. If you are running Microsoft Windows, you can download such an application. My personal favorite is PuTTY. It is free and does a very good job of allowing us to save our configuration so we don't have to type configurations each time. Type [putty](#) in a search window and you'll soon come to a page that supports a download, or you can go to www.putty.org.

Download PuTTY to your Microsoft Windows machine. Then run [putty.exe](#). You should see a configuration window, which will look something like the following screenshot:



Type `inet addr` from the previous page in the **Host Name** space and make sure the SSH selection is selected. I save this configuration under Raspberry Pi so I can load it every 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 that 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 as shown in the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows the following text:

```
pi@raspberrypi: ~
login as: pi
pi@10.25.155.176's password:
Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 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.

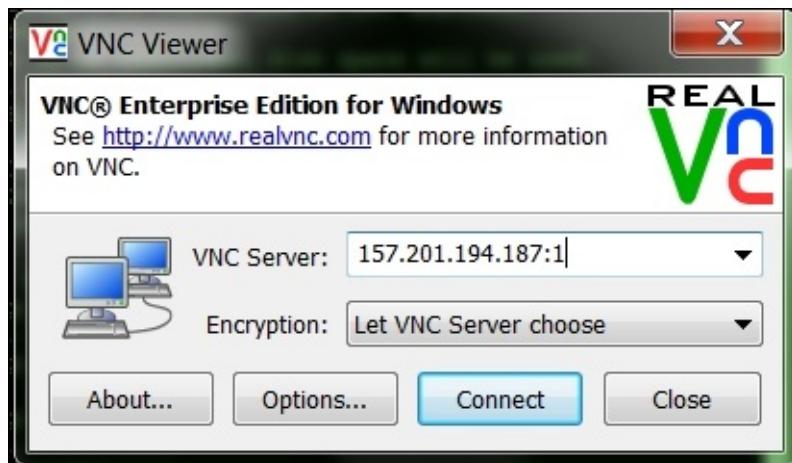
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Nov  1 21:30:18 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $
```

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 `sshpi157.201.194.187 -p 22`. This will then bring you to the login screen of your Raspberry Pi, which should look similar to the previous screenshot.

SSH is a really useful tool to communicate with your Raspberry Pi. However, sometimes you need a graphical look at your system, and you don't necessarily want to connect a display. You can get this on your remote computer using an application called vncserver. You'll need to install a version of this on your Raspberry Pi by typing `sudoapt-getinstall tightvncserverin` in a terminal window on your Raspberry Pi. By the way, this is a perfect opportunity to use SSH.

TightVNC is an application that will allow you to remotely view your complete Windows system. Once you have it installed, you'll need to start the server by typing `vncserver` in a terminal window on Raspberry Pi. You will then be prompted for a password, be prompted to verify it, 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.

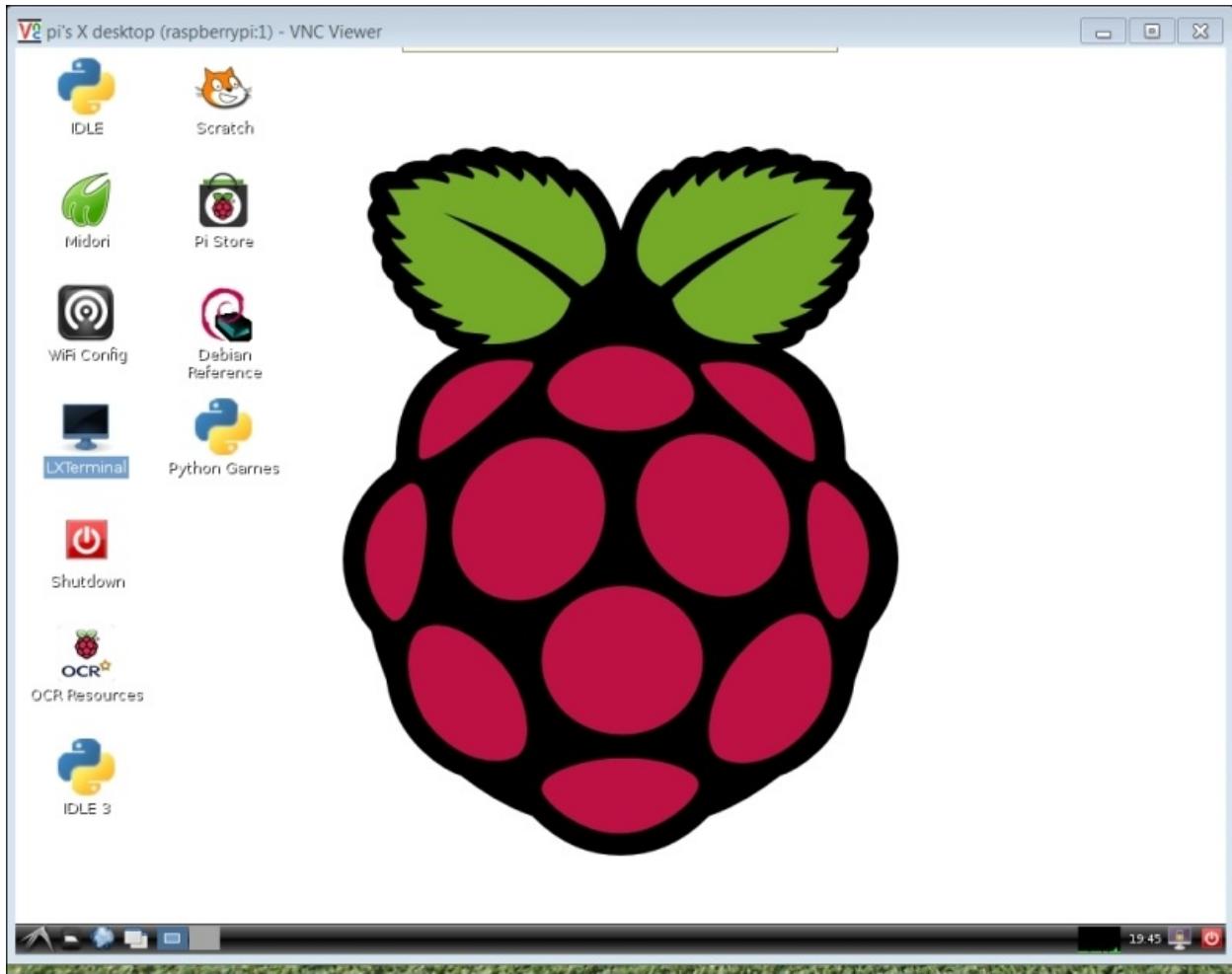
You'll need a VNC Viewer application for your remote computer. On my Windows system, I use an application called Real VNC. When I start the application, it gives me the following screenshot:



Enter the **VNC Server** address, which is the IP address of your Raspberry Pi, and click on **Connect**. You will get this pop-up window:



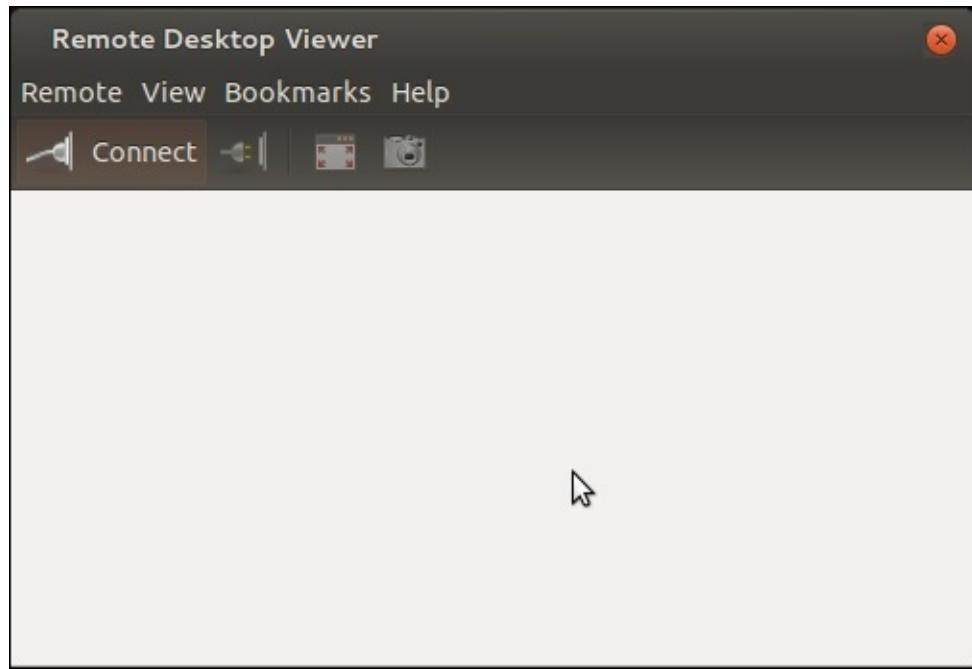
Type in the password you just entered while starting the vncserver; you should then get a graphical view of your Raspberry Pi that looks like the following screenshot:



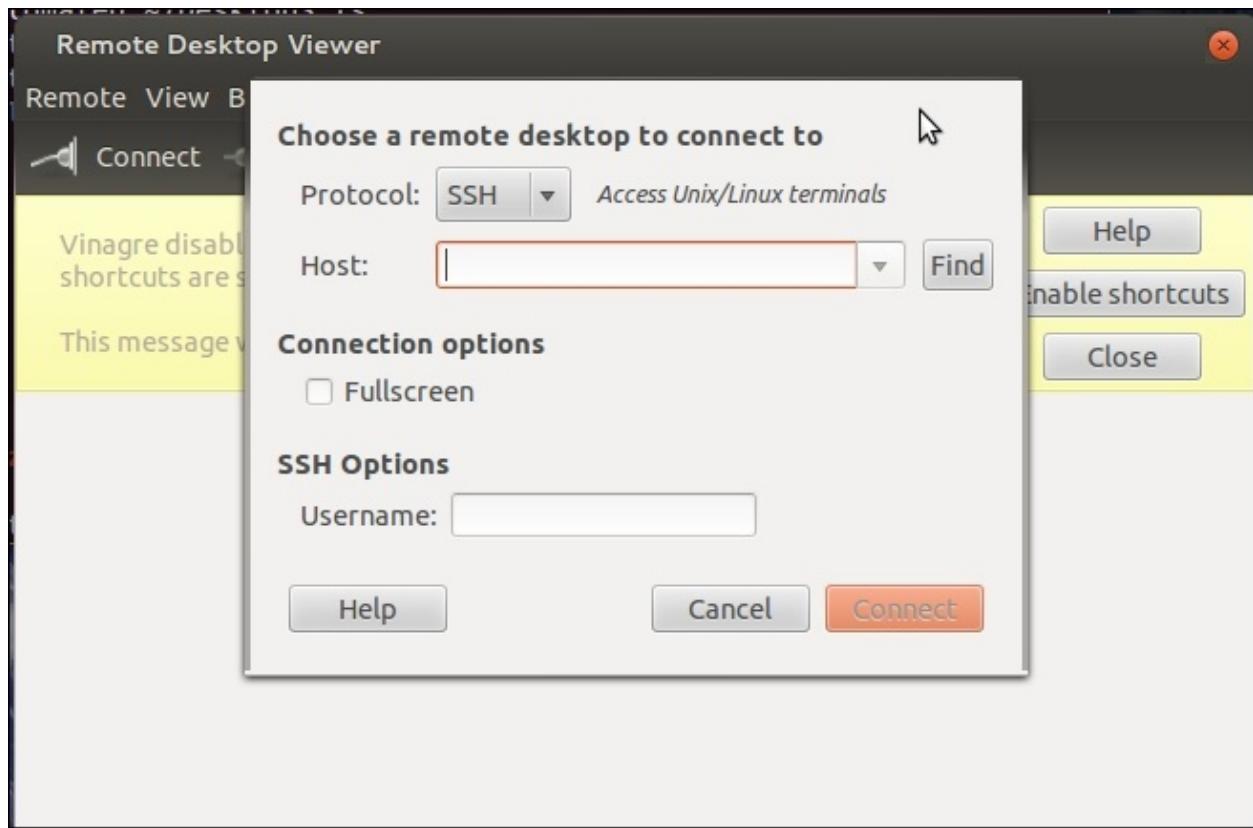
You can now access all the capabilities of your system, albeit they may be slower if you are doing graphics-intense data transfer. Just a note: there are ways to make your vncserver start automatically on boot. I have not used them; I choose to type the command `vncserver` from an SSH application when I want the application running. This keeps your running applications to a minimum and, more importantly, ensures fewer security risks. If you'd like to start yours each time your system boots, there are several places on the Internet that will show you how to configure this. You can try <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 update software application based on the type of Linux system you have. Once you have the software, run the application and you should see the

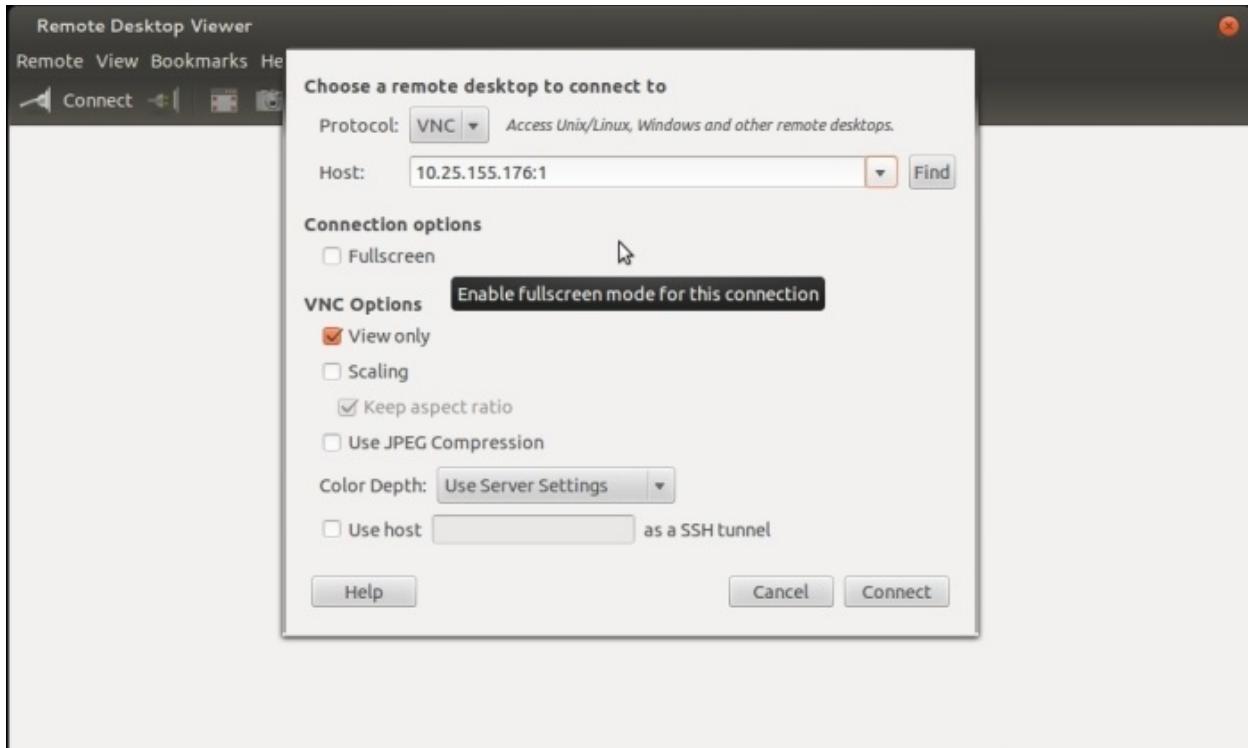
following screenshot:



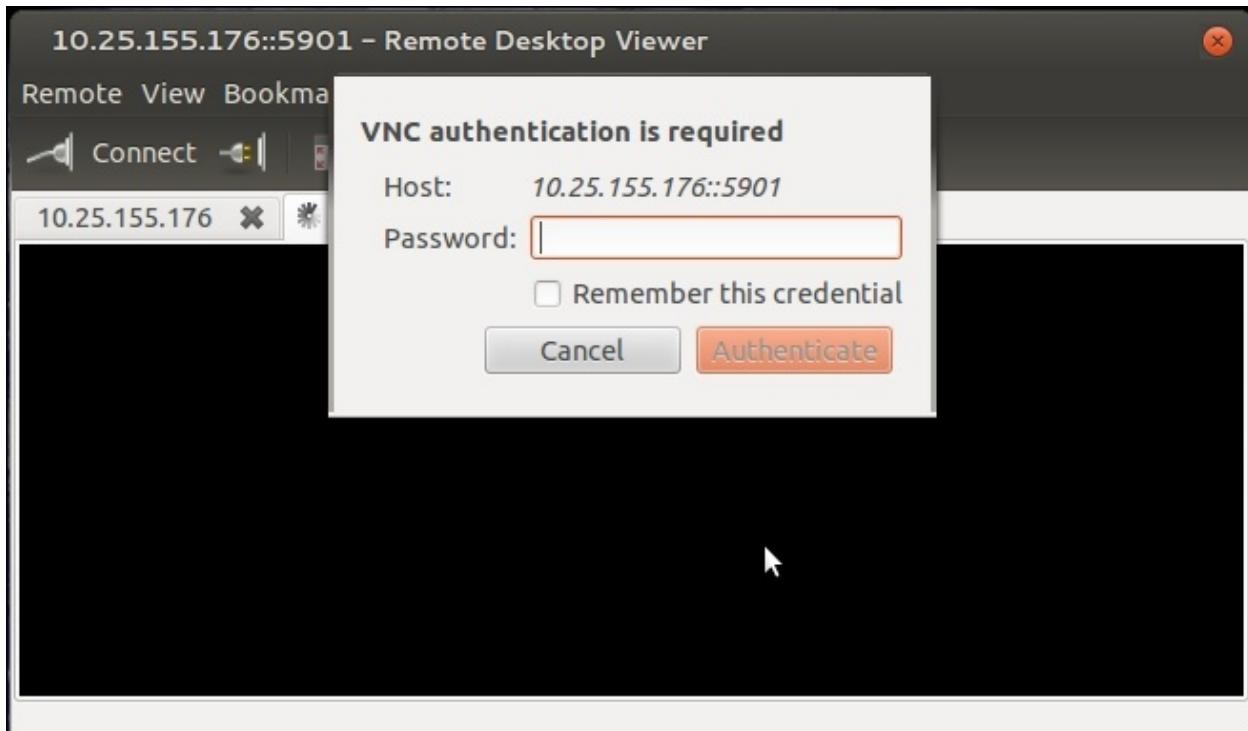
Make sure vncserver is running on Raspberry Pi; the easiest way to do this is to log in using SSH and run vncserver at the prompt. Now click on **Connect** on the **Remote Desktop Viewer** running in Linux, shown in the following screenshot:



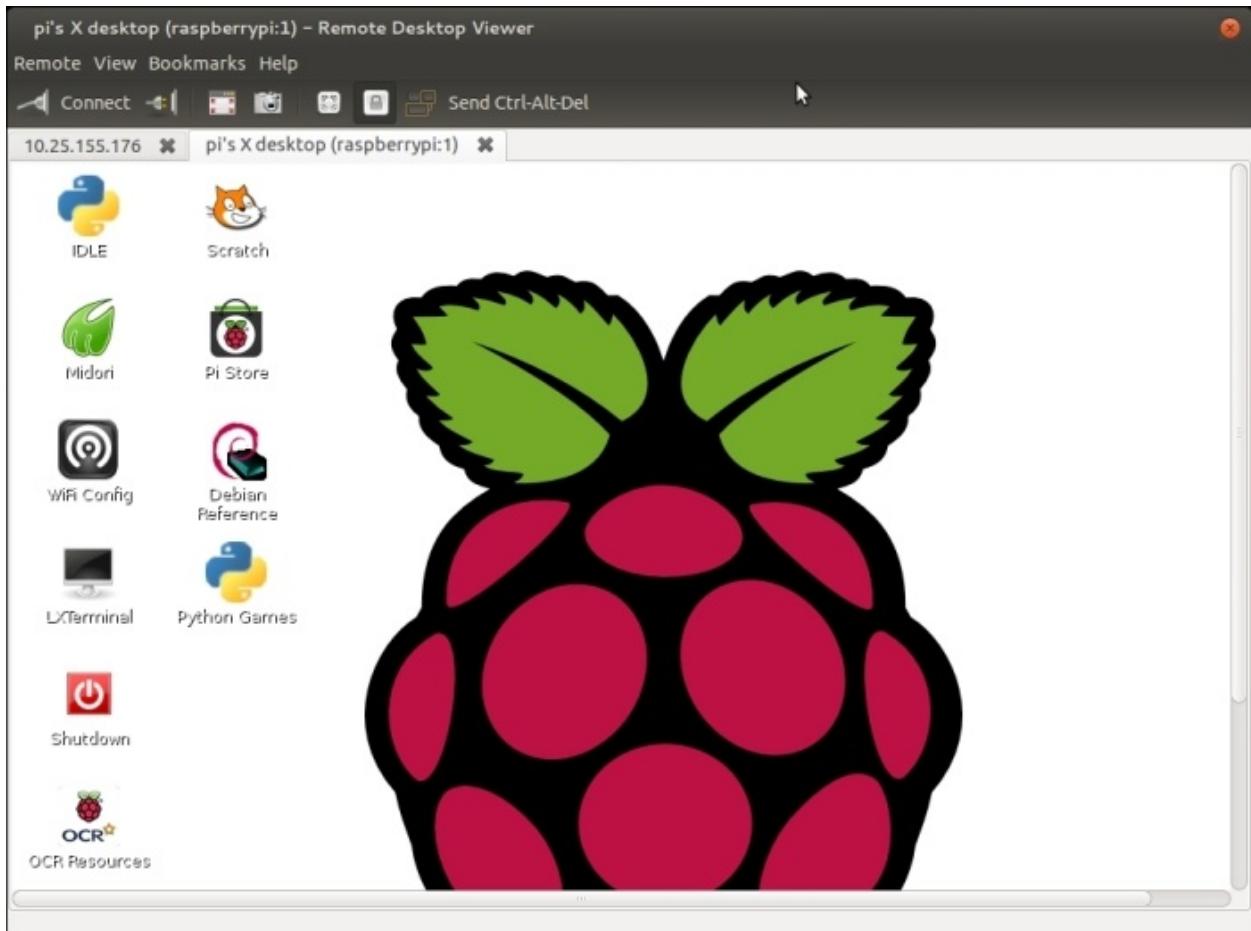
Under the **Protocol** selection, choose **VNC**, as shown in the following screenshot:



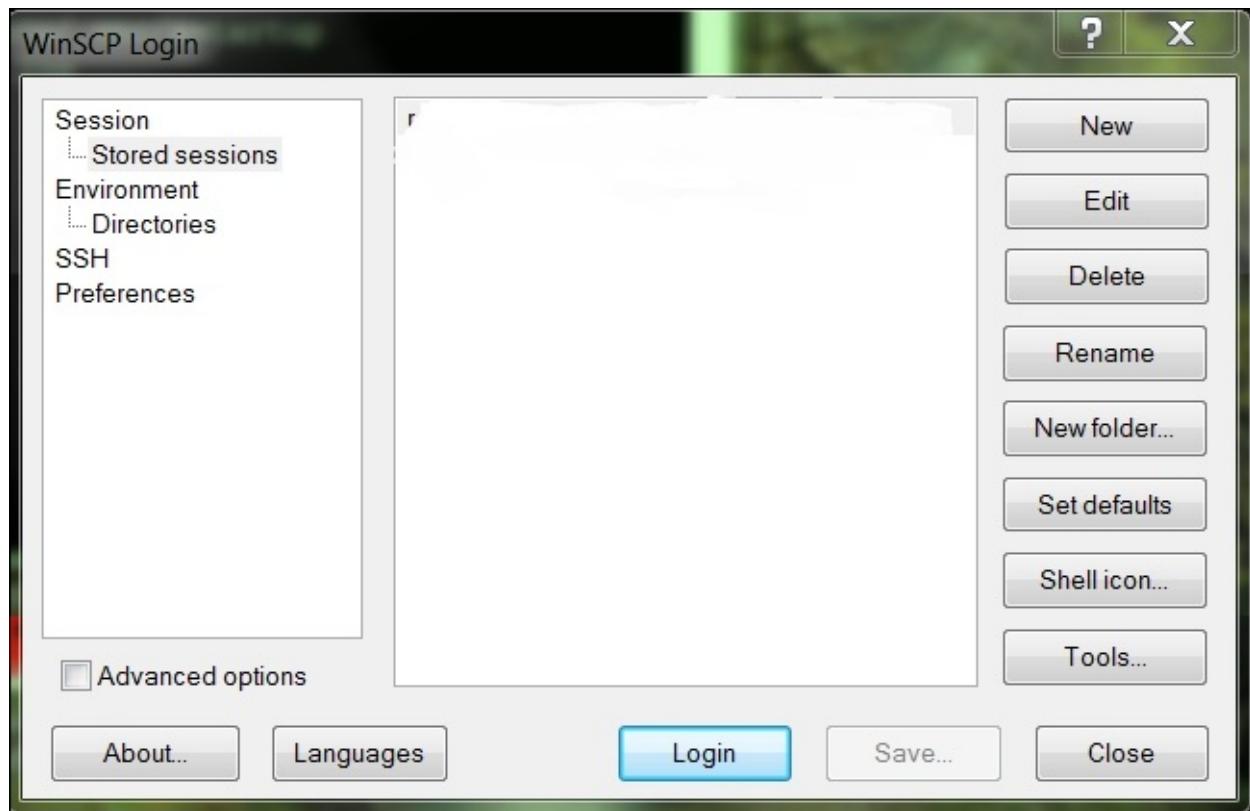
Now enter the **Host** inet address, make sure you include a `:1` at the end, and then click on **Connect**. You'll need to enter the vncserver password you set up, as shown in the following screenshot:



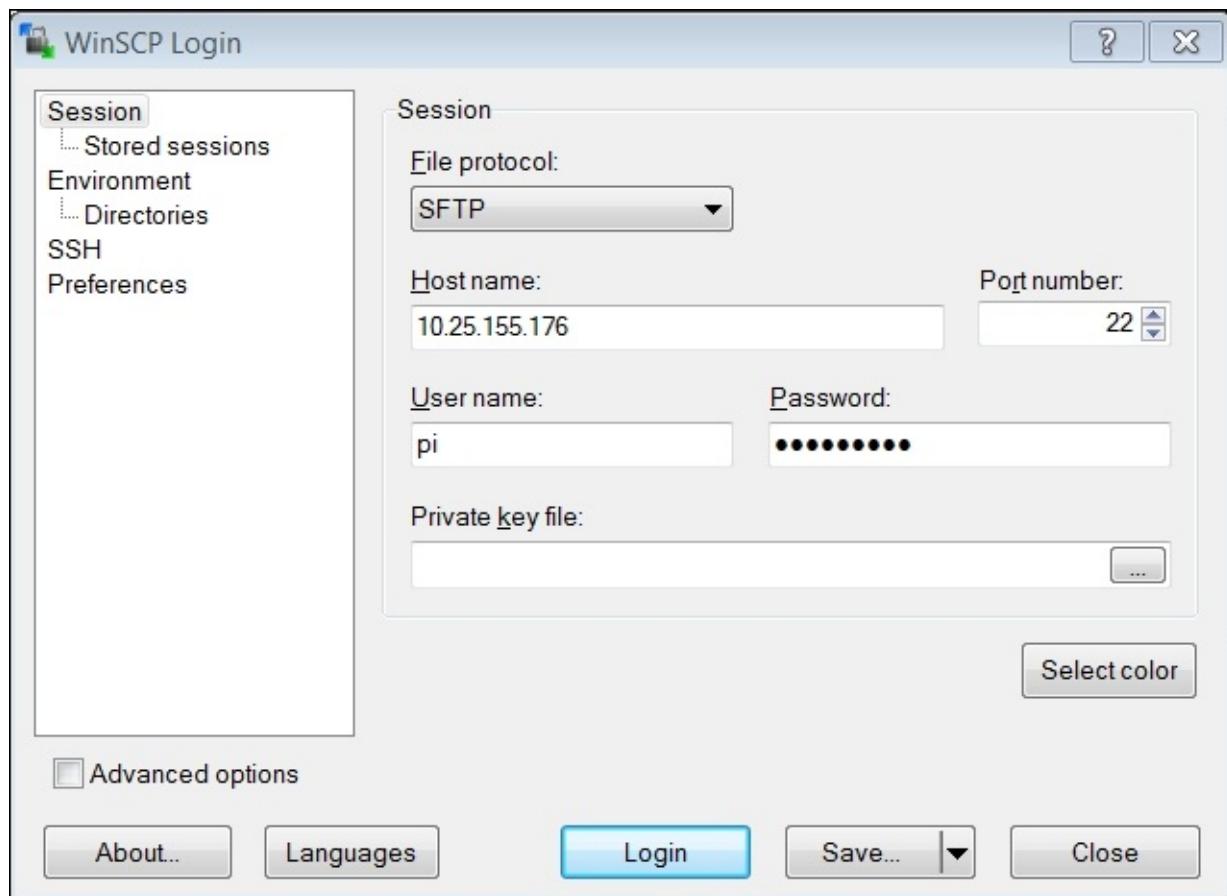
Now you should see your LXDE window that is running on Raspberry Pi, as shown in the following screenshot:



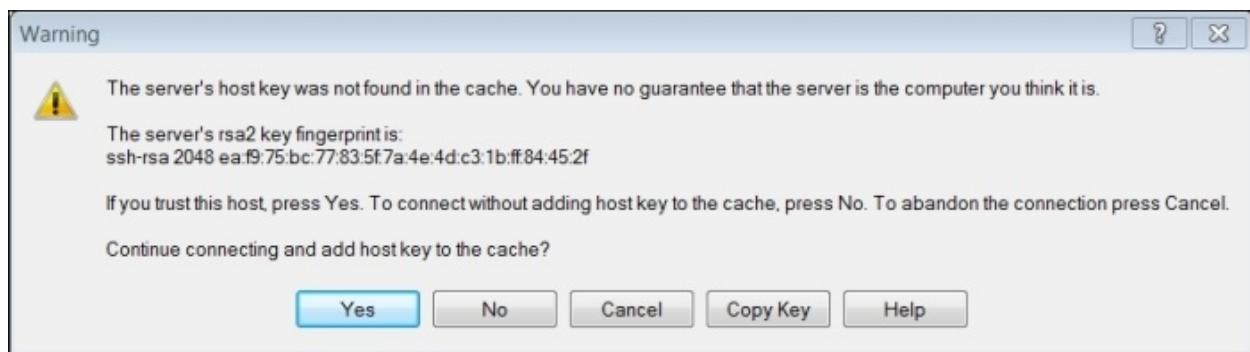
The final piece of software I like to use with my Windows system is a free application called WinSCP. To download and install this software, simply search the Web for WinSCP and follow the instructions. Once installed, run the program; it will open the following dialog box:



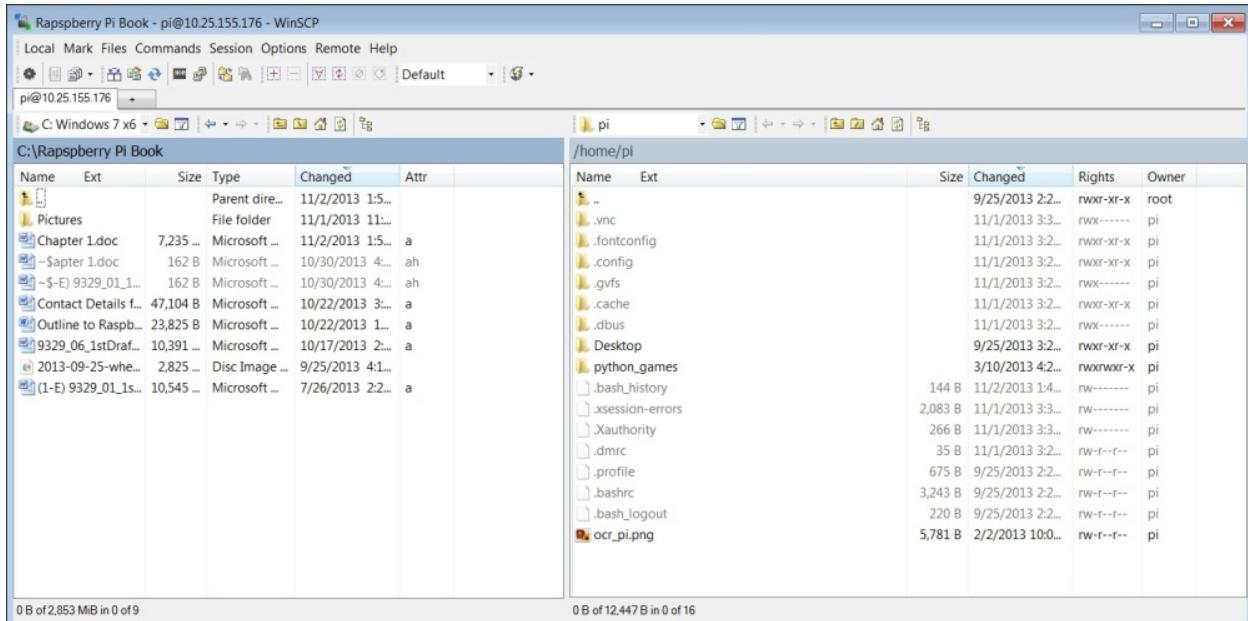
Click on **New** and you will get the following screenshot:



Here, fill in the IP address in **Host name** (pi in the user name) and the password (not the vncserver password) in the password space. Click on Login and you should see the following warning displayed:



The host computer, again, doesn't know the remote computer. Click on **Yes** and the application will display the following screenshot:

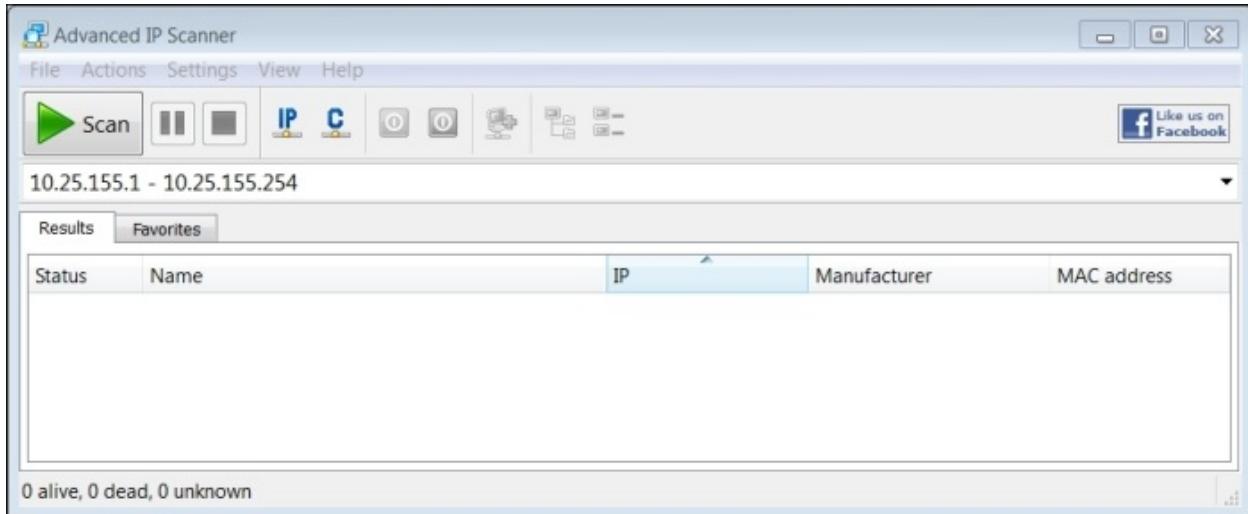


Now we can drag-and-drop files from one system to the other. You can also do similar things on Linux using the command line. To transfer a file to the remote Raspberry Pi, you can use the command `scp user@host.domain:path`, where `file` is the filename and `user@host.domain:path` is the location you want to copy. For example, if I want to copy the term `robot.py` from my Linux system to Raspberry Pi, I would type `scp robot.py pi@10.25.155.176:/home/pi/`. Now the system will ask me for the remote password; this is the login for Raspberry Pi. Enter the password and the file should be transferred. Once we've completed this step, we can now access our system remotely without connecting a display, keyboard, and mouse. Now your system will look like the following image:



You need to connect the power and LAN. If you need to issue simple commands, you can connect via SSH. If you need a more complete set of graphical functionalities, you can access them via vncserver. Finally, if you are using a Windows system and want to transfer files back and forth, you need access to WinSCP. Now you have the toolkit to build your first set of capabilities.

One of the challenges of accessing the system remotely is that you need to know the IP address of your board. If you have the board connected to a keyboard and display, you can always just run [ifconfig](#) to get this info. But you're going to use the board in applications where you don't have this information. There is a way to discover this by using an IP scanner application. There are several available for free; on Windows, I use an application known as Advanced IP Scanner. When I start the program, it looks like the following screenshot:



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 Nmap. To install Nmap, type `sudo apt-get installnmap`. To run Nmap, type `sudonmap -sP 10.25.155.1/254`, and the scanner will scan the addresses from 10.25.155.1 to 10.25.155.254.

These scanners let you know which addresses are being used; this should then let you find your Raspberry Pi address without typing `ipconfig`.

Your system has lots of capabilities. Feel free to play with the system; try to get an understanding of what is already there and what you'll want to add from a software perspective. One advanced possibility is to connect Raspberry Pi via a wireless LAN connection so that you don't have to connect a LAN connection when you want to communicate with it. There are several good tutorials on the Internet. Try <http://learn.adafruit.com/adafruits-raspberry-pi-lesson-3-network-setup/setting-up-wifi-with-occidentalis> or <http://www.howtogeek.com/167425/how-to-setup-wi-fi-on-your-raspberry-pi-via-the-command-line/>.

Remember, there is limited power on your USB port, so make sure you have a powered USB hub before trying this.

Summary

Congratulations! You've completed the first stage of your journey. You have your Raspberry Pi up and working. No gathering dust in the bin for this piece of hardware. It is now ready to start connecting to all sorts of interesting devices in all sorts of interesting ways. You should have installed a Debian operating system, learned how to connect all the appropriate peripherals, and even mastered how to access the system remotely so that the only connections you need are a power supply cable and a LAN cable.

You are now ready to start commanding your Raspberry Pi to do something. The next chapter will introduce you to the Linux operating system, the Emacs text editor, and also show you some basic programming concepts in both the Python and C programming languages. You'll then be ready to add open source software to inexpensive hardware to start building your robotics projects.

Chapter 2. Programming Raspberry Pi

Now that your system is up and running, you'll want Raspberry Pi to start working. Almost always, this requires you to either create your own programs or edit someone else's programs. This chapter will provide a brief introduction to editing a file and programming.

While it is fun to build hardware (and you'll spend a good deal of time designing and building your robots), your robots won't get very far without programming. This chapter will help introduce you to file editing and programming concepts so you'll feel comfortable creating some of the fairly simple programs that we'll discuss in this book. You'll also learn how to change existing programs, making your robot do even more amazing things.

In this chapter, we will cover the following topics:

- Basic Linux commands and navigating the filesystem on Raspberry Pi
- Creating, editing, and saving files on Raspberry Pi
- Creating and running Python programs on Raspberry Pi
- Some of the basic programming constructs on Raspberry Pi
- How the C programming language is both similar and different to Python so you can understand when you need to change C code files

We're going to use the basic configuration that you created in [Chapter 1, Getting Started with Raspberry Pi](#). You can accomplish the tasks in this chapter by connecting a keyboard, mouse, and monitor to Raspberry Pi, or remotely logging in to Raspberry Pi using vncserver or SSH. All of these methods will work in executing the examples in this chapter.

Basic Linux commands on Raspberry Pi

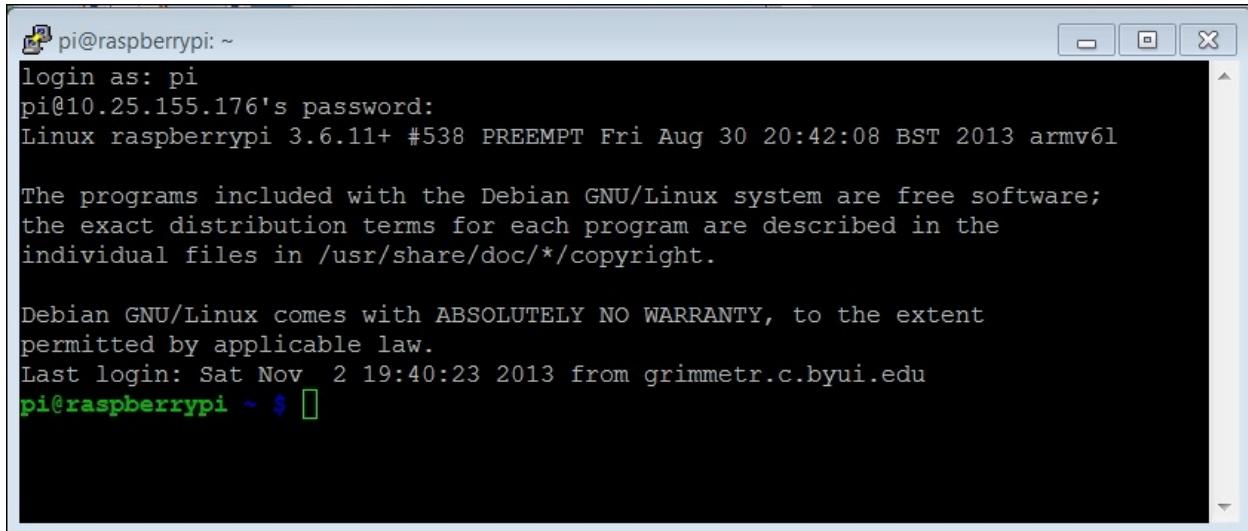
After completing the tasks in [Chapter 1](#), *Getting Started with Raspberry Pi*, you should have a working Raspberry Pi running a version of Linux called Raspbian. We selected the Linux version because it is the most popular version, and thus has the largest range of supported hardware and software. The commands reviewed in this chapter should also work with other versions of Linux, but the examples shown in this chapter use Raspbian.

So, power up your Raspberry Pi and log in using a valid username and password. If you are going to log in remotely, go ahead and establish the connection and do so. Now we can take a quick tour of Linux. This will not be extensive, but we will walk through some of the basic commands.

Once you have logged in, you should open up a terminal window. If you are logging in using a keyboard, mouse, and monitor, or using vncserver, you'll find the terminal selection by selecting the **LXTerminal** application to the left-hand side of the screen as shown in the following screenshot:



If you are using SSH, you should already be at the terminal emulator program. Either way, the terminal should look something as shown in the following screenshot:

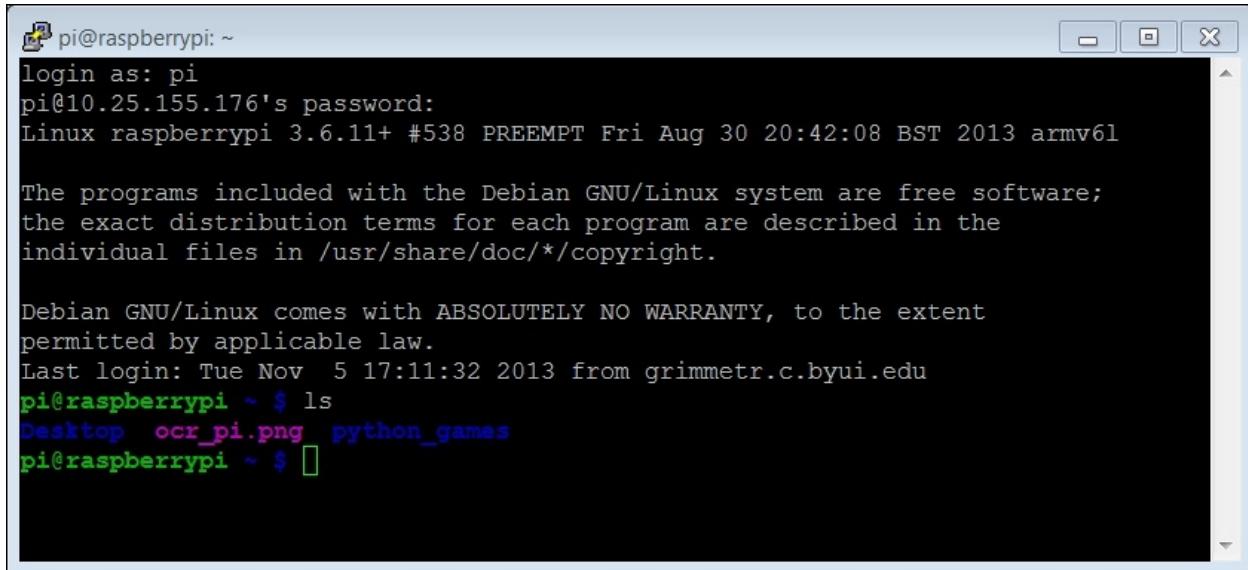


```
pi@raspberrypi: ~
login as: pi
pi@10.25.155.176's password:
Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 armv6l

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 Nov  2 19:40:23 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $
```

Your cursor is at the command prompt. Unlike with Microsoft Windows or Apple's OS, with Linux, most of our work will be done by actually typing commands in the command line. So, let's try a few commands. First, type `ls`; you should see the result as shown in the following screenshot:



```
pi@raspberrypi: ~
login as: pi
pi@10.25.155.176's password:
Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 armv6l

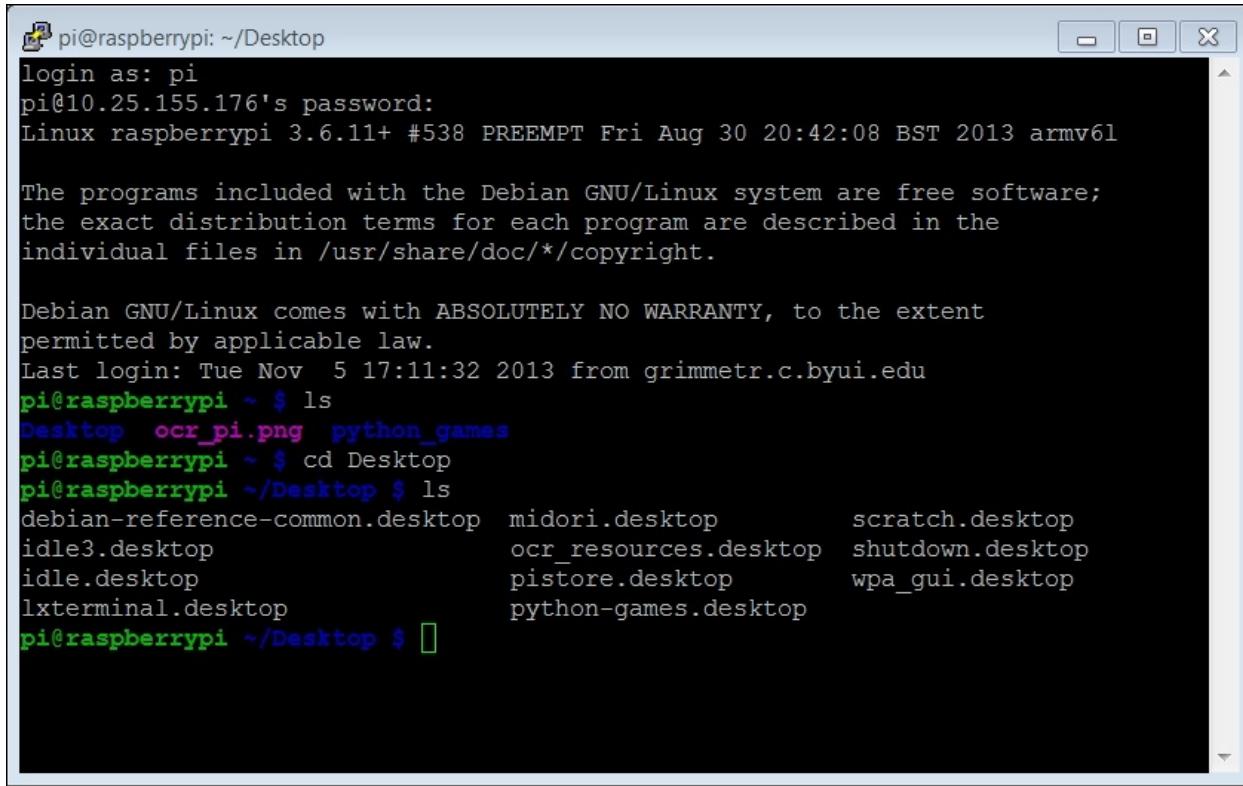
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Nov  5 17:11:32 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $ ls
Desktop  ocr_pi.png  python_games
pi@raspberrypi ~ $
```

In Linux, the command `ls` is short for list-short, and it lists all the files and directories in our current directory. You can tell apart the different file types and directories because they are normally in different colors.

You can move around the directory structure by issuing the `cd` (change directory) command. For example, if you want to see what is in the `Desktop` directory, type `cd ./Desktop`. If you issue the `ls` command now,

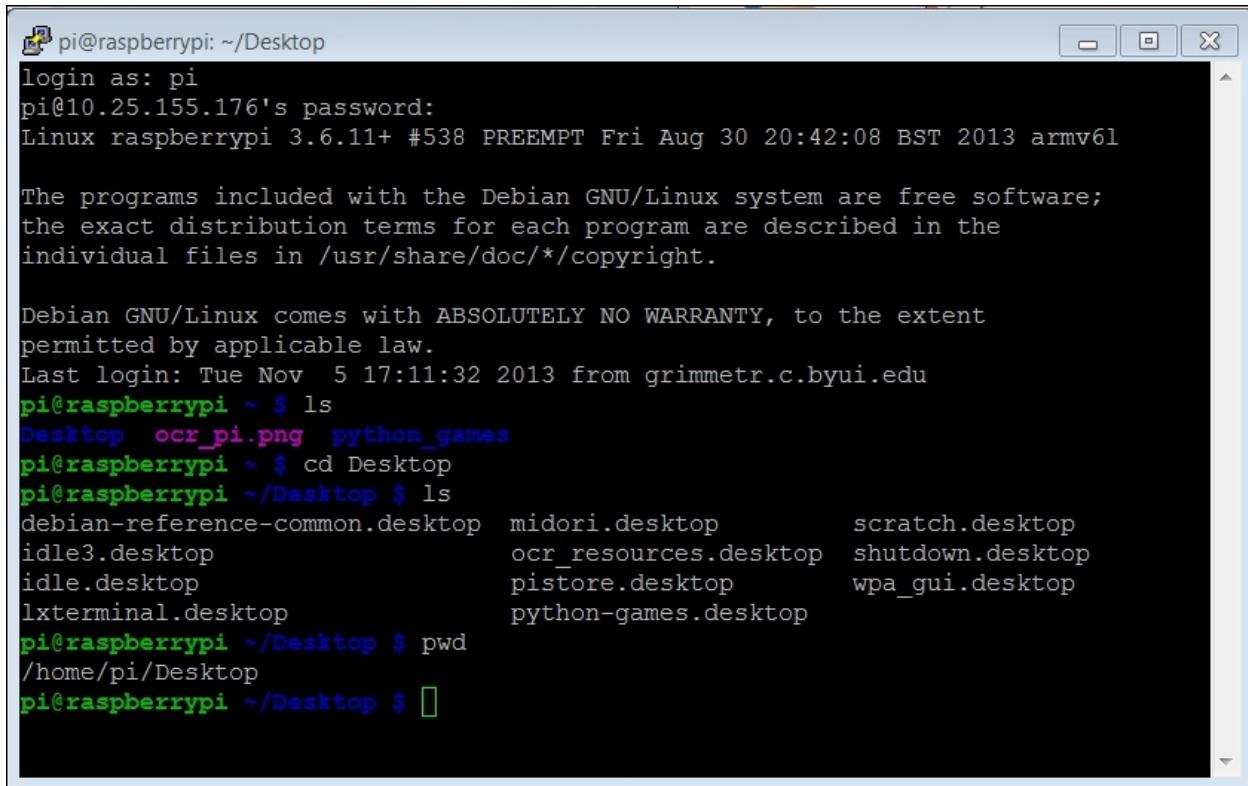
you should see something as shown in the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~/Desktop". The window shows a standard Linux login sequence where the user "pi" logs in from IP address "10.25.155.176". It then displays the kernel version and date ("Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 armv6l"). Following this, it shows the copyright notice for the Debian GNU/Linux system. Below the notice, it states that the system comes with "ABSOLUTELY NO WARRANTY". The next few lines show the last login information ("Last login: Tue Nov 5 17:11:32 2013 from grimmetr.c.byui.edu") and the current working directory ("pi@raspberrypi ~ \$"). The user then runs the command "ls" to list files in the Desktop directory. The output shows several desktop environment files: "debian-reference-common.desktop", "idle3.desktop", "idle.desktop", "lxterminal.desktop", "midori.desktop", "ocr_resources.desktop", "pistore.desktop", "python-games.desktop", "scratch.desktop", "shutdown.desktop", and "wpa_gui.desktop". The terminal ends with a prompt "pi@raspberrypi ~/Desktop \$".

This directory mostly has definitions for what should appear on this [Desktop](#) folder. Now, I should point out that we used a shortcut when we typed `cd ./Desktop`. The dot (.) character is a shortcut for the default directory. The command `cd` is short for change directory. You may also have typed `cd /homepi/Desktop` and received the exact same result; this is because you were in the [/homepi](#) directory, which is the directory where you always start when you first log in to the system.

If you ever want to see which directory you are in, simply type `pwd`, which stands for print working directory. If you do that, you should get the result shown in the following screenshot:



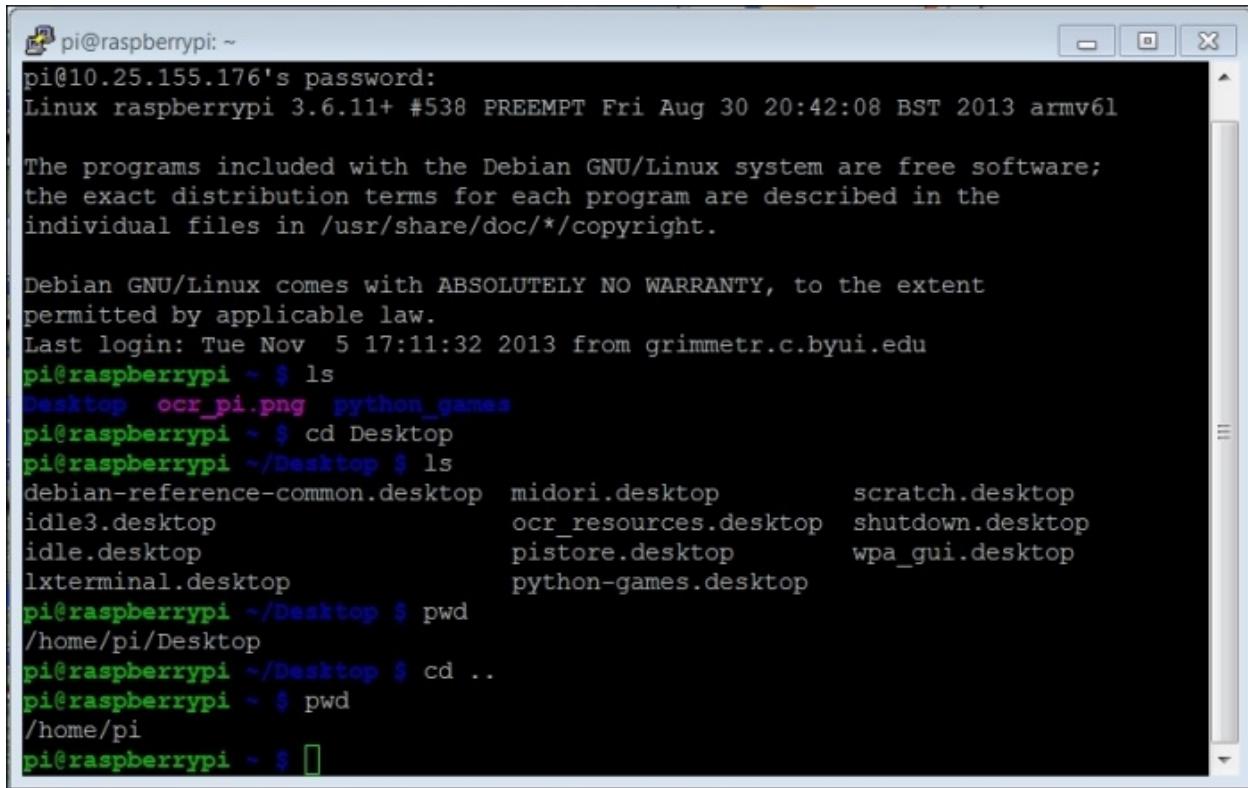
A screenshot of a terminal window titled "pi@raspberrypi: ~/Desktop". The window shows a standard Linux login sequence where the user "pi" logs in from IP address "10.25.155.176". It then displays the kernel version and date ("Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 armv6l"). A copyright notice follows, stating that the programs are free software with individual files containing copyright information. The terminal then shows the user navigating to the "/Desktop" directory, listing files like "ocr_pi.png" and "python_games". Finally, the user runs the "pwd" command to show the current working directory as "/home/pi/Desktop".

```
pi@raspberrypi: ~/Desktop
login as: pi
pi@10.25.155.176's password:
Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 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: Tue Nov  5 17:11:32 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $ ls
Desktop ocr_pi.png python_games
pi@raspberrypi ~ $ cd Desktop
pi@raspberrypi ~/Desktop $ ls
debian-reference-common.desktop  midori.desktop      scratch.desktop
idle3.desktop                   ocr_resources.desktop shutdown.desktop
idle.desktop                     pistore.desktop    wpa_gui.desktop
lxterminal.desktop               python-games.desktop
pi@raspberrypi ~/Desktop $ pwd
/home/pi/Desktop
pi@raspberrypi ~/Desktop $
```

The result of running the `pwd` command is `homepi/Desktop`. Now, you can use two different shortcuts to navigate back to the default directory. The first is to type `cd ..` in the terminal; this will take you to the directory just above the current directory in the hierarchy. Then type `pwd`; you should see the following screenshot as a result:



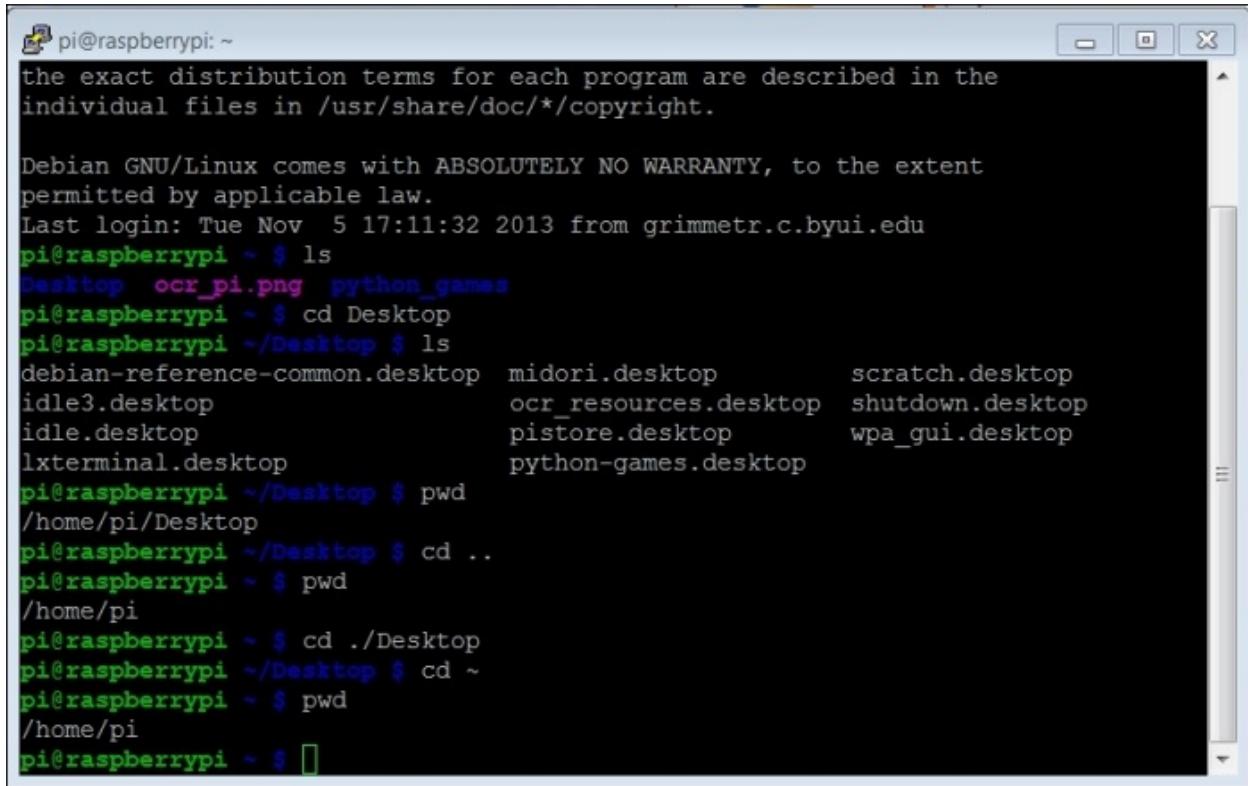
A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows a standard Linux boot message, the kernel version (3.6.11+ #538 PREEMPT), the date (Fri Aug 30 20:42:08 BST 2013), and the architecture (armv6l). It then displays the copyright notice for the Debian GNU/Linux system. Below that, it shows the warranty notice, the last login information (Tue Nov 5 17:11:32 2013 from grimmetr.c.byui.edu), and the user's current directory (~). The user runs several commands: "ls" to list files in the Desktop directory, "cd Desktop" to change to that directory, "ls" again to show desktop files, "pwd" to print the current working directory, "cd .." to move up one directory level, and finally "pwd" again to show the home directory. The terminal window has a standard title bar with close, minimize, and maximize buttons.

```
pi@raspberrypi: ~
pi@10.25.155.176's password:
Linux raspberrypi 3.6.11+ #538 PREEMPT Fri Aug 30 20:42:08 BST 2013 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: Tue Nov  5 17:11:32 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $ ls
Desktop  ocr_pi.png  python_games
pi@raspberrypi ~ $ cd Desktop
pi@raspberrypi ~/Desktop $ ls
debian-reference-common.desktop  midori.desktop      scratch.desktop
idle3.desktop                   ocr_resources.desktop shutdown.desktop
idle.desktop                     pistore.desktop    wpa_gui.desktop
lxterminal.desktop               python-games.desktop
pi@raspberrypi ~/Desktop $ pwd
/home/pi/Desktop
pi@raspberrypi ~/Desktop $ cd ..
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

The other way to get back to the home directory is by typing `cd ~`, as this will always return you to your home directory. If you were to do this from the `Desktop` directory and then type `pwd`, you should see the result as shown in the following screenshot:



The terminal window shows a session on a Raspberry Pi. The user has run the 'ls' command in their home directory (~) and then navigated to the 'Desktop' directory. Inside 'Desktop', they have run 'ls' again, showing various desktop environment files like 'midori.desktop', 'ocr_resources.desktop', etc. They then used 'cd ..' to move back up to the home directory, and finally 'cd .' to move back into the current directory (~).

```
pi@raspberrypi: ~
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: Tue Nov  5 17:11:32 2013 from grimmetr.c.byui.edu
pi@raspberrypi ~ $ ls
Desktop  ocr_pi.png  python_games
pi@raspberrypi ~ $ cd Desktop
pi@raspberrypi ~/Desktop $ ls
debian-reference-common.desktop  midori.desktop      scratch.desktop
idle3.desktop                   ocr_resources.desktop shutdown.desktop
idle.desktop                     pistore.desktop    wpa_gui.desktop
lxterminal.desktop               python-games.desktop
pi@raspberrypi ~/Desktop $ pwd
/home/pi/Desktop
pi@raspberrypi ~/Desktop $ cd ..
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $ cd ./Desktop
pi@raspberrypi ~/Desktop $ cd ~
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

Another way to go to a specific file is using its entire pathname. In this case, if you want to go to the [homeRaspbian/Desktop](#) directory from anywhere in the filesystem, simply type [cd homeRaspbian/Desktop](#).

There are a number of other Linux commands that you might find useful as you program your robot. The following is a table with some of the more useful commands:

Linux command	What it does
ls	List-short: This command lists all the files and directories in the current directory by just their names.
rm filename	Remove: This command deletes the file specified by filename .

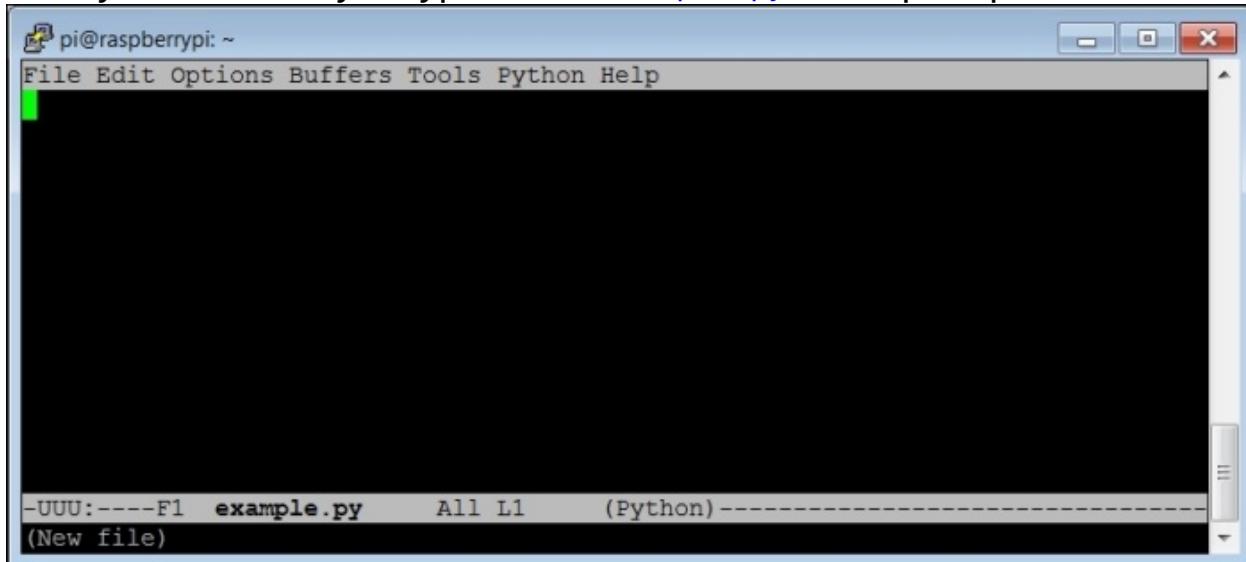
<code>mv filename1 filename2</code>	Move: This command renames <code>filename1</code> to <code>filename2</code> .
<code>cp filename1 filename2</code>	Copy: This command copies <code>filename1</code> to <code>filename2</code> .
<code>mkdir directoryname</code>	Make directory: This command creates a directory with the name specified by <code>directoryname</code> ; this will be made in the current directory unless otherwise specified.
<code>clear</code>	Clear: This command clears the current terminal window.
<code>sudo</code>	Super user: If you type the <code>sudo</code> command at the beginning of any command, it will execute that command as the super user. This may be required if the command or program you are trying to execute needs super user permissions. If, at any point in this book, you type a command or the name of the program you want to run and the result seems to suggest that the command does not exist or permission is denied, try doing it again with <code>sudo</code> at the beginning.

Now you can play around with the commands and look at your system and the files that are available to you. But, be a bit careful! Linux is not like Windows: it will not warn you if you try to delete or replace a file.

Creating, editing, and saving files on Raspberry Pi

Now that you can log in and move easily between directories and see which files are in them, you'll want to be able to edit those files. To do this, you'll need a program that allows you to edit the characters in a file. If you are used to working on Microsoft Windows, you have probably used programs such as Microsoft Notepad, WordPad, or Word to do this. As you can imagine, these programs are not available in Linux. There are several choices for editors, all of which are free. In this chapter, we will use an editor program called **Emacs**. Other possibilities are programs such as **nano**, **vi**, **vim**, and **gedit**. Programmers have strong feelings about which editor to use, so if you already have a favorite, you can skip this section.

If you want to use Emacs, download and install it by typing `sudo apt-get install emacs`. Once installed, you can run Emacs simply by typing `emacs filename`, where `filename` is the name of the file you want to edit. If the file does not exist, Emacs will create it. The following screenshot shows what you will see if you type `emacs example.py` in the prompt:



Notice that unlike Windows, Linux doesn't automatically assign file extensions; it is up to us to specify the kind of file we want to create. Notice that the Emacs editor has indicated, in the lower-left corner, that

you have opened a new file. Now, if you are using Emacs in the LXDE windows interface, either because you have a monitor, keyboard, and mouse hooked up or because you are running vncserver, you can use the mouse in much the same way as you do in Microsoft Word.

However, if you are running Emacs from SSH, you won't have the pointer available. So you'll need to navigate the file using the cursor keys. You'll also have to use some keystroke commands to save your file as well as accomplish a number of other tasks that you would normally use the mouse to select. For example, when you are ready to save the file, you must press *Ctrl + X* and *Ctrl + S*, and that will save the file under the current filename. When you want to quit Emacs, you must press *Ctrl + X* and *Ctrl + C*. This will stop Emacs and return you to the command prompt. If you are going to use Emacs, the following are a number of keystroke commands you might find useful:

The Emacs command	What it does
<i>Ctrl + X Ctrl + S</i>	Save: This command saves the current file
<i>Ctrl + X Ctrl + C</i>	Quit: This command causes you to exit Emacs and return to the command prompt
<i>Ctrl + K</i>	Kill: This command erases the current line
<i>Ctrl + _</i>	Undo: This command undoes the last action
Left-click and text selection followed by cursor placement and right-click	Cut and paste: If you select the text you want to paste by clicking the mouse, move the cursor to where you want to paste the code and then right-click on it; the code will be pasted in that location

Now that you have the capability to edit files, in the next section, you'll use this capability to create programs.

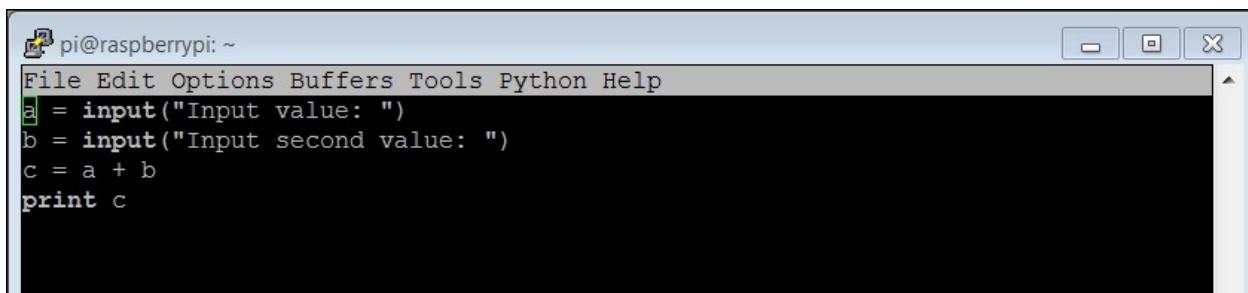
Creating and running Python programs on Raspberry Pi

Now that you are ready to begin programming, you'll need to choose a language. There are many available: C, C++, Java, Python, Perl, and a great deal of other possibilities. I'm going to initially introduce you to Python for two reasons: it is a simple language that is intuitive and very easy to use and it avails a lot of the open source functionality of the robotics world. We'll also cover a bit of C/C++ in this chapter as well, as some functionalities are only available in C/C++. But, it makes sense to start in Python. To work through the examples in this section, you'll need a version of Python installed. Fortunately, the basic Raspbian system has one already, so you are ready to begin.

We are only going to cover some of the very basic concepts here. 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 tutorials, visit www.codecademy.com or <http://www.learnpython.org/> and give it a try.

In this section, we'll cover how to create and run a Python file. It turns out that Python is an interactive language, so you could run it and then type in commands one at a time. But, we want to use Python to create programs, so we are going to type in our commands using Emacs and then run them in the command line by invoking Python. Let's get started.

Open an example Python file by typing `emacs example.py`. Now, let's put some code in the file. Start with the lines shown in the following screenshot:

A screenshot of an Emacs window on a Raspberry Pi. The title bar says "pi@raspberrypi: ~". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", "Python", and "Help". The main buffer contains the following Python code:

```
a = input("Input value: ")
b = input("Input second value: ")
c = a + b
print c
```

The cursor is positioned after the first line of code.

(Note that your code may be colored. I have removed the coloring here, so it is easier to read.)

Tip

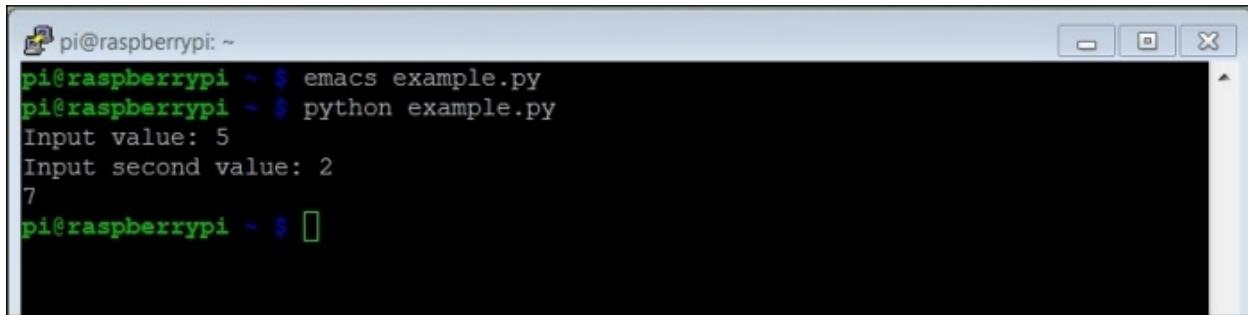
Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. 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.

Let's go through the code to see what is happening. The code lines are as follows:

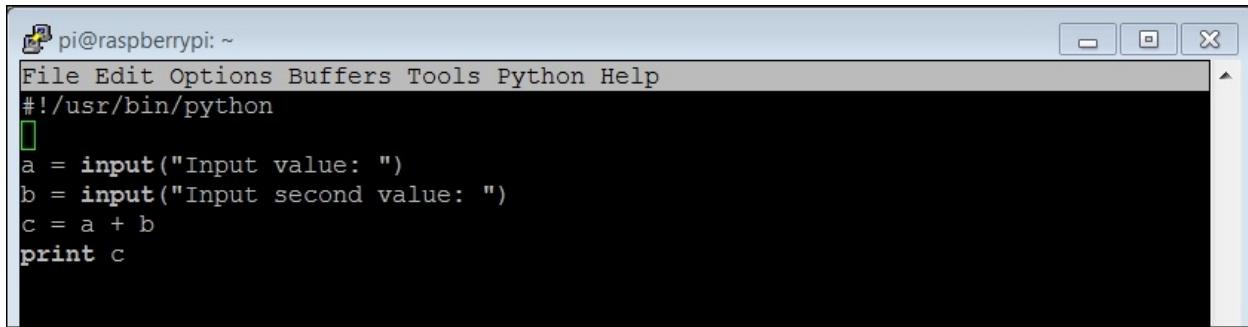
- `a = input("Input value: ")`— One of the basic purposes of a program is to get input from the user. `raw_input` allows us to do that. The data will be input by the user and stored in `a`. The prompt `Input value:` will be shown to the user.
- `b = input("Input second value: ")`— This data will also be input by the user and stored in `b`. The prompt `Input second value:` will be shown to the user.
- `c = a + b`— This is an example of something you can do with the data; in this example, you can add `a` and `b`.
- `print c`— Another basic purpose 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 `python example.py`. You should see the result as shown in the following screenshot:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example.py
pi@raspberrypi ~ $ python example.py
Input value: 5
Input second value: 2
7
pi@raspberrypi ~ $
```

You can also run the program right from the command line without typing `python example.py` by adding the line `#!/usr/bin/python` to the program. Then the program looks as shown in the following screenshot:



```
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
[green square icon]
a = input("Input value: ")
b = input("Input second value: ")
c = a + b
print c
```

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 `chmod +x example.py`. This will change the file's execution permissions, so the computer will now accept it and execute it. You should be able to simply type `./example.py` and see the program run, as shown in the following screenshot:

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example.py
pi@raspberrypi ~ $ python example.py
Input value: 5
Input second value: 2
7
pi@raspberrypi ~ $ emacs example.py
pi@raspberrypi ~ $ chmod +x example.py
pi@raspberrypi ~ $ ./example.py
Input value: 6
Input second value: 1
7
pi@raspberrypi ~ $ 
```

Notice that if you simply type `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 the system a path to it. In this case, `./` is the current directory.

Basic programming constructs on Raspberry Pi

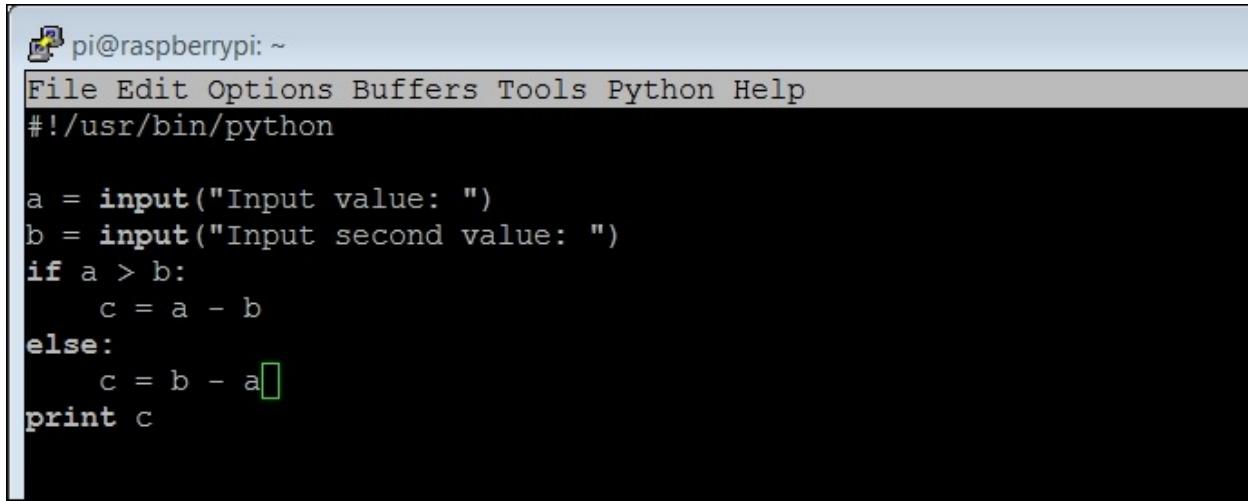
Now that you know how to enter and run a simple Python program on Raspberry Pi, let's look at some more complex programming tools. Specifically, we'll cover what to do when we want to determine what instructions to execute and show how to loop our code to do that more than once. I'll give a brief introduction on how to use libraries in the Python Version 2.7 code and how to organize statements into functions. Finally, we'll very briefly cover object-oriented code organization.

Note

Indentation in Python is very important; it will specify which group of statements are associated with a given loop or decision set, so watch your indentation carefully.

The if statement

As you have seen in previous examples, your programs normally start by executing the first line of code and then continue with the next lines until your program runs out of code. This is fine; but, what if you want to decide between two different courses of action? We can do this in Python using an **if statement**. The following screenshot shows some example code:



```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

a = input("Input value: ")
b = input("Input second value: ")
if a > b:
    c = a - b
else:
    c = b - a
print c
```

The following is the detail of the code shown in the previous screenshot, line by line:

- `#!/usr/bin/python` – This is included so you can make your program executable.
- `a = input("Input value: ")` – One of the basic needs of a program is to get input from the user. `raw_input` 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.
- `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.
- `if a > b:` – This is an `if` statement. The expression evaluated in this case is `a > b`. If it is `true`, the program will execute the next statement(s) that is indented. If not, it will skip that statement(s); in this case, `c = a - b`.
- `else:` – The `else` statement is an optional part of the command. If the expression in the `if` statement is evaluated as `false`, the indented statement(s) will be executed; in this case, `c = b - a`.
- `print c` – Another basic purpose of our program is to print out results. The `print` command prints out the value of `c`.

You can run the previous program a couple of times, checking both the `true` and `false` possibilities of the `if` expression as shown in the following screenshot:

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ./example.py
Input value: 5
Input second value: 2
3
pi@raspberrypi ~ $ ./example.py
Input value: 3
Input second value: 8
5
pi@raspberrypi ~ $ []
```

The while statement

Another useful construct is the **while** construct; it will allow us to execute a set of statements over and over until a specific condition has been met. The following screenshot shows a set of code that uses this construct:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

a = 0
b = 1
while a != b:
    a = input("Input value: ")
    b = input("Input second value: ")
    c = a + b
    print c
```

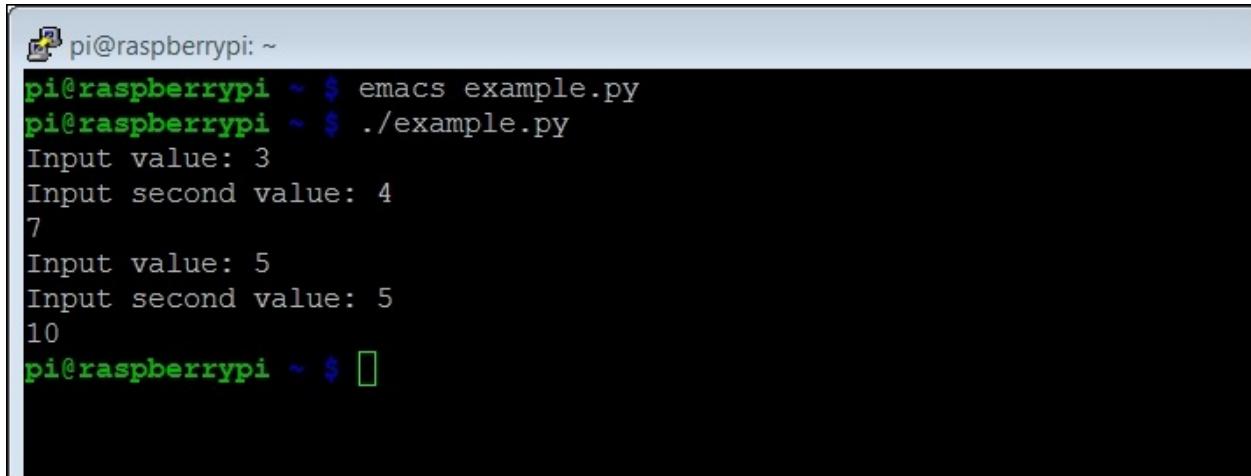
The following are the details of the code shown in the previous screenshot:

- `#!/usr/bin/python` – This is included so you can make your program executable.
- `a = 0` – This line sets the value of variable `a` to `0`. We'll need this only to make sure we execute the loop at least once.
- `b = 1` – This line sets the value of the variable `b` to `1`. We'll need this

only to make sure we execute the loop at least once.

- `while a != b:` – The expression `a != b` (in this case, `!=` means not equal to) is verified. If it is `true`, the statement(s) that is indented is executed. When the statement is evaluated as `false`, the program jumps to the statements after the indented section.
- `a = input("Input value: ")` – One of the basic purposes of a program is to get input from the user. `raw_input` 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.
- `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.
- `c = a + b` – The variable `c` is loaded with the sum of `a` and `b`.
- `print c` – The `print` command prints out the value of `c`.

Now you can run the program. Notice that when you enter the same value for `a` and `b`, the program stops, as shown in the following screenshot:



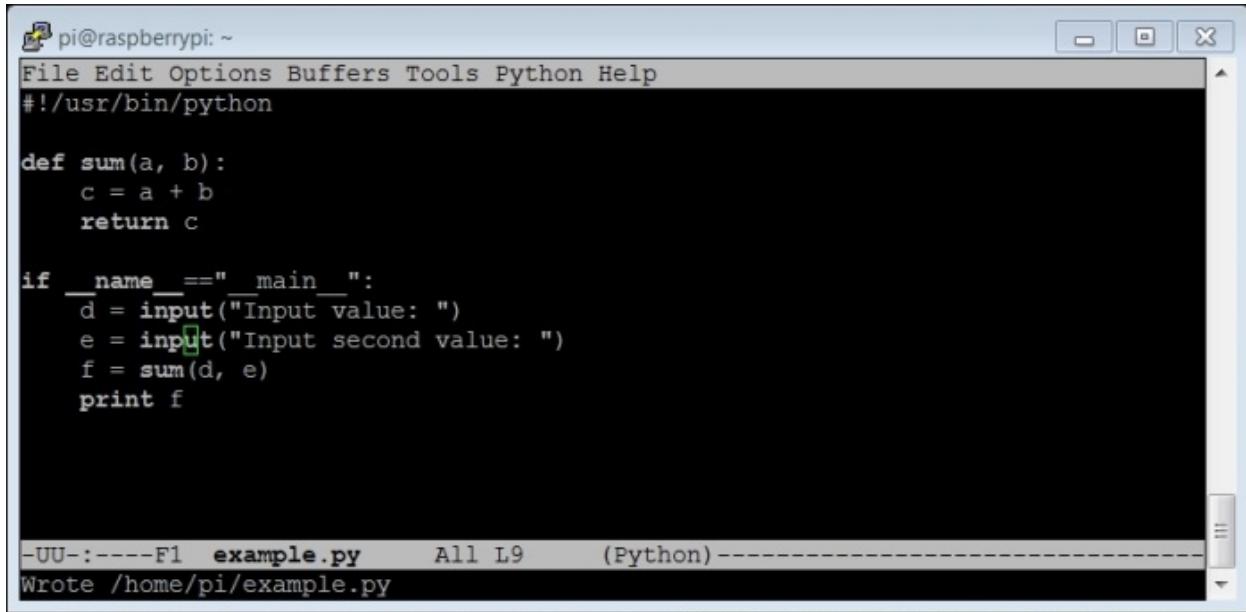
The screenshot shows a terminal window on a Raspberry Pi. The session starts with the user logging in as 'pi'. The user runs the command `emacs example.py` to edit a file named `example.py`. After saving the file, the user runs the command `./example.py`. The program prompts for two inputs: 'Input value:' and 'Input second value:'. Both inputs are set to 5. The program then calculates the sum (10) and prints it. Finally, the user exits the terminal.

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example.py
pi@raspberrypi ~ $ ./example.py
Input value: 3
Input second value: 4
7
Input value: 5
Input second value: 5
10
pi@raspberrypi ~ $ []
```

Working with functions

The next concept we need to cover is how to put a set of statements into a function. We use functions to organize code, grouping sets of statements together when it makes sense that they be organized and in the same location. For example, if we have a specific calculation that we might want to perform many times, instead of copying the set of statements each time we want to perform it, we group them into a

function. I'll use a fairly simple example here, but if the calculation takes a significant number of programming statements, you can see how that would make our code significantly more efficient. The following screenshot shows the code:



The screenshot shows a terminal window titled "example.py" with the following Python code:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

def sum(a, b):
    c = a + b
    return c

if __name__ == "__main__":
    d = input("Input value: ")
    e = input("Input second value: ")
    f = sum(d, e)
    print f
```

The status bar at the bottom of the terminal window indicates: "-UU-----F1 example.py All L9 (Python)-" and "Wrote /home/pi/example.py".

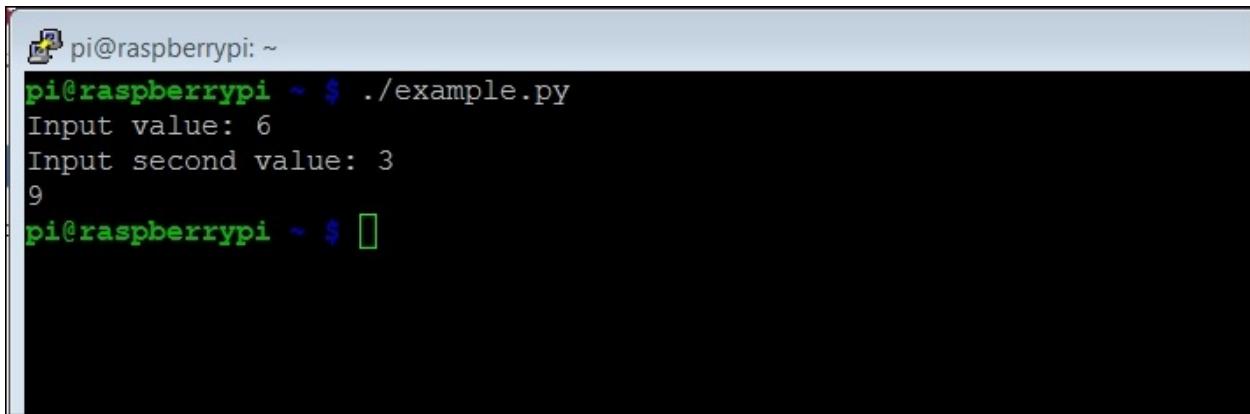
The following is the explanation of the code from our previous example:

- `#!/usr/bin/python` – This is included so you can make your program executable.
- `def sum(a, b):` – This line defines a function named `sum`. The `sum` function takes `a` and `b` as arguments.
- `c = a + b` – Whenever this function is called, it will add the values in the variable `a` to the values in variable `b`.
- `return c` – When the function is executed, it will return the variable `c` to the calling expression.
- `if __name__ == "__main__":` – In this particular case, you don't want your program to start executing each statement from the top of the file; you would rather it started at a particular point. This line tells the program to begin its execution at this particular point.
- `d = input("Input 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.
- `e = 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.

- `f = sum(d, e)` – The function `sum` is called. In the `sum` function, the value in variable `d` is copied to the variable `a` and the value in the variable `e` is copied to the variable `b`. The program then goes to the `sum` function and executes it. The return value is then stored in the variable `f`.
- `print f` – The `print` command prints out the value of `f`.

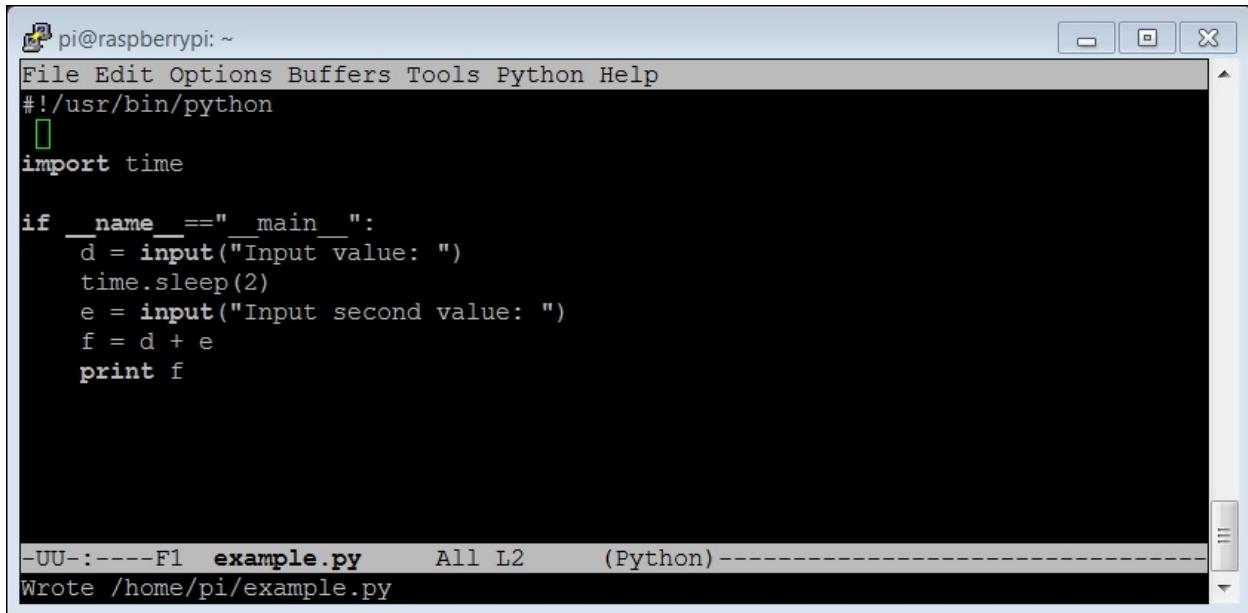
The following screenshot is the result received when you run the code:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ./example.py
Input value: 6
Input second value: 3
9
pi@raspberrypi ~ $
```

Libraries/modules in Python

The next topic we need to cover is how to add functionality to our programs using libraries/modules. Libraries, or modules as they are sometimes called in Python, include functionality that someone else has created and that you want to add to your code. As long as the functionality exists and your system knows about it, you can include the library in the code. So, let's modify our code again by adding the library as shown in the following screenshot:



A screenshot of a terminal window titled "example.py" in a Python environment. The window shows the following code:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
[green square icon]
import time

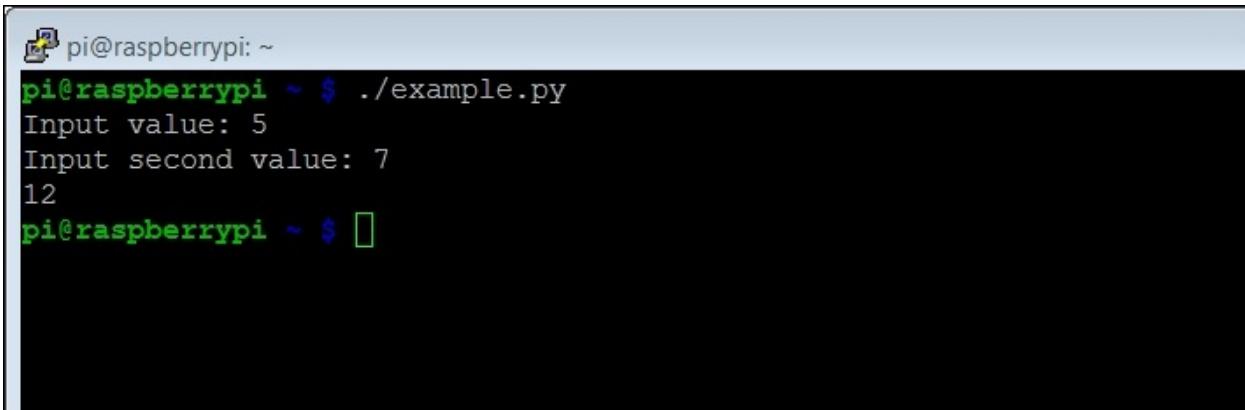
if __name__ == "__main__":
    d = input("Input value: ")
    time.sleep(2)
    e = input("Input second value: ")
    f = d + e
    print f

-UU-----F1  example.py      All L2      (Python)-----
Wrote /home/pi/example.py
```

The following is the line-by-line description of the code:

- `#!/usr/bin/python` – This is included so you can make your program executable.
- `import time` – This includes the `time` library. The `time` library includes a function that allows you to pause for a specified number of seconds.
- `if __name__ == "__main__":` – In this particular case, you don't want your program to start executing each statement from the top of the file; you would rather it started from a particular line. This line tells the program to begin its execution at this specified point.
- `d = input("Input 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.
- `time.sleep(2)` – This line calls the `sleep` function in the `time` library, which will cause a two-second delay.
- `e = 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.
- `f = d + e` – The `f` variable is loaded with the value `d + e`.
- `print f` – The `print` command prints out the value of `f`.

The following screenshot shows the result after running the previous example code:

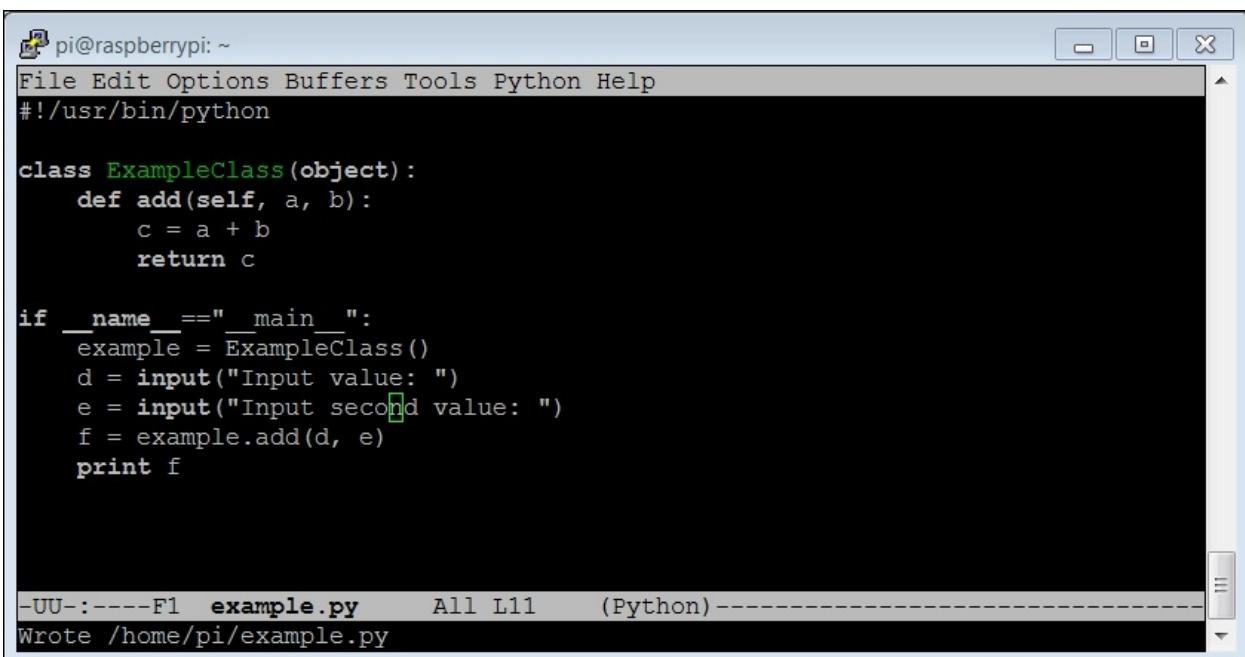


```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ./example.py
Input value: 5
Input second value: 7
12
pi@raspberrypi ~ $
```

Of course, this looks very similar to other results. But, you will notice a pause between you entering the first value and the second value appearing.

The object-oriented code

The final topic we need to cover is object-oriented organization in our code. In object-oriented code, we organize a set of related functions in an object. If, for example, we have a set of functions that are all related, we can place them in the same class and then call them by associating them with a specified class. This is a complex and difficult topic, but let me just show you a simple example in the following screenshot:



```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

class ExampleClass(object):
    def add(self, a, b):
        c = a + b
        return c

if __name__=="__main__":
    example = ExampleClass()
    d = input("Input value: ")
    e = input("Input second value: ")
    f = example.add(d, e)
    print f

-UU-----F1  example.py      All L11      (Python)-----
Wrote /home/pi/example.py
```

The following is an explanation of the code in the previous screenshot:

- `#!/usr/bin/python` – This is included so you can make your program executable.
- `class ExampleClass(object):` – This defines a class named `ExampleClass`. This class can have any number of functions associated with it.
- `def add(self, a, b):` – This defines the function `add` as part of `ExampleClass`. We can have functions that have the same names as long as they belong to different classes. This function takes two arguments: `a` and `b`.
- `c = a + b` – This statement indicates the simple addition of two values.
- `return c` – This function returns the result of the addition.
- `if __name__=="__main__":` – In this particular case, you don't want your program to start executing each statement from the top of the file; you would rather it started from a specific line. This line tells the program to begin its execution at the specified point.
- `example = ExampleClass()` – This defines a variable named `example`, the type of which is `ExampleClass`. This variable now has access to all the functions and variables associated with the class `ExampleClass`.
- `d = input("Input value: ")` – This data will also be input by the user and stored in the variable `b`. The prompt `Input second value:` will be shown to the user.
- `e = 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.
- `f = example.add(d,e)` – The instance of `ExampleClass` is called and its function `add` is executed by sending `d` and `e` to that function. The result is returned and stored in `f`.
- `print f` – The `print` command prints out the value of the variable `f`.

The result after running the previous example code is as shown in the following screenshot:

 pi@raspberrypi: ~
pi@raspberrypi ~ \$./example.py
Input value: 9
Input second value: 12
21
pi@raspberrypi ~ \$

The result shown in the previous screenshot is the same as the other codes discussed earlier, and there is no functionality difference; however, object-oriented techniques have been used to keep similar functions organized together to make the code easier to maintain. This also makes it easier for others to use your code.

Now that you have a feel for the basics of Python coding, I'll introduce you very briefly to the C coding language.

Introduction to the C/C++ programming language

Now that you've been introduced to a simple programming language in Python, we need to spend a bit of time talking about a more complex but powerful language called C. C is the original language of Linux and has been around for many decades, but it is still widely used by open source developers. It is similar to Python, but is also a bit different; since you may need to understand and make changes to C code, you should be familiar with it and how it is used.

As with Python, you will need to have access to the language capabilities of C. These come in the form of a compiler and build system, which turns your text files that contain programs into machine code that the processor can actually execute. To do this, type `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 required for C/C++ are installed, let's walk through some simple examples. The following screenshot shows the first C/C++ code example:

The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window contains a C++ code editor with the following code:

```
#include <iostream>

int main()
{
    int a;
    int b;
    int c;

    std::cout << "Input value: ";
    std::cin >> a;
    std::cout << "Input second value: ";
    std::cin >> b;

    c = a + b;

    std::cout << c << std::endl;

    return 0;
}
```

The status bar at the bottom of the window displays "-UU:----F1 example2.cpp All L13 (C++/l Abbrev)-----" and "wrote /home/pi/example2.cpp".

The following is an explanation of the code shown in the previous screenshot:

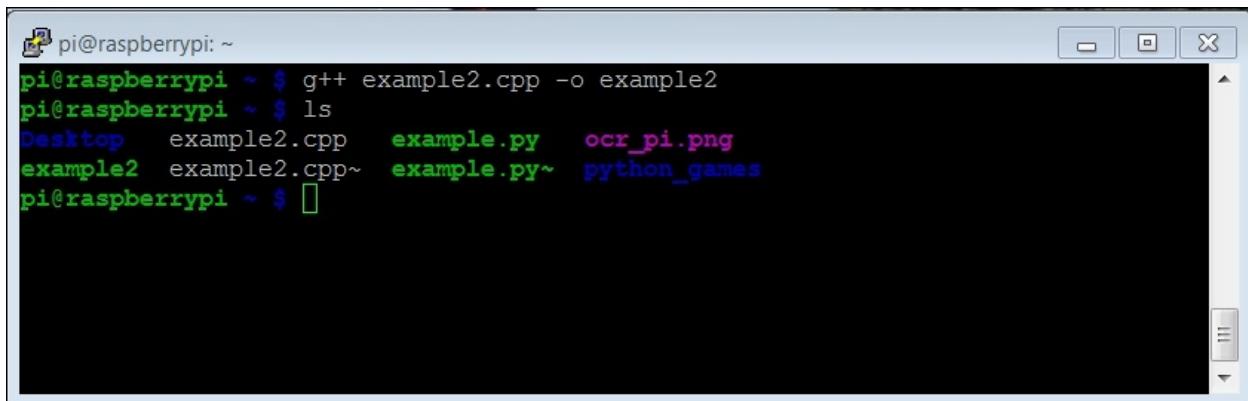
- `#include <iostream>` – This is a library that is included so your program can input data using the keyboard and output information to the screen.
- `int main()` – As with Python, we can place functions and classes in the file, but you will always want to start a program's execution at a known point; C defines this known point as the main function.
- `int a;` – This defines a variable named `a`, which is of the type `int`. C is what we call a strongly typed language, which means we need to declare the type of the variable we are defining. The normal types are as follows:
 - `int`: This is used for numbers that have no decimal points
 - `float`: This is used for numbers that require decimal points
 - `char`: This is used for a character of text
 - `bool`: This is used for a true or false value

Also note that every line in C ends with the `;` character.

- `int b;` – This defines a variable named `b`, which is of the type `int`.
- `int c;` – This defines a variable named `c`, which is of the 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 be stored in 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 `a`.
- `c = a + b;` – The statement is the 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 of this line 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 `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` (specified as the name after the `-o` option).

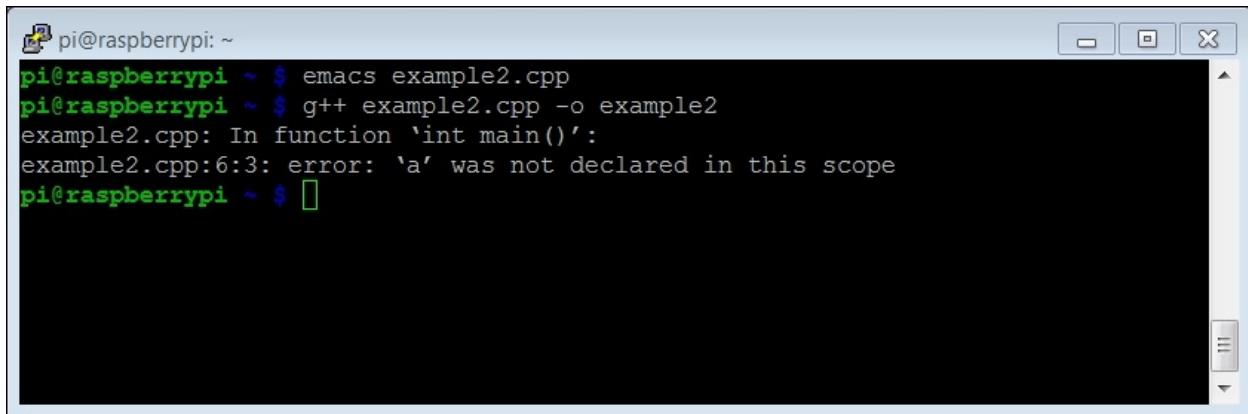
If you run an `ls` command on your directory after you have compiled this program, you should see the `example2` file in your directory as shown in the following screenshot:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ g++ example2.cpp -o example2
pi@raspberrypi ~ $ ls
Desktop  example2.cpp  example.py  ocr_pi.png
example2  example2.cpp~  example.py~  python_games
pi@raspberrypi ~ $
```

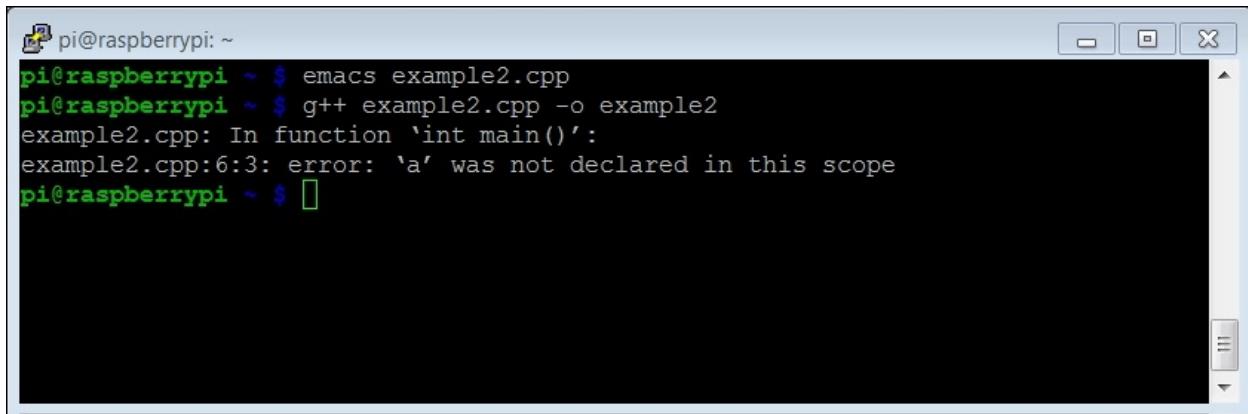
By the way, if you run into a problem, the compiler will try to help you figure it out. If, for example, you were to forget to include the `int` type

before the `int a;` declaration, you would get the error shown in the following screenshot when you try to compile the previous code:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example2.cpp
pi@raspberrypi ~ $ g++ example2.cpp -o example2
example2.cpp: In function 'int main()':
example2.cpp:6:3: error: 'a' was not declared in this scope
pi@raspberrypi ~ $
```

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 `./example2` and you should be able to obtain the following result:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example2.cpp
pi@raspberrypi ~ $ g++ example2.cpp -o example2
example2.cpp: In function 'int main()':
example2.cpp:6:3: error: 'a' was not declared in this scope
pi@raspberrypi ~ $
```

We will not repeat the Python tutorial for C in this book; there are several good tutorials on the Internet that can help; for example, <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 seems fairly straightforward. However, if you have your functionality distributed among a lot of files, or need lots of libraries for your programs, the command-line approach to executing a compile can get unwieldy.

The C development environment provides a way to automate the compile

process; this is called the make process. When performing this process, you create a text program named `makefile` that defines the files you want to include and compile. Instead of typing a long command or set of commands, you simply type `make` and the system will execute a compile based on the definitions in the `makefile` program. There are several good tutorials that discuss this system more; try visiting <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.

If you are going to do a significant amount of coding, you'll want to install an **IDE (Integrated Development Environment)**. These environments make it much easier to see, edit, compile, and debug your programs. The most popular of these programs in the Linux world is called **Eclipse**. If you'd like to know more, start with a Google search or go to <http://www.eclipse.org/>.

Summary

In this chapter, you've hopefully learned how to interact with the Raspbian operating system using the command line and also create and edit files using Emacs. You have also been exposed to both the Python and C programming languages. If this is your first experience with programming, don't be surprised if you are still very uneasy with programing in general, and if and while statements in particular. You probably felt just as uncomfortable during your first introduction to the English language: you may not remember it.

It is always a bit difficult to try new things. However, I will try to give you explicit instructions on what to type so that you can be successful. There is one major challenge in working with computers. They always do exactly what you tell them to do and not necessarily what you want them to. So if you encounter problems, check several times to make sure that your code matches the example exactly. Now, on to some actual coding!

In the next chapter, you'll start in earnest adding additional functionality that will enable you to create amazing robotics projects. You'll start by providing your system with the capability to speak and also listen to your commands.

Chapter 3. Providing Speech Input and Output

Now that your Raspberry Pi is up and operating, let's start giving your projects some basic functionality. You'll use this functionality in later chapters as you build wheeled robots, tracked robots, robots that can walk, and even sail or fly. We're going to start with speech; it is a good basic project and offers several examples of adding capability in both hardware and software. So, buckle up and get ready to learn the basics of interfacing with our board by facilitating speech.

You'll be adding a microphone and speaker to your Raspberry Pi. You'll also be adding functionality so the robot can recognize voice commands and respond via the speaker. Additionally, you'll be able to issue voice commands and make the robot respond with an action. When you're free from typing in commands, you can interact with your robotic projects in an impressive way. This project will require adding both hardware and software.

Interfacing with your projects via speech is more fun than typing in commands, and it allows interaction with your project without using a keyboard or mouse. Besides, which self-respecting robot wants to carry around a keyboard? No, you want to interact in natural ways with your projects, and this chapter will teach you how. Interfacing via speech also helps you find your way around the board, learn how to use available free functionality, and become familiar with the community of open source developers. This chapter covers the capabilities needed before you can see or move the project.

In this chapter, we'll specifically cover the following points:

- Hooking up the hardware to make and input sound
- Using **Espeak** to allow your projects to respond in a robot voice
- Using **PocketSphinx** to interpret your commands
- Providing the capability to interpret your commands and having your robot initiate action

Before beginning this project, you'll need a working Raspberry Pi that has power connection and a LAN connection (refer to [Chapter 1, Getting Started with Raspberry Pi](#), for instructions). Additionally, this project requires a USB microphone or speaker adapter. The board itself has an audio out but does not have an audio in. The HDMI output does support audio, but most of your robotics projects will not be connected to HDMI monitors with speaker capability.

You'll need the following three pieces of hardware:

- A USB device that supports microphone in and speaker out.



- A microphone that can plug into the USB device.



- A powered speaker that can plug into the USB device.



Fortunately, these devices are very inexpensive and widely available. Make sure the speaker is powered because your board will generally not be able to drive a passive speaker with enough power for your applications. The speaker can use either internal battery power or an externally powered USB hub. Many of your robotics projects will require a powered USB hub, so it's a good investment.

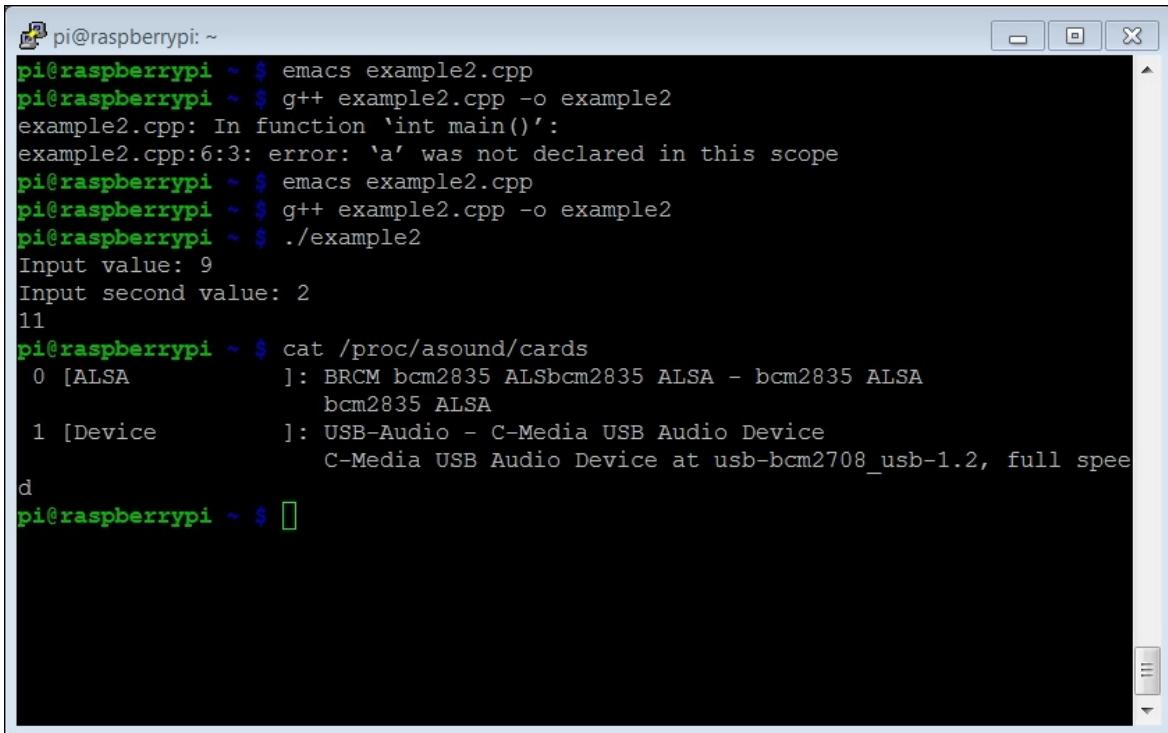
Hooking up the hardware to make and input sound

For this task, we are going to hook up our hardware so that we can record and play sound. To do this, reassemble your Raspberry Pi. Plug in the LAN cable. Connect the powered USB hub and plug in the microphone or speaker USB device. Also, plug in your speakers and the microphone. The entire system should look like the following image:



Plug in the power. You can execute all of the following instructions in one of the following ways:

- If you are still connected to the display, keyboard, and mouse, log into the system and use the Windows system by opening an LXTerminal window.
- If you are only connected via LAN, you can do all of this using an SSH terminal window; so as soon as your board flashes that it has power, open up an SSH terminal window using PuTTY or some similar terminal emulator. Once the terminal window comes up, log in with your username and password. Now, type in `cat` `procasound/cards`. You should see the response, as shown in the following screenshot:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ emacs example2.cpp
pi@raspberrypi ~ $ g++ example2.cpp -o example2
example2.cpp: In function 'int main()':
example2.cpp:6:3: error: 'a' was not declared in this scope
pi@raspberrypi ~ $ emacs example2.cpp
pi@raspberrypi ~ $ g++ example2.cpp -o example2
pi@raspberrypi ~ $ ./example2
Input value: 9
Input second value: 2
11
pi@raspberrypi ~ $ cat /proc/asound/cards
 0 [ALSA]           : BRCM bcm2835 ALSbcm2835 ALSA - bcm2835 ALSA
                      bcm2835 ALSA
 1 [Device]         : USB-Audio - C-Media USB Audio Device
                      C-Media USB Audio Device at usb-bcm2708_usb-1.2, full spee
d
pi@raspberrypi ~ $
```

Notice that the system thinks there are two possible audio devices. The first is the internal Raspberry Pi audio connected to the audio port and the second is your USB audio plugin. Although you could use the USB audio plugin to record sound and the Raspberry Pi audio out to play the sound, it is easier to just use the USB audio plugin to both create and record sound.

First, let's play some music to test if the USB sound device is working. You'll need to configure your system to look for your USB audio plugin and use it as the default plugin to play and record sound. To do this, you'll need to add a couple of libraries to your system. The first of these are some ALSA libraries. **ALSA** stands for **Advanced Linux Sound Architecture**. It will enable your sound system on Raspberry Pi by performing the following steps:

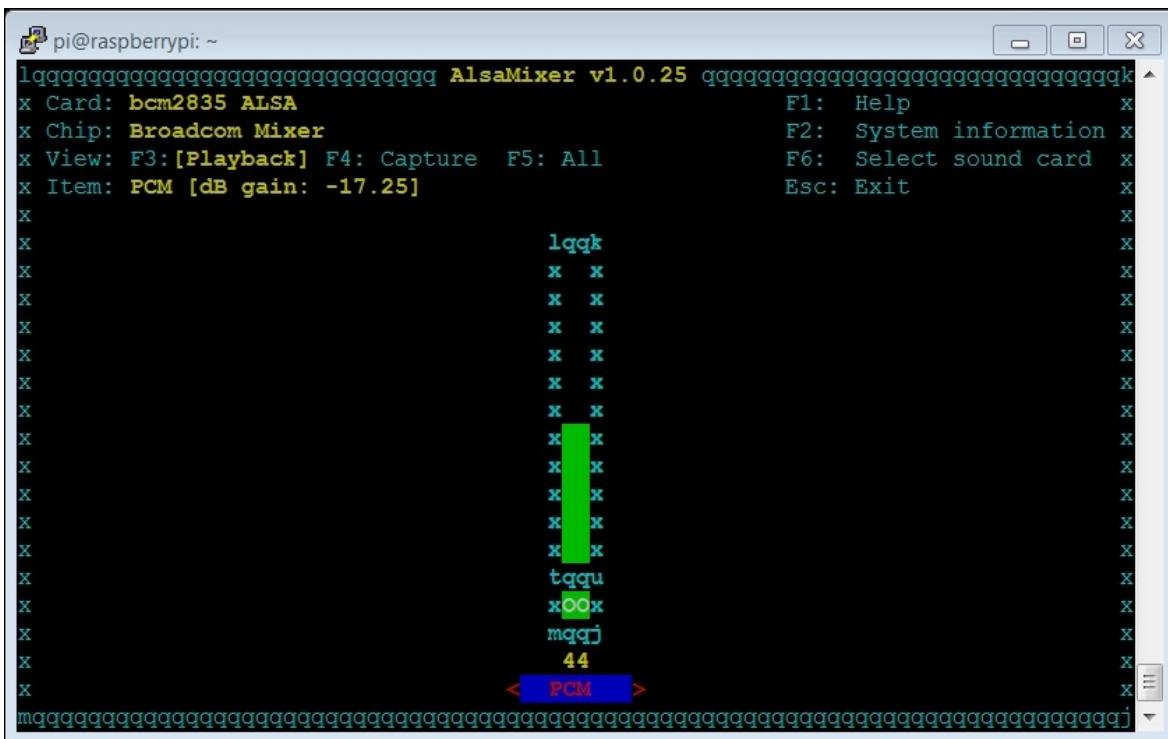
1. Firstly, install two libraries associated with ALSA by typing `sudo apt-get install alsa-base alsa-utils`.
2. Then, install some files that help provide the sound library by typing `sudo apt-get install libasound2-dev`.

If your system already contains these libraries, Linux will simply tell you that they are already installed or that they are up to date. After installing

both libraries, reboot your Raspberry Pi. It takes time, but the system needs a reboot after new libraries or hardware is installed.

Now we'll use an application named `alsamixer` to control the volume of both the input and the output of our USB sound card. To do this, perform the following steps:

1. Type `alsamixer` on the prompt. You should see a screen that looks as follows:



2. Press `F6` and select your USB sound device using the arrow keys. For example, refer to the following screenshot:

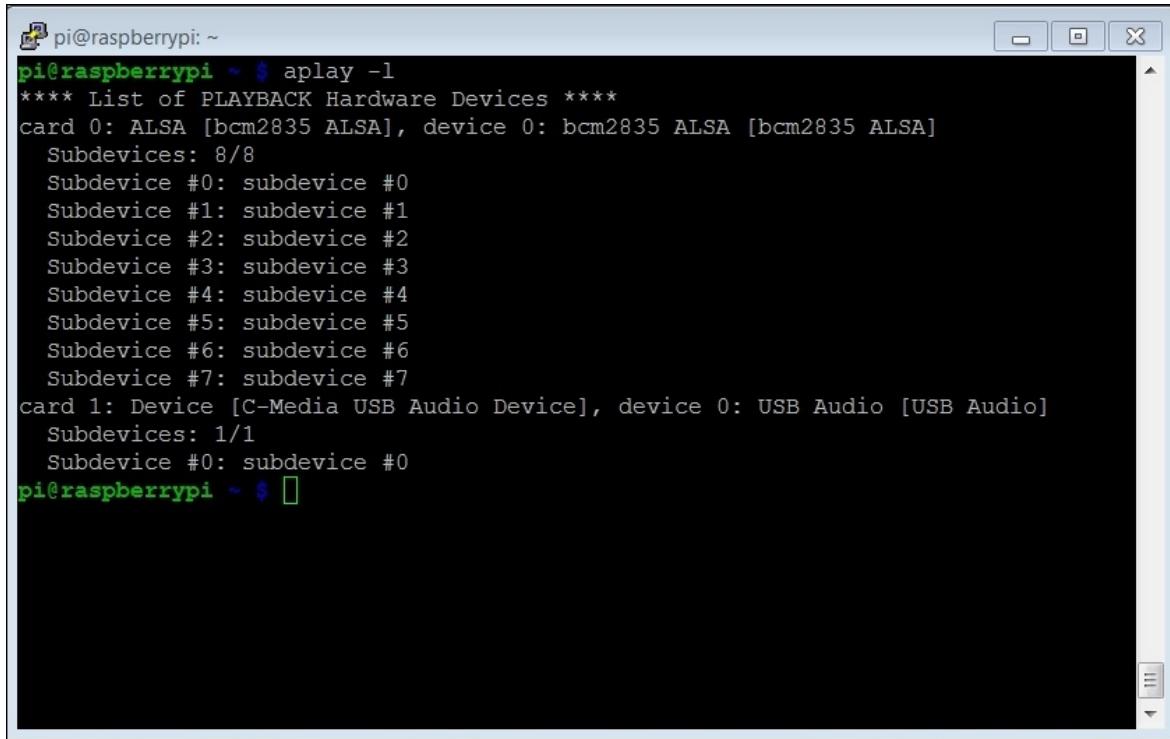
3. **C-Media USB Audio Device** is my USB audio device. You should now see a screen that looks like the following screenshot:

```
pi@raspberrypi: ~
lqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq AlsaMixer v1.0.25 qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
x Card: C-Media USB Audio Device
x Chip: USB Mixer
x View: F3:[Playback] F4: Capture F5: All
x Item: Speaker [dB gain: -6.63, -6.63]
x
x          lqqk          lqqk
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          x x          x x
x          tqqu          tqqu          lqqk
x          xOOx          xMMx          xOOx
x          mqqj          mqqj          mqqj
x          66<->66          52
x < Speaker > Mic Auto Gain Control
```

4. You can use the arrow keys to set the volume for both, the speakers

and the microphone. Use the **M** key to unmute the microphone. In the preceding screenshot, **MM** is mute and **∞** is unmute.

- Let's make sure our system knows about our USB sound device. On the prompt, type `aplay -l`. You should now see the following screenshot:

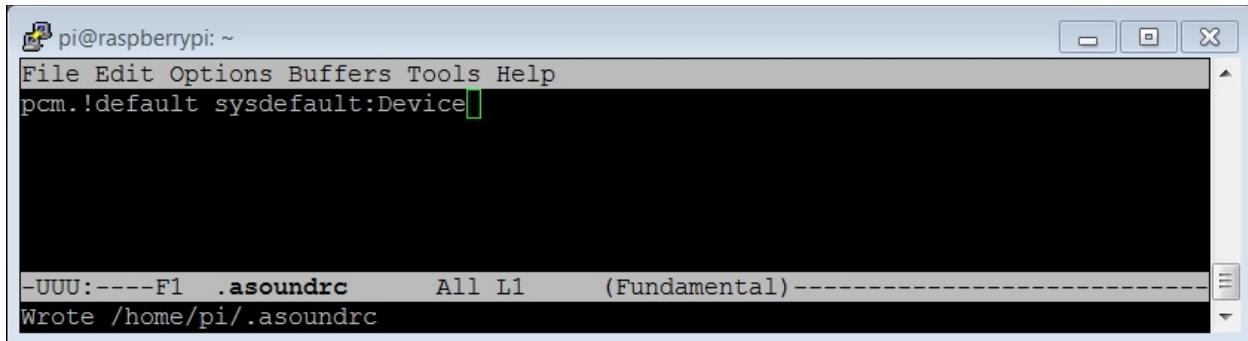


```
pi@raspberrypi: ~
pi@raspberrypi ~ $ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: ALSA [bcm2835 ALSA], device 0: bcm2835 ALSA [bcm2835 ALSA]
    Subdevices: 8/8
    Subdevice #0: subdevice #0
    Subdevice #1: subdevice #1
    Subdevice #2: subdevice #2
    Subdevice #3: subdevice #3
    Subdevice #4: subdevice #4
    Subdevice #5: subdevice #5
    Subdevice #6: subdevice #6
    Subdevice #7: subdevice #7
card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]
    Subdevices: 1/1
    Subdevice #0: subdevice #0
pi@raspberrypi ~ $
```

If this did not work, try `sudo aplay -l`. Once you have added the libraries, you'll need to add a file. You are going to add a file in your home directory with the name `.asoundrc`. This will be read by your system and used to set your default configuration. To do this, perform the following steps:

- Open the file named `.asoundrc` using your favorite editor.
- Type in `pcm.!default sysdefault:Device`.
- Save the file.

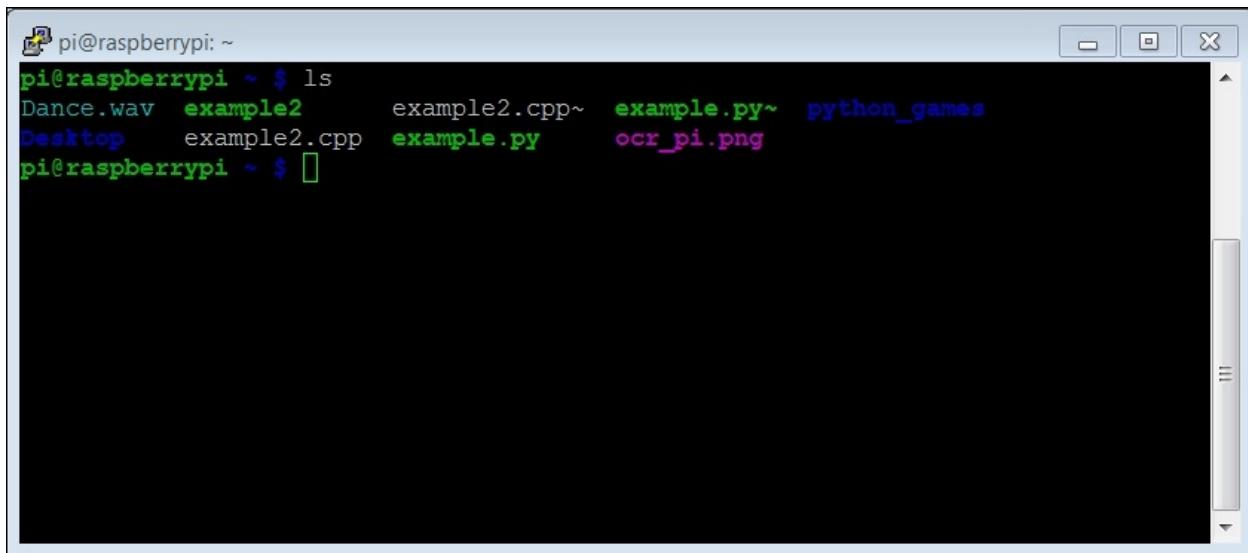
The file should look as follows:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows a file editor with the file ".asoundrc" open. The content of the file is "pcm.!default sysdefault:Device". The status bar at the bottom indicates "-UUU:---F1 .asoundrc All L1 (Fundamental)---" and "Wrote /home/pi/.asoundrc".

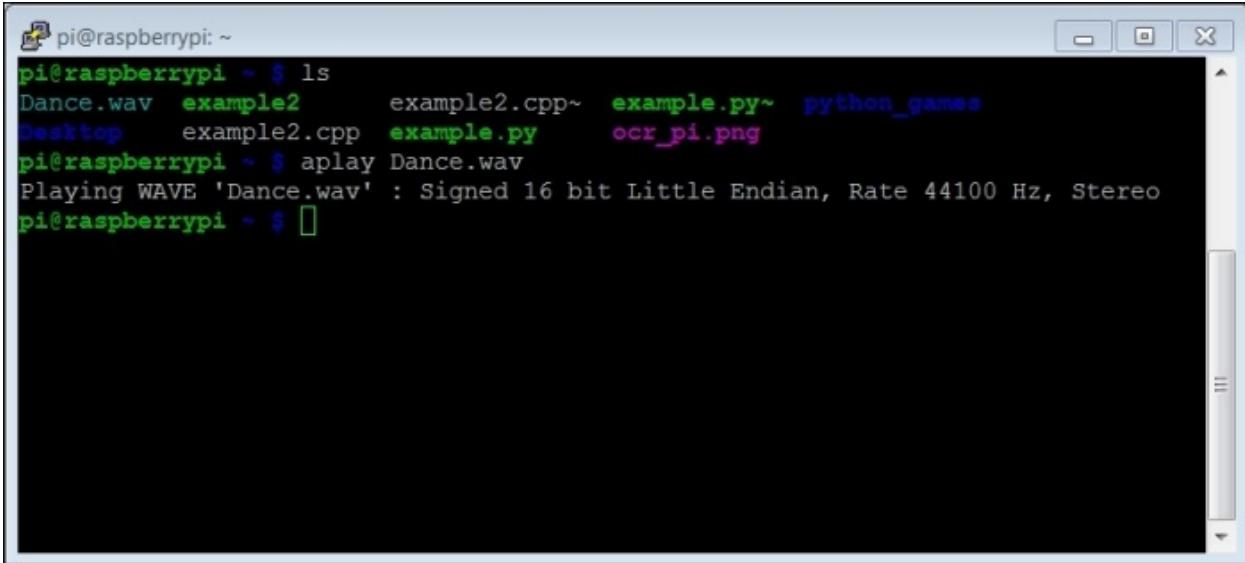
This will tell the system to use your USB device as a default. Once you have completed this, reboot your system again.

Now we'll play some music. To do this, you need a sound file and a device to play it. I used **WinSCP** from my Windows machine to transfer a simple [.wav](#) file to my Raspberry Pi. If you are using a Linux machine as your host, you can also use [scp](#) from the command line to transfer the file. You could also just download some music to Raspberry Pi using a web browser if you have a keyboard, mouse, and display connected. You can use an application named [aplay](#) to play your sound. You should see the music file by simply typing [ls](#), which stands for **list short**, as shown in the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window shows the output of the "ls" command, which lists files in the current directory: "Dance.wav", "example2", "example2.cpp", "example2.cpp~", "example.py", "example.py~", "python_games", "Desktop", and "ocr_pi.png".

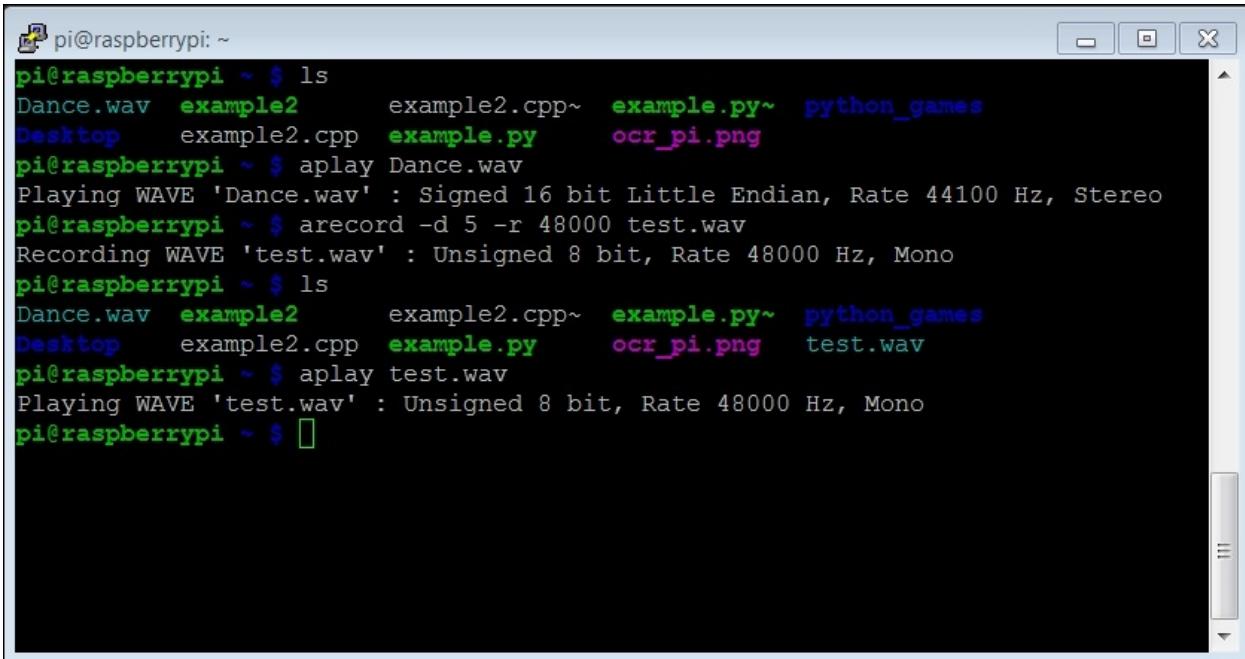
Now, type [aplay Dance.wav](#) to see if you can play music using the [aplay](#) music player. You will see the result (and hopefully hear it) as shown in the following screenshot:



```
pi@raspberrypi: ~
pi@raspberrypi: ~ $ ls
Dance.wav  example2  example2.cpp~  example.py~  python_games
desktop   example2.cpp  example.py    ocr_pi.png
pi@raspberrypi: ~ $ aplay Dance.wav
Playing WAVE 'Dance.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo
pi@raspberrypi: ~ $
```

If you aren't hearing any music, check the volume you set with [alsamixer](#) and the power of your speaker. Also, [aplay](#) can be a bit finicky about the type of files it accepts, so you may have to try different [.wav](#) files until [aplay](#) accepts one. One more thing to try if the system doesn't seem to know about the program is to type [sudo aplay Dance.wav](#).

Now that we can play sound, let's record some sound. To do this, we're going to use the [arecord](#) program. On the prompt, type [arecord -d 5 -r 48000 test.wav](#). This will record the sound at a sample rate of 48000 Hz per 5 seconds. Once you have typed the command, either speak into the microphone or make some other recognizable sound. You should see the following output in the terminal:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ls
Dance.wav  example2  example.cpp~  example.py~  python_games
Desktop   example2.cpp  example.py    ocr_pi.png
pi@raspberrypi ~ $ aplay Dance.wav
Playing WAVE 'Dance.wav' : Signed 16 bit Little Endian, Rate 44100 Hz, Stereo
pi@raspberrypi ~ $ arecord -d 5 -r 48000 test.wav
Recording WAVE 'test.wav' : Unsigned 8 bit, Rate 48000 Hz, Mono
pi@raspberrypi ~ $ ls
Dance.wav  example2  example.cpp~  example.py~  python_games
Desktop   example2.cpp  example.py    ocr_pi.png    test.wav
pi@raspberrypi ~ $ aplay test.wav
Playing WAVE 'test.wav' : Unsigned 8 bit, Rate 48000 Hz, Mono
pi@raspberrypi ~ $
```

Once you have created the file, play it with `aplay`. Type `aplay test.wav` and you should hear the recording. If you can't hear your recording, check `alsamixer` to make sure your speakers and microphone are both unmuted.

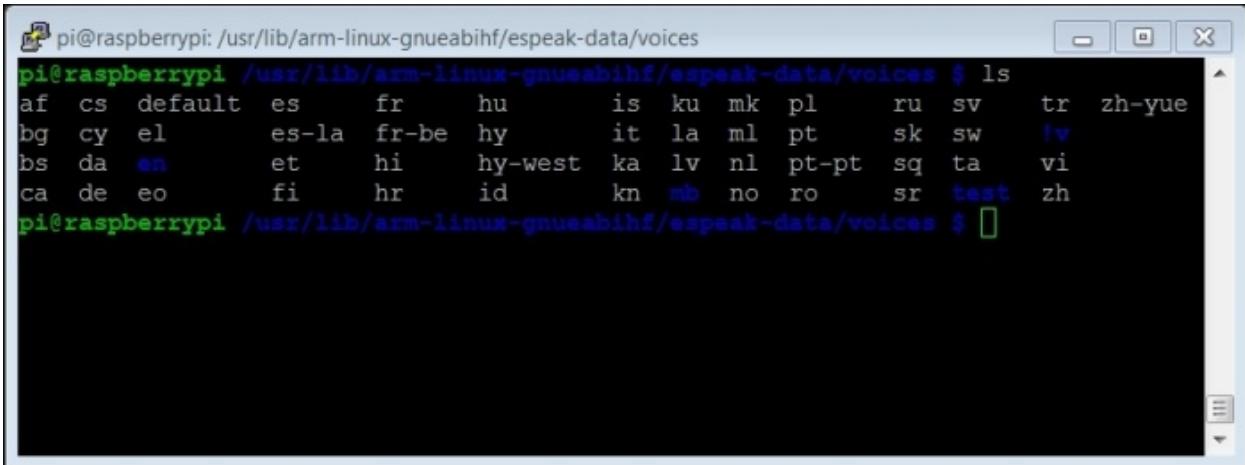
Now you can play music or other sound files using your Raspberry Pi. You can change the volume of your speaker and record your voice or other sounds on the system. You're now ready for the next step.

Using Espeak to allow our projects to respond in a robot voice

Sound is an important tool in our robotic toolkit, but you will want to do more than just play music. Let's allow our robot to speak. You're going to start by enabling Espeak, an open source application that provides us with a computer voice. Espeak is an open source voice generation application. To get this free functionality, download the Espeak library by typing `sudo apt-get install espeak` on the prompt. The download may take a while, but the prompt will reappear when it is complete. Now, let's see if Raspberry Pi has a voice. Type the `espeak "hello"` command. The speaker should emit a computer-voiced hello. If it does not, check the speakers and volume level.

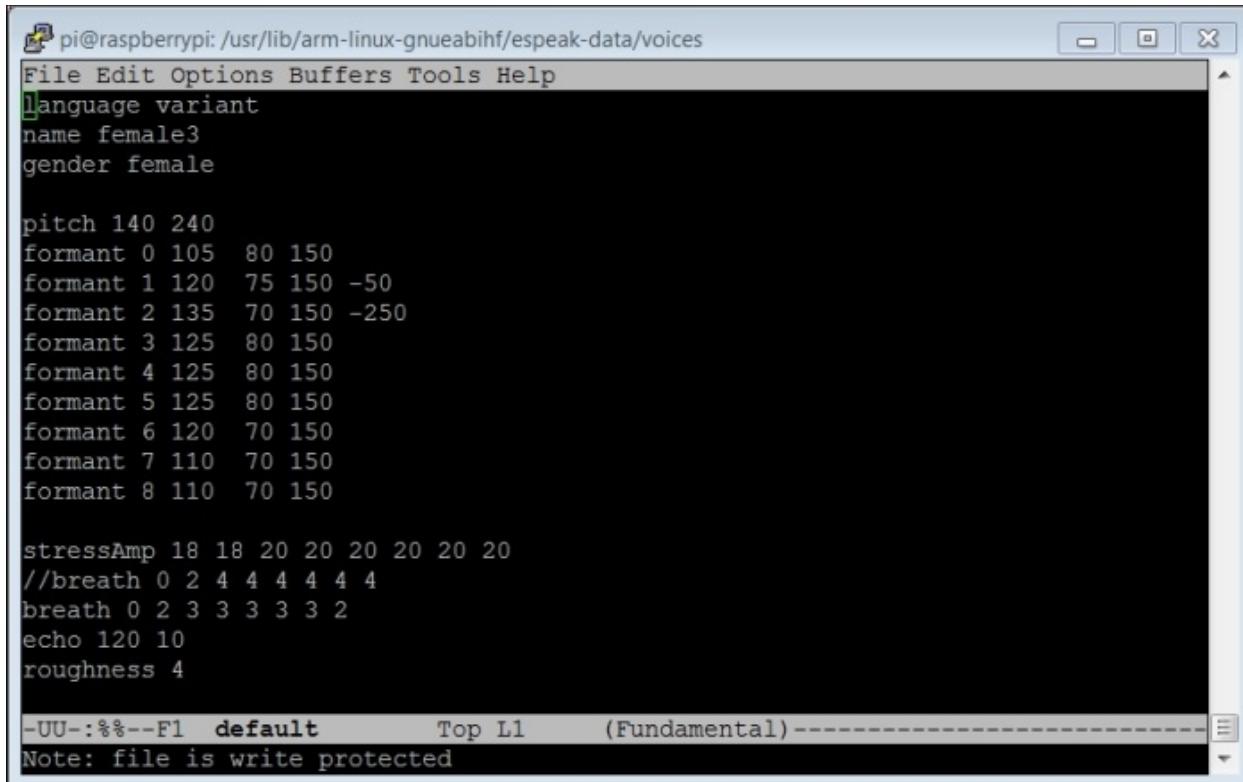
Now that we have a computer voice, you may want to customize it. Espeak offers a fairly complete set of customization features, including a large number of languages, voices, and other options. To access these, you can type in the options on the command-line prompt. For example, type in `espeak -v+f3 "hello"` and you should hear a female voice. You can add a Scottish accent by typing `espeak -ven-sc+f3 "hello"`. Once you have selected the kind of voice you'd like for your projects, you can make it the default setting so that you don't always have to include it in the command line.

To create the default settings, go to the default file definition for `espeak`, which is in the `/usr/lib/arm-linux-gnueabihf/espeak-data/voices` directory. You should see something like as follows:



```
pi@raspberrypi: /usr/lib/arm-linux-gnueabihf/espeak-data/voices
pi@raspberrypi /usr/lib/arm-linux-gnueabihf/espeak-data/voices $ ls
af  cs  default  es   fr    hu      is   ku   mk   pl    ru   sv   tr   zh-yue
bg  cy  el       es-la fr-be  hy      it   la   ml   pt    sk   sw   !v
bs  da  en       et    hi    hy-west ka   lv   nl   pt-pt sq   ta   vi
ca  de  eo       fi    hr    id      kn   mb   no   ro   sr   test  zh
pi@raspberrypi /usr/lib/arm-linux-gnueabihf/espeak-data/voices $
```

The default file is the one that Espeak uses to choose a voice. To get your desired voice, say one with a female tone, you need to copy a file into the default file. The file, that is, the female tone, is in the `!v` directory. Type `\!v` whenever you want to specify this directory. We need to type the `\` character because the `!` character is a special character in Linux, and if we want to use it as a regular old character, we need to put a `\` character before it. Before starting the process, copy the current default into a file named `default.old`, so it can be retrieved later if needed. The next step is to copy the `f3` voice as your default file. Type the `sudo cp ./\!v/f3 default` command. If you open this file, it will look as follows:



```
pi@raspberrypi: /usr/lib/arm-linux-gnueabihf/espeak-data/voices
File Edit Options Buffers Tools Help
language variant
name female3
gender female

pitch 140 240
formant 0 105 80 150
formant 1 120 75 150 -50
formant 2 135 70 150 -250
formant 3 125 80 150
formant 4 125 80 150
formant 5 125 80 150
formant 6 120 70 150
formant 7 110 70 150
formant 8 110 70 150

stressAmp 18 18 20 20 20 20 20 20
//breath 0 2 4 4 4 4 4 4
breath 0 2 3 3 3 3 3 2
echo 120 10
roughness 4

-UU-:%%--F1 default      Top L1      (Fundamental)-----
Note: file is write protected
```

This has all the settings for your female voice. Now you can simply type [espeak](#) and the desired text. You will now get your female computer voice.

Now your project can speak. Simply type [espeak](#) followed by the text you want to speak in quotes and out comes your speech. If you want to read an entire text file, you can do that as well using the [-f](#) option and then typing the name of the file. Try this by using your editor to create a text file called [speak](#); then type the [espeak -f speak.txt](#) command.

There are lots of choices with respect to the voices you might use with [espeak](#). Feel free to play around and choose your favorite. Then, edit the default file to set it to that voice. However, don't expect that you'll get the kind of voices that you hear from computers in the movies; those are actors and not computers. Although one day we will hopefully reach a stage where computers will sound a lot more like real people.

Using PocketSphinx to accept your voice commands

Sound is cool and speech is even cooler, but you'll also want to be able to communicate with your projects through voice commands. This section will show you how to add speech recognition to your robotic projects. This isn't nearly as simple as the speaking part, but thankfully, you have some significant help from the open source development community. You are going to download a set of capabilities named PocketSphinx, which will allow our project to listen to our commands.

The first step is downloading the PocketSphinx capabilities. Unfortunately, this is not quite as user-friendly as the `espeak` process, so follow along carefully. There are two possible ways to do this. If you have a keyboard, mouse, and display connected or want to connect via `vncserver`, you can do this graphically by performing the following steps:

1. Go to the Sphinx website hosted by **Carnegie Mellon University (CMU)** at <http://cmusphinx.sourceforge.net/>. This is an open source project that provides you with the speech recognition software. With our smaller, embedded system, we will be using the PocketSphinx version of this code.
2. You will need to download two pieces of software modules: sphinxbase and PocketSphinx. Select the **Download** option at the top of the page and then find the latest version of both of these packages. Download the `.tar.gz` version of these and move them to the `homepi` directory of your Raspberry Pi.

However, before you build these, you need two libraries. The first library is `libasound2-dev`. If you skipped the first two objectives of this chapter, you'll need to download them now using `sudo apt-get install libasound2-dev`. If you're unsure whether or not it's installed, try it again. The system will let you know if it's already installed.

The second of these libraries is a library called `Bison`. This is a general-purpose, open source parser that will be used by PocketSphinx. To get this package, type `sudo apt-get install bison`.

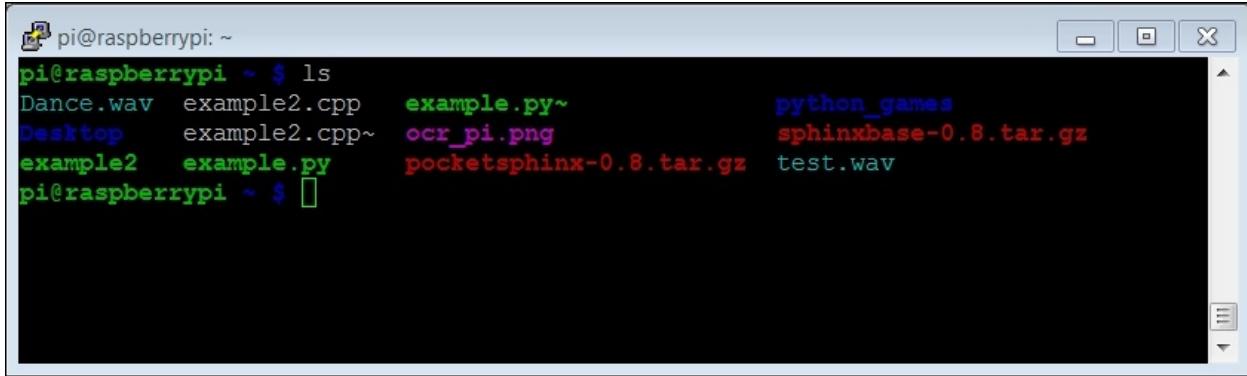
Another way to accomplish this is to use `wget` directly on the command-line prompt of Raspberry Pi. If you want to do it this way, perform the following steps:

1. To use `wget` on your host machine, find the link to the file you wish to download. In this case, go to the Sphinx website hosted by Carnegie Mellon University at <http://cmusphinx.sourceforge.net/>. This is an open source project that provides you with the speech recognition software. With your smaller, embedded system, you will be using the PocketSphinx version of this code.
2. You will need to download two pieces of software modules, namely sphinxbase and PocketSphinx. Select the **Download** option at the top of the page and then find the latest version of both of these packages. Right-click on the `sphinxbase-0.8.tar.gz` file (if 0.8 is the latest version) and select **Copy Link Location**. Now open a PuTTY window in Raspberry Pi, and after logging in, type `wget` and paste the link you just copied. This will download the `.tar.gz` version of `sphinxbase`. Now follow the same procedure with the latest version of PocketSphinx.

Before you build these, you need two libraries. The first library is `libasound2-dev`. If you skipped the first two objectives of this chapter, you'll need to download it now using `sudo apt-get install libasound2-dev`. If you're unsure whether or not it's installed, try it again. The system will let you know if it's already installed.

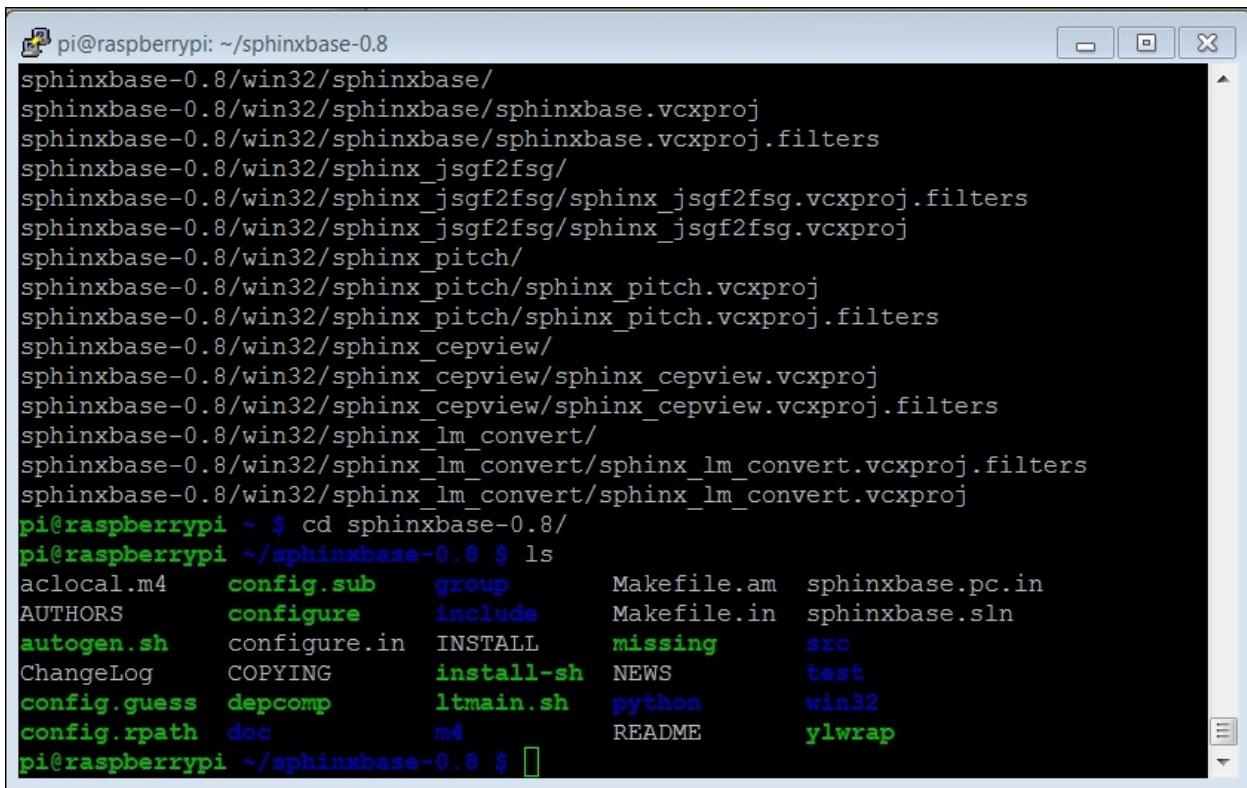
The second of these libraries is called `Bison`. This is a general purpose, open source parser that will be used by PocketSphinx. To get this package, type `sudo apt-get install bison`.

Once everything is installed and downloaded, you can build PocketSphinx. Firstly, your home directory, with the `.tar.gz` files of both PocketSphinx and sphinxbase, should look as follows:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ls
Dance.wav  example2.cpp  example.py~          python_games
Desktop   example2.cpp~  ocr_pi.png          sphinxbase-0.8.tar.gz
example2  example.py    pocketsphinx-0.8.tar.gz  test.wav
pi@raspberrypi ~ $
```

To unpack and build the `sphinxbase` module, type `sudo tar -xzvf sphinxbase-0.y.tar.gz`, where `y` is the version number; in our example, it is `8`. This should unpack all the files from the archive into a directory named `sphinxbase-0.8`. Now type `cd sphinxbase-0.8`. Listing the files should show something like the following screenshot:



```
pi@raspberrypi: ~/sphinxbase-0.8
sphinxbase-0.8/win32/sphinxbase/
sphinxbase-0.8/win32/sphinxbase/sphinxbase.vcxproj
sphinxbase-0.8/win32/sphinxbase/sphinxbase.vcxproj.filters
sphinxbase-0.8/win32/sphinx_jsgf2fsg/
sphinxbase-0.8/win32/sphinx_jsgf2fsg/sphinx_jsgf2fsg.vcxproj.filters
sphinxbase-0.8/win32/sphinx_jsgf2fsg/sphinx_jsgf2fsg.vcxproj
sphinxbase-0.8/win32/sphinx_pitch/
sphinxbase-0.8/win32/sphinx_pitch/sphinx_pitch.vcxproj
sphinxbase-0.8/win32/sphinx_pitch/sphinx_pitch.vcxproj.filters
sphinxbase-0.8/win32/sphinx_cepview/
sphinxbase-0.8/win32/sphinx_cepview/sphinx_cepview.vcxproj
sphinxbase-0.8/win32/sphinx_cepview/sphinx_cepview.vcxproj.filters
sphinxbase-0.8/win32/sphinx_lm_convert/
sphinxbase-0.8/win32/sphinx_lm_convert/sphinx_lm_convert.vcxproj.filters
sphinxbase-0.8/win32/sphinx_lm_convert/sphinx_lm_convert.vcxproj
pi@raspberrypi ~ $ cd sphinxbase-0.8/
pi@raspberrypi ~/sphinxbase-0.8 $ ls
aclocal.m4  config.sub  group      Makefile.am  sphinxbase.pc.in
AUTHORS     configure   include    Makefile.in  sphinxbase.sln
autogen.sh  configure.in  INSTALL   missing     src
ChangeLog   COPYING     install-sh NEWS      test
config.guess depcomp    ltmain.sh  python    win32
config.rpath doc       m4        README    ylwrap
pi@raspberrypi ~/sphinxbase-0.8 $
```

To build the application, start by issuing the command `sudo ./configure --enable-fixed`. This command will check that everything is ok with the system and then configure a build.

Now you are ready to actually build the `sphinxbase` code base. This is a

two-step process, which is as follows:

1. Type `make` and the system will build all the executable files.
2. Type `sudo make install` and this will install all the executables onto the system.

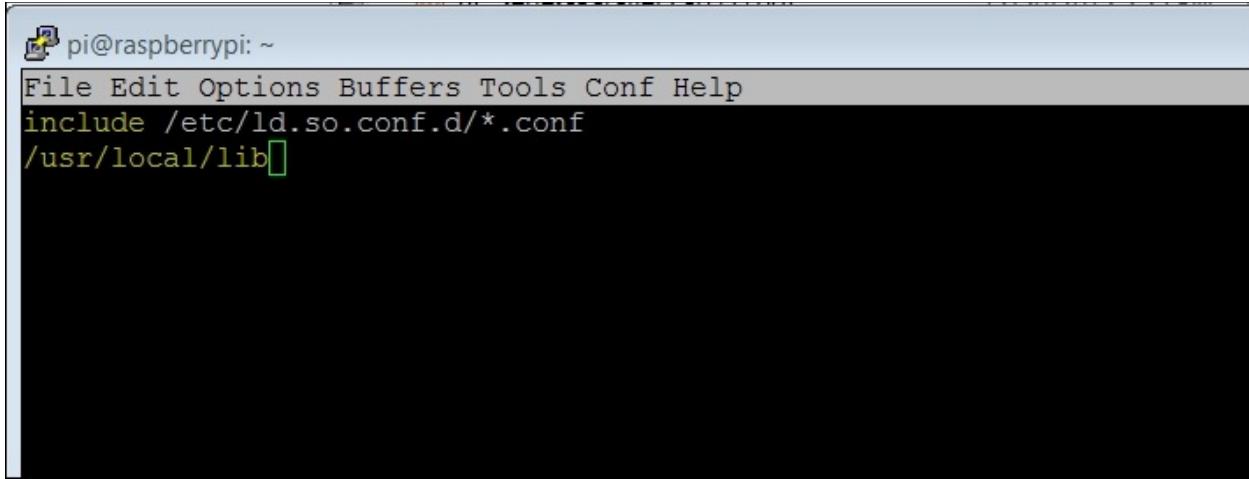
Now we need to make the second part of the system: the `PocketSphinx` code itself. Go to the home directory and decompress and unarchive the code by typing `tar -xzvf pocketsphinx-0.8.tar.gz`. The files should now be unarchived, and we can now build the code. Installing these files is a three-step process as follows:

1. Type `cd` in the `PocketSphinx` directory, and then type `./configure` to see if we are ready to build the files.
2. Type `make` and wait for a while for everything to build.
3. Type `sudo make install`.

Note

Several possible additions to our library installations will be useful later if you are going to use your `PocketSphinx` capability with Python as a coding language. You can install Python-Dev using `sudo apt-get install python-dev` and Cython using `sudo apt-get install cython`. You can also choose to install `pkg-config`, a utility that can sometimes help deal with complex compiles. Install it using `sudo apt-get install pkg-config`.

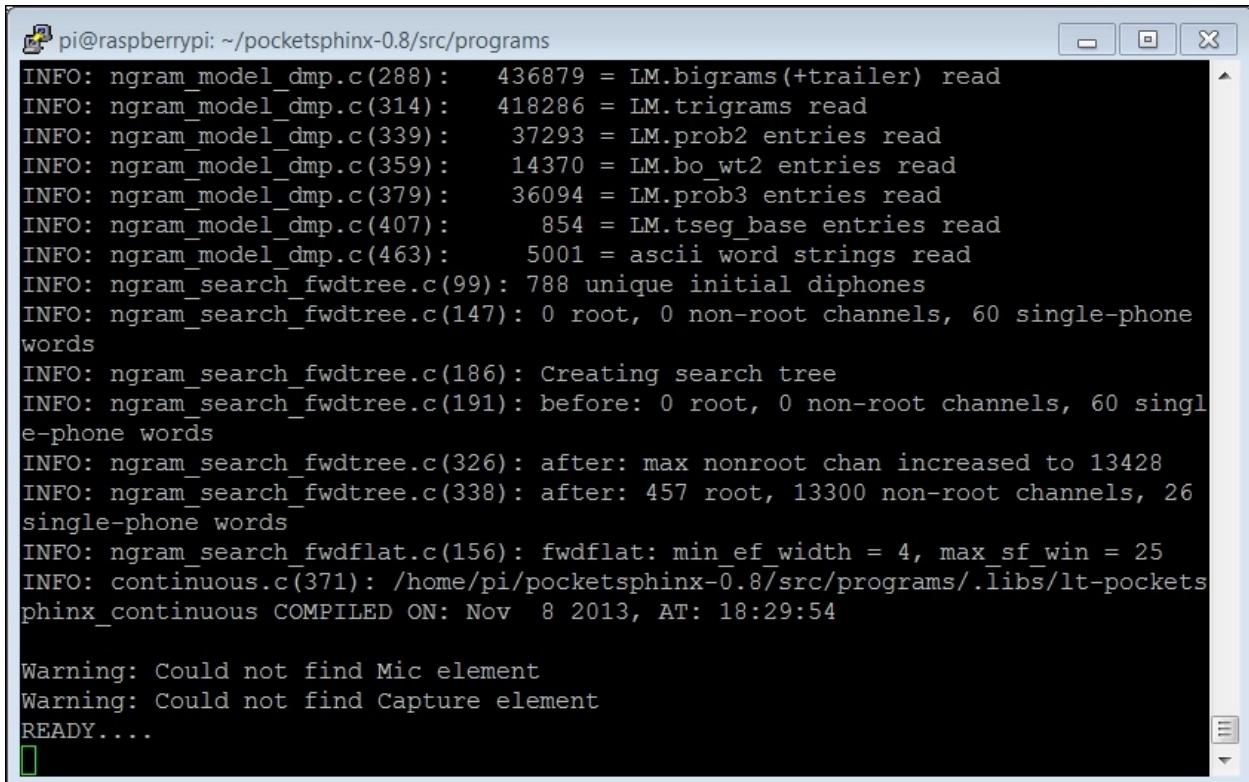
Once the installation is complete, you'll need to let the system know where our files are. To do this, you will need to edit the `etclib.so.conf` path as the root by typing `sudo emacs etclib.so.conf`. You will add the last line to the file, so it should now look like the following screenshot:



```
pi@raspberrypi: ~
File Edit Options Buffers Tools Conf Help
include /etc/ld.so.conf.d/*.conf
/usr/local/lib[]
```

Now type `sudo sbinldconfig`, and the system will now be aware of your `PocketSphinx` libraries.

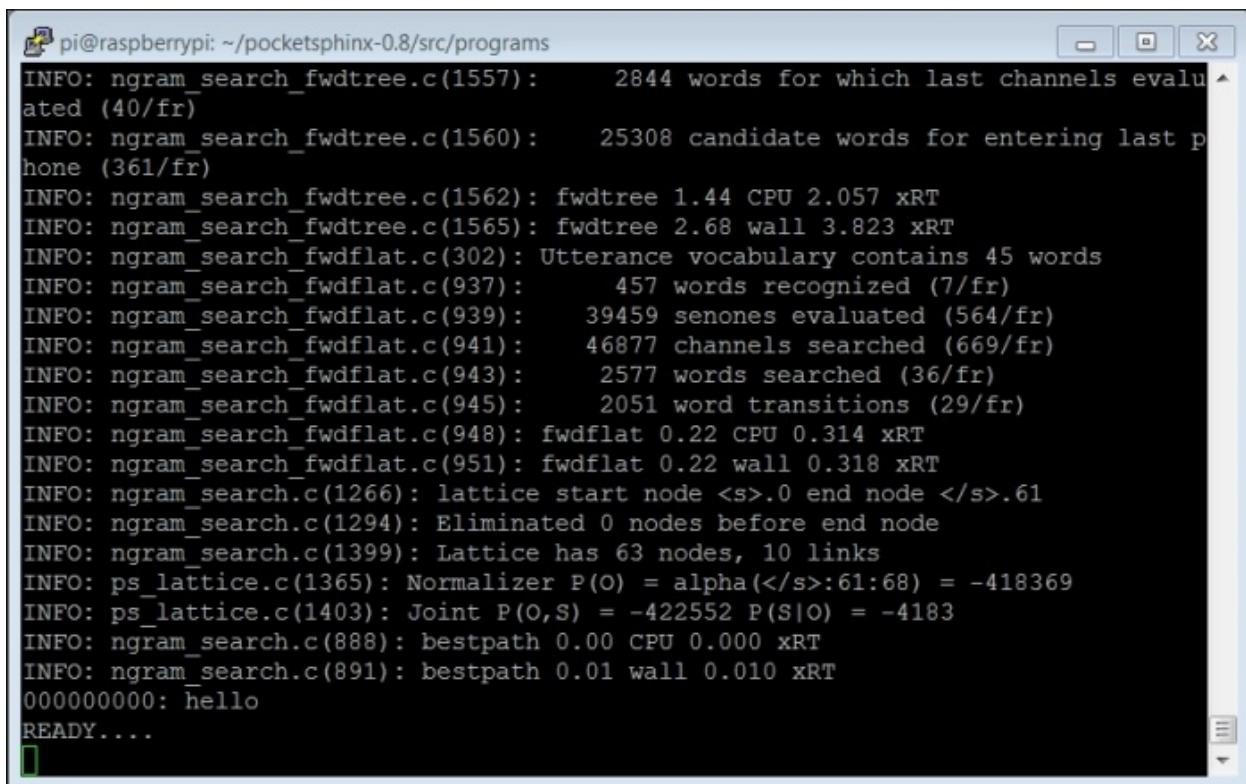
Everything is installed, so you can now try our speech recognition. Type `cd` in the `homepi/pocketsphinx-0.8/src/programs` directory to try a demo program; then type `pocketsphinx_continuous`. This program takes input from the microphone and turns it into speech. After running the command, you'll get a lot of irrelevant information, and then you will see the following screenshot:



```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
INFO: ngram_model_dmp.c(288): 436879 = LM.bigrams(+trailer) read
INFO: ngram_model_dmp.c(314): 418286 = LM.trigrams read
INFO: ngram_model_dmp.c(339): 37293 = LM.prob2 entries read
INFO: ngram_model_dmp.c(359): 14370 = LM.bo_wt2 entries read
INFO: ngram_model_dmp.c(379): 36094 = LM.prob3 entries read
INFO: ngram_model_dmp.c(407): 854 = LM.tseg_base entries read
INFO: ngram_model_dmp.c(463): 5001 = ascii word strings read
INFO: ngram_search_fwdtree.c(99): 788 unique initial diphones
INFO: ngram_search_fwdtree.c(147): 0 root, 0 non-root channels, 60 single-phone words
INFO: ngram_search_fwdtree.c(186): Creating search tree
INFO: ngram_search_fwdtree.c(191): before: 0 root, 0 non-root channels, 60 single-phone words
INFO: ngram_search_fwdtree.c(326): after: max nonroot chan increased to 13428
INFO: ngram_search_fwdtree.c(338): after: 457 root, 13300 non-root channels, 26 single-phone words
INFO: ngram_search_fwdflat.c(156): fwdflat: min_ef_width = 4, max_sf_win = 25
INFO: continuous.c(371): /home/pi/pocketsphinx-0.8/src/programs/.libs/lt-pocketsphinx_continuous COMPILED ON: Nov 8 2013, AT: 18:29:54

Warning: Could not find Mic element
Warning: Could not find Capture element
READY....
```

The [INFO](#) and [Warning](#) statements come from the C or C++ code and are there for debugging purposes. Initially, they will warn you that they cannot find your [Mic](#) and [Capture](#) elements, but when they find them, they will print out [READY....](#). If you have set things up as previously described, you should be ready to give them a command. Say "hello" into the microphone. When they sense that you have stopped speaking, they will process your speech and give lots of irrelevant information again, but they should eventually show the commands in the following screenshot:



A terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window displays a log of the PocketSphinx search process. It shows various informational messages (INFO) such as word counts, CPU usage, and search parameters. Near the bottom, it receives a speech input represented by a sequence of zeros: "000000000: hello". This triggers a "bestpath" search, which is completed almost immediately with a wall time of 0.010 seconds. The process ends with the message "READY....".

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
INFO: ngram_search_fwdtree.c(1557):      2844 words for which last channels evaluated (40/fr)
INFO: ngram_search_fwdtree.c(1560):      25308 candidate words for entering last phone (361/fr)
INFO: ngram_search_fwdtree.c(1562): fwdtree 1.44 CPU 2.057 xRT
INFO: ngram_search_fwdtree.c(1565): fwdtree 2.68 wall 3.823 xRT
INFO: ngram_search_fwdflat.c(302): Utterance vocabulary contains 45 words
INFO: ngram_search_fwdflat.c(937):      457 words recognized (7/fr)
INFO: ngram_search_fwdflat.c(939):      39459 senones evaluated (564/fr)
INFO: ngram_search_fwdflat.c(941):      46877 channels searched (669/fr)
INFO: ngram_search_fwdflat.c(943):      2577 words searched (36/fr)
INFO: ngram_search_fwdflat.c(945):      2051 word transitions (29/fr)
INFO: ngram_search_fwdflat.c(948): fwdflat 0.22 CPU 0.314 xRT
INFO: ngram_search_fwdflat.c(951): fwdflat 0.22 wall 0.318 xRT
INFO: ngram_search.c(1266): lattice start node <s>.0 end node </s>.61
INFO: ngram_search.c(1294): Eliminated 0 nodes before end node
INFO: ngram_search.c(1399): Lattice has 63 nodes, 10 links
INFO: ps_lattice.c(1365): Normalizer P(O) = alpha(</s>:61:68) = -418369
INFO: ps_lattice.c(1403): Joint P(O,S) = -422552 P(S|O) = -4183
INFO: ngram_search.c(888): bestpath 0.00 CPU 0.000 xRT
INFO: ngram_search.c(891): bestpath 0.01 wall 0.010 xRT
000000000: hello
READY....
```

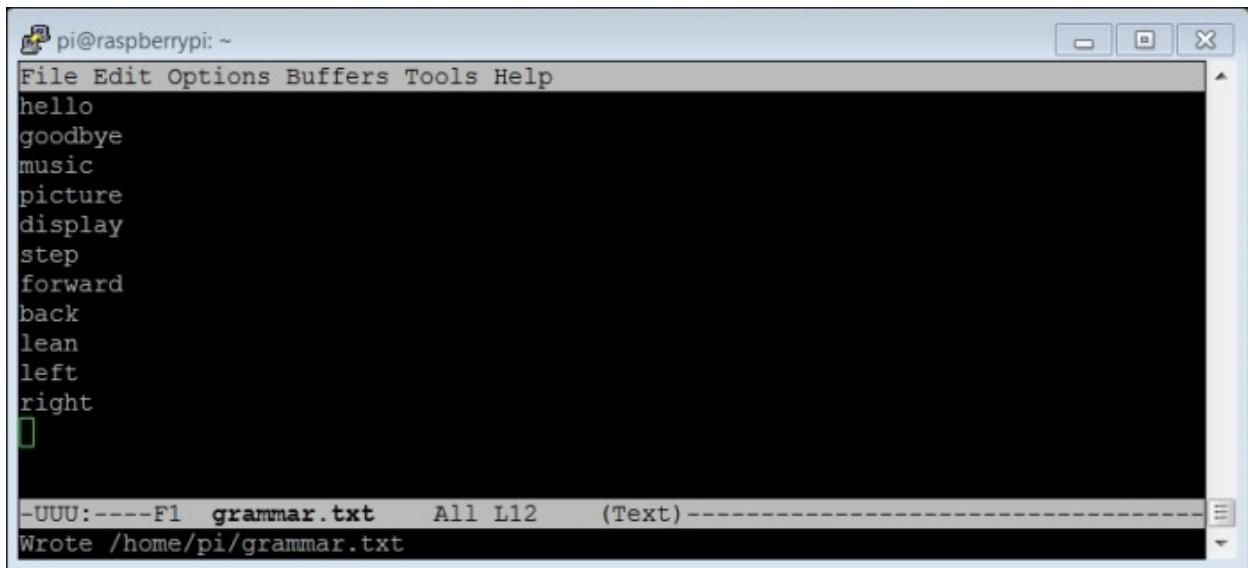
Notice the [000000000: hello](#) command. It recognized your speech! You can try other words and phrases too. The system is very sensitive, so it may pick up background noise. You are also going to find that it is not very accurate. We'll deal with that in a moment. To stop the program, type [ctrl-c](#).

There are two ways to make your voice recognition more accurate. One is to train the system to more accurately understand your voice. This is a bit complex and if you want to know more, go to the [PocketSphinx](#) website of CMU.

The second way to improve accuracy is to limit the number of words that

your system uses to determine what you are saying. The default has literally thousands of word possibilities, so if two words are close, it may choose the wrong word. To avoid this, you can make your own grammar rules to restrict the words it has to choose from.

The first step is to create a file with the words or phrases that you want the system to recognize. Then, you use a web tool to create two files that the system will use to define our grammar. I'll do this through the [vncserver](#) command because I'll need to use a web browser on Raspberry Pi to turn a text file into a set of grammar files. Begin by editing a file; type [emacs grammar.txt](#) and insert the text as shown in the following screenshot:

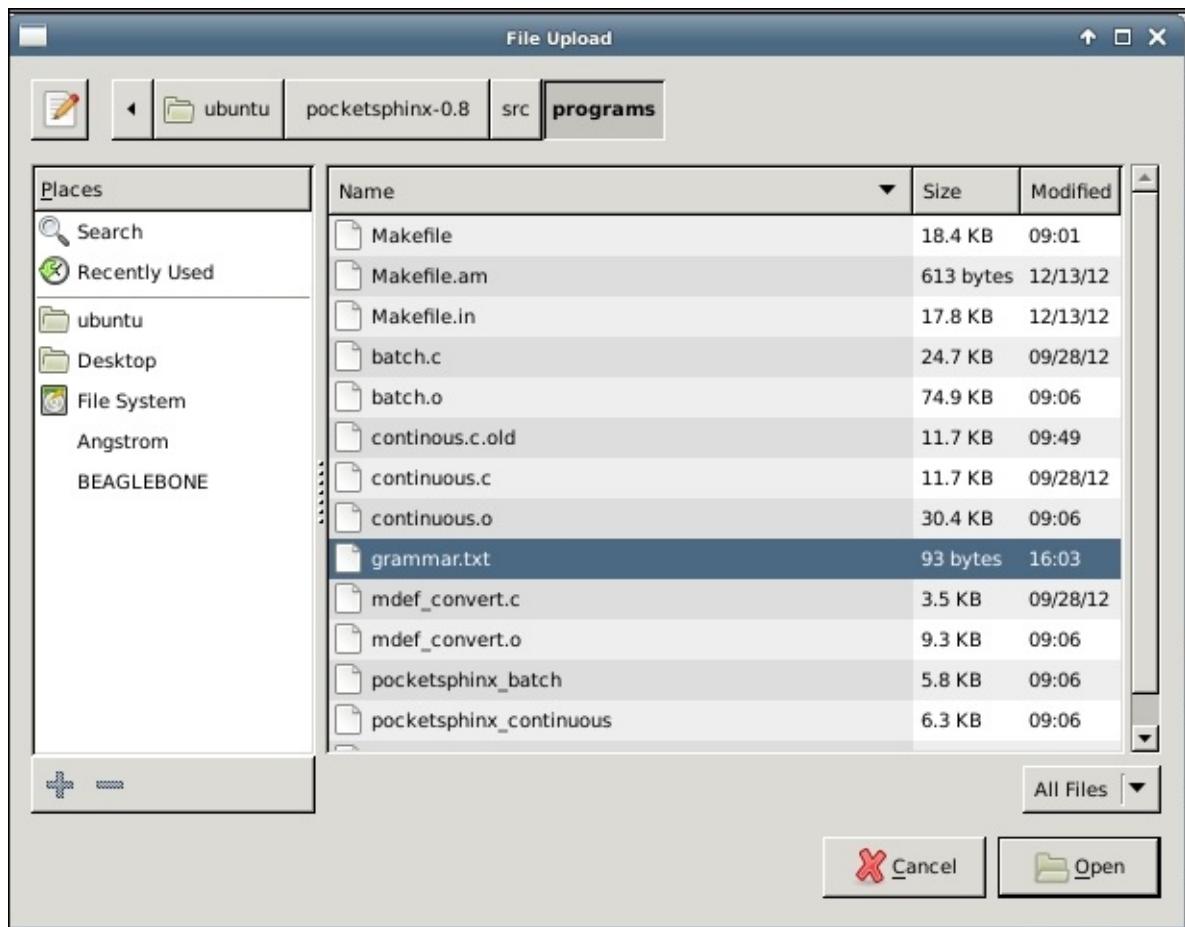


A screenshot of an Emacs text editor window. The title bar says "pi@raspberrypi: ~". The menu bar includes "File", "Edit", "Options", "Buffers", "Tools", and "Help". The main buffer contains the following text:

```
hello
goodbye
music
picture
display
step
forward
back
lean
left
right
```

The status bar at the bottom shows "-UUU:----F1 grammar.txt All L12 (Text)---" and "Wrote /home/pi/grammar.txt".

Now you must use the CMU web tool to turn this file into two files that the system can use to define its dictionary. On my system, I have already installed Firefox using [sudo apt-get install firefox](#). So, now I can open a web browser window and go to <http://www.speech.cs.cmu.edu/tools/lmtool-new.html>. If you hit the **Browse** button, you can find and select the file. It should look something like the following screenshot:



Open the [grammar.txt](#) file; then, on the web page, select **COMPILE KNOWLEDGE BASE**, and a window should pop up, as shown in the following screenshot:

The screenshot shows a Mozilla Firefox browser window. The address bar displays the URL: www.speech.cs.cmu.edu/tools/product/1374186047_27854/. The main content area features a large title: **Sphinx knowledge base generator [lmtool.3]**. Below the title, a message states: "Your request for a Sphinx knowledge base appears to have been successfully processed!" It also mentions the base name for the set is **1565**, and provides a link to **TAR1565.tgz** which is described as the compressed version. A note says the files are internally consistent and best used together. An **IMPORTANT:** note advises downloading the files as soon as possible, as they will be deleted in approximately a half hour. A code block shows session logs:

```
SESSION 1374186047_27854
[_INFO_] Found corpus: 11 sentences, 15 unique words
[_INFO_] Found 0 words in extras (0)
[_INFO_] Language model completed (0)
[_INFO_] Pronounce completed (0)
[_STAT_] Elapsed time: 0.021 sec
```

Please include these messages in bug reports.

Name	Size	Description
1565.dic	207	Pronunciation Dictionary
1565.lm	1.8K	Language Model
1565.log_pronounce	159	Log File
1565.sent	192	Corpus (processed)

You need to download the **.tgz** file created; in this case, the **TAR1565.tgz** file. This will download into your **homepi/** directory. Move it to the **homepi/pocketsphinx-0.8/src/programs** directory and unarchive it using **tar -xzvf** and the filename.

Now you can invoke the **pocketsphinx_continuous** program to use this dictionary by typing **./pocketsphinx_continuous -lm 1565.lm -dict 1565.dic**, and it will look in that directory to find matches to your commands.

You can also do this on your remote computer using Windows or Linux by creating the file in a text editor such as WordPad or Emacs. Once you have created the required grammar files, you can download them to your

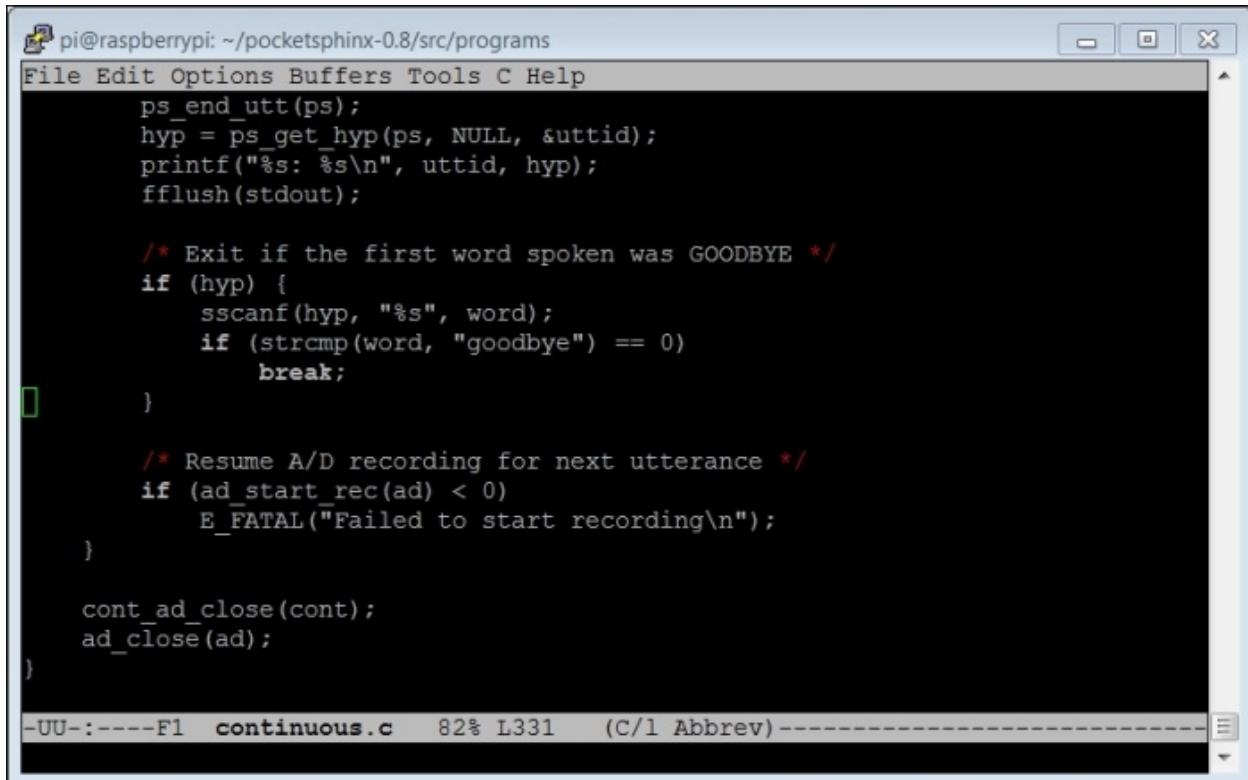
Raspberry Pi using WinSCP, if you are using Windows or `scp` from the command line, if you are using Linux.

Your system can now understand your specific set of commands! In the next section of this chapter, you'll learn how to use this input to have the project respond.

Interpreting commands and initiating actions

Now that the system can both hear and speak, you'll want to provide the capability to respond to your speech and execute some commands based on the speech input. Now, you're going to configure the system to respond to simple commands.

In order to respond, we're going to edit the `continuous.c` code in the `homepi/src/programs` directory. We could create our own C file, but this file is already set up in the `makefile` system and is an excellent starting spot. You can save a copy of the current file in `continuous.c.old` so that you can always get back to the starting program if required. Then, you will need to edit the `continuous.c` file. It is very long and a bit complicated, but you are specifically looking for the section in the code, which is shown in the following screenshot. Look for the comment line `/* Exit if the first word spoken was GOODBYE */`.



The screenshot shows a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains the source code for the `continuous.c` file. The code includes several sections of C code, with one specific section highlighted:

```
    ps_end_utt(ps);
    hyp = ps_get_hyp(ps, NULL, &uttid);
    printf("%s: %s\n", uttid, hyp);
    fflush(stdout);

    /* Exit if the first word spoken was GOODBYE */
    if (hyp) {
        sscanf(hyp, "%s", word);
        if (strcmp(word, "goodbye") == 0)
            break;
    }

    /* Resume A/D recording for next utterance */
    if (ad_start_rec(ad) < 0)
        E_FATAL("Failed to start recording\n");
}

cont_ad_close(cont);
ad_close(ad);
}
```

The status bar at the bottom of the terminal window indicates the file name is `continuous.c`, the line number is 82%, and the column number is L331. The status bar also shows "(C/l Abbrev)".

In this section of the code, the word has already been decoded and is held in the `hyp` variable. You can add code here to make your system do things based on the value associated with the word we have decoded. First, let's try adding the capability to respond to "hello" and "goodbye" to see if we can get the program to stop. Make changes to the code in the following manner:

1. Find the comment `/* Exit if the first word spoken was GOODBYE */.`
2. In the statement `if (strcmp(hyp, "good bye") == 0)`, change `word` to `hyp` and `good bye` to `GOODBYE`.
3. Insert brackets around the `break;` statement and add the `system ("espeak" \\"good bye\\");` statement just before the `break;` statement.
4. Add the other `else if` statement to the clause by typing `else if (strcmp(hyp, "HELLO") == 0)`. Add brackets after the `else if` statement, and inside the brackets, type `system ("espeak" \\"good bye\\");`.

The file should now look as follows:

A screenshot of a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains C code for a program named "continuous.c". The code includes logic to handle commands like "GOODBYE" and "HELLO" by calling the system command "espeak". The terminal status bar at the bottom shows the file name "continuous.c", a 80% completion status, and "L329" lines.

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
File Edit Options Buffers Tools C Help
    ps_end_utt(ps);
    hyp = ps_get_hyp(ps, NULL, &uttid);
    printf("%s: %s\n", uttid, hyp);
    fflush(stdout);

    /* Exit if the first word spoken was GOODBYE */
    if (hyp) {
        sscanf(hyp, "%s", word);
        if (strcmp(hyp, "GOODBYE") == 0)
        {
            system("espeak \"good bye\"");
            break;
        }
        else if (strcmp(hyp, "HELLO") == 0)
        {
            system("espeak \"hello\"");
        }
    }

    /* Resume A/D recording for next utterance */
if (ad_start_rec(ad) < 0)
-UU-:***-F1  continuous.c  80% L329  (C/l Abbrev)-----
```

Now you need to rebuild your code. As the `make` system already knows how to build the program `pocketsphinx_continuous`, it will rebuild the application if you make a change to the `continuous.c` file any point of time. Simply type `make`, and the file will compile and create a new version of `pocketsphinx_continuous`. To run your new version, type `./pocketsphinx_continuous -lm 1565.lm -dict 1565.dic`. Make sure you type the `./` command at the start.

If you haven't created your own dictionary, or would like to use the default dictionary, you can still have your robot respond. You'll just need to change the `GOODBYE` and `HELLO` in the `if (strcmp(hyp, "GOODBYE") == 0)` and `else if (strcmp(hyp, "HELLO") == 0)` statements to words that your system is currently recognizing. Simply say your command, see what the system is printing for the word it is recognizing, and replace "`GOODBYE`" or "`HELLO`" with that word.

If everything is set correctly, saying "hello" should result in a response of "hello" from your Raspberry Pi. Saying "good bye" should elicit a response of "good bye" and also shut down the program. Notice that the system command can be used to run any program that runs with a

command line. Now you can use this program to start and run other programs based on the commands.

Your Raspberry Pi will both listen and respond and execute the commands that you give it. By the way, you may be tempted to use Python to do this, but the reason that we did not use Python is that to get Python to recognize real-time speech for [PocketSphinx](#), you would need a way to stream the data to the application. There is an example of this in the [pocketSphinx-0.8/src/gst-plug-in](#) directory titled [livedemo.py](#). If you would like to try this, keep in mind that you will need to install the [gtk](#) and [gstreamer](#) tools onto Raspberry Pi, which requires many different packages and a significant amount of disk space as it does not come with the default release.

Summary

Now your project can both hear and speak. You can use these functions later when you want to interact with your project without typing commands or using a display. You should also feel more comfortable installing new hardware and software into your system. We'll be using this skill throughout the book as we look at more complex projects.

In the next chapter, we'll look at adding a capability that will allow your robots to see and use vision to track objects, motion, or whatever else your robot needs to track.

Chapter 4. Adding Vision to Raspberry Pi

In the previous chapter, you learned how to communicate with Raspberry Pi via voice. In this chapter, we are going to add vision with a webcam; you'll use this functionality in lots of different applications. Fortunately, adding hardware and software for vision is both easy and inexpensive.

To do this, you'll have to add a USB webcam to your system. Having the standard USB interface on your board opens a wide range of amazing possibilities. Furthermore, there are several amazing open source libraries that offer complex capabilities, which we can use in our projects without spending months coding them.

Vision will open a set of possibilities for your project. These can range from simple motion detection to advanced capabilities such as facial recognition, object identification, and even object tracking. The robot can also use vision to detect its surroundings and avoid obstacles.

In this chapter, we will cover the following topics:

- Connecting our USB camera to our Raspberry Pi and viewing the images
- Downloading and installing OpenCV, a full-featured vision library
- Using the vision library to detect colored objects

To add vision to our projects, we'll need a Raspberry Pi with a LAN connection and a 5V power supply. We'll also need to add a USB webcam; try to find a recently manufactured one. You may have an older webcam sitting on your project shelf, but it will probably cause problems as Linux may not have driver support for these devices, and the money you save will not be worth the frustration you might have later. You should stick with webcams from major players, such as Logitech or Creative Labs.

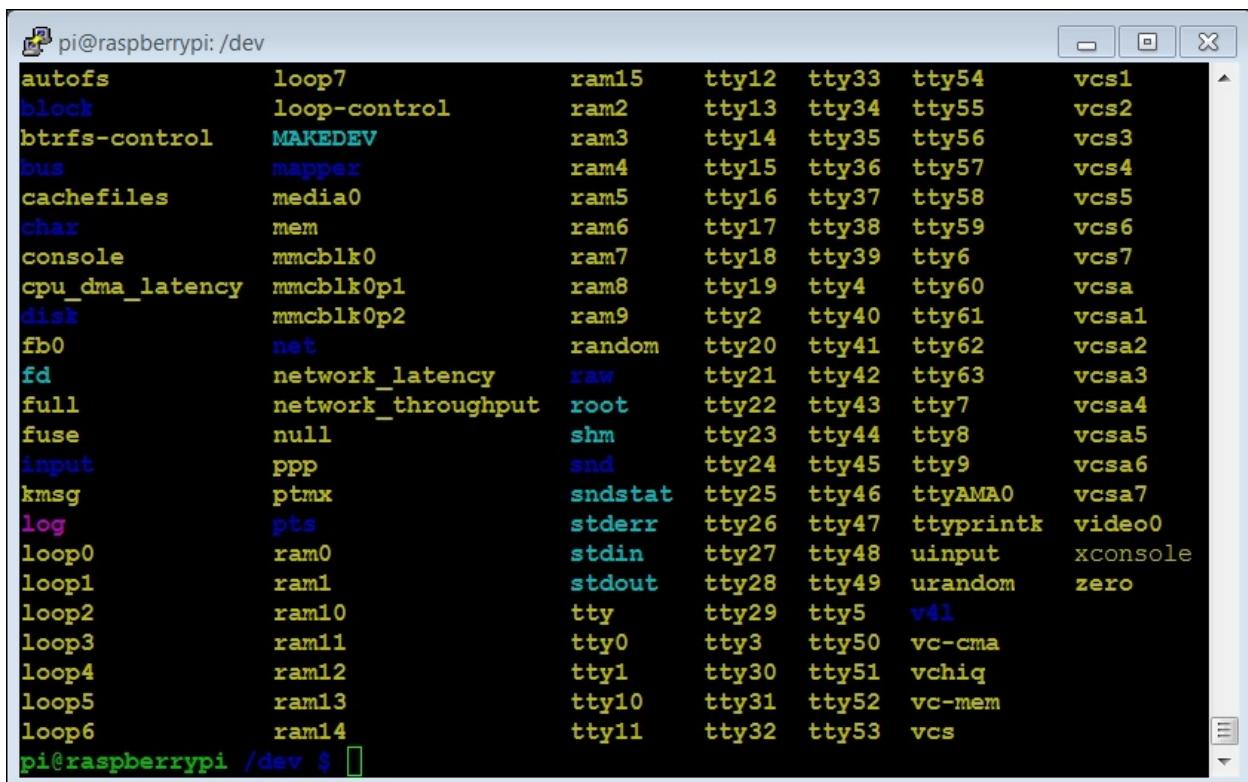
In most cases, you won't need to connect this device through a powered USB hub; however, if you encounter problems, for example, if the system

does not recognize that your webcam is connected, realize that lack of USB power could be the problem.

Connecting the USB camera to Raspberry Pi and viewing the images

The first step in enabling computer vision is connecting the USB camera to the USB port. I am using a Logitech C110 camera in my example. To access the USB webcam, I like to use a Linux program called [guvcview](#). Install this by entering `sudo apt-get install guvcview`.

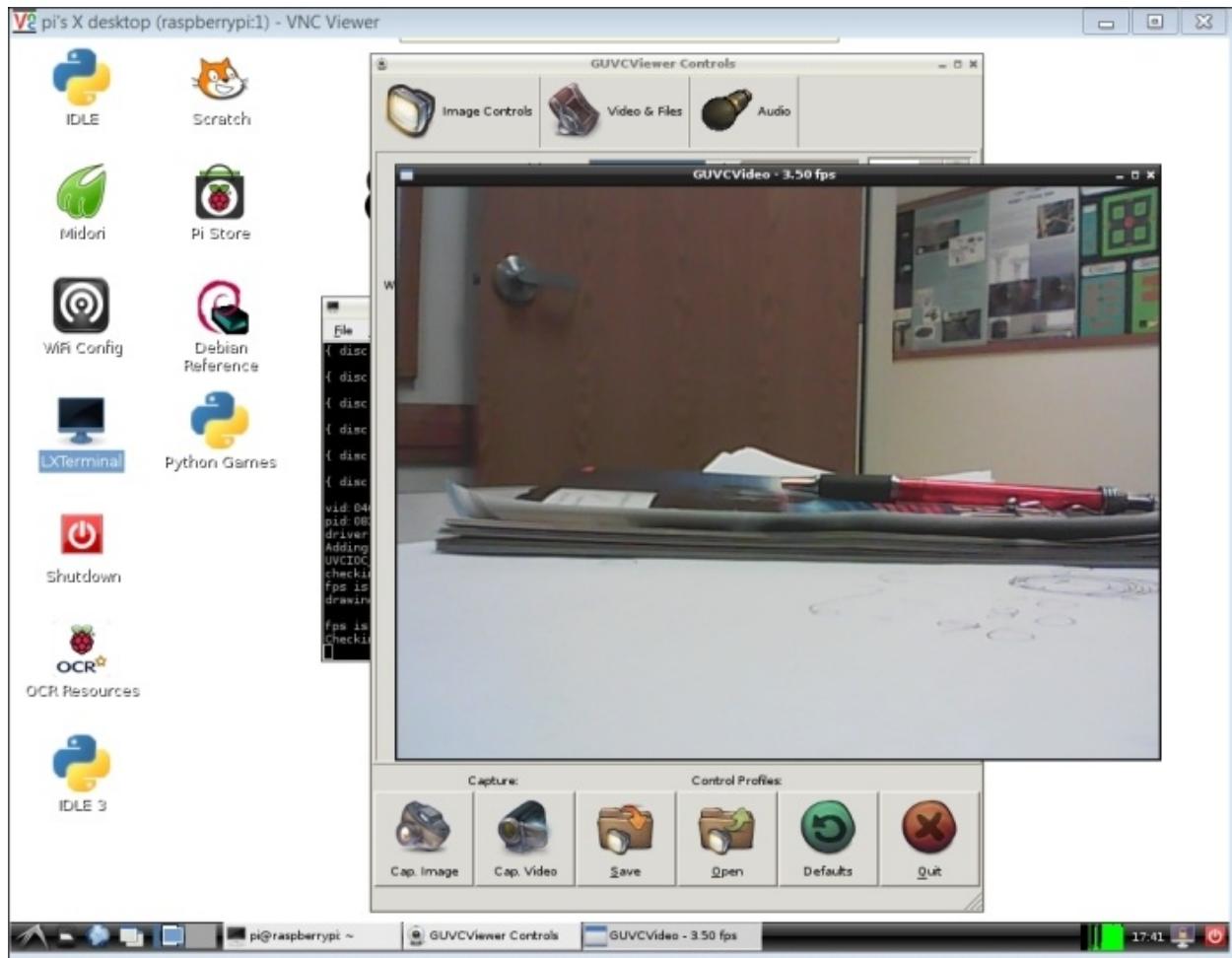
Connect your USB camera and make sure your LAN cable is plugged in. Then, apply power to Raspberry Pi. After the system is booted, you can check to see if Raspberry Pi has found your USB camera. Go to the `/dev` directory and type `ls`. You should see something as shown in the following screenshot:



```
pi@raspberrypi: /dev
loop7          ram15      tty12  tty33  tty54      vcs1
loop-control   ram2       tty13  tty34  tty55      vcs2
MAKDEV        ram3       tty14  tty35  tty56      vcs3
mapper         ram4       tty15  tty36  tty57      vcs4
media0         ram5       tty16  tty37  tty58      vcs5
mem            ram6       tty17  tty38  tty59      vcs6
mmcblk0        ram7       tty18  tty39  tty6      vcs7
mmcblk0p1     ram8       tty19  tty4   tty60      vcsa
mmcblk0p2     ram9       tty2   tty40  tty61      vcsa1
net            random    tty20  tty41  tty62      vcsa2
network_latency raw       tty21  tty42  tty63      vcsa3
network_throughput root    tty22  tty43  tty7      vcsa4
null           shm      tty23  tty44  tty8      vcsa5
ppp            snd       tty24  tty45  tty9      vcsa6
ptmx           sndstat   tty25  tty46  ttyAMA0    vcsa7
pts             stderr   tty26  tty47  ttyprintk  video0
ram0           stdin    tty27  tty48  uinput    xconsole
ram1           stdout   tty28  tty49  urandom   zero
ram10          tty      tty29  tty5   v4l
ram11          tty0     tty3   tty50  vc-cma
ram12          tty1     tty30  tty51  vchiq
ram13          tty10    tty31  tty52  vc-mem
ram14          tty11    tty32  tty53  vcs
pi@raspberrypi /dev $
```

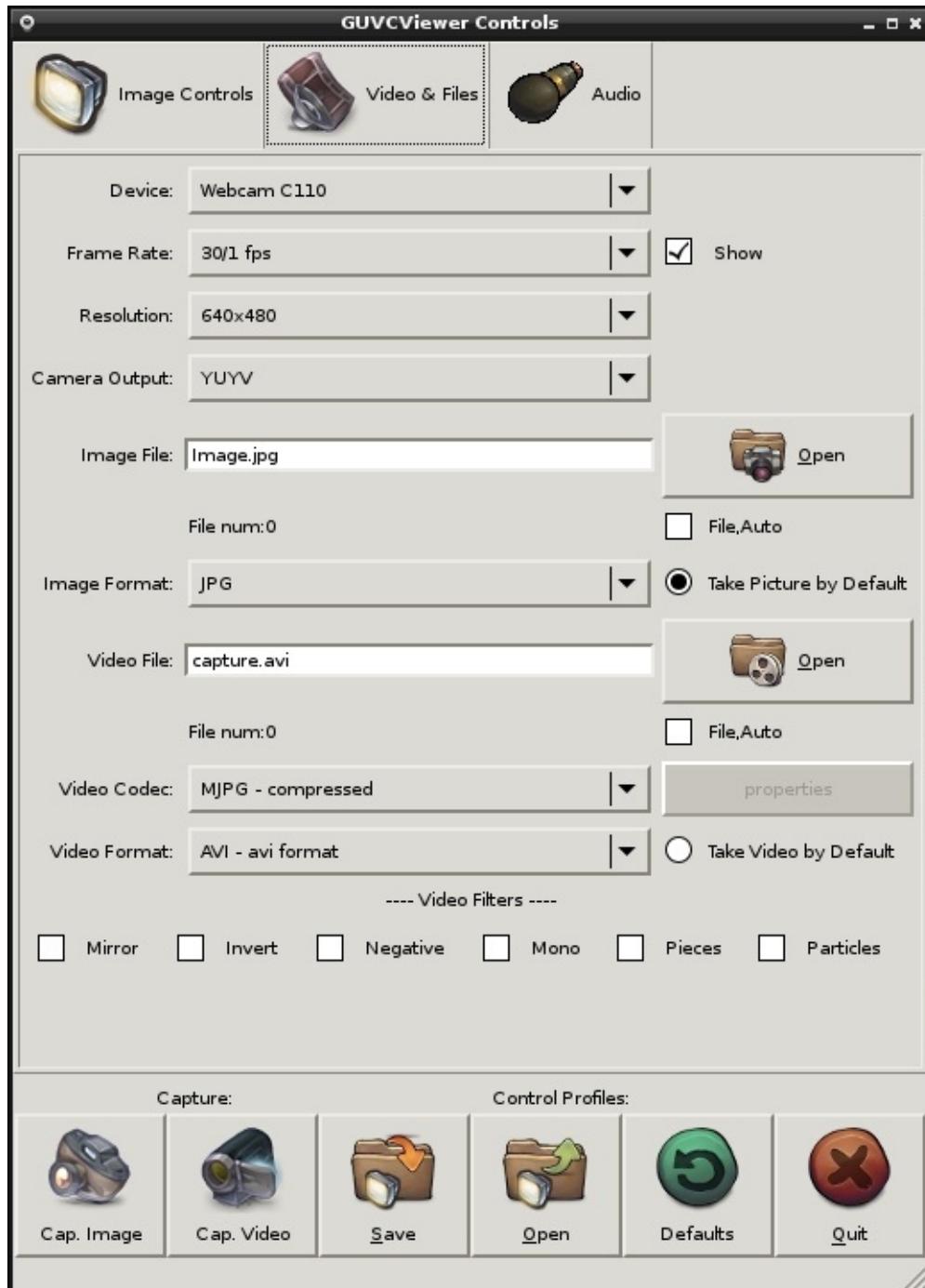
Look for `video0`, as this is the entry for your webcam. If you see it, the system knows your camera is there.

Now, let's use `guvcview` to see the output of the camera. Since it will need to output some graphics, you either need to use a monitor connected to the board, as well as a keyboard and mouse, or you can use vncserver as described in [Chapter 1, Getting Started with Raspberry Pi](#). If you are going to use vncserver, make sure you start the server on Raspberry Pi by typing `vncserver` via SSH. Then, start up **VNC Viewer** as described in [Chapter 1, Getting Started with Raspberry Pi](#). Open a terminal window and type `sudo guvcview`. You should see something as shown in the following screenshot:



The video window displays what the webcam sees, and the **GUVCViewer Controls** window controls the different characteristics of the camera. The default settings of the Logitech C110 camera work fine.

However, if you get a black screen for the camera, you may need to adjust the settings. Select the **GUVCViewer Controls** window and the **Video & Files** tab. You will see a window where you can adjust the settings for your camera, as shown in the following screenshot:



The most important setting is **Resolution**. If you see a black screen, lower the resolution; this will often resolve the issue. This window will

also tell you what resolutions are supported by your camera. Also, you can display the frame rate by checking the box to the right of the **Frame Rate** setting. Be aware, however, that if you are going through **vncviewer**, the refresh rate (how quickly the video window will update itself) will be much slower than if you're using Raspberry Pi and a monitor directly.

Once you have the camera up and running and the desired resolution set, we can go on to download and install **OpenCV**.

Note

You can connect more than one webcam to the system. Follow the same steps, but connect to cameras via a USB hub. List the devices in the `/dev` directory. Use `guvcview` to see the different images. One challenge, however, is that connecting too many cameras can overwhelm the bandwidth of the USB port.

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. The most popular of these for computer vision is OpenCV. To do this, you'll need to install OpenCV. There are several possible ways of doing this; I'm going to suggest ones that I follow to install it on my systems. Once you have booted the system and opened a terminal window, type the following commands in the following order:

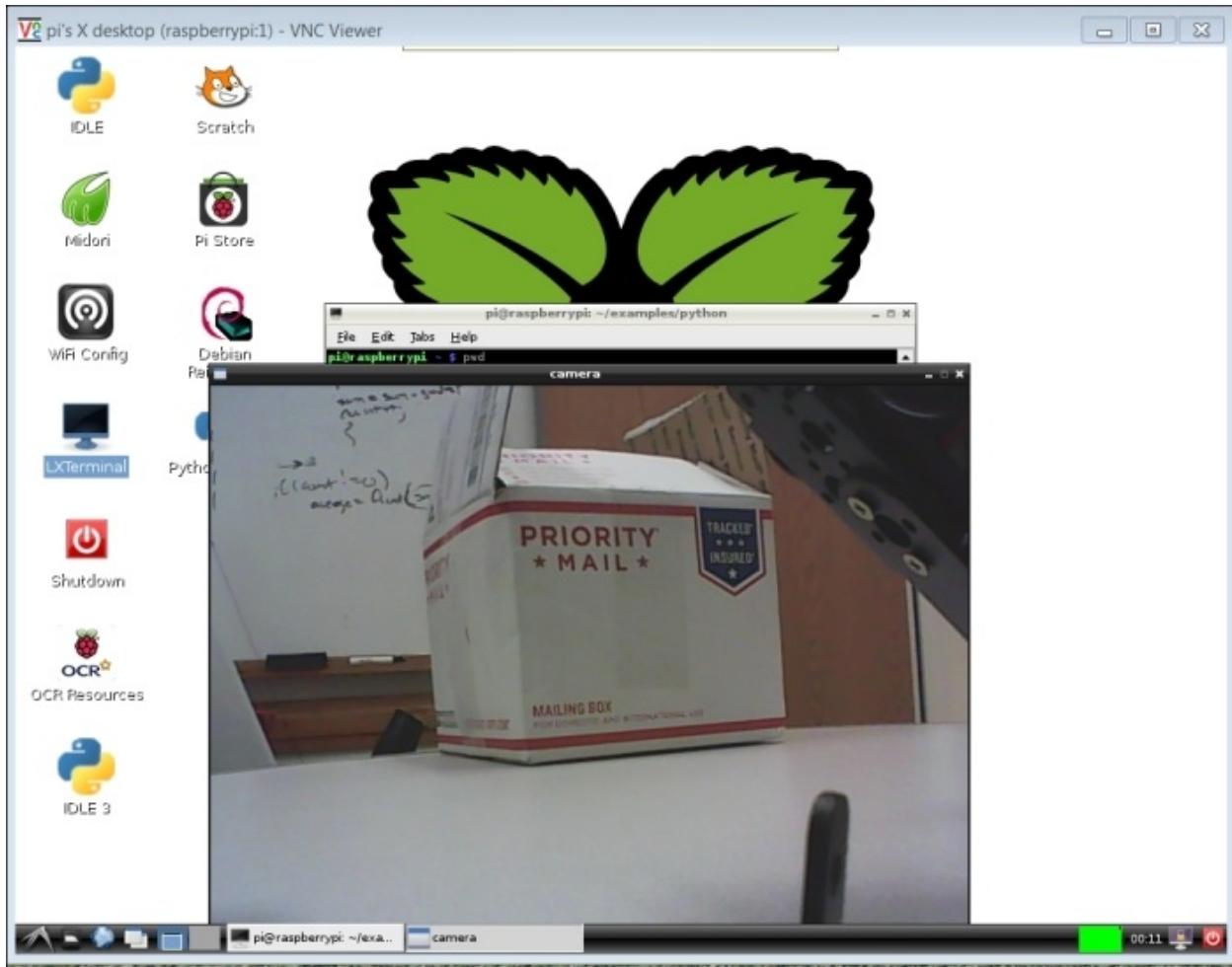
- `sudo apt-get update` – If you haven't done this in a while, it is a good idea to do this now before you start. You're going to download a number of new software packages, so it is good to make sure everything is up to date.
- `sudo apt-get install build-essential` – We have done this in a previous chapter. In case you skipped that part, you will have to refer to it now, as you need this package.
- `sudo apt-get install libavformat-dev` – This library provides a way to code and decode audio and video streams.
- `sudo apt-get install ffmpeg` – This library provides a way to transcode audio and video streams.
- `sudo apt-get install libcv2.3 libcvaux2.3 libhighgui2.3` – 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 2.3 does not work, either try 2.4 or google for the latest version of OpenCV.
- `sudo apt-get install python-opencv` – This is the Python development kit needed for OpenCV, as you are going to use Python.
- `sudo apt-get install opencv-doc` – This command will show the documentation for OpenCV just in case we need it.
- `sudo apt-get install libcv-dev` – This command shows the header file and static libraries to compile OpenCV.

- `sudo apt-get install libcvaux-dev` – This command shows more development tools for compiling OpenCV.
- `sudo apt-get install libhighgui-dev` – This is another package that provides header files and static libraries to compile OpenCV.

Make sure you are in your home directory, and then type `cp -r usrshare/doc/opencv-doc/examples`. This will copy all the examples to your home directory.

Now you are ready to try out the OpenCV library. I prefer to use Python while programming simple tasks; hence, I'll show the Python examples. If you prefer the C examples, feel free to explore. In order to use the Python examples, you'll need one more library. So type `sudo apt-get install python-numpy`, as you will need this to manipulate the matrices that OpenCV uses to hold images.

Now that you have these, you can try one of the Python examples. Switch to the directory with the Python examples by typing `cd homepi/examples/python`. In this directory, you will find a number of useful examples; we'll only look at the most basic, which is called `camera.py`. If `camera.py` is not created, you can create it by typing in the code shown in the next few pages. You can try running this example; however, to do this you'll either need to have a display connected to Raspberry Pi or you can do this over the vncserver connection. Bring up the **LXTerminal** window and type `python camera.py`. You should see something as shown in the following screenshot:



The camera window is quite large; you can change the resolution of the image to a lower one, which will make the update rate faster and the storage requirement for the image smaller. To do this, edit the [camera.py](#) file and add two lines, as shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~examples/python". The window contains Python code for a camera application using OpenCV. The code creates a window named "camera", captures frames from the default camera (index 0), sets the width to 360 pixels and height to 240 pixels, and displays the frames in the window. A loop continues until the Esc key is pressed, at which point it breaks out of the loop. The terminal window also shows the file name "camera.py", the number of lines (L3), and the Python interpreter. The status bar at the bottom indicates the file was written to "/home/pi/examples/python/camera.py".

```
pi@raspberrypi: ~examples/python
File Edit Options Buffers Tools Python Help
import cv2.cv as cv
import time
cv.NamedWindow("camera", 1)

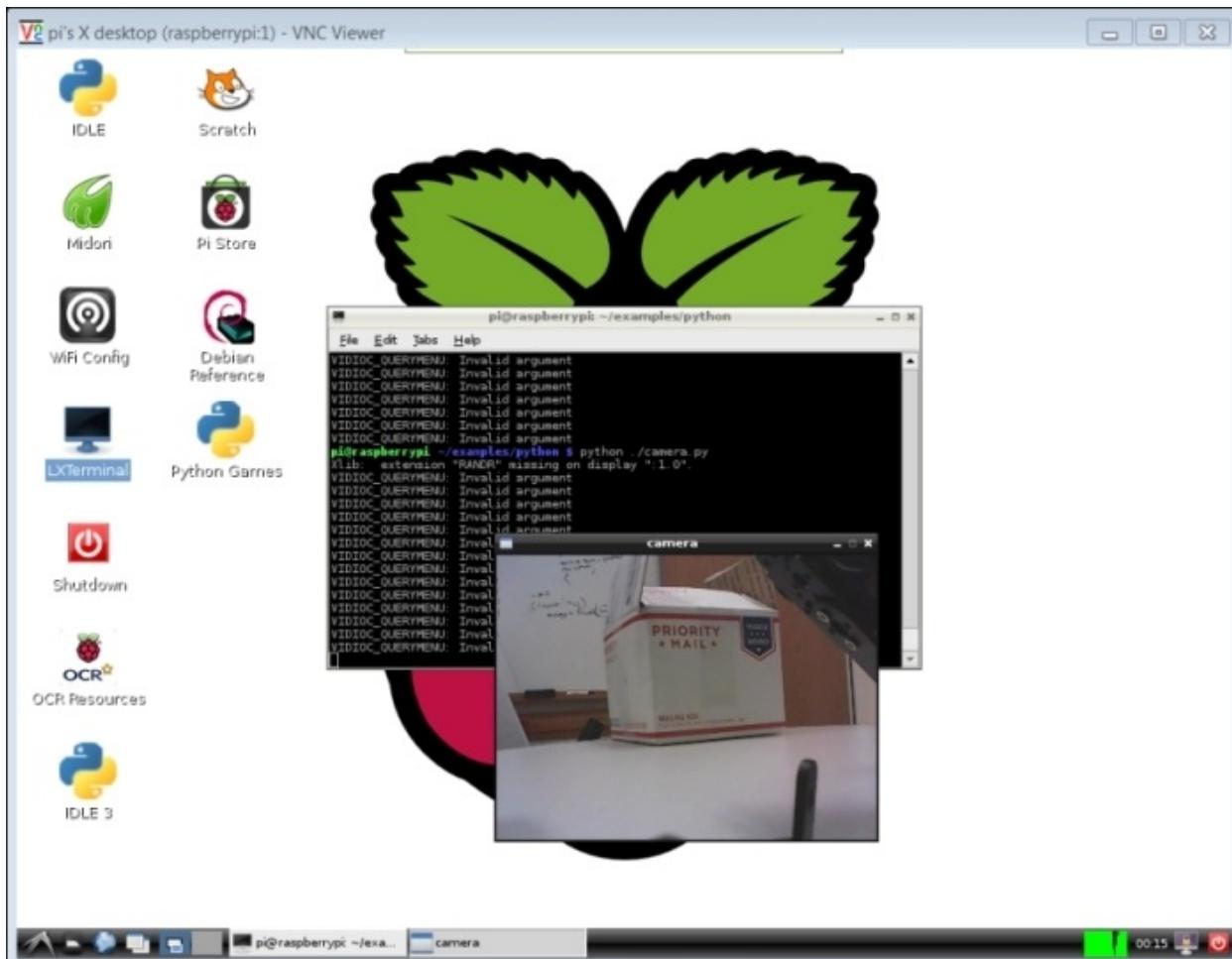
capture = cv.CaptureFromCAM(0)
cv.SetCaptureProperty(capture, 3, 360)
cv.SetCaptureProperty(capture, 4, 240)
while True:
    img = cv.QueryFrame(capture)
    cv.ShowImage("camera", img)
    if cv.WaitKey(10) == 27:
        break

-UU-----F1  camera.py      All L3      (Python)-----
wrote /home/pi/examples/python/camera.py
```

Here is an explanation of the Python code:

- `import cv2.cv as cv` – This line imports the OpenCV library so you can access its functionality.
- `import time` – This line imports the time library so you can access the time functionality.
- `cv.NamedWindow("camera", 1)` – This line creates a window that you will use to display your image.
- `capture = cvCaptureFromCAM(0)` – This line creates a structure that knows how to capture images from the connected webcam.
- `cv.SetCaptureProperty(capture, 3, 360)` – This line sets the image width to 360 pixels.
- `cv.SetCaptureProperty(capture, 4, 240)` – This line sets the image height to 240 pixels.
- `while True:` – Here you are creating a loop that will capture and display the image over and over until you press the *Esc* key.
- `img = cv.QueryFrame(capture)` – This line captures the image and stores it in the `img` data structure.
- `cv.ShowImage("camera", img)` – This line maps the `img` variable to the camera window you created previously.
- `If cv.WaitKey(10) == 27:` – This if statement checks if a key has been pressed, and if the pressed key is the *Esc* key, it executes the `break`, which stops the `while` loop and the program reach its end and stop. You need this statement in your code because it also signals OpenCV to display the image now.

Now run `camera.py`, and you should see the following screenshot:



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. We'll play with this more as we actually ask our 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.

Your project can now see! You will use this capability to do a number of impressive tasks that will use this vision capability.

Using the vision library to detect colored objects

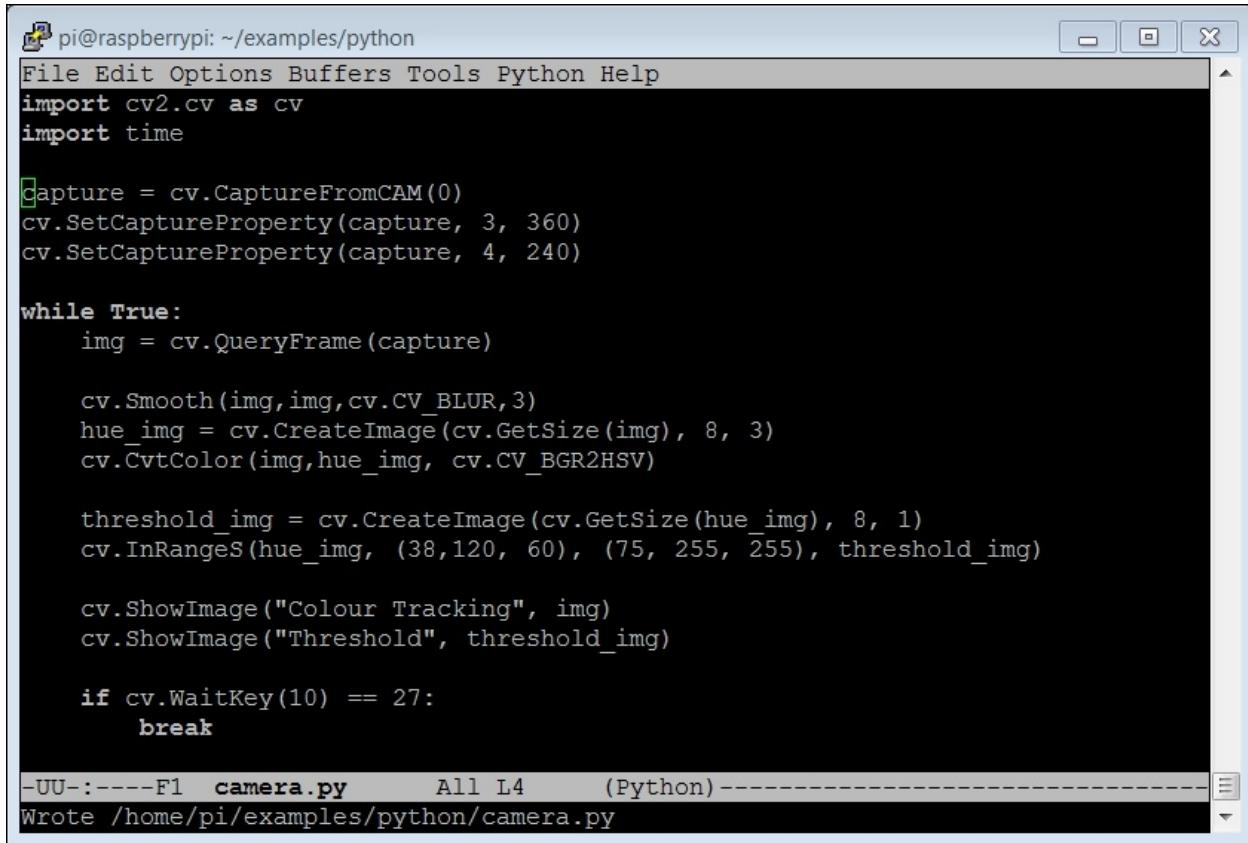
OpenCV and your webcam can track objects. This might be useful if you are building a system that needs to track and follow a colored ball.

OpenCV makes this amazingly simple by providing some high-level libraries that can help us with this task. I'm going to do this in Python, as I find it much easier to work with than C. If you feel more comfortable with C, these instructions should be fairly easy to translate. Also, performance will be better if implemented in C, so you can create the initial capability in Python and then finalize the code in C.

If you'd like, you can create a directory to hold your image-based work. To do this, perform the following steps:

1. From your home directory, create a directory named `imageplay` by typing `mkdir imageplay` while in your home directory. Then, switch from the home directory to the `imageplay` directory by typing `cd imageplay`.
2. Once there, let's bring over your `camera.py` file as a starting point by typing `cp /home/ubuntu/examples/python/camera.py camera.py`.

Now you are going to edit the file until it looks something as shown in the following screenshot:



The screenshot shows a terminal window titled "pi@raspberrypi: ~/examples/python". The window contains Python code for camera tracking. The code imports cv2.cv and time, initializes a camera capture, and enters a loop where it queries frames, smooths them, creates a hue image, converts it to HSV, and creates a threshold image. It then shows the original image and the threshold image. A key press of '27' breaks the loop. The status bar at the bottom shows "-UU-----F1 camera.py All L4 (Python)-----" and "wrote /home/pi/examples/python/camera.py".

```
pi@raspberrypi: ~/examples/python
File Edit Options Buffers Tools Python Help
import cv2.cv as cv
import time

capture = cv.CaptureFromCAM(0)
cv.SetCaptureProperty(capture, 3, 360)
cv.SetCaptureProperty(capture, 4, 240)

while True:
    img = cv.QueryFrame(capture)

    cv.Smooth(img,img,cv.CV_BLUR,3)
    hue_img = cv.CreateImage(cv.GetSize(img), 8, 3)
    cv.CvtColor(img,hue_img, cv.CV_BGR2HSV)

    threshold_img = cv.CreateImage(cv.GetSize(hue_img), 8, 1)
    cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)

    cv.ShowImage("Colour Tracking", img)
    cv.ShowImage("Threshold", threshold_img)

    if cv.WaitKey(10) == 27:
        break

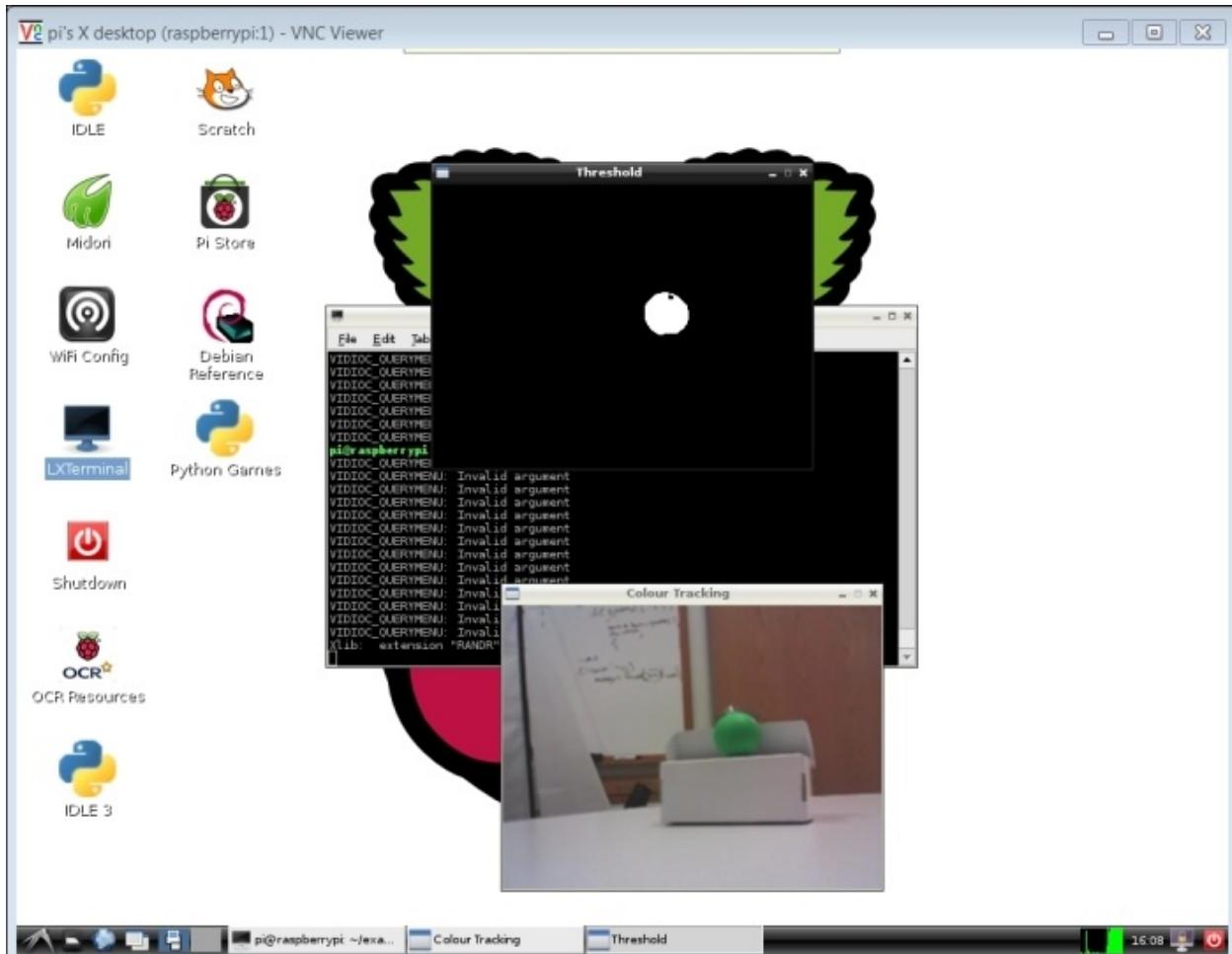
-UU-----F1 camera.py All L4 (Python)-----
wrote /home/pi/examples/python/camera.py
```

Let's look specifically at the following changes you need to make to [camera.py](#):

- `cv.Smooth(img, img, cv.CV_BLUR, 3)` – You are going to use the OpenCV library first to smoothen the image, taking out any large deviations.
- `hue_img = cv.CreateImage(cv.GetSize(img), 8, 3)` – This statement creates a default image that can hold the hue image you create in the next statement.
- `cv.CvtColor(img, hue_img, 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.CreateImage(cv.GetSize(hue_img), 8, 1)` – You are going to create yet another image, this time a black and white image that is black for any pixel that is not between two certain color values.

- `cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)` – The `(38, 160, 60), (75, 255, 255)` parameters determine the color range. In this case, I have a green ball and I want to detect the color green. For a good tutorial on using hue to specify color, try <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.
- `cv.ShowImage("Colour Tracking", img)` – This shows a window with the original image in it.
- `cv.ShowImage("Threshold", threshold_img)` – This shows a window with just the threshold image.

Now run the program. You'll need to either have a display, keyboard, and mouse connected to the board, or you can run it remotely using vncserver. Run the program by typing `sudo python ./camera.py`. You should see a single black image, but move this window and you will expose the original image window as well. Now take your target (I used my green ball) and move it into the frame. You should see something as 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 [camera.py](#) file to look as follows:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/examples/python". The window contains Python code for color tracking using OpenCV. The code imports cv2.cv and time, initializes a camera capture, and enters a loop where it smooths the image, converts it to HSV, creates a threshold image, finds contours, and draws rectangles around them. It then shows the image and waits for a key press. The status bar at the bottom shows "-UU-----F1 camera.py All L11 (Python)".

```
pi@raspberrypi: ~/examples/python
File Edit Options Buffers Tools Python Help
import cv2.cv as cv
import time

capture = cv.CaptureFromCAM(0)
cv.SetCaptureProperty(capture, 3, 360)
cv.SetCaptureProperty(capture, 4, 240)

while True:
    img = cv.QueryFrame(capture)

    cv.Smooth(img,img, cv.CV_BLUR,3)
    hue_img = cv.CreateImage(cv.GetSize(img), 8, 3)
    cv.CvtColor(img,hue_img, cv.CV_BGR2HSV)

    threshold_img = cv.CreateImage(cv.GetSize(hue_img), 8, 1)
    cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)

    storage = cv.CreateMemStorage(0)
    contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)

    points = []
    while contour:
        rect = cv.BoundingRect(list(contour))
        contour = contour.h_next()
        size = (rect[2] * rect[3])
        if size > 100:
            pt1 = (rect[0], rect[1])
            pt2 = (rect[0] + rect[2], rect[1] + rect[3])
            cv.Rectangle(img, pt1, pt2, (38, 160, 60))

    cv.ShowImage("Colour Tracking", img)

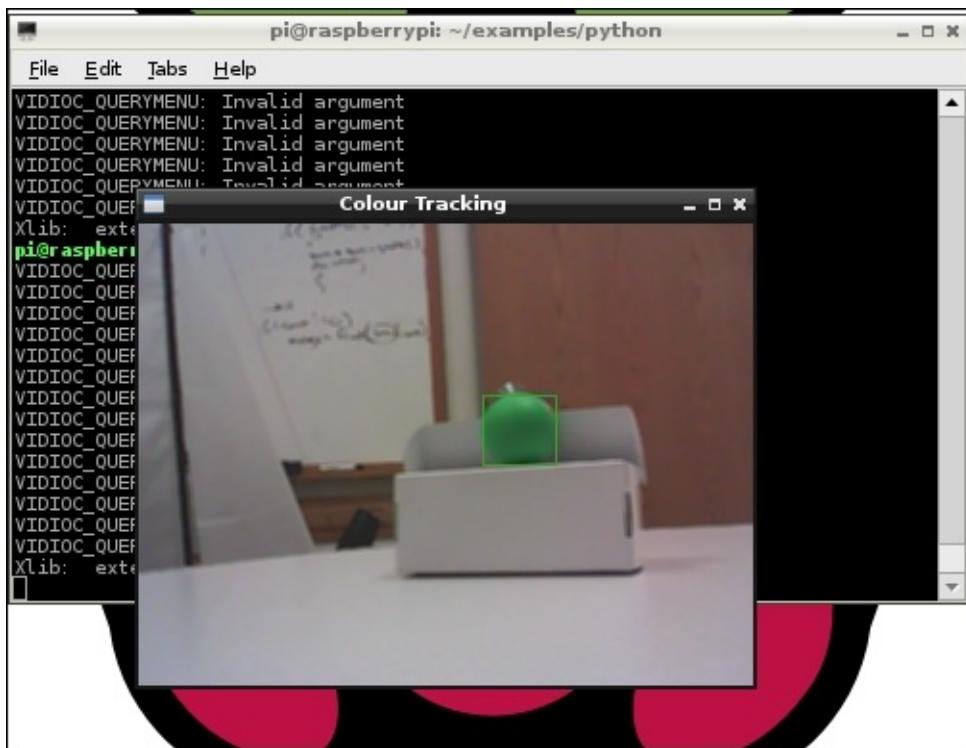
    if cv.WaitKey(10) == 27:
        break
-UU-----F1 camera.py All L11 (Python)
```

Start by editing just below the line `cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)`. The lines used are as follows:

- `storage = cv.CreateMemStorage(0)` – This line creates some memory for you to manipulate images in.
- `contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)` – This finds all the areas on your image that are within the threshold. There could be more than one, so you may want to capture them all.
- `points = []` – The creates an array for us to hold all the different possible color points.

- `while contour`: – Now add a while loop that will let you step through all the possible contours. By the way, it is important to note that if there is another larger green blob in the background, you will "find" that location. Just to keep this simple, we'll assume your green ball is unique.
- `rect = cv.BoundingRect(list(contour))` – This gets a bounding rectangle for each area of color. The rectangle is defined by the corners of a rectangle around the "blob" of color.
- `contour = contour.h_next()` – This will prepare you for the next contour, if one exists.
- `size = (rect[2] * rect[3])`: This calculates the diagonal length of the rectangle you are evaluating. The data structure `rect` contains four integers; `0` and `1` for the pixel values of the lower-left corner of the box, and `2` and `3` for the size in pixels of the rectangle.
- `if size > 100`: – Here you check to see if the area is big enough to be of concern. `100` tells your program to not worry about any rectangles that are less than 100 pixels in area. You may want to vary this based on the application.
- `pt1 = (rect[0], rect[1])` – Define a `pt1` variable and set its two values to the x and y coordinates of the left side of the blob's rectangular location.
- `pt2 = (rect[0] + rect[2], rect[1] + rect[3])` – Define a `pt2` variable and set its two values to the x and y coordinates of the right side of the blob's rectangular location.
- `cv.Rectangle(img, pt1, pt2, (38, 160, 60))` – Now you add a rectangle to your original image by identifying where it is located.

Now that the code is ready, you can run it. You should see something as shown in the following screenshot:



You can now track your object.

Now that you have the code, you can modify the color or add more colors. You also have the location of your object, so later you can attempt to follow the object or manipulate it in some way.

OpenCV is an amazing, powerful library of functions. You can do all sorts of incredible things with just a few lines of code. Another common feature you may want to add to your projects is motion detection. If you'd like to try, there are several good tutorials; try looking at the following links:

- <http://derek.simkowiak.net/motion-tracking-with-python/>
- <http://stackoverflow.com/questions/3374828/how-do-i-track-motion-using-opencv-in-python>
- <https://www.youtube.com/watch?v=8QouvYMfmQo>
- <https://github.com/RobinDavid/Motion-detection-OpenCV>

Having a webcam connected to your system provides all kinds of complex vision capabilities. You can get 3D vision with OpenCV using two cameras. There are several good places; for example, the code in the `samples/cpp` directory that came with OpenCV has a sample `stereo_match.cpp`. For more information, refer to

<http://code.google.com/p/opencvstereovision/source/checkout>.

Summary

As we learned in this chapter, your projects can now speak and see! You can issue commands, and your projects can respond to changes in the physical environment sensed by the webcam. In the next chapter, you will add mobility using motors, servos, and other methods.

Chapter 5. Creating Mobile Robots on Wheels

You can now talk to the board and it can talk back and even see. Now you will add the capability to move the entire project using wheels. Perhaps the easiest way to make your projects mobile is to use a wheeled platform. In this chapter, you will be introduced to some of the basics of controlling DC motors and using Raspberry Pi to control the speed and direction of your wheeled platform.

In this chapter, you will learn how to perform the following actions:

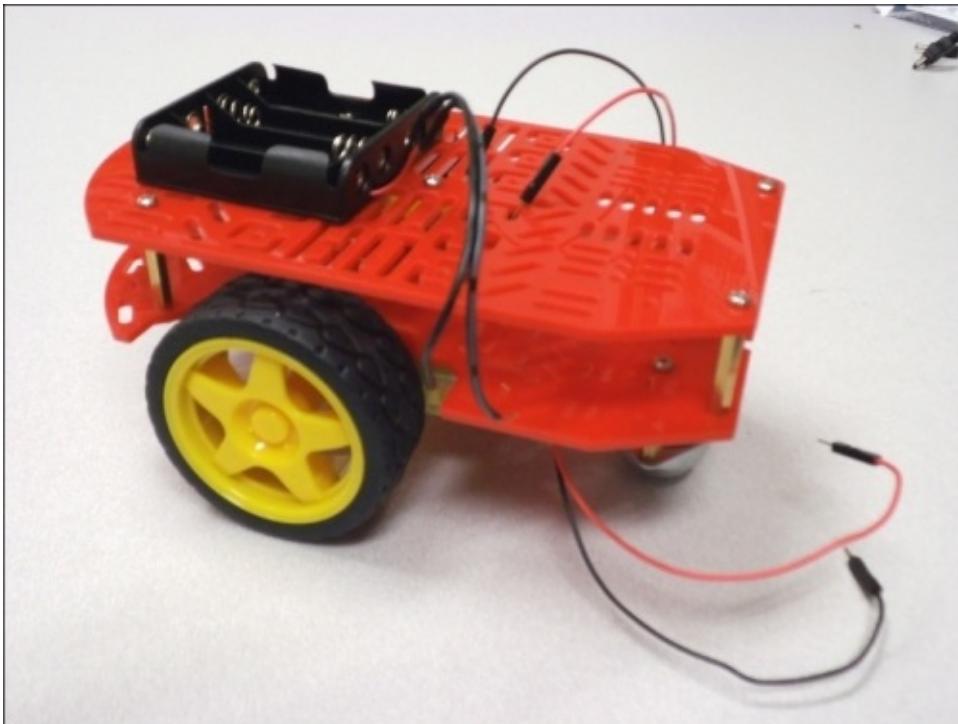
- Gather the required hardware
- Use a motor controller to control the speed of your platform
- Control your mobile platform programmatically using Raspberry Pi
- Make your platform truly mobile by issuing voice commands

Gathering the required hardware

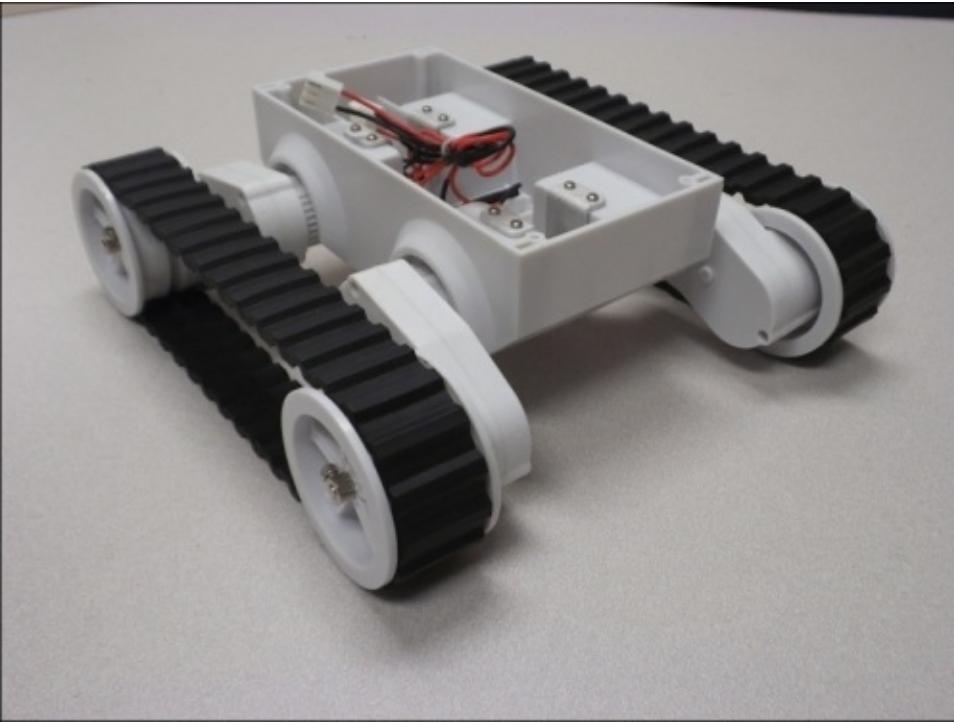
You'll need to add some hardware, specifically a wheeled or tracked platform, to make your project mobile. You're going to use a platform that will use differential motion to propel and steer the vehicle. This simply means that instead of turning the wheels, you're going to vary the speed and direction of the two motors that drive the wheels or tracks. There are a lot of choices: some are completely assembled, while others require some assembly; or, you can buy the components and construct your own custom mobile platform.

Throughout this book, I'm going to assume that you don't want to do any soldering or mechanical machining yourself, so let's look at a couple of the more popular variants that are available completely assembled or that can be assembled with simple tools (a screwdriver and/or pliers). The simplest mobile platform is one that has two DC motors, each of which controls a single wheel with a small ball in the front or at the back. DC motors are very simple to control; as you vary the DC voltage, the speed on the motor varies. The following is an image of one called the Magic

Chassis sold by SparkFun at www.sparkfun.com:



I did have to assemble this platform, but it was fairly straightforward. For more choices of two-wheeled platforms, go to <http://www.robotshop.com/2-wheeled-development-platforms-1.html> or dx.com/es/p/diy-disc-type-2-wheel-smart-car-model-body-black-yellow-184395. You could also choose a tracked platform instead of a wheeled platform. A tracked platform has more traction, but is not as nimble in that it takes a longer distance to turn. Additionally, manufacturers make preassembled units. The following is an image of a preassembled tracked platform made by Dagu; it's called the Dagu Rover 5 Tracked Chassis:

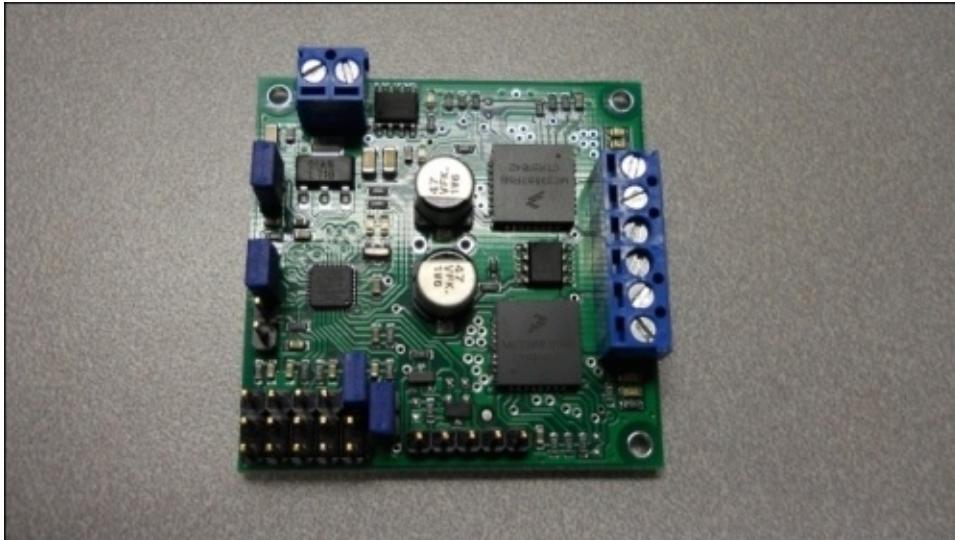


Since you have a mobile platform, you'll need a mobile power supply for Raspberry Pi. I personally like the external 5V rechargeable cell phone batteries that are available at almost any place that sells cell phones. These batteries can be charged using a USB cable connected either through a DC power supply or directly to a computer USB port. I like to choose one that comes with two USB output connectors so I can power Raspberry Pi and the powered USB hub if I need to, as you can see in the following image:



You'll also need a USB cable to connect your battery to Raspberry Pi, but you can just use the cable supplied with Raspberry Pi.

Now that you have the mobile platform, you'll need to connect your Raspberry Pi to one more bit of hardware. You'll need some hardware that will take the control signals from your Raspberry Pi and turn them into voltage to control the speed of the motors. Unfortunately, the board cannot source enough current to power the motors directly, so you'll need a circuit that can do this. I strongly suggest that you purchase a motor controller instead of designing and building your own. There are numerous possibilities; however, I'm going to suggest one that requires no internal programming and allows you to talk over USB to control the motors. You'll want one that can also control two motors. The one I prefer is the Pololu TReX Jr Dual Motor Controller DMC02 motor controller from Pololu, which you can order from www.pololu.com. The following image shows the Pololu TReX Jr Dual Motor Controller DMC02:



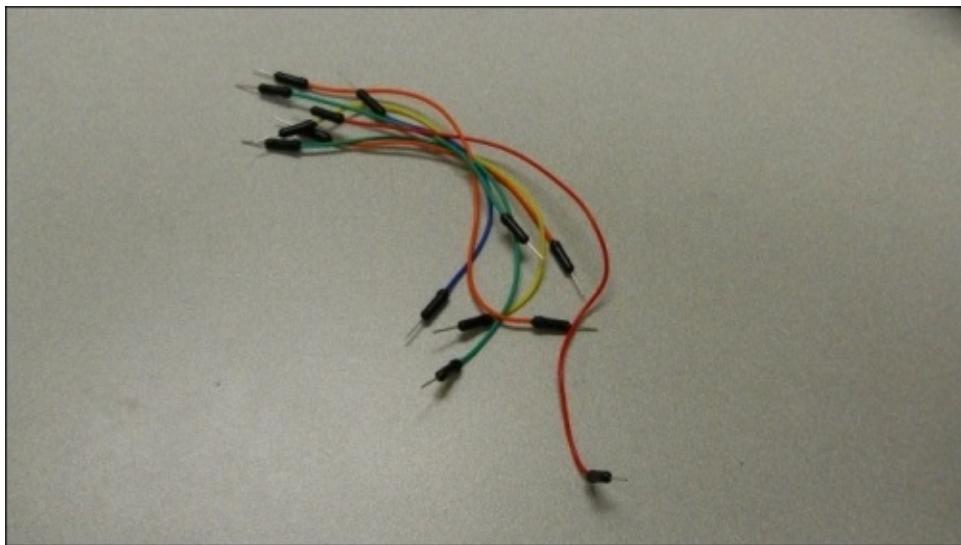
The piece of hardware shown in the previous image will turn USB commands into voltage that controls your motors.

The motor controller is going to want a serial input, but we'll find it easiest to talk over USB. You'll connect the controller to Raspberry Pi via a USB using a TTL serial cable, which are available at www.amazon.com and a number of other suppliers. You can also build one yourself, although that would take some soldering, as shown in the following image:



The final component you will need is four short pieces of wire that are less than 2 inches long and stripped at the end. You can also buy these wires, called male-male jumper wires, pre-made online, for example, at www.pololu.com or www.amazon.com. They also come in male-female

and female-male versions. The following is an image of a set of jumper wires:



Again, I've selected components in such a way as to avoid the requirement of soldering.

Now that you have all the hardware, let's walk through a quick tutorial of how the system works and then follow some step-by-step instructions to make your project mobile.

Using a motor controller to control the speed of your platform

The first step to make the platform mobile is adding a motor controller. This allows us to control the speed of each wheel (or track) independently. Before you get started, let's spend some time understanding the basics of motor control. Whether you choose the two-wheeled mobile platform or the tracked platform, the basic movement control is the same. The unit moves by engaging the motors. If the desired direction is forward, both motors are run at the same speed. If you want to turn the unit, the motors are run at different speeds. The unit can turn in a circle if you run one motor forward and one backwards.

DC motors are fairly straightforward devices. The speed and direction of the motor is controlled by the magnitude and polarity of the voltage applied to its terminals. The higher the voltage, the faster the motor turns. If you reverse the polarity of the voltage, you can reverse the direction in which the motor is turning.

The magnitude and polarity of the voltage are not the only important factors when you think about controlling the motors. The power that your motor can apply to move your platform is also determined by the voltage and current supplied at its terminals.

There are **general-purpose input/output (GPIO)** pins on Raspberry Pi that you could use to create the control voltage and drive your motors. These GPIO pins provide direct access to some of the control lines made available by the processor itself. However, the unit cannot source enough current and your motors will not be able to generate enough power to move your mobile platform. You can also cause physical damage to your Raspberry Pi board. That is why you will need to use a motor controller, as it will provide both voltage and current so that your platform can move reliably. In this case, I have chosen to hook up the motor controller via USB, making the connections and programming much simpler.

The next step in making your project mobile is connecting the motor controller to the platform. There are two connections you need to make: you'll first need to connect a battery to the controller and then connect the motors themselves.

I'm going to use the tracked platform as my project. To connect the battery, find the output connectors on the battery holder. It should look as shown in the following image:

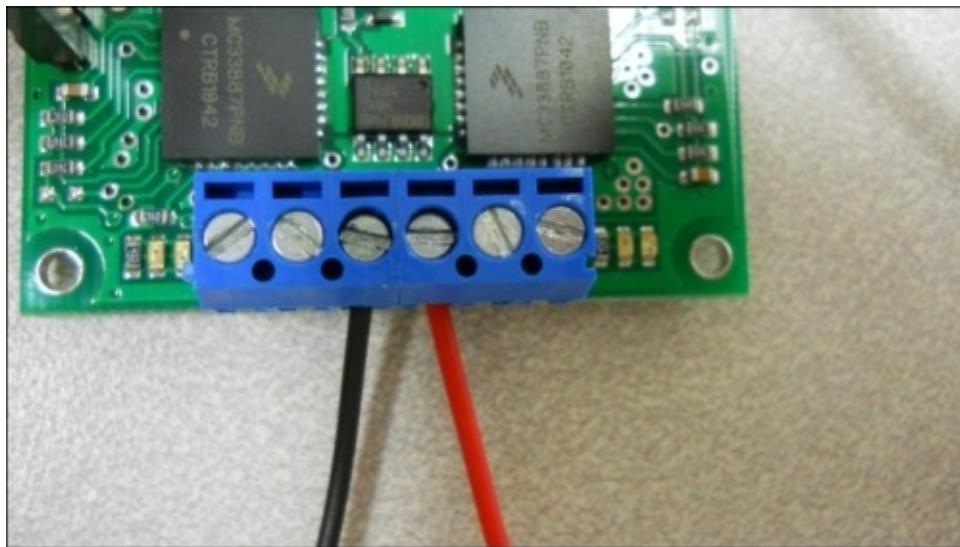


On the back of the motor controller, notice the labels VIN, OUTB, OUTA, and GND as shown in the following image:

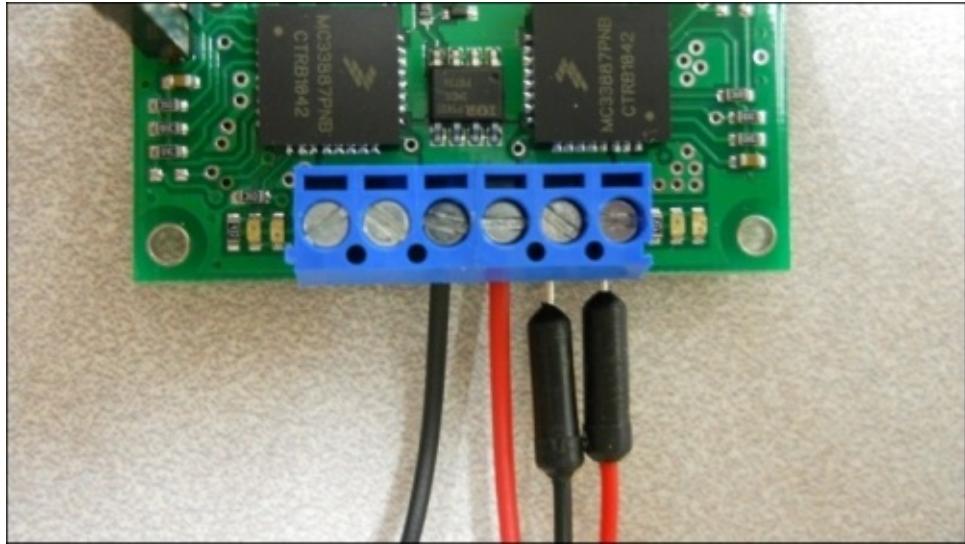


Once you have the battery pack ready, do the following:

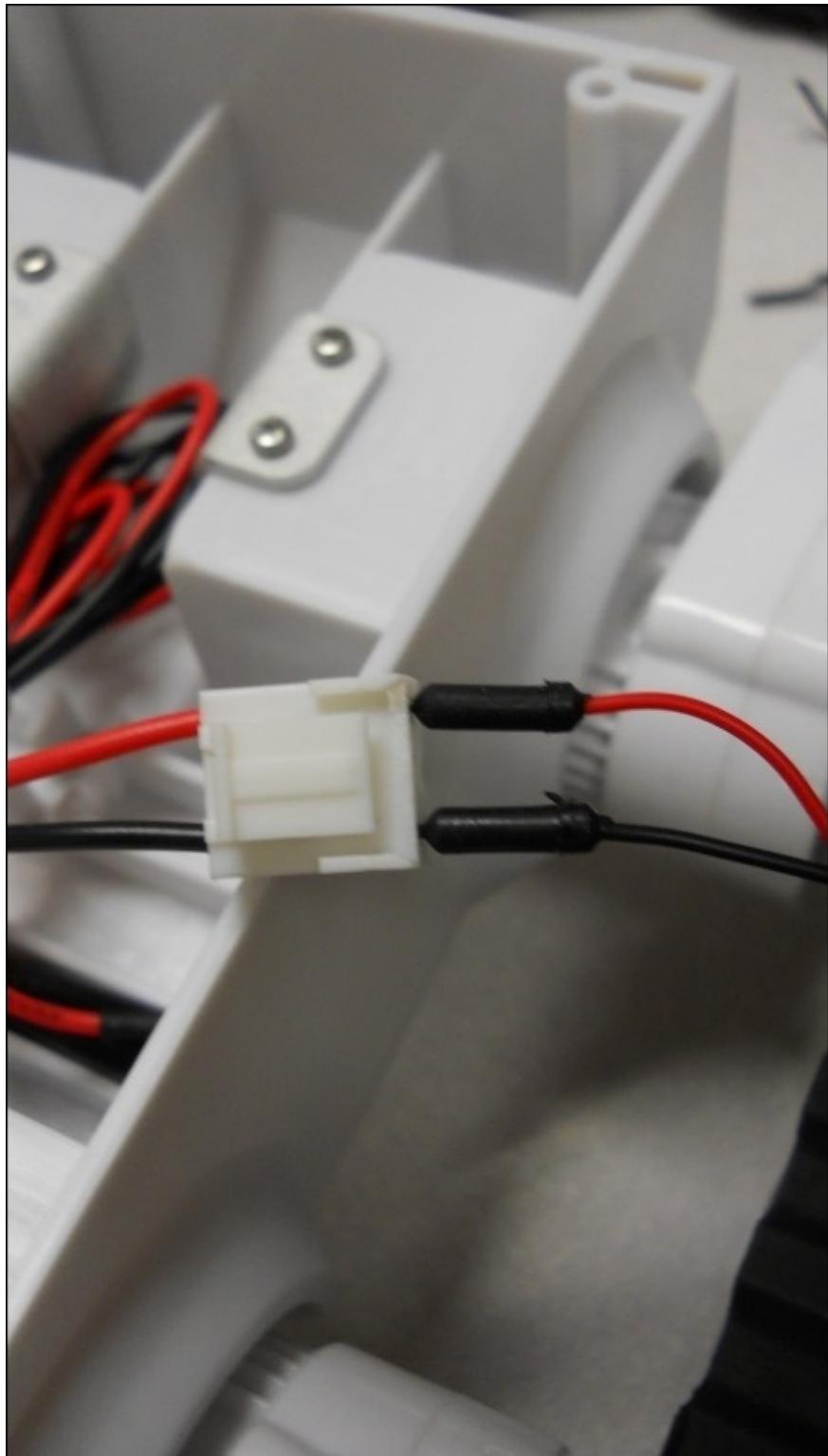
1. Insert the wires into the motor controller using the blue connectors marked VIN and GND. VIN is the constant DC voltage from your batteries and GND is the ground connection from your batteries. A and B are the control signals to your DC motors. You'll notice that on the battery connector, one of the wires is red. Insert it into the VIN connector and then tighten the screw connector.
2. On the battery connector, you'll notice that one of the wires is black. Insert that wire into the connector marked GND and tighten the screw connector. The connections will look as shown in the following image:



3. Now connect one of the motors to the motor controller by connecting the red and black wires with the male connectors to the two inner, blue screw connectors—the red one to A and the black one to B—using the male-male jumper wires. The connection to the motor controller should look as shown in the following image:



4. The connections to the motor should look as shown in the following image:

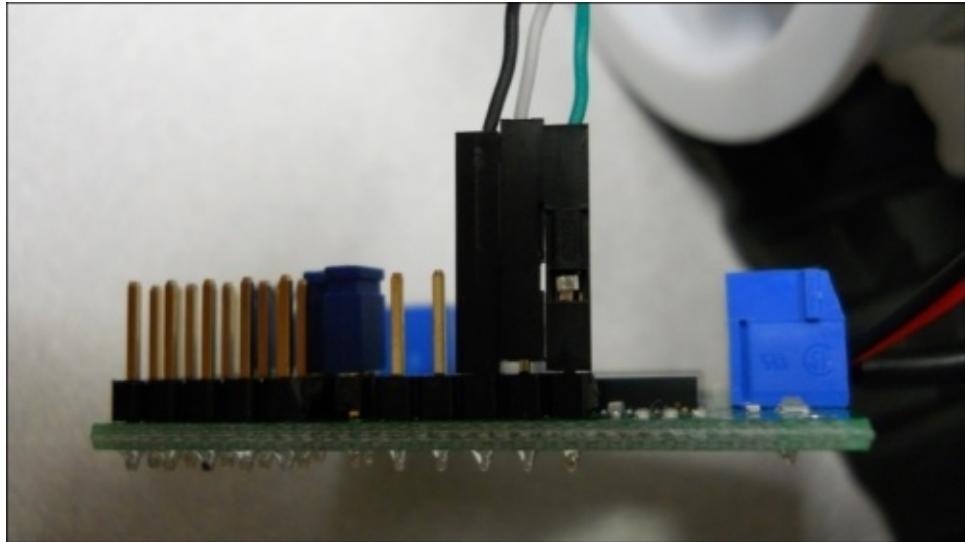


You can now test the system using software provided by Pololu. It is worthwhile to try it on a Windows PC first. To do this, perform the following steps:

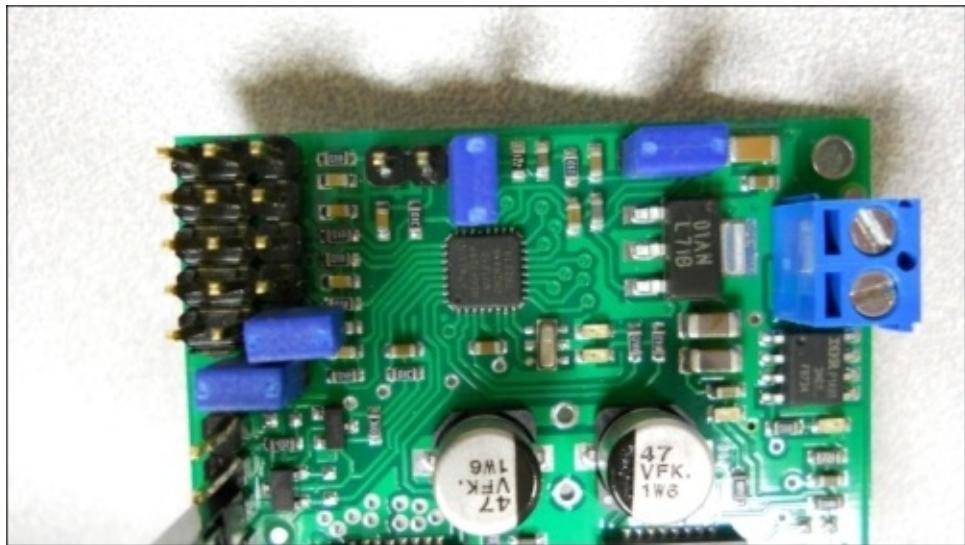
1. Install the Windows driver software from the site <http://www.pololu.com/product/767/resources> by clicking on **TReX Configurator utility for Windows version 100608**. Unzip the software.
2. Before running the software, you need to connect your motor controller to the USB using a TTL serial cable so that your computer can talk to it. To do this, find the serial connector on the side of the board. On the underside of the board, you will see the labels shown in the following image:



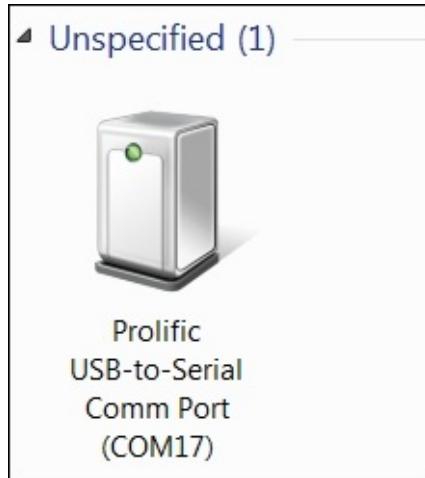
3. Now connect the green cable to the SI connection, the white cable to the SO connection, and the black cable to the G connection. The following is an image of what this should look like:



4. This is the last step before you connect the board to the computer. Remove the jumper wire at the top of the board so that the board will accept serial commands. The following is an image of the jumper, and I have just turned it sideways so that I don't lose sight of the jumper:

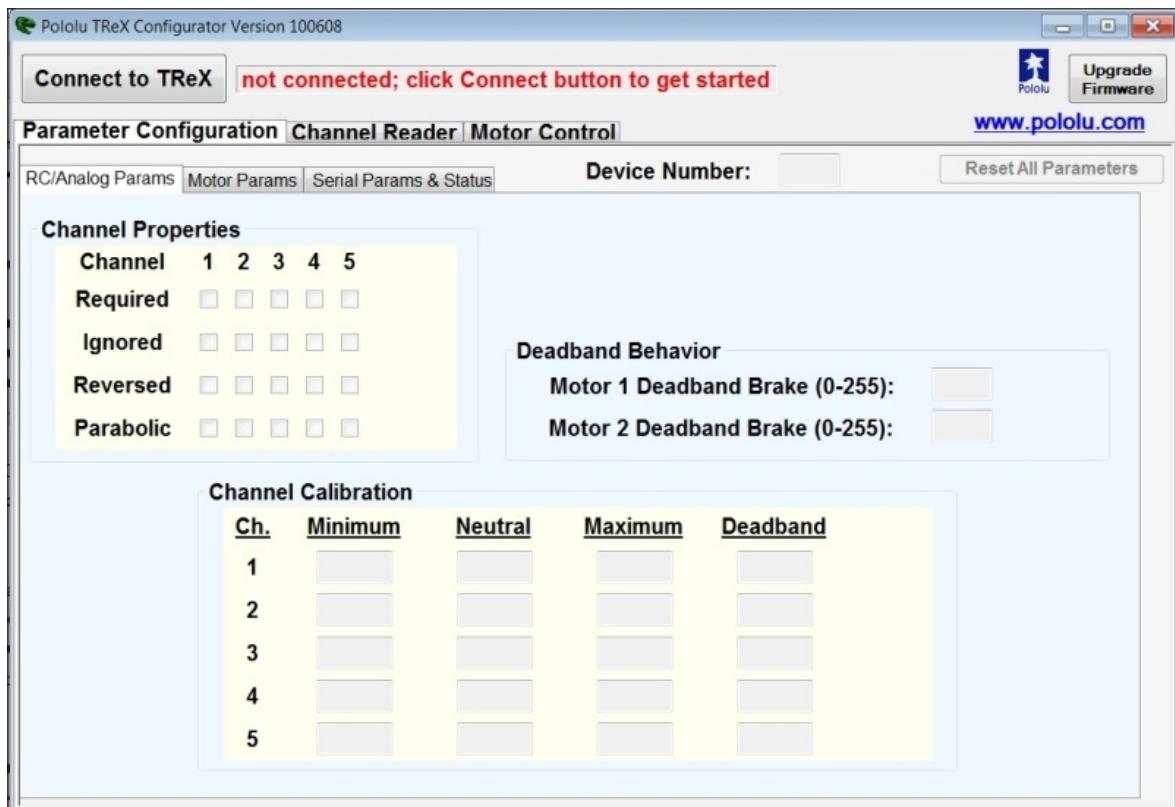


Now you should supply the unit with power by making sure you have batteries in the battery holder and then connect your motor controller to the PC. Your PC should recognize the board and create a COM connection as shown in the following screenshot:

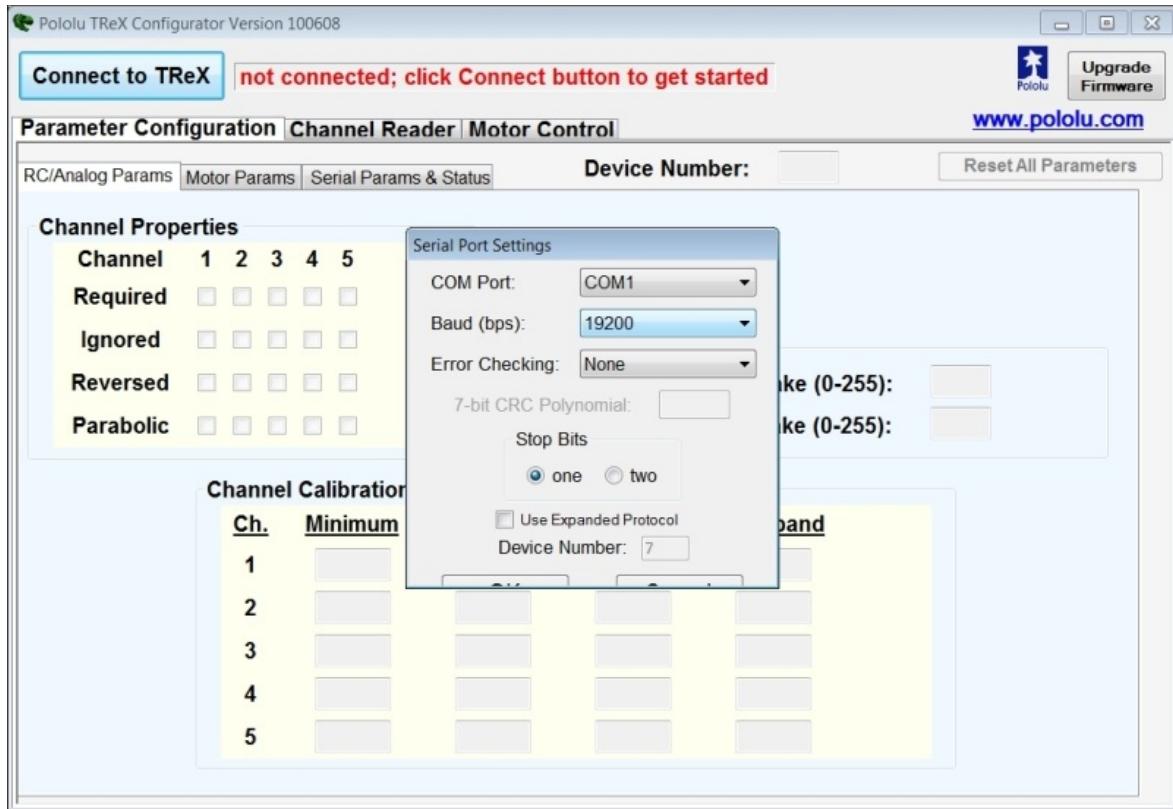


Now you can start the software by performing the following steps:

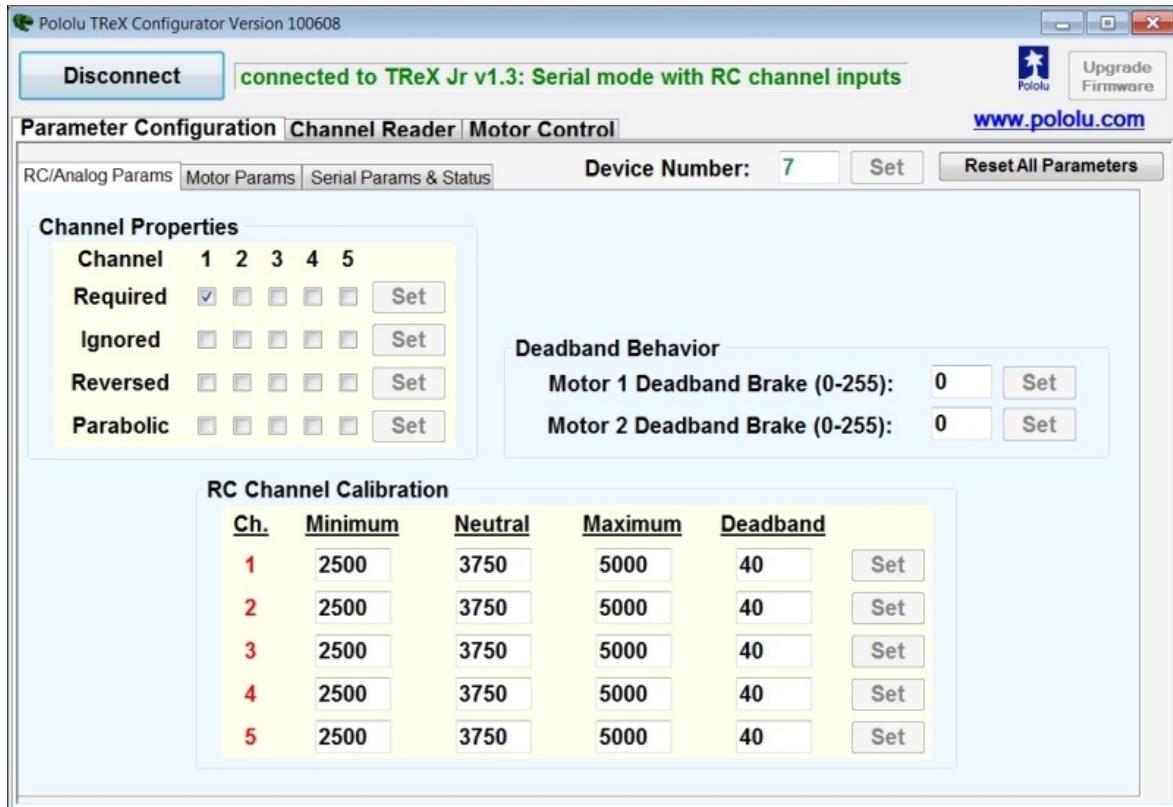
1. Go to the directory where you installed the software and select the [setup.exe](#) file. Once you start the software, you should see what is shown in the following screenshot:



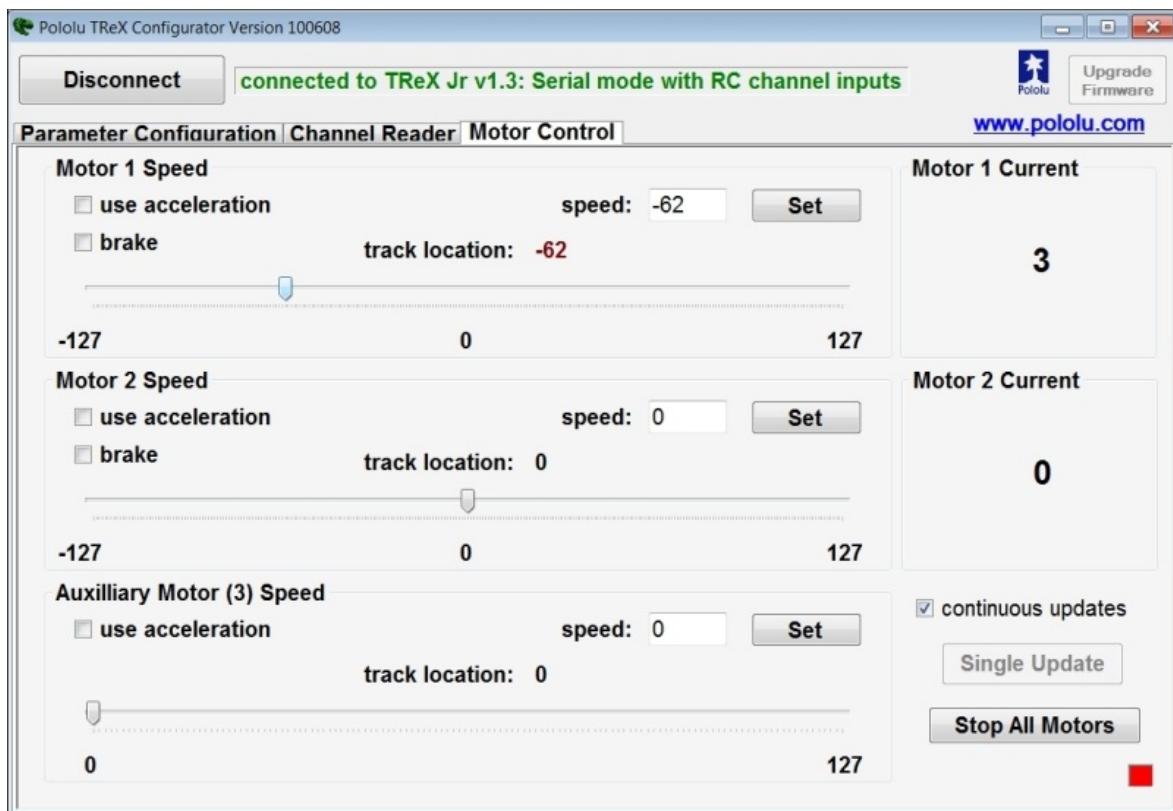
2. Click on the **Connect to TReX** button; you will see what is shown in the following screenshot:



3. Click on the **COM Port** selection and select the COM port your device is connected to; this is, in my case, COM 17. You should now see what is shown in the following screenshot:



4. You can now control the motor by going to the **Motor Control** tab on the screen. You should now be able to control the speed of Motor 1 by moving the slider. The motor should then run when you select a value other than **0**, as shown in the following screenshot:



The motor should now be running! You have control of the DC motor. Now connect the second motor, repeating the connection steps detailed earlier in this chapter. Now you should be able to control the speed of both Motor 1 and Motor 2.

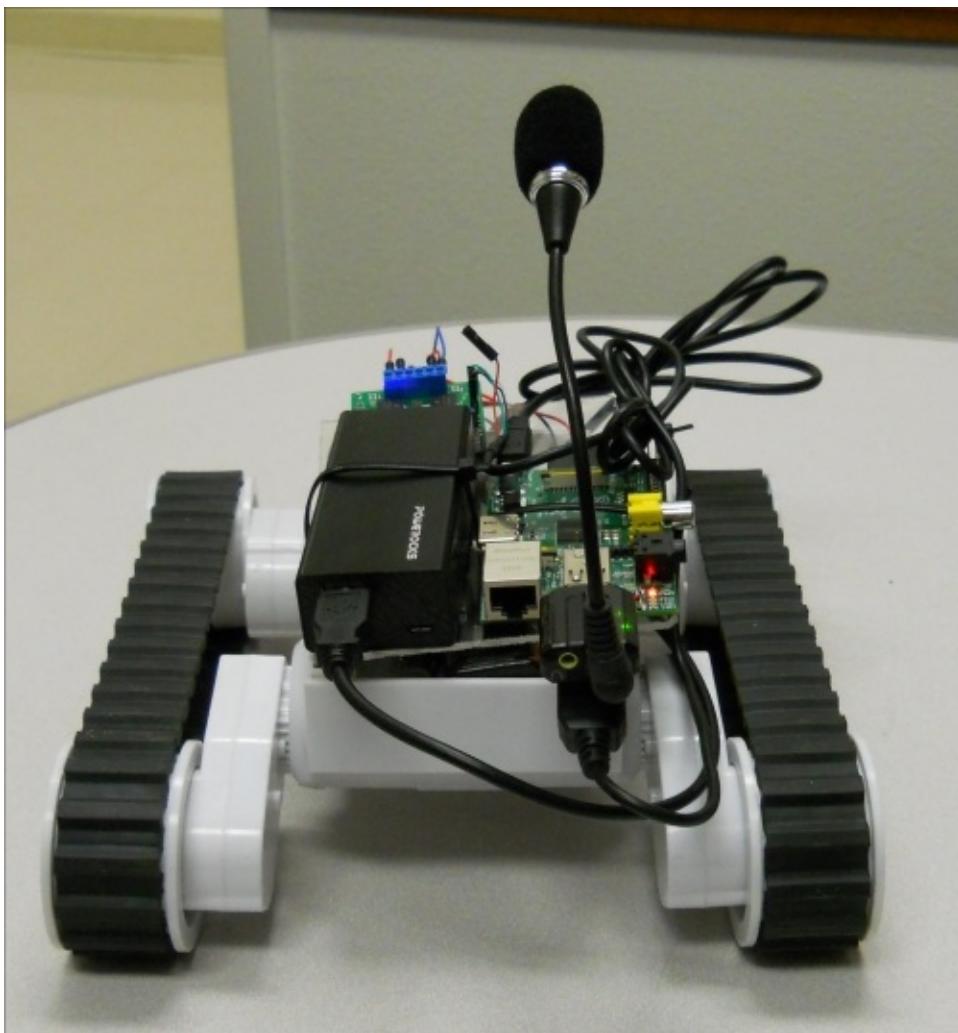
The next step is to control your motor controller through your Raspberry Pi. If you are going to do this remotely, log in through PuTTY. If you are doing this directly on a monitor, simply log in.

Now that you have your motor running, your next step is to plug both motor controllers into the USB hub and control them programmatically using Raspberry Pi.

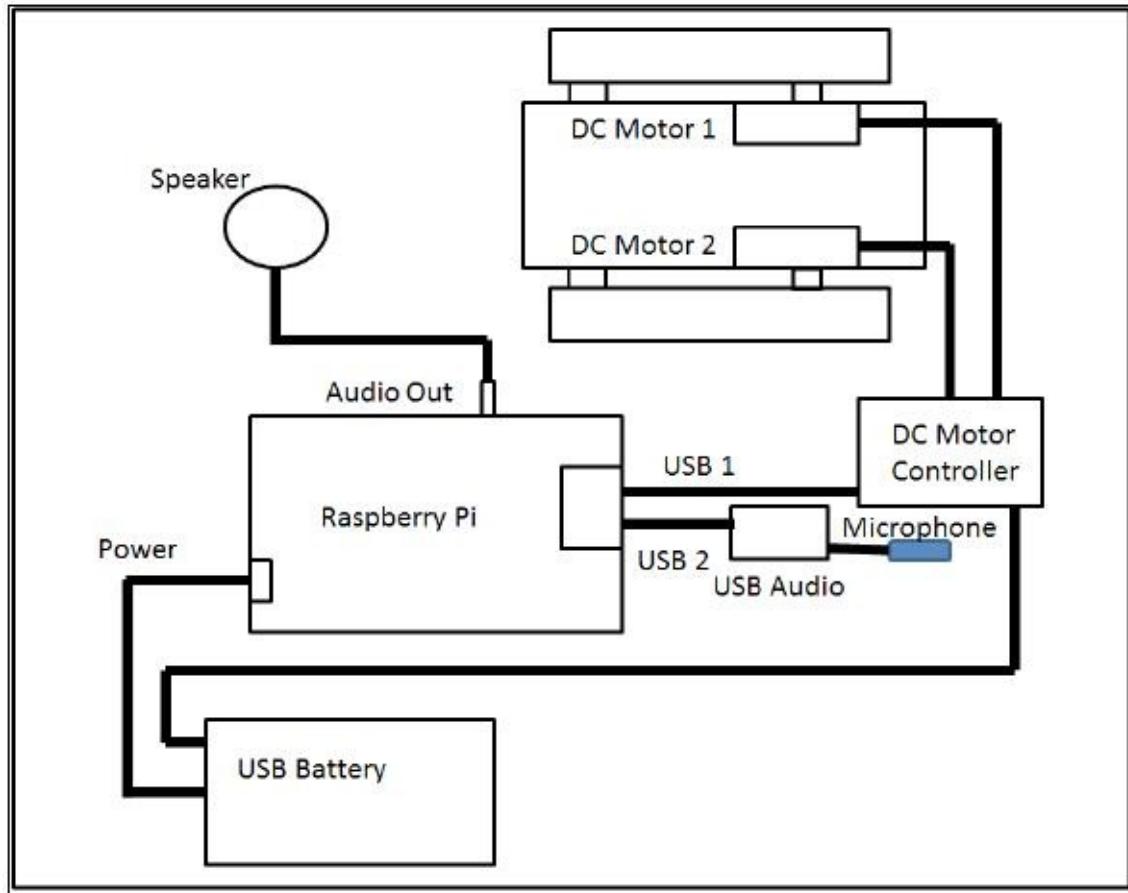
Controlling your mobile platform programmatically using Raspberry Pi

Now that you have your basic motor controller functionality up and running, you need to connect both motor controllers to Raspberry Pi. This section will cover that and also show you how to control your entire platform programmatically.

First, let's configure all of the hardware on top of the mobile platform, as shown in the following image:



I tend to use lots of cable ties, but if you'd like it to look even more polished, feel free to use more metal and nuts and bolts. The following is a diagram of the connections:



I suggest that you use Python in your initial attempts to control the motor. It is very straightforward to code, run, and debug your code in Python. I am going to include the directions for Python in this chapter; you can also go to the Pololu website at www.pololu.com/ and find instructions for how to access the capabilities in C.

The first Python program you are going to create is shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/track". The window contains Python code for controlling a DC motor via a serial port. The code defines a function `setSpeed` that takes parameters `ser`, `motor`, `direction`, and `speed`. It uses `chr` to convert integers to bytes and `ser.write` to send the command. The main part of the script initializes a serial connection to '/dev/ttyUSB0' at 19200 baud, sets speeds for both motors, and then closes the connection. The status bar at the bottom indicates the file is named "dcmotor.py", has 11 lines, and is in Python mode.

```
pi@raspberrypi: ~/track
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time

def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)

    ser.write(sendByte)
    ser.write(chr(speed))

ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
setSpeed(ser, 0, 0, 100)
setSpeed(ser, 1, 1, 100)
time.sleep(1)
setSpeed(ser, 0, 0, 0)
setSpeed(ser, 1, 0, 0)
time.sleep(1)
ser.close()

-UU:----F1  dcmotor.py      All L1      (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

1. To create the program shown in the previous screenshot, create a directory called `track` in your home directory by typing `mkdir track` and then type `cd track`.
2. You should now be in the `track` directory.
3. Now open the file by typing `emacs dcmotor.py` (if you are using a different editor, open a new file with the name `dcmotor.py`).
4. Now enter the program. Let's go through the program step-by-step:
 - `#!/usr/bin/python` – This is the first line which allows your program to be run outside of the Python environment. You'll use it later when you want to execute your code using voice commands.
 - `import serial` – The next line imports the serial library. You need this to talk to your motor controllers. In order to run this program, you'll need the serial library. Install it by typing `sudo apt-get install Python-serial` in the prompt. You'll then need to add yourself to the `dialout` group; do this by typing `sudo`

`adduser ubuntu dialout`. Then type a `sudo reboot` command to enable all these changes.

- `import time` – This line imports the time library; you'll need this to add some delays in your code.
- `def setSpeed(ser, motor, direction, speed)` – This function sets the speed and direction of one of your two motors. Since you are going to control the motors throughout your program, it is easier if you put this functionality in a function.
- `if motor == 0 and direction == 0:` – This `if` statement tests to see if Motor 1 is going in the forward direction.
- `sendByte = chr(0xC2)` – This line sets the address if you are going to send data to Motor 1 to move it in the forward direction.
- `if motor == 1 and direction == 0:` – This `if` statement tests to see if Motor 2 is going in the forward direction.
- `sendByte = chr(0xCA)` – This line sets the address if you are going to send data to Motor 2 in the forward direction.
- `if motor == 0 and direction == 1:` – This `if` statement tests to see if Motor 1 is going in the reverse direction.
- `sendByte = chr(0xC1)` – This line sets the address if you are going to send data to Motor 1 in the reverse direction.
- `if motor == 1 and direction == 1:` – This `if` statement tests to see if Motor 2 is going in the reverse direction.
- `sendByte = chr(0xC9)` – This line sets this address if you are going to send data to Motor 2 in the reverse direction.
- `ser.write(sendByte)` – This line writes the `sendByte` command to the motor controller, which sets the motor's movements in different directions.
- `ser.write(chr(speed))` – This line writes the speed to the motor controller, setting the speed.
- `ser = serial.Serial('devttyUSB0', 19200, timeout = 1)` – This line opens the serial port.
- `setSpeed(ser, 0, 0, 100)` – This line calls the function and sets the speed of the motor from `1` to `100`.
- `setSpeed(ser, 1, 1, 100)` – This line calls the function and sets the speed of the motor from `2` to `100`. Note that you have to make this motor go in the opposite direction if you want the platform to move forward.
- `time.sleep(1)` – This line makes the program wait for one second.

- `setSpeed(ser, 0, 0, 0)` – This line calls the function and sets the speed of the motor from `1` to `0`.
 - `setSpeed(ser, 1, 0, 0)` – This line calls the function and sets the speed of the motor from `2` to `0`.
 - `time.sleep(1)` – This line again makes the program wait for one second.
 - `ser.close()`: This line closes the serial port.
5. After the installation process, you can run your program. To do this, type `python dcmotor.py`. Your motor should run for one second and then stop. You can now control the motor through Python. Additionally, you'll want to make this program available to run from the command line. Type `chmod +x dcmotor.py`. If you now do `ls` (list programs), you'll see that your program is now green, which means you can execute it directly. Now you can type `./dcmotor.py`.

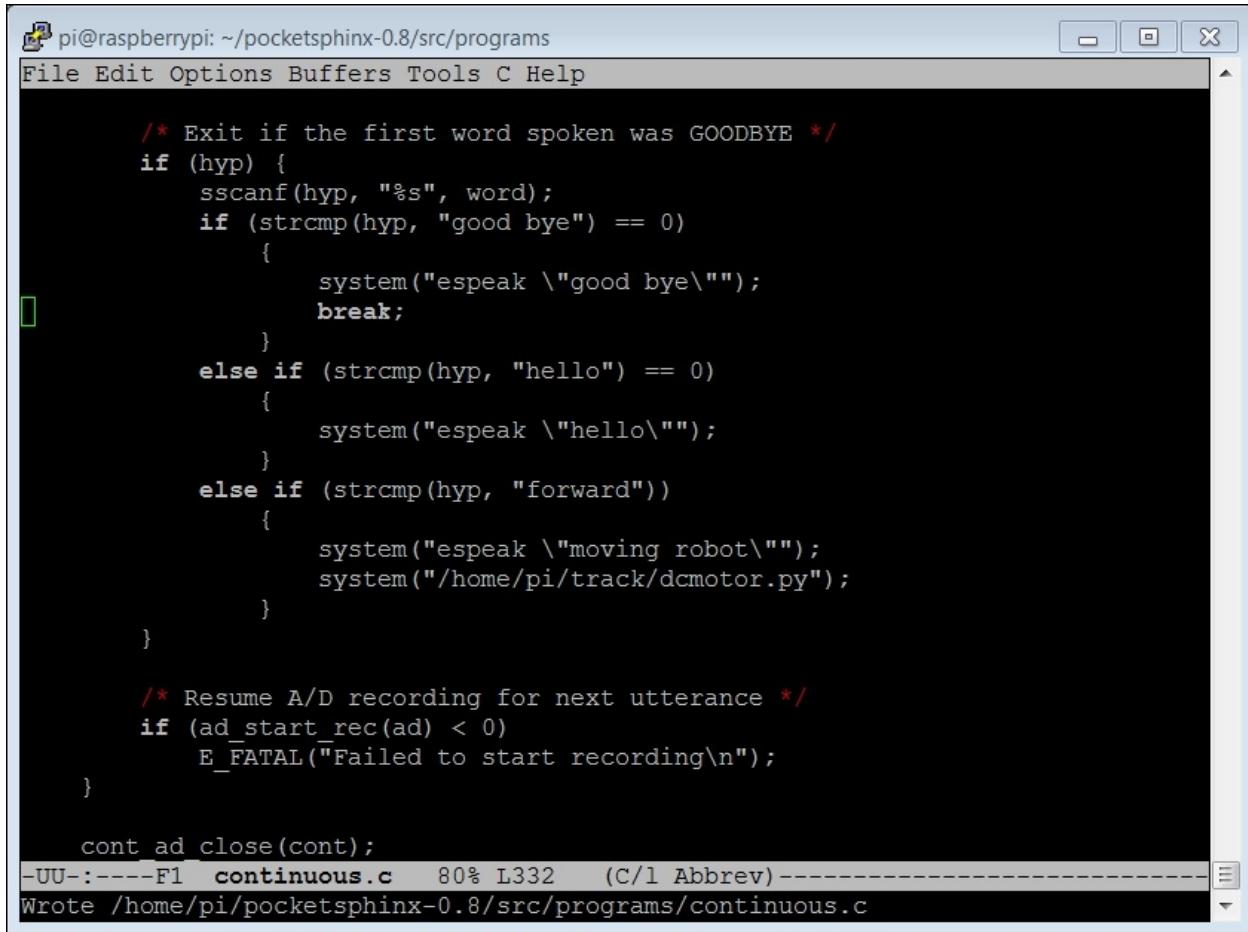
Now that you know the basics of commanding your mobile platform, feel free to add even more `setSpeed` commands to make your mobile platform move. Running just one motor will make the platform turn, as will running both motors in opposite directions.

The platforms you've looked at up until now had two DC motors. It would be easy to add even more motors. There are several platforms that provide DC motors for all four wheels. In such a case, you could just add more motor controllers and then update the code to access each individual controller.

Making your mobile platform truly mobile by issuing voice commands

You should now have a mobile platform that you can program to move in any number of ways. Unfortunately, you still have your LAN cable connected, so the platform isn't completely mobile. Once you have begun the program, you can't alter its behavior. In this section, you will use the principles from [Chapter 3, Providing Speech Input and Output](#), to issue voice commands and initiate movement.

You'll need to modify your voice recognition program so it will run your Python program when it gets a voice command. If you feel your knowledge is rusty on how this works, review [Chapter 3, Providing Speech Input and Output](#). You are going to make a simple modification to the `continuous.c` program in `homeubuntu/pocketsphinx-0.8/src/programs`. To do this, type `cd homeubuntu/ pocketsphinx-0.8/src/programs` and then type `emacs continuous.c`. The changes will appear in the same section as your other voice commands and will look as shown in the following screenshot:



The screenshot shows a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains C code for a voice command program. The code includes logic to exit if the first word spoken was "GOODBYE", handle commands like "hello" and "forward", and resume A/D recording for the next utterance. It also includes a check for failed recording start and a command to close the audio device. The status bar at the bottom indicates the file is "continuous.c", line 332, 80% complete, and shows the command "Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c".

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
File Edit Options Buffers Tools C Help

    /* Exit if the first word spoken was GOODBYE */
    if (hyp) {
        sscanf(hyp, "%s", word);
        if (strcmp(hyp, "good bye") == 0)
        {
            system("espeak \"good bye\"");
            break;
        }
        else if (strcmp(hyp, "hello") == 0)
        {
            system("espeak \"hello\"");
        }
        else if (strcmp(hyp, "forward"))
        {
            system("espeak \"moving robot\"");
            system("/home/pi/track/dcmotor.py");
        }
    }

    /* Resume A/D recording for next utterance */
    if (ad_start_rec(ad) < 0)
        E_FATAL("Failed to start recording\n");
}

cont ad close(cont);
-UU-----F1 continuous.c 80% L332 (C/l Abbrev)-----
Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c
```

The additions are pretty straightforward. Let's walk through them:

- `else if (strcmp(hyp, "forward") == 0)`: This line checks the word as recognized by your voice command program. If it corresponds with the word FORWARD, you will execute everything that is in the `if` statement. You use `{ }` to group and tell the system which commands go with this `else if` clause.
- `system("espeak \"moving robot\")")`: This line executes `espeak`, which should tell us that you are about to run your robot program. By the way, you need to type `\"` because the `"` character is a special character in Linux, and if you want the actual `"` character, you need to have the `\` character precede it.
- `system("homepi/track/dcmotor.py")`: This is the program you will execute. In this case, your mobile platform will do whatever the program `dcmotor.py` tells it to.

Next, you will need to recompile the program, so type `make`; the

executable `pocketsphinx_continuous` will be created. Run the program by typing `/pocketsphinx_continuous`. Disconnect the LAN cable; the mobile platform will now take the `forward` voice command and execute your program.

You should now have a complete mobile platform! When you execute your program, the mobile platform can now move around based on what you have programmed it to do.

You don't have to put all of your capabilities in one program. You can create several programs, each with a different function, and then connect each of them to their appropriate voice commands. Perhaps one command could move your robot forward, a different one can move it backwards, and another can turn it right or left.

Now you have your mobile platform up and ready to move around. You can give commands using your voice. In the next chapter, you'll be introduced to a different kind of mobile platform, that is, one with legs.

You have already covered how to add vision to your Raspberry Pi project. A great addition to your mobile robot is the ability to follow a colored object attached to a target.

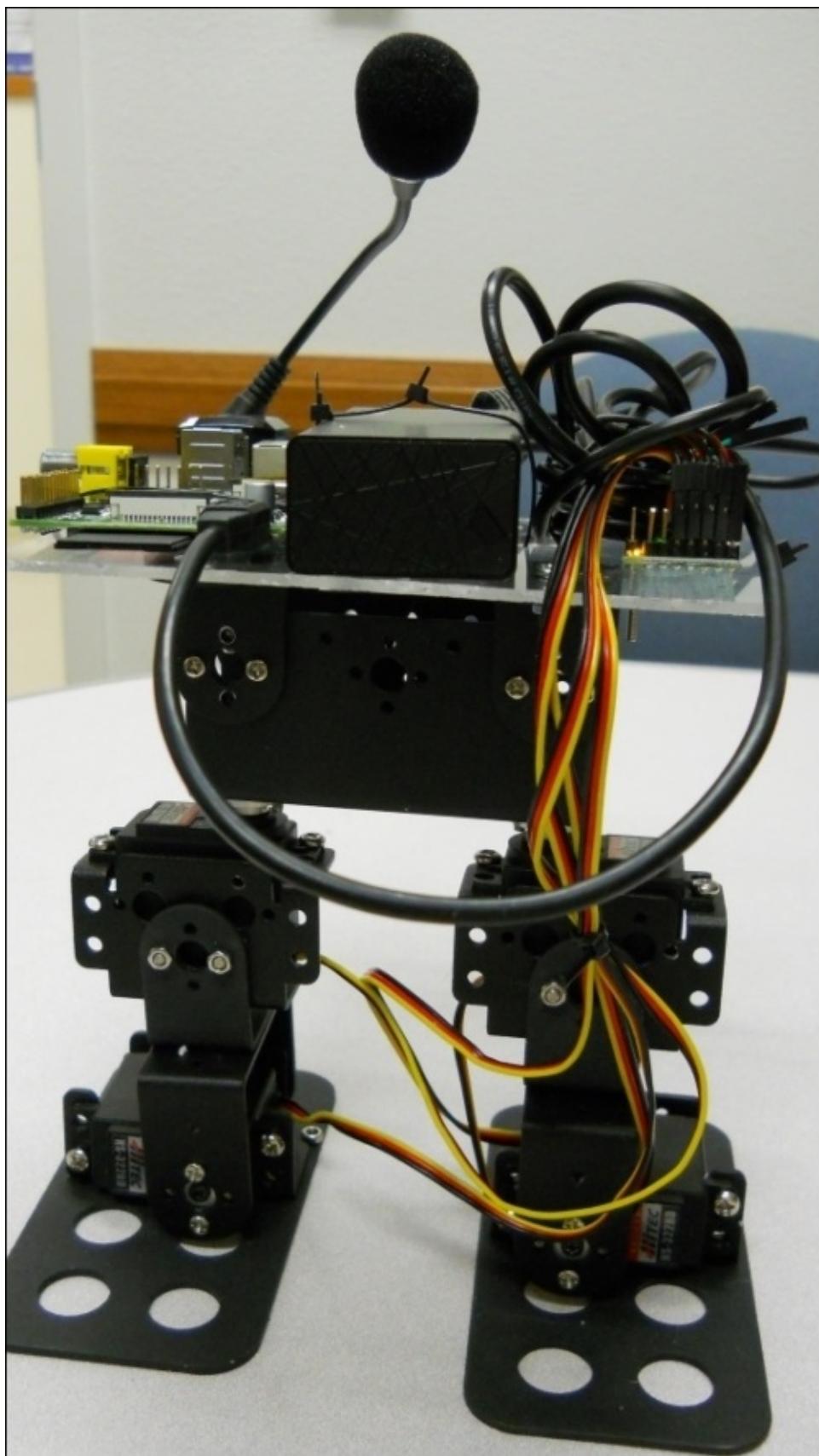
Previously, you have learned how to use OpenCV to find a colored object and then found out where in your field of view (left or right, up or down) it existed. You can use this to decide whether to move your mobile platform right, left, forward, or backward. Try this and then move the target to see whether or not your mobile robot can follow it.

Summary

This chapter provided you with the opportunity to make your robot mobile. Whether you choose a wheeled or tracked platform, your robot should now be able to move around. In the next chapter, you'll learn how to build a robot with legs—an even more flexible mobile platform.

Chapter 6. Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs

In the previous chapter, we covered wheeled and tracked movement. That's cool enough, but what if you want your robot to navigate uneven ground? Now you will add the capability to move the entire project using legs. In this chapter, you will be introduced to some of the basics of servo motors and using Raspberry Pi to control the speed and direction of your legged platform. The following is an image of a finished project:



Even though you've learned to make your robot mobile by adding wheels or tracks, this mobile platform will only work well on smooth, flat surfaces. Often, you'll want your robot to work in environments where the path is not smooth or flat; perhaps you'll even want your robot to go up stairs or around curbs. In this chapter, you'll learn how to attach your board, both mechanically and electrically, to a platform with legs so that your projects can be mobile in many more environments. Robots that can walk! What could be more amazing than that?

In this chapter, we will cover the following topics:

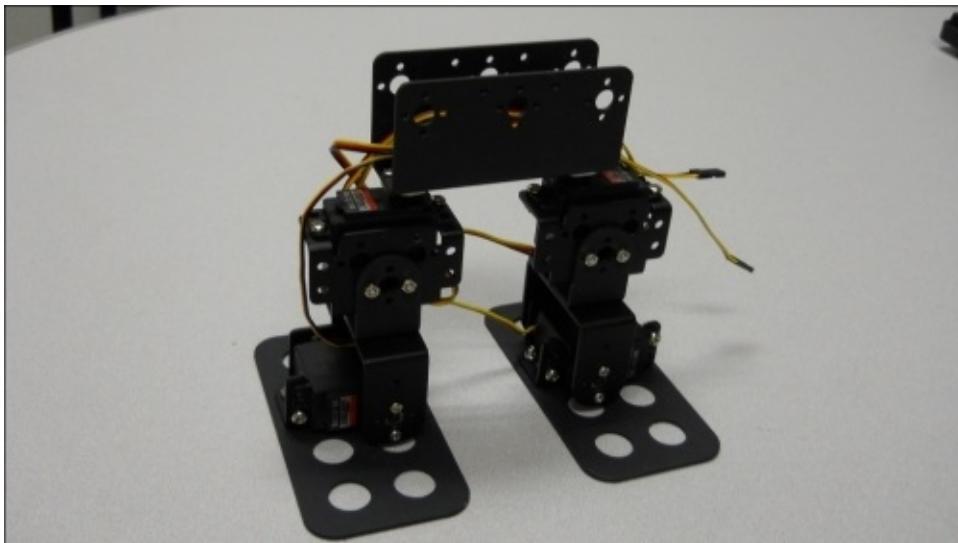
- Connecting Raspberry Pi to a two-legged mobile platform using a servo motor controller
- Creating a program in Linux so that you can control the movement of the two-legged mobile platform
- Making your robot truly mobile by adding voice control

Gathering the hardware

In this chapter, you'll need to add a legged platform to make your project mobile.

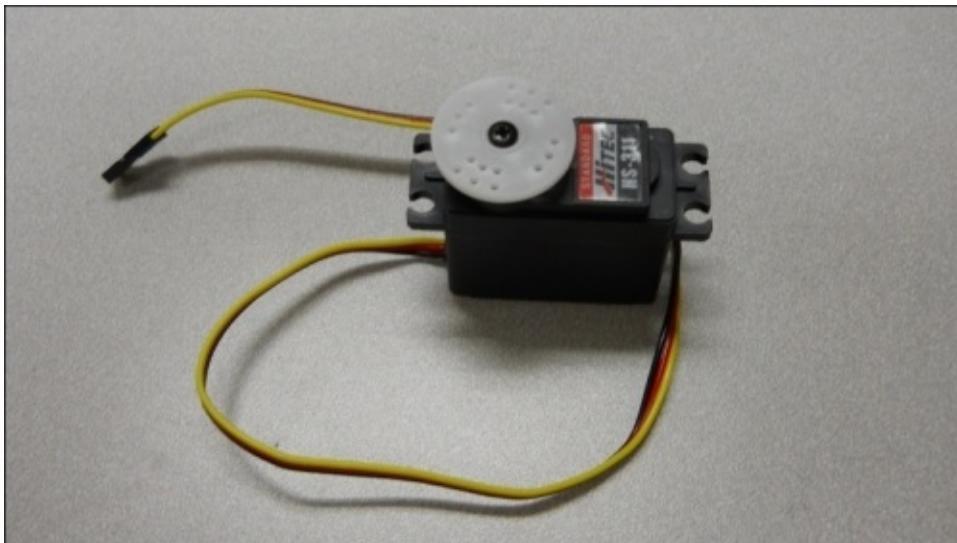
For a legged robot, there are a lot of choices for hardware. As seen in [Chapter 5, Creating Mobile Robots on Wheels](#), some are completely assembled, others require some assembly, and you may even choose to buy the components and construct your own custom mobile platform. Also I'm going to assume that you don't want to do any soldering or mechanical machining yourself, so let's look at several choices of hardware that are available completely assembled or can be assembled using simple tools (a screwdriver and/or pliers).

One of the simplest legged mobile platforms is one that has two legs and four servo motors. The following is an image of this type of platform:



We'll use this legged mobile platform in this chapter because it is the simplest to program and the least expensive, requiring only four servos. To construct this platform, you must purchase the parts and then assemble them yourself. Find the instructions and parts list at <http://www.lynxmotion.com/images/html/build112.htm>. Another easy way to get all the mechanical parts (except servos) is by purchasing a biped robot kit with six **DOF (degrees of freedom)**. This will contain the parts needed to construct your four-servo biped. These six DOF bipeds can be purchased on eBay or at <http://www.robotshop.com/2-wheeled-development-platforms-1.html>.

You'll also need to purchase the servo motors. Servo motors are similar to the DC motors you may have used in [Chapter 5, Creating Mobile Robots on Wheels](#), except that servo motors are designed to move at specific angles based on the control signals that you send. For this type of robot, you can use standard-sized servos. I like the Hitec HS-311 or HS-322 for this robot. They are inexpensive but powerful enough in operations. You can get them on Amazon or eBay. The following is an image of an HS-311 servo:



As in the last chapter, you'll need a mobile power supply for Raspberry Pi. I personally like the 5V cell phone rechargeable batteries that are available at almost any place that supplies cell phones. Choose one that comes with two USB connectors; you can use the second port to power your servo controller. The mobile power supply shown in the following image mounts well on the biped hardware platform:



You'll also need a USB cable to connect your battery to Raspberry Pi.

You should already have one of those.

Now that you have the mechanical parts for your legged mobile platform, you'll need some hardware that will turn the control signals from your Raspberry Pi into voltage levels that can control the servo motors. Servo motors are controlled using a signal called PWM. For a good overview of this type of control, see

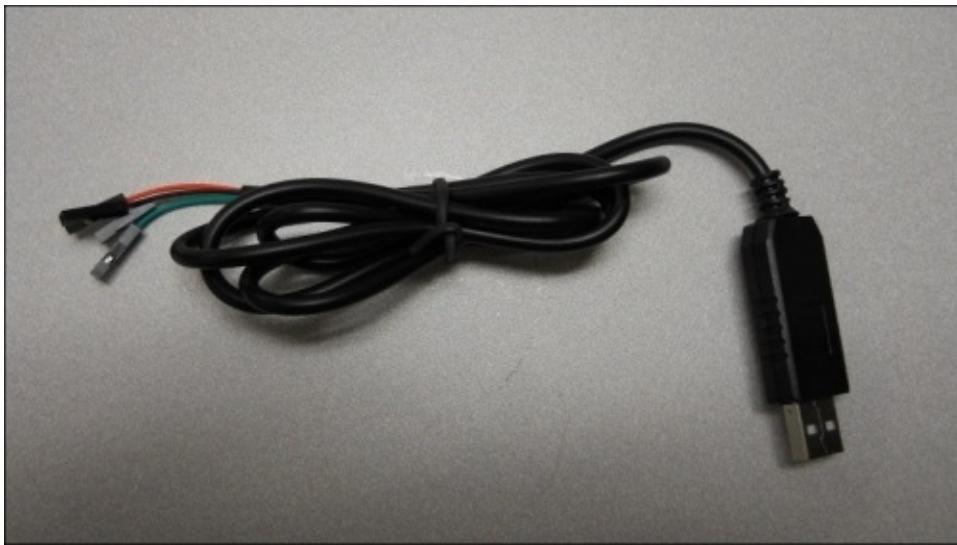
http://pcbheaven.com/wikipages/How_RC_Servos_Works/ or <https://www.ghielectronics.com/docs/18/pwm>. You can find tutorials that show you how to control servos directly using Raspberry Pi's **GPIO (General Purpose Input/Output)** pins, for example, those at <http://learn.adafruit.com/adafruit-16-channel-servo-driver-with-raspberry-pi/> and <http://www.youtube.com/watch?v=ddIDgUymbxc>. For ease of use, I've chosen to purchase a servo controller that can talk over a USB and control the servo motor. These controllers protect my board and make controlling many servos easy. My personal favorite for this application is a simple servo motor controller utilizing a USB from Pololu that can control six servo motors—the Micro Maestro 6-Channel USB Servo Controller (Assembled). The following is an image of the unit:



Make sure you order the assembled version. This piece of hardware will turn USB commands into voltage levels that control your servo motors. Pololu makes a number of different versions of this controller, each able to control a certain number of servos. Once you've chosen your legged platform, simply count the number of servos you need to control and choose a controller that can control that many servos. In this book, we

will use a two-legged, four-servo robot, so I will illustrate the robot using the six-servo version. Since you are going to connect this controller to Raspberry Pi via USB, you'll also need a USB A to mini-B cable.

You'll also need a power cable running from the battery to your servo controller. You'll want to purchase a USB to FTDI cable adapter that has female connectors, for example, the **PL2303HX USB to TTL to UART RS232 COM** cable available on [amazon.com](https://www.amazon.com). The TTL to UART RS232 cable isn't particularly important, other than that the cable itself provides individual connectors to each of the four wires in aUSB cable. The following is an image of the cable:



Now that you have all the hardware, let's walk through a quick tutorial of how a two-legged system with servos works and then some step-by-step instructions to make your project walk.

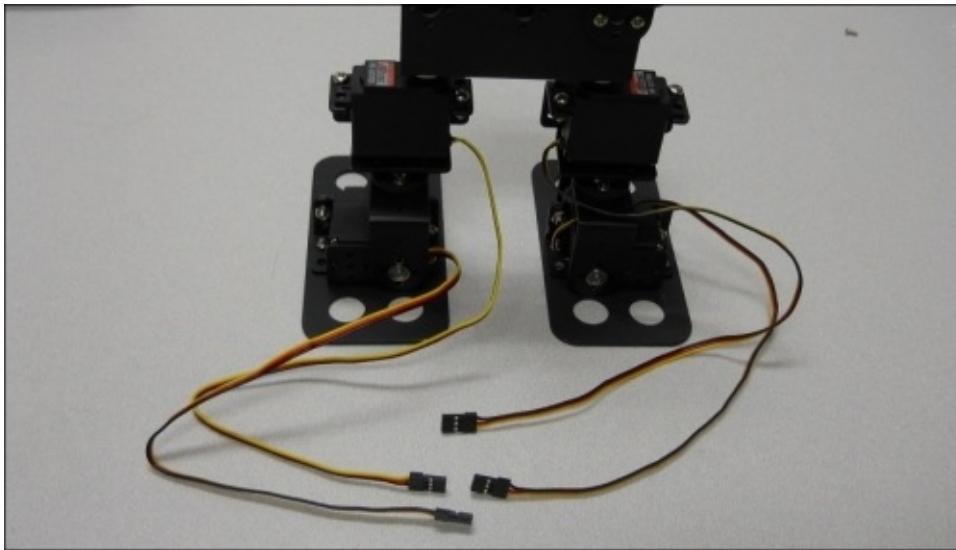
Connecting Raspberry Pi to the mobile platform using a servo controller

Now that you have a legged platform and a servo motor controller, you are ready to make your project walk! Before you begin, you'll need some background on servo motors. 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, servo motors are generally designed to move at angles within a limited set. In other words, in the DC motor world, you generally want your motors to spin at a continuous rotation speed that you control. In the servo world, you want to control the movement of your motor to a specific position. For more information on how servos work, visit <http://www.seattlerobotics.org/guide/servos.html> or http://www.societyofrobots.com/actuators_servos.shtml.

Connecting the hardware

To make your project walk, you first need to connect the servo motor controller to the servos. There are two connections you need to make: the first is to the servo motors and the second is to the battery holder. In this section, you'll connect your servo controller to your PC or Linux machine to check to see whether or not everything is working. The steps for that are as follows:

1. Connect the servos to the controller. The following is an image of your two-legged robot and the four different servo connections:



- In order to be consistent, let's connect your four servos to the connections marked **0** through **3** on the controller using the following configurations:
 - 0: Left foot
 - 1: Left hip
 - 2: Right foot
 - 3: Right hip

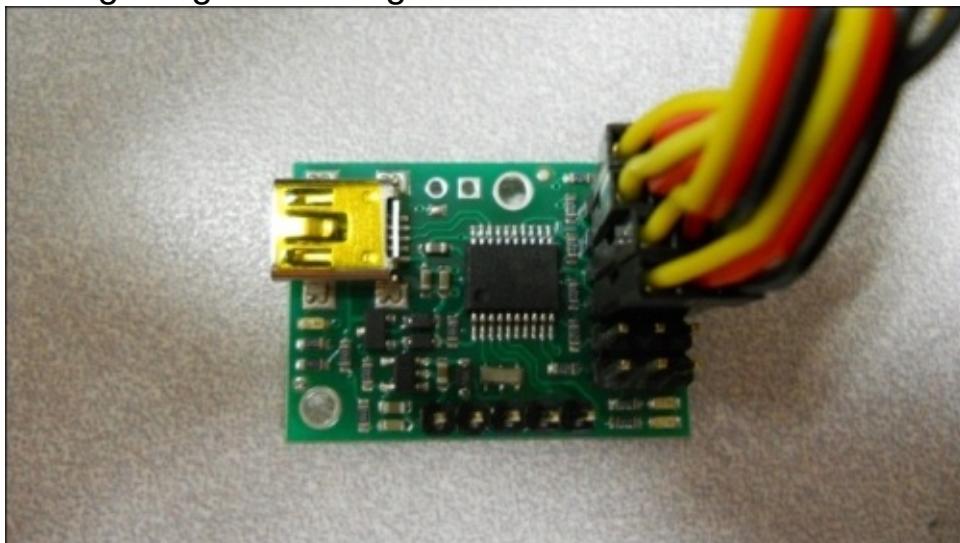
The following is an image of the back of the controller; it will show you where to connect your servos:



- Connect these servos to the servo motor controller as follows:

- The left foot to the 0 to the top connector, the black cable to the outside (-)
- The left hip to the 1 connector, the black cable out
- The right foot to the 2 connector, the black cable out
- The right hip to the 3 connector, the black cable out

See the following image indicating how to connect servos to the



controller:

- Now you need to connect the servo motor controller to your battery. You'll use the USB to FTDI UART cable; plug the red and black cables into the power connector on the servo controller, as shown in the

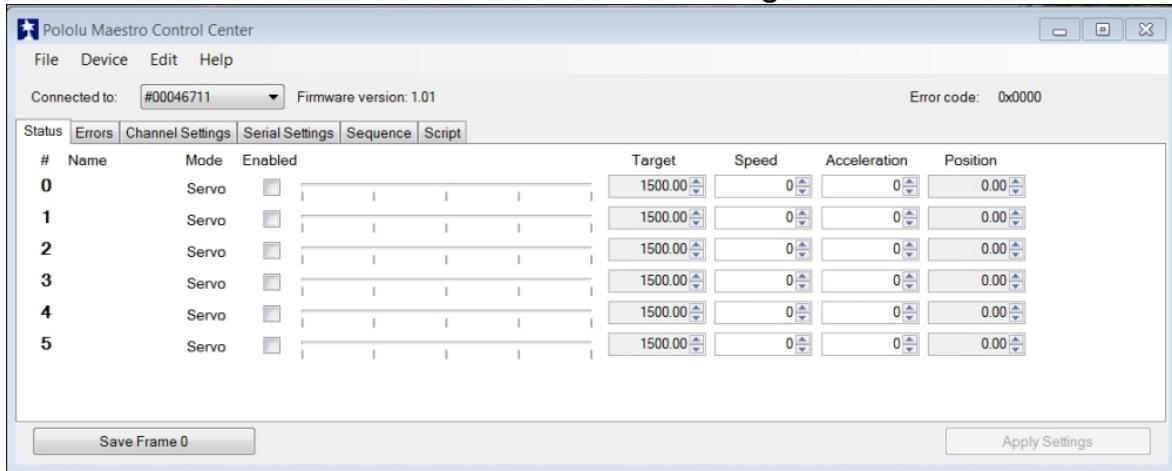


following image:

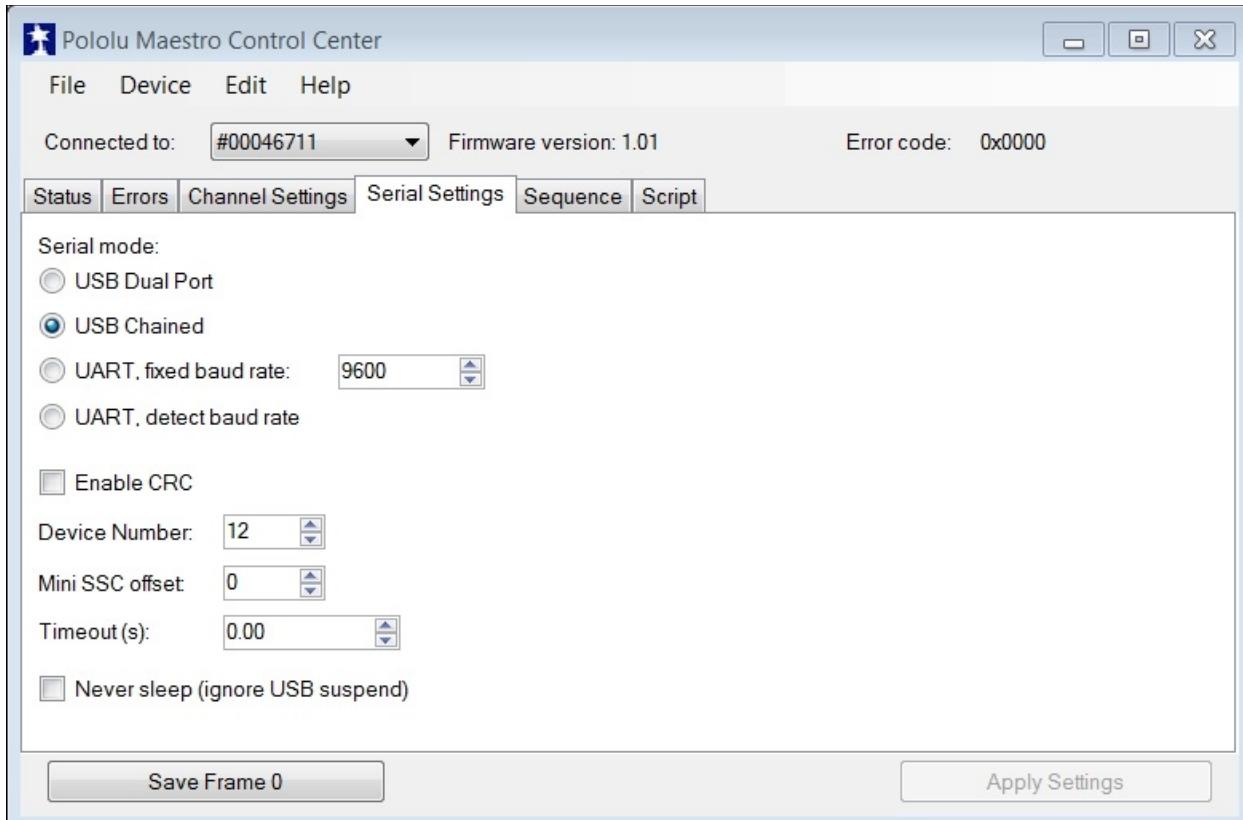
Configuring the software

Now you can connect the motor controller to your PC or Linux machine to see whether or not you can talk to it. Once the hardware is connected, you can use some of the software provided by Polulu to control the servos. It is easiest to do this using your personal computer or Linux machine. The steps to do so are as follows:

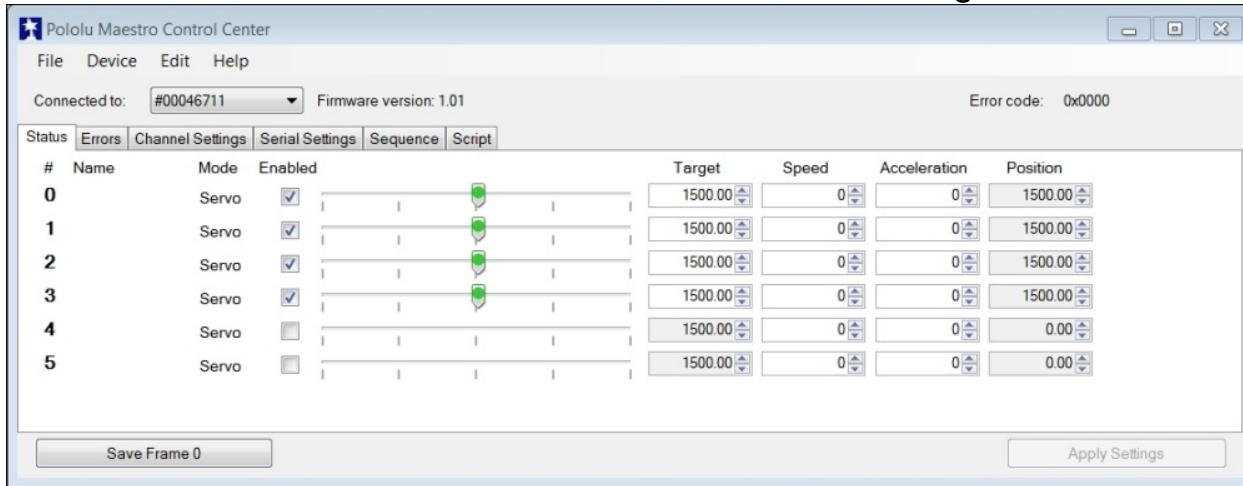
1. Download the Polulu software from <http://www.pololu.com/docs/0J40/3.a> and install it based on the instructions on the website. Once it is installed, run the software; you should see the window shown in the following screenshot:



- You will first need to change the **Serial mode** configuration in **Serial Settings**, so select the **Serial Settings** tab; you should see the window shown in the following screenshot:

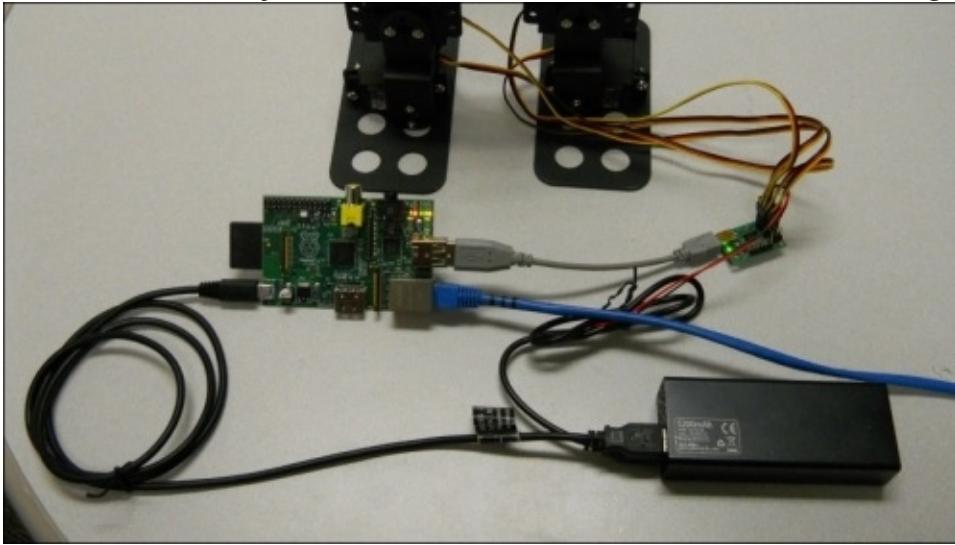


- Make sure that **USB Chained** is selected; this will allow you to connect to and control the motor controller over the USB. Now go back to the main screen by selecting the **Status** tab; now you can turn on the four servos. The screen should look as shown in the following screenshot:



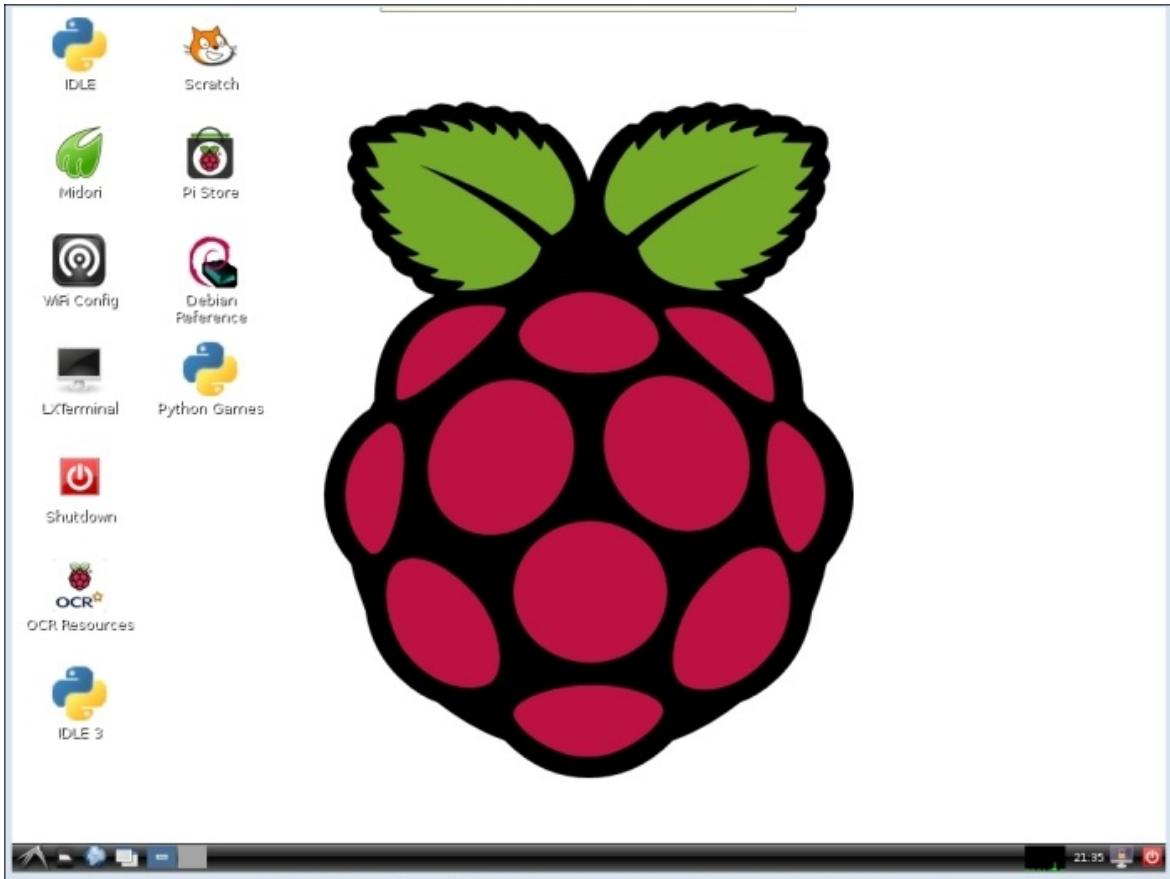
- Now you can use the sliders to control the servos. Enable the four servos and make sure that the servo 0 moves the left foot, 1 the left hip, 2 the right foot, and 3 the right hip.
- You've checked the motor controllers and the servos and you'll now connect the motor controller to Raspberry Pi to control the servos from

there. Remove the USB cable from the PC and connect it to Raspberry Pi. The entire system will look as shown in the following image:

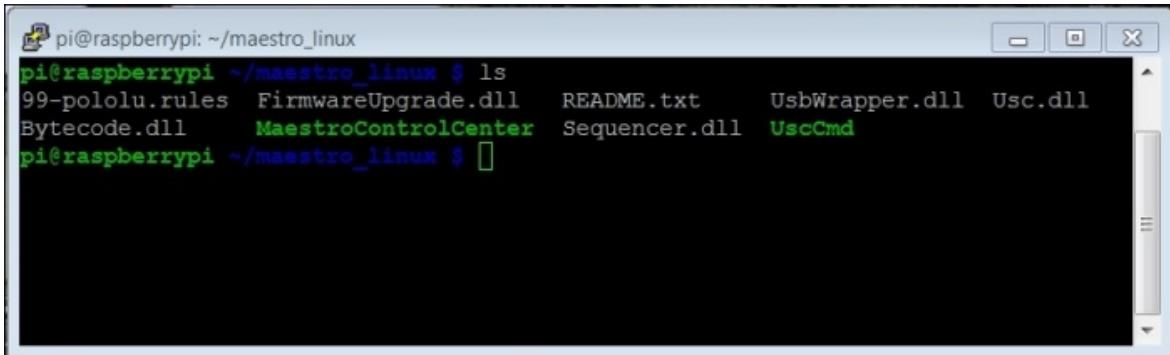


Let's now talk to the motor controller by downloading the Linux code from Pololu at <http://www.pololu.com/docs/0J40/3.b>. Perhaps the best way to do this is by logging on to Raspberry Pi using vncserver and opening a **VNC Viewer** window on your PC. To do this, log in to your Raspberry Pi using PuTTY and then type `vncserver` at the prompt to make sure vncserver is running. Then, perform the following steps:

1. On your PC, open the **VNC Viewer** application, enter your IP address, and then click on **Connect**. Then, enter the password that you created for the vncserver; you should see the Raspberry Pi viewer screen, which should look as shown in the following screenshot:

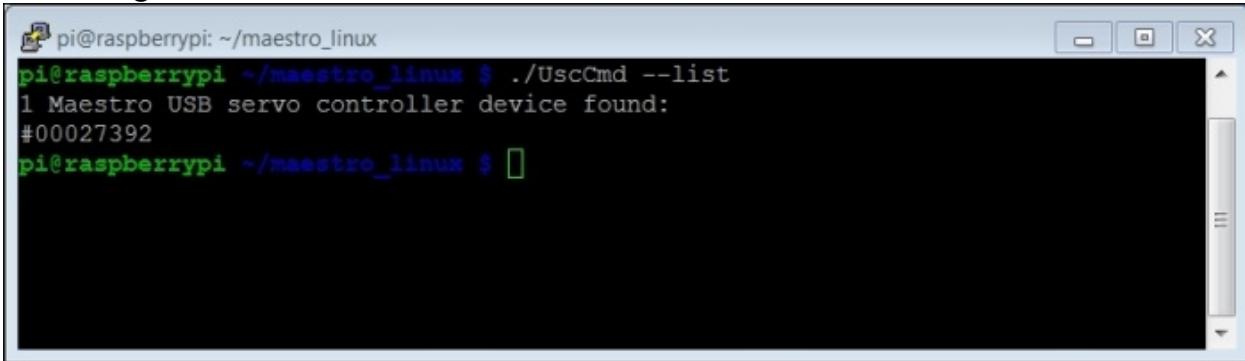


2. Open a Firefox browser window and go to <http://www.pololu.com/docs/0J40/3.b>. Click on the **Maestro Servo Controller Linux Software** link. You will need to download the file `maestro_linux_100507.tar.gz` to the `Download` directory. You can also use wget to get this software by typing `wget http://www.pololu.com/file/download/maestro-linux-100507.tar.gz?file_id=0J315` in a terminal window.
3. Go to your `Download` directory, move it to your home directory by typing `mv maestro_linux_100507.tar.gz ..` and then you can go back to your home directory.
4. Unpack the file by typing `tar -xzf maestro_linux_011507.tar.gz`. This will create a directory called `maestro_linux`. Go to that directory by typing `cd maestro_linux` and then type `ls`. You should see the output as shown in the following screenshot:



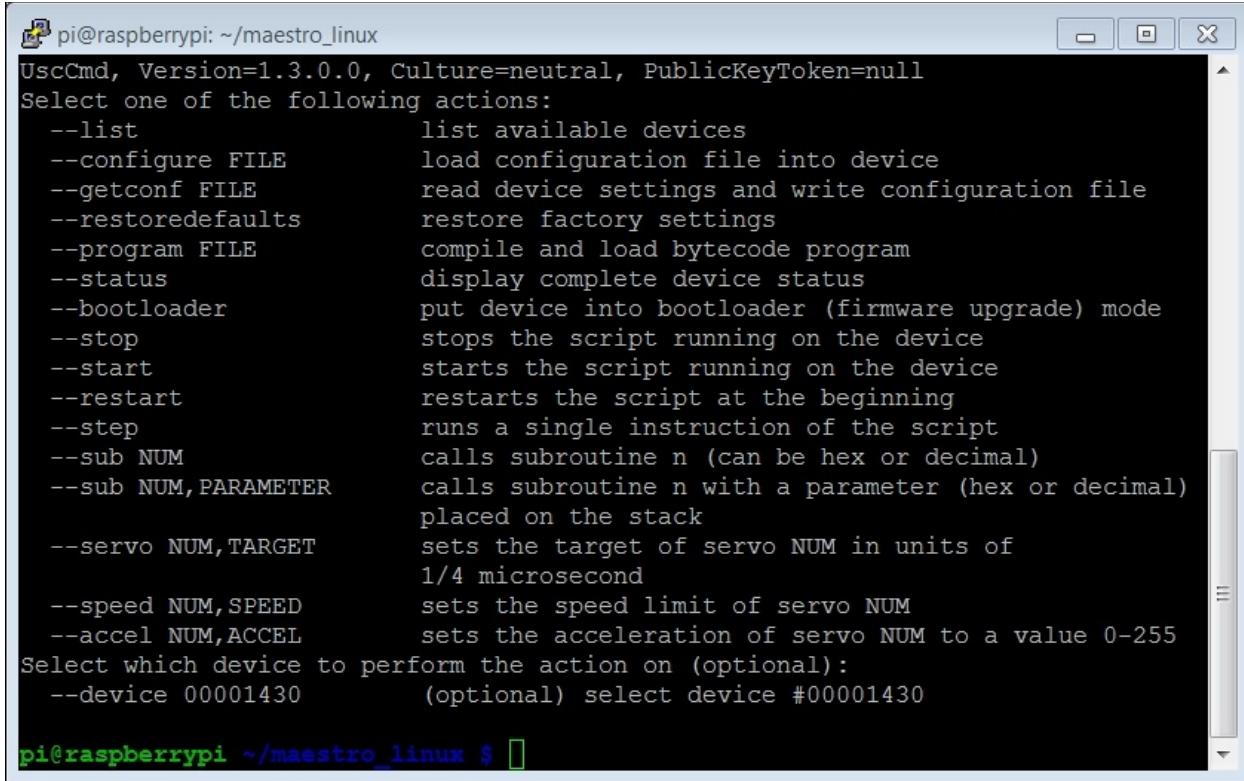
```
pi@raspberrypi: ~/maestro_linux
pi@raspberrypi ~/maestro_linux $ ls
99-pololu.rules  FirmwareUpgrade.dll  README.txt  UsbWrapper.dll  Usc.dll
Bytecode.dll     MaestroControlCenter  Sequencer.dll  UscCmd
pi@raspberrypi ~/maestro_linux $
```

The document [README.txt](#) will give you explicit instructions on how to install the software. Unfortunately, you can't run **MaestroControlCenter** on your Raspberry Pi. Our version of windowing 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 screenshot:



```
pi@raspberrypi: ~/maestro_linux
pi@raspberrypi ~/maestro_linux $ ./UscCmd --list
1 Maestro USB servo controller device found:
#00027392
pi@raspberrypi ~/maestro_linux $
```

The unit sees your servo controller. If you just type [./UscCmd](#), you can see all the commands you could send to your controller. When you run this command, you can see the result as shown in the following screenshot:



```
pi@raspberrypi: ~/maestro_linux
UscCmd, Version=1.3.0.0, Culture=neutral, PublicKeyToken=null
Select one of the following actions:
--list                      list available devices
--configure FILE            load configuration file into device
--getconf FILE              read device settings and write configuration file
--restoredefaults           restore factory settings
--program FILE              compile and load bytecode program
--status                     display complete device status
--bootloader                put device into bootloader (firmware upgrade) mode
--stop                       stops the script running on the device
--start                      starts the script running on the device
--restart                    restarts the script at the beginning
--step                       runs a single instruction of the script
--sub NUM                   calls subroutine n (can be hex or decimal)
--sub NUM, PARAMETER         calls subroutine n with a parameter (hex or decimal)
                           placed on the stack
--servo NUM, TARGET          sets the target of servo NUM in units of
                           1/4 microsecond
--speed NUM, SPEED           sets the speed limit of servo NUM
--accel NUM, ACCEL           sets the acceleration of servo NUM to a value 0-255
Select which device to perform the action on (optional):
--device 00001430           (optional) select device #00001430

pi@raspberrypi ~/maestro_linux $
```

Notice that you can send a servo a specific target angle, although if the target angle is not within range, it makes it a bit difficult to know where you are sending your servo. Try typing `./UscCmd --servo 0, 10`. The servo will most likely 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 software that will translate your angles to the electronic signals that will move the servos.

If you haven't run the Maestro Controller tool and set the **Serial Settings** setting to **USB Chained**, your motor controller may not respond.

Creating a program in Linux to control the mobile platform

Now that you can control your servos using a basic command-line program, let's control them by programming some movement in Python. In this section, you'll create a Python program that will let you talk to your servos a bit more intuitively. You'll issue commands that tell a servo to go to a specific angle and it will go to that angle. You can then add a set of such commands to allow your legged mobile robot to lean left or right or even take a step forward.

Let's start with a simple program that will make your legged mobile robot's servos turn at 90 degrees; this should be somewhere close to the middle of the 180-degree range you can work within. However, the center, maximum, and minimum values can vary from one servo to another, so you may need to calibrate them. To keep things simple, we will not cover that here. The following screenshot shows the code required for turning the servos:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/maestro_linux". The window contains Python code for controlling a Pololu Micro Maestro servo controller via serial communication. The code defines a class `PololuMicroMaestro` with methods for setting servo angles and closing the connection. It also includes a main block that creates an instance of the class and sets angles for four servos. The terminal window has a status bar at the bottom indicating it's running on Python and providing Emacs key information.

```
pi@raspberrypi: ~/maestro_linux
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time

class PololuMicroMaestro(object):
    def __init__(self, port= "/dev/ttyACM0"):
        self.ser = serial.Serial(port = port)
    def setAngle(self, channel, angle):
        minAngle = 0.0
        maxAngle = 180.0
        minTarget = 256.0
        maxTarget = 13120.0
        scaledValue = int((angle / ((maxAngle - minAngle) / (maxTarget - minTarget))) + minTarget)
        commandByte = chr(0x84)
        channelByte = chr(channel)
        lowTargetByte = chr(scaledValue & 0x7F)
        highTargetByte = chr((scaledValue >> 7) & 0x7F)
        command = commandByte + channelByte + lowTargetByte + highTargetByte
        self.ser.write(command)
        self.ser.flush()
    def close(self):
        self.ser.close()
if __name__ == "__main__":
    robot = PololuMicroMaestro()
    robot.setAngle(0,80)
    robot.setAngle(1,80)
    robot.setAngle(2,80)
    robot.setAngle(3,75)

-UU-----F1  robot.py      All L1      (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

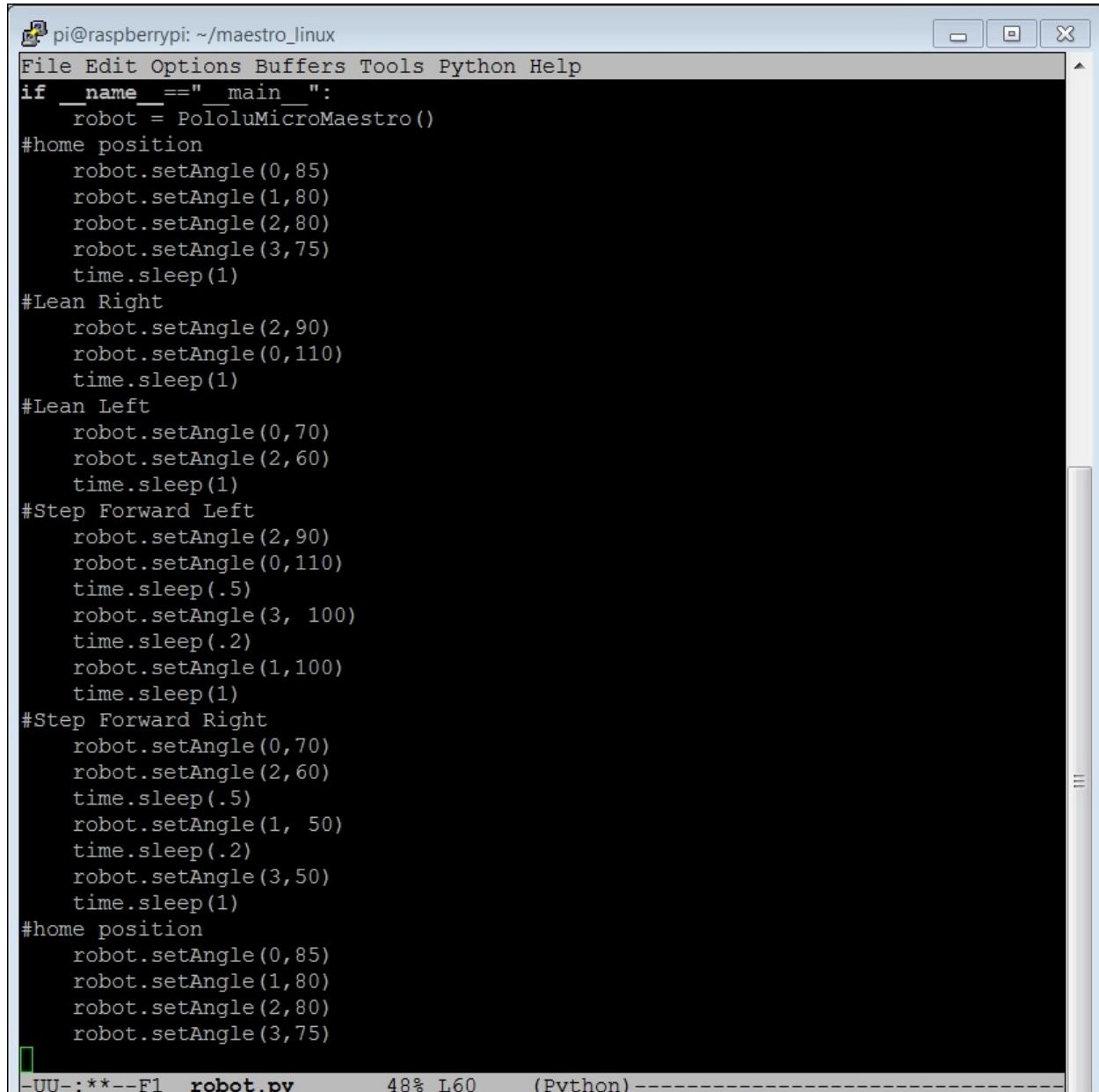
The following is an explanation of the code:

- The `#!/user/bin/python` line allows you to make this Python file available for execution from the command line. It will allow you to call this program from your voice command program. We'll talk about that in the next section.
- The `import serial` and `import time` lines include a `serial` and `time` library. You need the `serial` library to talk to your unit via USB. If you have not installed this library, type `sudo apt-get install python-serial`. You will use the `time` library later to wait between servo commands.
- The class `PololuMicroMaestro` holds the methods that will allow you to communicate with your motor controller.
- The first method, the `__init__` method, opens the USB port

associated with your servo motor controller.

- The next method, `setAngle`, converts your desired settings for the servo and angle to the serial command that the servo motor controller needs. The values, such as `minTarget` and `maxTarget`, and the structure of the communications—`channelByte`, `commandByte`, `lowTargetByte`, and `highTargetByte`—come from the manufacturer.
- The last method, `close`, closes the serial port.
- Now that you have the class, the `__main__` statement of the program instantiates an instance of your servo motor controller class so that you can call it.
- Now you can set each servo to the desired position. The default would be to set each servo to 90 degrees. However, the servos were exactly centered, so I found that I needed to set the angle of each servo such that my robot has both feet on the ground and both hips centered.

Once you have the basic home position set, you can ask your robot to do some things; the following screenshot shows some examples in simple Python code:



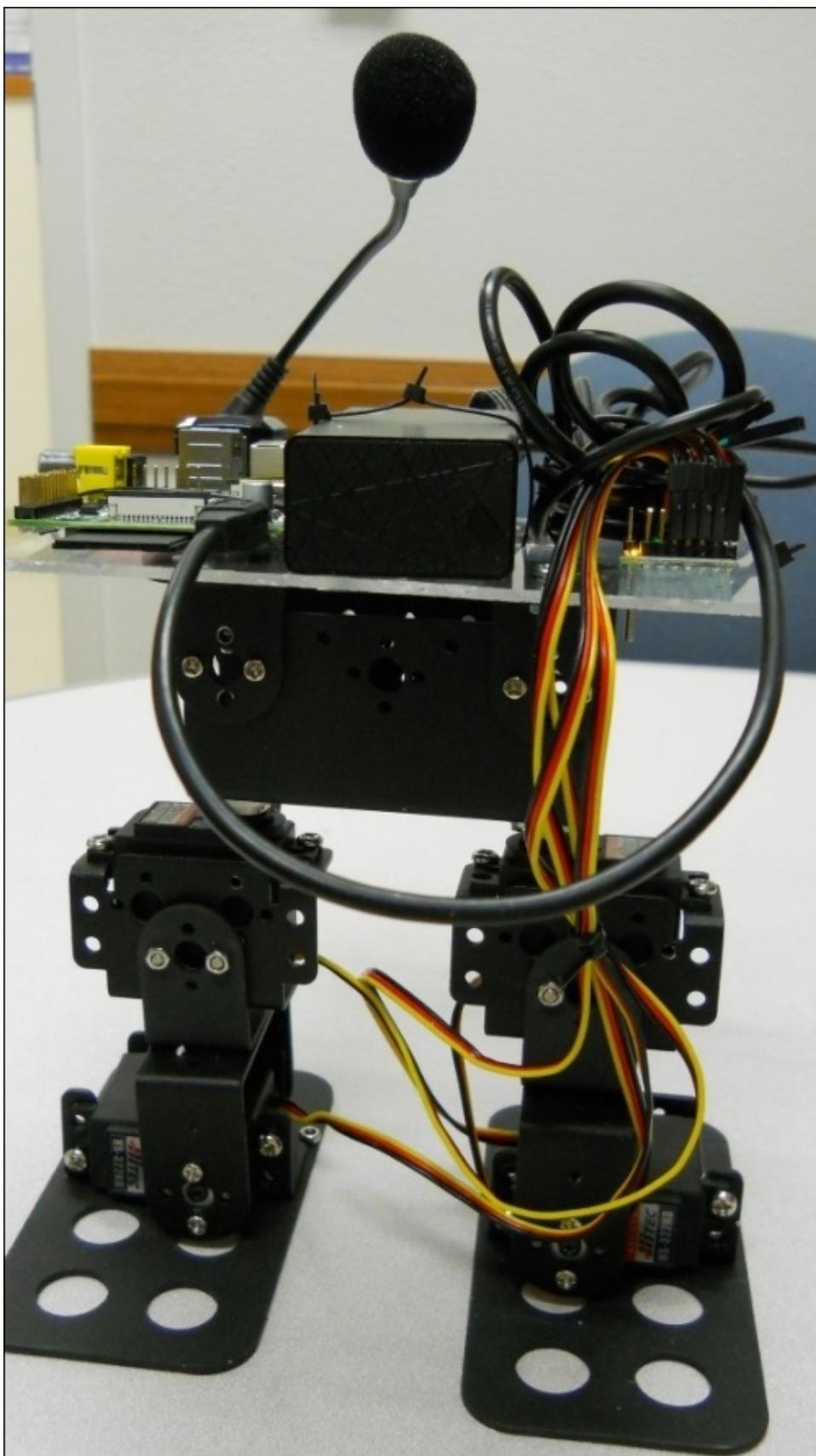
A screenshot of a terminal window titled "pi@raspberrypi: ~/maestro_linux". The window contains Python code for a robot. The code includes comments for various movements: "home position", "Lean Right", "Lean Left", "Step Forward Left", "Step Forward Right", and another "home position". It uses the `PololuMicroMaestro()` class to set servos at specific angles and includes `time.sleep()` statements for timing.

```
pi@raspberrypi: ~/maestro_linux
File Edit Options Buffers Tools Python Help
if __name__ == "__main__":
    robot = PololuMicroMaestro()
#home position
    robot.setAngle(0,85)
    robot.setAngle(1,80)
    robot.setAngle(2,80)
    robot.setAngle(3,75)
    time.sleep(1)
#Lean Right
    robot.setAngle(2,90)
    robot.setAngle(0,110)
    time.sleep(1)
#Lean Left
    robot.setAngle(0,70)
    robot.setAngle(2,60)
    time.sleep(1)
#Step Forward Left
    robot.setAngle(2,90)
    robot.setAngle(0,110)
    time.sleep(.5)
    robot.setAngle(3, 100)
    time.sleep(.2)
    robot.setAngle(1,100)
    time.sleep(1)
#Step Forward Right
    robot.setAngle(0,70)
    robot.setAngle(2,60)
    time.sleep(.5)
    robot.setAngle(1, 50)
    time.sleep(.2)
    robot.setAngle(3,50)
    time.sleep(1)
#home position
    robot.setAngle(0,85)
    robot.setAngle(1,80)
    robot.setAngle(2,80)
    robot.setAngle(3,75)

UU-:***-F1  robot.py  48% L60  (Python)
```

In this case, you are using your `setAngle` command to set your servos to manipulate your robot. This set of commands first sets your robot to the home position. Then, you can use the feet to lean to the right and then to the left, and then you can use a combination of commands to make your robot step forward with the left and then the right foot.

Once you have the program working, you'll want to package all your hardware onto the mobile robot. There is no right or wrong way to do this, but I like to use a small piece of transparent plastic because it is easy to cut and drill. The following is an image of my robot:



By following these principles, you can make your robot do many amazing things, such as walk forward and backward, dance, and turn around—any number of movements are possible. The best way to learn these movements is to try positioning the servos in new and different ways.

Making your mobile platform truly mobile by issuing voice commands

Now that your robot can move, wouldn't it be neat to have it obey your commands?

You should now have a mobile platform that you can program to move in any number of ways. Unfortunately, you still have your LAN cable connected, so the platform isn't completely mobile. Once you have started executing the program, you can't alter its behavior. In this section, you will use the principles from [Chapter 2, Programming Raspberry Pi](#), to issue voice commands to initiate movement.

You'll need to modify your voice recognition program so that it will run your Python program when it gets a voice command. If you feel your knowledge of how this works is rusty, review [Chapter 2, Programming Raspberry Pi](#). You are going to make a simple modification to the `continuous.c` program in `homeubuntu/pocketsphinx-0.8/src/`. To do this, type `cd homeubuntu/ pocketsphinx-0.8/src/programs` and then type `emacs continuous.c`. The changes will appear in the same section as your other voice commands and will look as shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains C code for a program named "continuous.c". The code includes logic for handling voice commands like "GOODBYE", "hello", and "forward". A cursor is visible near the end of the "forward" command block. The terminal also shows the file status "-UU-----F1 continuous.c 80% L339 (C/l Abbrev)" and the command "Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c".

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
File Edit Options Buffers Tools C Help
    sscanf(hyp, "%s", word);
    if (strcmp(hyp, "GOODBYE") == 0)
    {
        system("espeak \"good bye\"");
        break;
    }
    else if (strcmp(hyp, "hello") == 0)
    {
        system("espeak \"hello\"");
    }
    else if (strcmp(hyp, "forward"))
    {
        system("espeak \"moving robot\"");
        system("home/pi/maestro_linux/robot.py");
    }
}

/* Resume A/D recording for next utterance */
if (ad_start_rec(ad) < 0)
    E_FATAL("Failed to start recording\n");
}
-UU-----F1 continuous.c 80% L339 (C/l Abbrev)
Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c
```

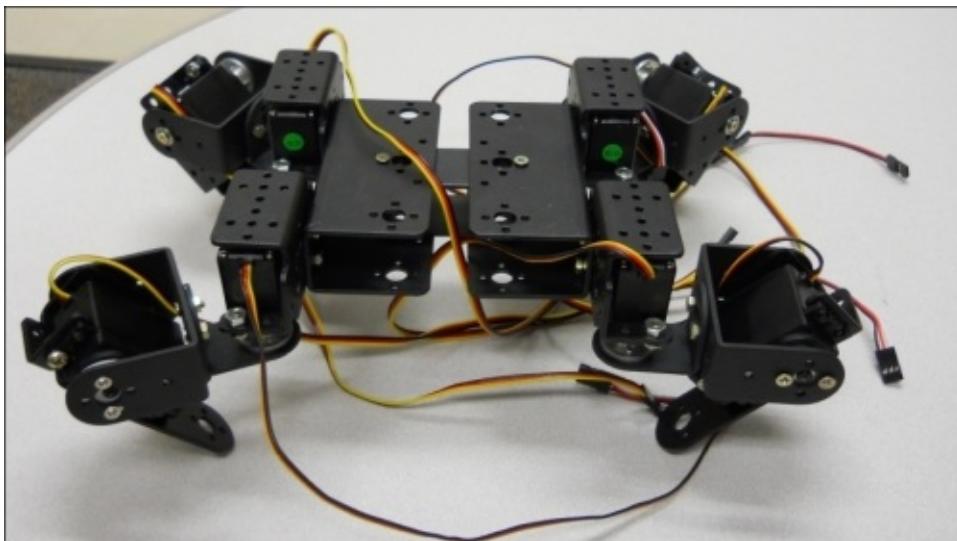
The additions are pretty straightforward. Let's walk through them.

- `else if (strcmp(hyp, "FORWARD") == 0)` checks the input word as recognized by your voice command program. If it corresponds with the word `FORWARD`, you will execute everything within the `if` statement. You use `{` and `}` to tell the system which commands go with this `else if` clause.
- `system("espeak \"moving robot\")")` executes Espeak, which should tell you that you are about to run your robot program.
- `system("home/ubuntu/maestro_linux/robot.py")` indicates the name of the program you will execute. In this case, your mobile platform will do whatever the program `robot.py` tells it to.

After doing this, you will need to recompile the program, so type `make` and the executable `pocketsphinx_continuous` will be created. Run the program by typing `./pocketsphinx_continuous`. Disconnect the LAN cable and the mobile platform will now take the forward voice command and execute your program. You should now have a complete mobile platform! When you execute your program, the mobile platform can now move around based on what you have programmed it to do.

You don't have to put all of your capabilities in one program. You can create several programs, each with a different function, and then connect each of the programs to their appropriate voice commands. Perhaps one command can move your robot forward, a different one can move it backwards, and another can turn it right or left.

Congratulations! Your robot should now be able to move around in any way you program it to move. You can even have the robot dance. You have now built a two-legged robot and you can easily expand on this knowledge to create robots with even more legs. The following is an image of the mechanical structure of a four-legged robot that has eight DOF and is fairly easy to create using many of the parts you have used to create your two-legged robot; this is my personal favorite because it doesn't fall over and break the electronics:



You'll need eight servos and lots of batteries. If you search eBay, you can often find kits for sale for four-legged robots with 12 DOF, but realize that the battery will need to be much bigger. For these kinds of applications, we often use RC (remote control) batteries. These are nice as they are rechargeable, but make sure you either purchase one that is 5 volts to 6 volts or include a way to regulate the voltage. The following is an image of such a battery, available at most hobby stores:



If you use this type of battery, don't forget its charger. The hobby store can help with choosing an appropriate match.

Summary

Now you have the capability of building not only wheeled robots, but also robots with legs. It is also easy to expand this capability to robots with arms: controlling the servos for an arm is the same as controlling them for legs. In later chapters, we can even use this capability to control the position of our sonar sensors or webcams. In the next chapter, we'll cover how to connect a sonar sensor and avoid or find obstacles.

Chapter 7. Avoiding Obstacles Using Sensors

In the previous two chapters, we covered wheeled and tracked movements and then legged movement. Now your robot can move around. But what if we want the robot to sense the outside world so that it doesn't run into things? In this chapter, we'll discover how to add some sensors to help us avoid colliding into barriers.

Your robot will take quite a beating if it continually runs into walls or runs off the edge of a surface. Let's help your robot avoid these so that it looks intelligent.

In this chapter, we will cover the following topics:

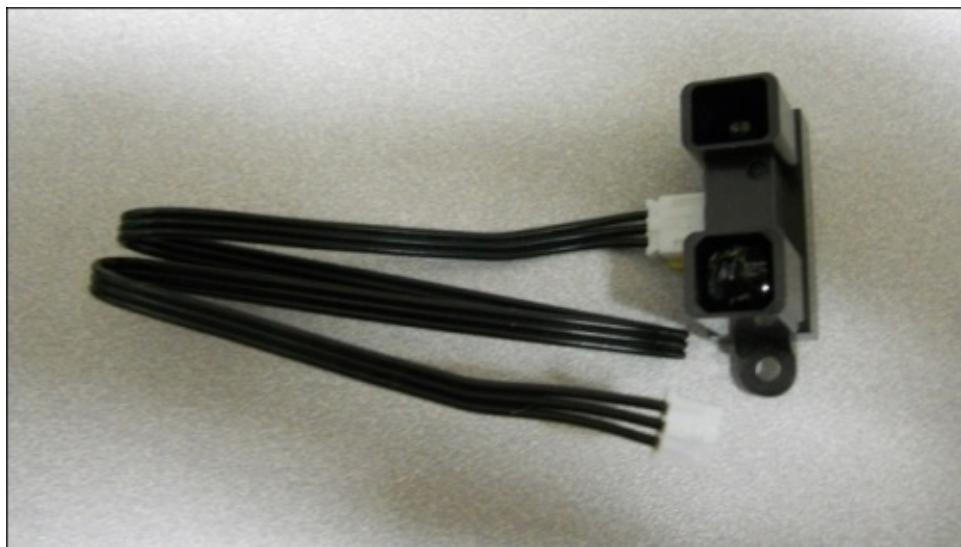
- Gathering the hardware
- Connecting Raspberry Pi to an **IR (infrared)** sensor in order to detect the world
- Connecting Raspberry Pi to a USB sonar sensor in order to detect the world
- Using a servo to change the position of your sensor so that a single sensor can view a large field, eliminating the need for additional sensors

Gathering the hardware

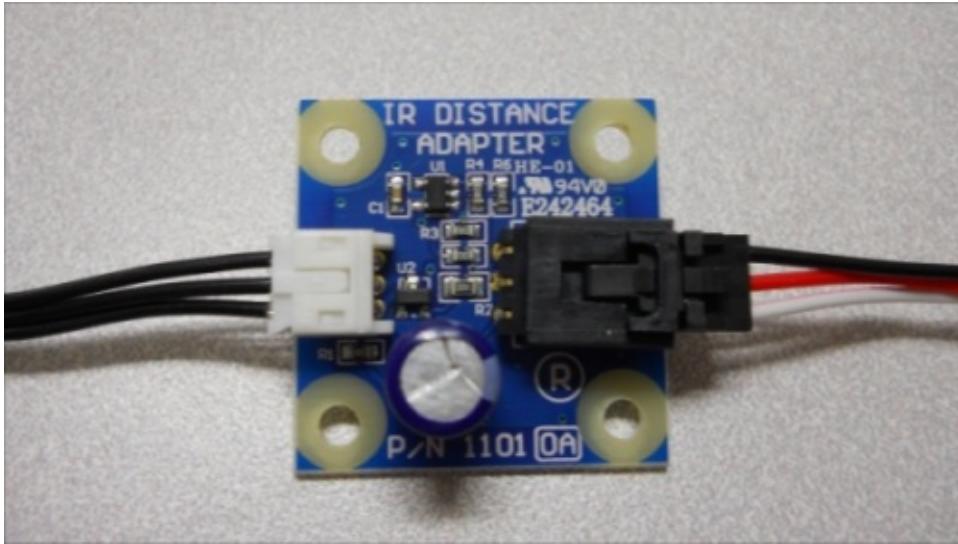
In this chapter, you'll need some sensors. I am going to show you how to interface with an infrared sensor and a sonar sensor. You'll probably want to choose one of the two. It is not an easy choice; both will do an adequate job. If you're not sure which sensor to use, I suggest you read through this chapter first and then choose the one you think will work in your specific application.

If you are going to choose the infrared sensor, I would advise you to use a set of parts that include not only the sensor but also a way to communicate with the sensor using USB. Raspberry Pi can use its GPIO

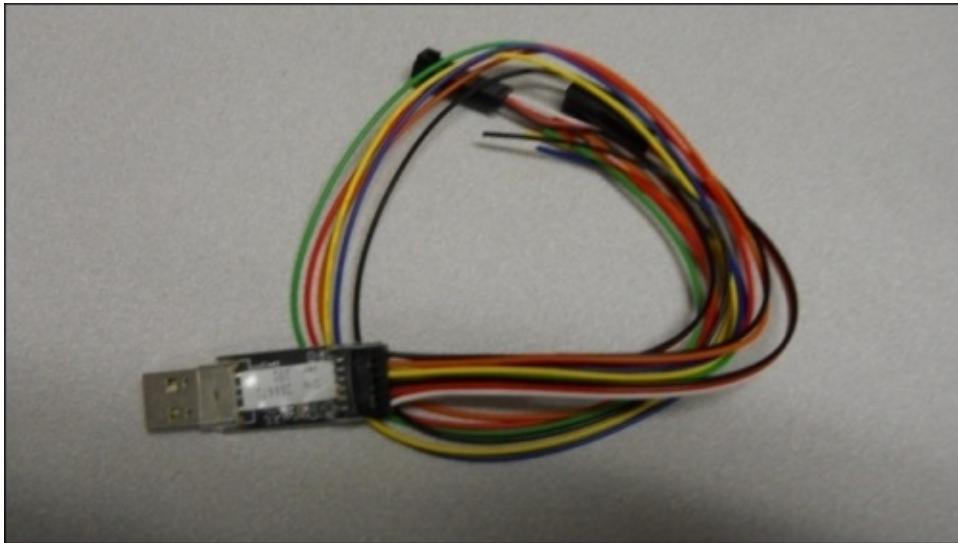
pins to communicate with an infrared sensor. See <http://blogs.arcsoftwareconsultancy.com/pi/2013/10/03/halloween/> and <http://www.raspberrypi-spy.co.uk/2013/01/cheap-pir-sensors-and-the-raspberry-pi-part-1/> for tutorials on sensors. However, I like the ease of use and ease of connection that an USB infrared set can provide. You'll need to purchase three parts, all from www.phidgets.com. The first part is the sensor itself. This sensor is made by Sharp and it comes in several different distance specifications. The following is an image of the 20-150 cm version:



The second part that you'll need is an IR distance adapter, which can provide the signals to the sensor and condition the signals coming back from the sensor. The model number is **1101_0 - IR Distance Adapter**. The following is an image of it:



The third part is an USB interface, also offered by 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 the following is an image of it:

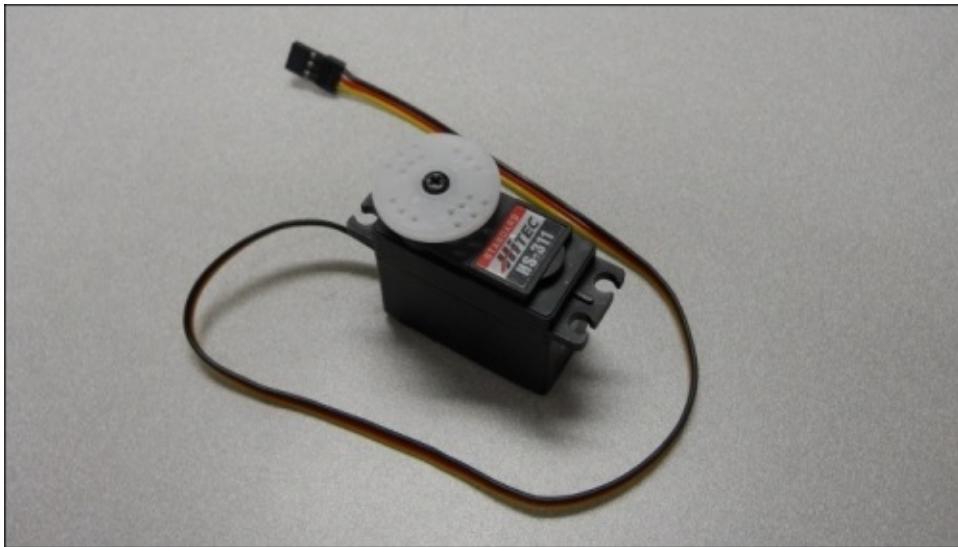


If you are going to go with the sonar sensor, the following is an image of the USB sonar sensor I like to use in my projects:

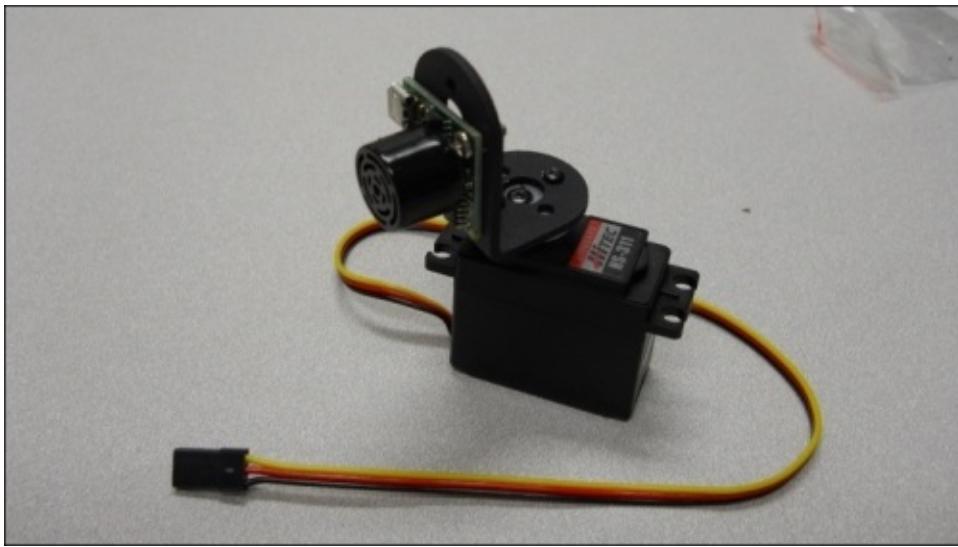


This is the **USB-ProxSonar-EZ** sensor, which can be purchased directly from MaxBotix or on Amazon. There are several models, each with a different distance specification; however, they all work in the same way.

Irrespective of whether you choose an infrared sensor or a sonar sensor, you may want to detect distance in more than just one direction. You have two choices. The first choice is to simply use a number of sensors, one in each direction. But, in the *Using a servo to move a single sensor* section of this chapter, you will learn how to use a servo to rotate the sensor. This way, you can use a single sensor and just turn it to a new direction. To complete configuring the hardware, you'll need a servo and a way to mount it on your project. Again, I like the Hitec series of servos and this is ready-made for an HS-311 servo, which should look as shown in the following image:

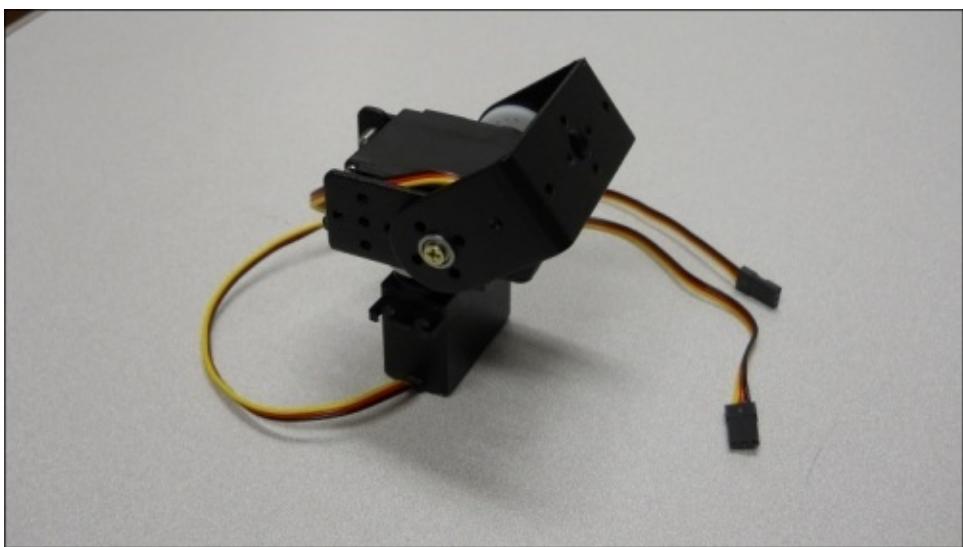


Here is a way to mount the sensor on a 90-degree angle bracket. I used one from a robot kit I purchased on eBay. It can connect to the servo as shown in the following image:



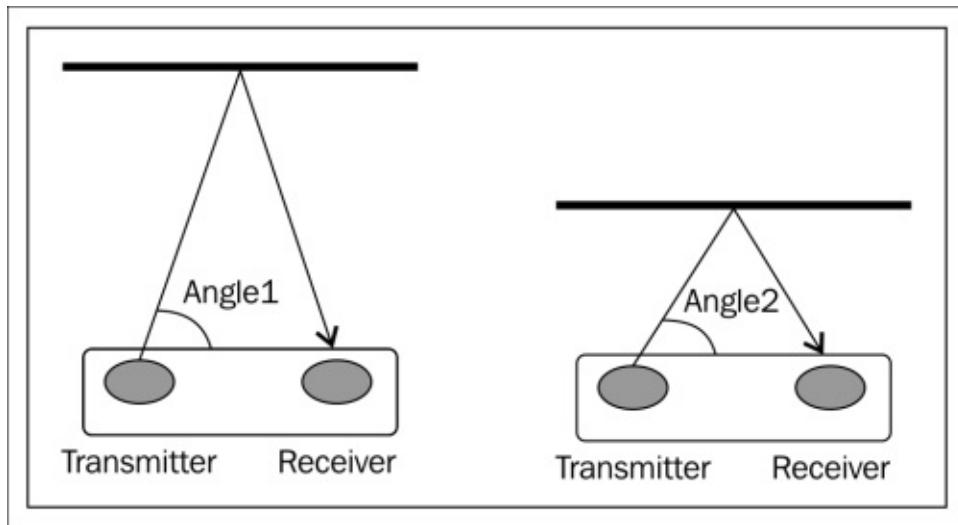
However, if you want to get really fancy, you can purchase a **Pan and Tilt assembly**. This contains two servos and allows you to rotate your sensor in both the vertical and horizontal axes. The assembly is available on online stores such as www.robotshop.com. You can also construct a Pan and Tilt assembly out of the components that you may have if you have purchased a legged robot kit.

The finished product with two servos looks as shown in the following image:



Connecting Raspberry Pi to an infrared sensor

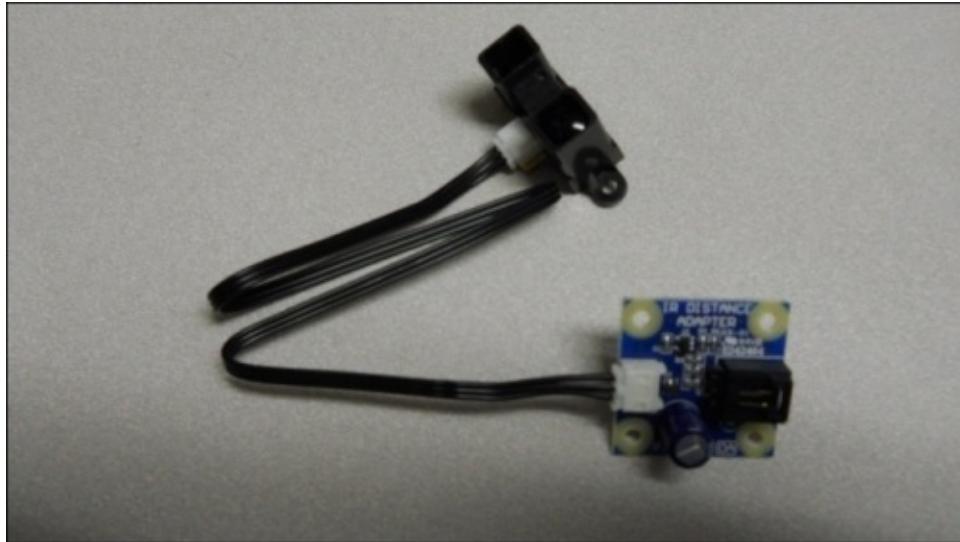
The ability of the robot to move around is impressive, but it won't be if your robot keeps running into barriers. To avoid this, you'll want to be able to sense a barrier or a target. One of the ways to do this is by using an IR sensor. Now, a little tutorial on IR sensors. The sensor you are using has both a transmitter and a receiver. 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 at the sensor, as shown in the following diagram:



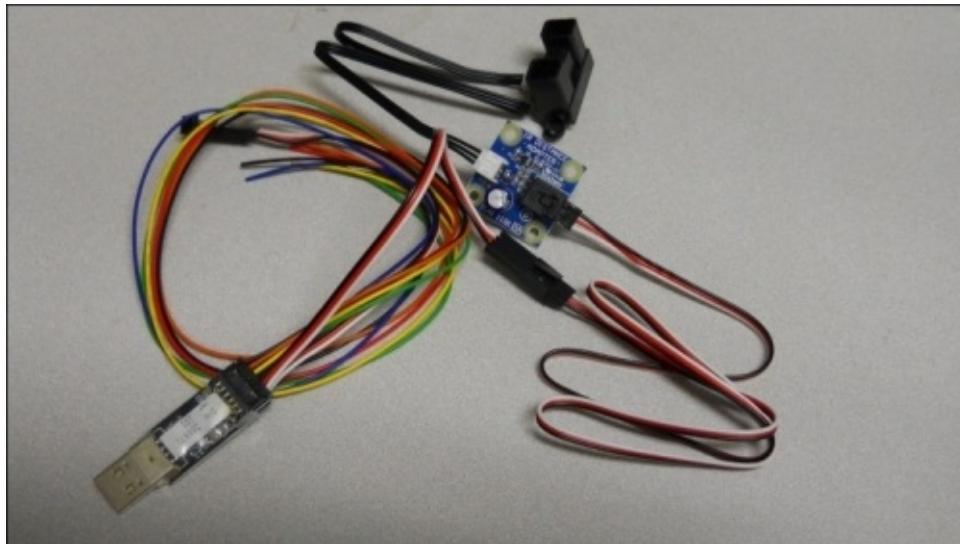
The different angles give you an indication of the distance from the object. Unfortunately, the relationship between the output of the sensor and the distance is not linear, so you'll need to do some calibration to predict the actual distance and its relationship to the output of the sensor.

Now, let's connect the sensor. The following are the steps to do so:

1. Connect the Sharp IR sensor to the IR Distance Adapter. Simply take the output of the sensor and insert it into the white connector on the adapter board as shown in the following image:



2. Now, you need to connect the IR Distance Adapter to the USB interface board. Connect the board using two cables, as shown in the following image (fortunately, there is only one way to connect them):

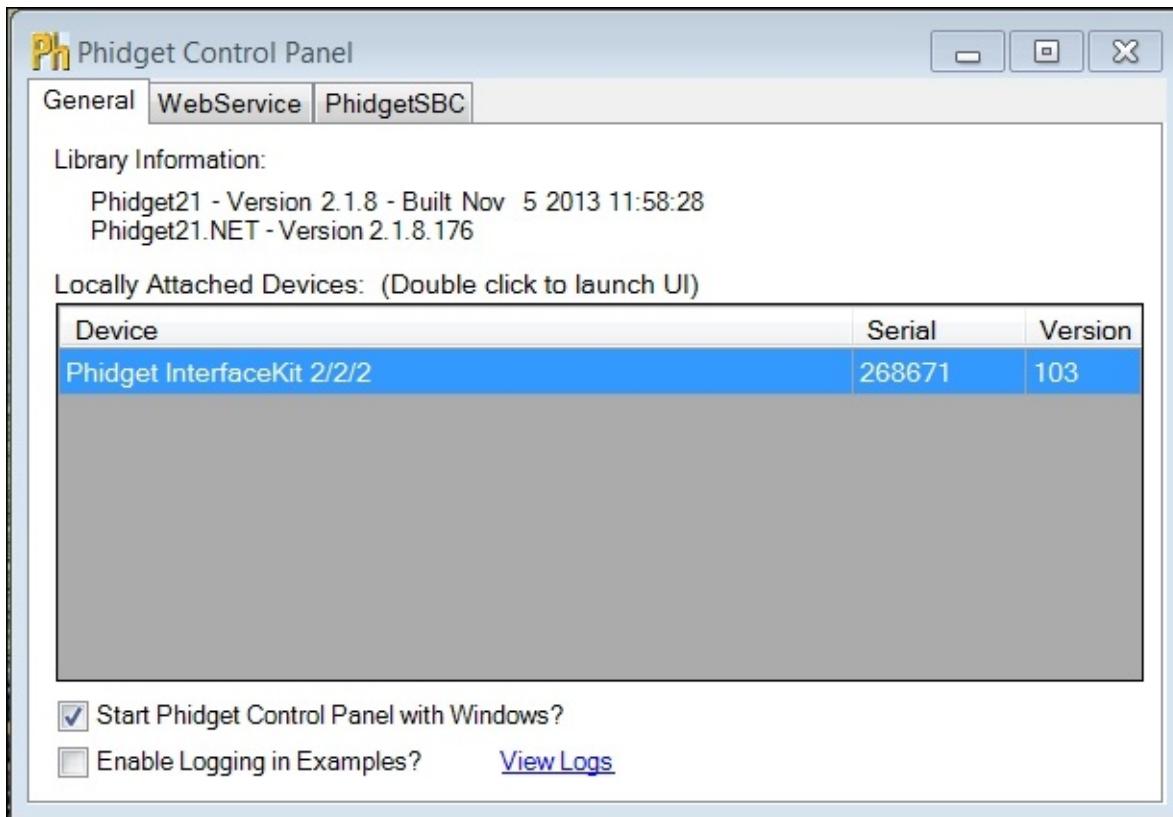


Now that everything is connected, you can begin to access the data from Raspberry Pi. To do this, perform the following steps:

1. The first step is to look at the data from a PC. For this, you'll first need to download some software from http://www.phidgets.com/docs/OS_-_Windows#Quick_Downloads. Go to this website and then to the *Getting Started with Windows* section.

2. After you have downloaded the software, run the downloaded installation software, and it will install the **Phidgets Control Panel** application in the **Phidgets** folder.
3. You can then run the software from the **Start** menu by selecting the **Phidgets** folder and then the **Phidgets Control Panel** application.

When you run the software with your Phidgets USB device plugged in, you should see the following screenshot:

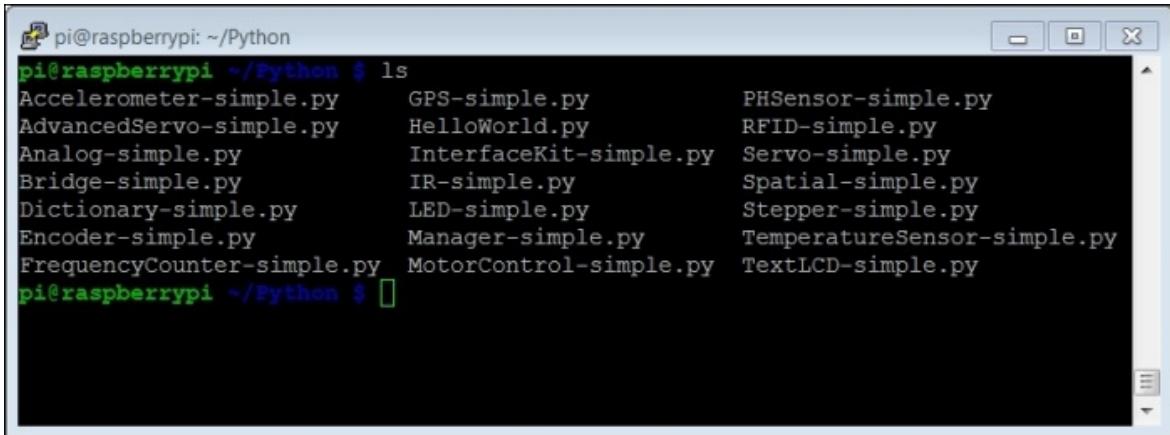


Note that your phidget device is being recognized by the system. Now you can move the device to Raspberry Pi. Plug the Phidgets device into Raspberry Pi. Then, follow the directions available at http://www.phidgets.com/docs/OS_-_Linux#Installing.

1. Download the Phidget Libraries from the website. You can either install these directly in your Raspberry Pi, or you can use WinSCP to first download them to your PC and then transfer them to Raspberry Pi or use wget to transfer them by typing `wget http://www.phidgets.com/downloads/libraries/libphidget.tar.gz`. In either case, you'll need to unzip them so they are now in your

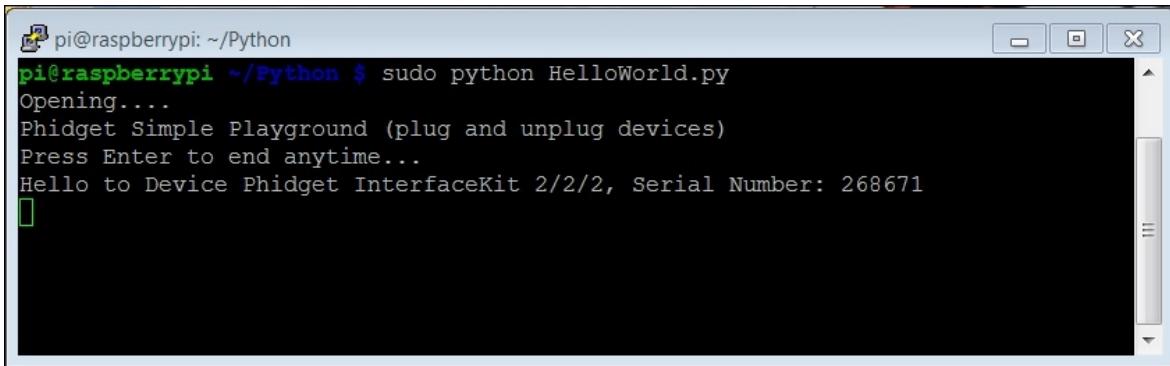
Raspberry Pi's home directory. You can use the `tar -xzvf` command to accomplish this.

2. Once you have unzipped the files, go to the directory created by the `tar -xzfv` command. Type `./configure`, then `make`, and then `sudo make install`.
3. The next step is to install the Python modules. Install these from http://www.phidgets.com/docs/Language_-_Python#Linux by selecting the **Phidget Python Module** option. You will then need to unzip this file using the `unzip` command. Go to the directory created by this unzipping process by typing `cd PhidgetsPython`. Then, type `sudo python setup.py install`.
4. Now install the Python examples by downloading the code from http://www.phidgets.com/docs/Language_-_Python#Use_Our_Examples_5. This will be a ZIP file, so you can unzip this using the `unzip` command and the `Python` directory will be created with a number of examples. Go to this directory by typing `cd Python`. You should see the following screenshot:



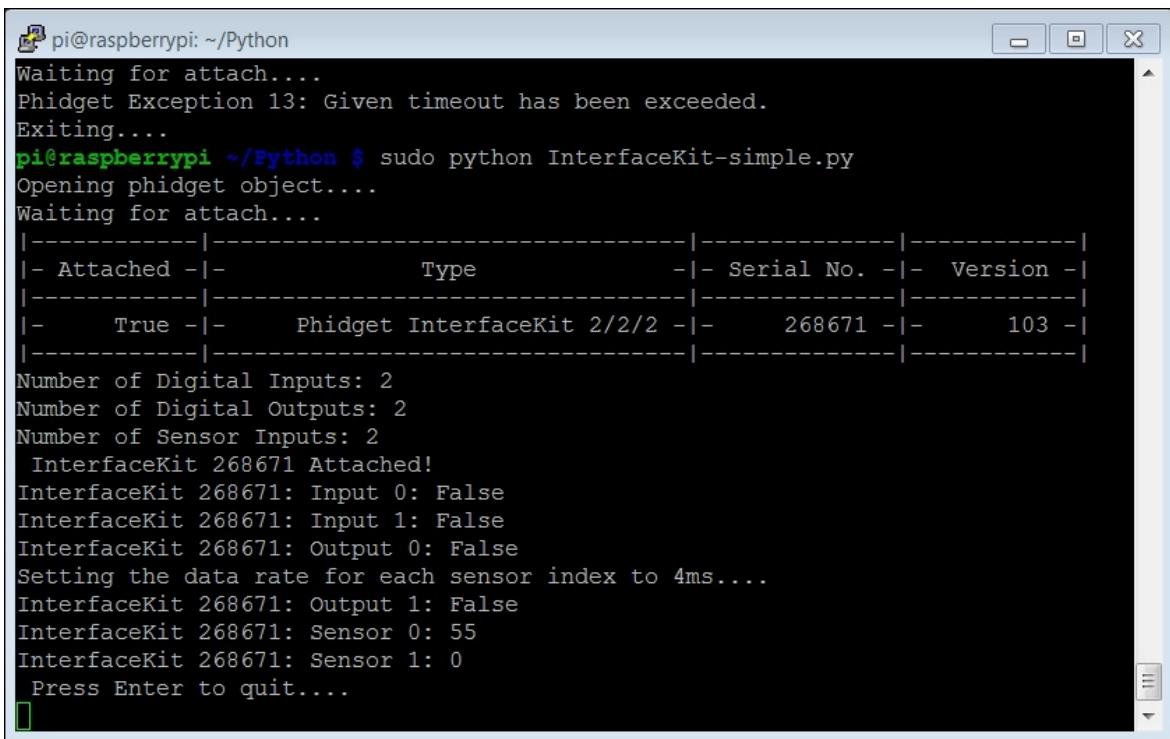
```
pi@raspberrypi: ~/Python
pi@raspberrypi ~/Python $ ls
Accelerometer-simple.py      GPS-simple.py          PHSensor-simple.py
AdvancedServo-simple.py       HelloWorld.py        RFID-simple.py
Analog-simple.py              InterfaceKit-simple.py Servo-simple.py
Bridge-simple.py              IR-simple.py         Spatial-simple.py
Dictionary-simple.py         LED-simple.py        Stepper-simple.py
Encoder-simple.py             Manager-simple.py   TemperatureSensor-simple.py
FrequencyCounter-simple.py   MotorControl-simple.py TextLCD-simple.py
pi@raspberrypi ~/Python $
```

5. Let's see whether the system can sense your Phidget device. Type `sudo python HelloWorld.py`. You should see the following screenshot:



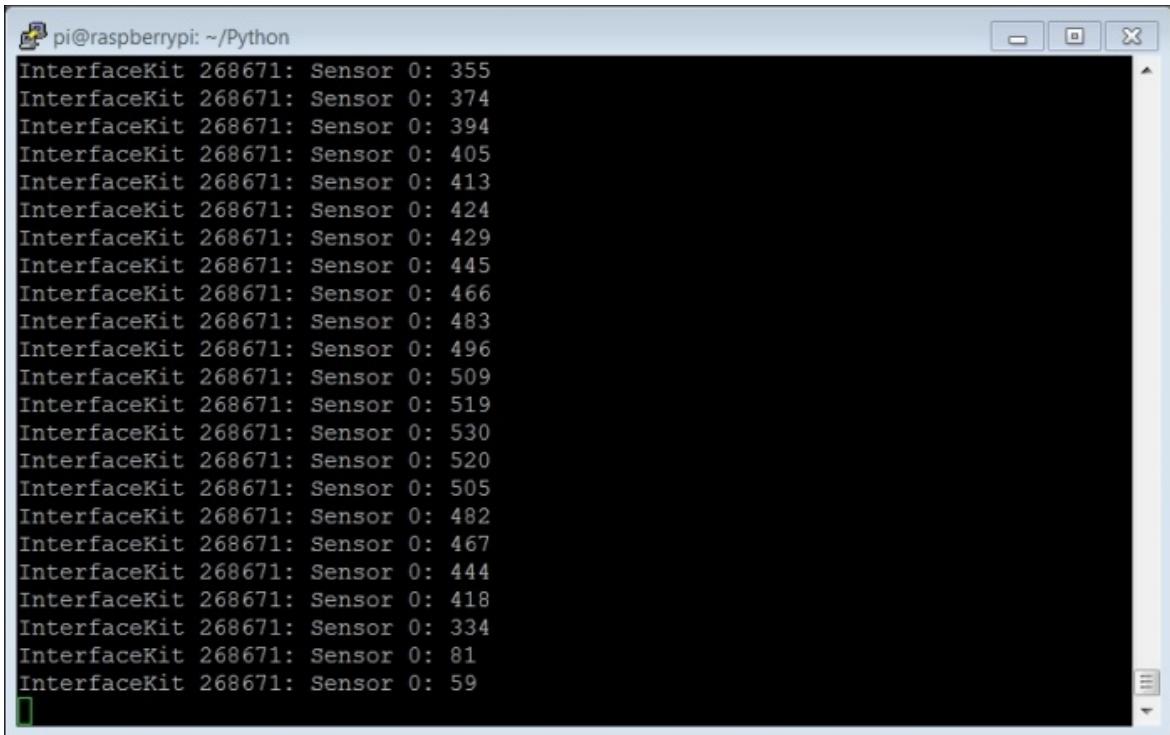
```
pi@raspberrypi: ~/Python
pi@raspberrypi ~/Python $ sudo python HelloWorld.py
Opening....
Phidget Simple Playground (plug and unplug devices)
Press Enter to end anytime...
Hello to Device Phidget InterfaceKit 2/2/2, Serial Number: 268671
```

- Now, you can run the [InterfaceKit-simple.py](#) example by typing `sudo python InterfaceKit-simple.py`. This will allow you to sense a target on your sensor. You should see the following screenshot if no targets are in front of the sensor:



```
pi@raspberrypi: ~/Python
Waiting for attach....
Phidget Exception 13: Given timeout has been exceeded.
Exiting....
pi@raspberrypi ~/Python $ sudo python InterfaceKit-simple.py
Opening phidget object....
Waiting for attach....
|-----|-----|-----|-----|
|- Attached -|- Type -|- Serial No. -|- Version -|
|-----|-----|-----|-----|
|- True -|- Phidget InterfaceKit 2/2/2 -|- 268671 -|- 103 -|
|-----|-----|-----|-----|
Number of Digital Inputs: 2
Number of Digital Outputs: 2
Number of Sensor Inputs: 2
    InterfaceKit 268671 Attached!
InterfaceKit 268671: Input 0: False
InterfaceKit 268671: Input 1: False
InterfaceKit 268671: Output 0: False
Setting the data rate for each sensor index to 4ms....
InterfaceKit 268671: Output 1: False
InterfaceKit 268671: Sensor 0: 55
InterfaceKit 268671: Sensor 1: 0
    Press Enter to quit....
```

- Place a target in front of the sensor and then remove it; you will see the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~/Python". The window displays a series of text lines representing sensor data. The data shows a sequence of values starting at 355 and increasing to 509, then decreasing back down to 59. This pattern repeats several times, indicating the sensor's response to a target approaching and then being removed.

```
pi@raspberrypi: ~/Python
InterfaceKit 268671: Sensor 0: 355
InterfaceKit 268671: Sensor 0: 374
InterfaceKit 268671: Sensor 0: 394
InterfaceKit 268671: Sensor 0: 405
InterfaceKit 268671: Sensor 0: 413
InterfaceKit 268671: Sensor 0: 424
InterfaceKit 268671: Sensor 0: 429
InterfaceKit 268671: Sensor 0: 445
InterfaceKit 268671: Sensor 0: 466
InterfaceKit 268671: Sensor 0: 483
InterfaceKit 268671: Sensor 0: 496
InterfaceKit 268671: Sensor 0: 509
InterfaceKit 268671: Sensor 0: 519
InterfaceKit 268671: Sensor 0: 530
InterfaceKit 268671: Sensor 0: 520
InterfaceKit 268671: Sensor 0: 505
InterfaceKit 268671: Sensor 0: 482
InterfaceKit 268671: Sensor 0: 467
InterfaceKit 268671: Sensor 0: 444
InterfaceKit 268671: Sensor 0: 418
InterfaceKit 268671: Sensor 0: 334
InterfaceKit 268671: Sensor 0: 81
InterfaceKit 268671: Sensor 0: 59
```

Note that the value increases as the target gets closer and then gets smaller when the target is removed.

You can calibrate your sensor by noting the value returned for a corresponding distance from the target. This allows your project to know how far the target is. The code for this capability is a bit complicated to be given in detail in this text, but basically it goes out, senses the presence of the Phidget device, sets the device to sense the IR sensor changes, and returns them to the user. To create your own code, follow the tutorial at http://www.phidgets.com/docs/Language_-_Python#Follow_the_Examples.

The following screenshot shows a simpler example of the code that may help you implement your IR capability:

```
pi@raspberrypi: ~/Python
File Edit Options Buffers Tools Python Help
#!/usr/bin/env python

from ctypes import *
import sys
import random
from Phidgets.PhidgetException import PhidgetErrorCodes, PhidgetException
from Phidgets.Events.Events import AttachEventArgs, DetachEventArgs, ErrorEventArgs, InputChangeEventArgs, OutputChangeEventArgs, SensorChangeEventArgs
from Phidgets.Devices.InterfaceKit import InterfaceKit

interfaceKit = InterfaceKit()
def interfaceKitAttached(e):
    attached = e.device
def interfaceKitDetached(e):
    detached = e.device
def interfaceKitInputChanged(e):
    source = e.device
    print("InterfaceKit %i: Input %i: %s" % (source.getSerialNum(), e.index, e.state))
def interfaceKitSensorChanged(e):
    source = e.device
    print("InterfaceKit %i: Sensor %i: %i" % (source.getSerialNum(), e.index, e.value))
def interfaceKitOutputChanged(e):
    source = e.device
    print("InterfaceKit %i: Output %i: %s" % (source.getSerialNum(), e.index, e.state))

#Main Program Code
interfaceKit.setOnAttachHandler(interfaceKitAttached)
interfaceKit.setOnDetachHandler(interfaceKitDetached)
interfaceKit.setOnInputChangeHandler(interfaceKitInputChanged)
interfaceKit.setOnOutputChangeHandler(interfaceKitOutputChanged)
interfaceKit.setOnSensorChangeHandler(interfaceKitSensorChanged)
interfaceKit.openPhidget()
interfaceKit.waitForAttach(10000)
for i in range(interfaceKit.getSensorCount()):
    interfaceKit.setDataRate(i, 4)

print("Press Enter to quit....")
chr = sys.stdin.read(1)
interfaceKit.closePhidget()
exit(0)
```

The following is a description of the code:

- `#!/usr/bin/env python` – This sets up the code so that you can run it from the command line without the Python directory.
- `from ctypes import *` – This imports the `ctypes` library, a library that allows Python to specify C data types.

- `import sys` – This imports the `sys` library.
- `import random` – This imports the random library and allows you to access random variables.
- `from Phidgets.PhidgetException import PhidgetErrorCodes, PhidgetException` – This imports libraries for Phidget's capabilities.
- `from Phidgets.Events.Events import AttachEventArgs, DetachEventArgs, ErrorEventArgs, InputChangeEventArgs, OutputChangeEventArgs, SensorChangeEventArgs` – This imports even more capabilities of Phidgets as a library.
- `from Phidgets.Devices.InterfaceKit import InterfaceKit` – This is one last import for the interface functionality from the Phidgets library.
- `interfaceKit = InterfaceKit()` – This creates an instance of an interface to your Phidgets device.
- `def interfaceKitAttached(e):` – The following lines will create callback functions or functions that are called by the device when certain events occur. This is the callback function for an attach event.
- `attached = e.device` – This simply attaches a device when asked.
- `def interfaceKitDetached(e):` – This is the callback function for an attach event.
- `detached = e.device` – This detaches the device when asked.
- `def interfaceKitInputChanged(e):` – This is the callback function for an input-changed event.
- `source = e.device` – This defines the device that is communicating with you.
- `print("InterfaceKit %i: Input %i: %s" % (source.getSerialNum(), e.index, e.state))` – This prints out the result of the event.
- `def interfaceKitSensorChanged(e):` – This is the callback function for a sensor-changed event. This is the function that is called when the value changes and will get the value from the sensor. It is here that you would put additional code to do other things with the sensor reading.
- `source = e.device` – This defines the device that is communicating with you.
- `print("InterfaceKit %i: Sensor %i: %i" % (source.getSerialNum(), e.index, e.value))` – This prints out the result of the event.
- `def interfaceKitOutputChanged(e):` – This is the callback function for an output-changed event.

- `source = e.device` – This defines the device that is communicating with you.
- `print("InterfaceKit %i: Output %i: %s" % (source.getSerialNum(), e.index, e.state))` – This prints out the result of the event.
- `#Main Program Code` – The following lines attach the callback functions to the device:

```
interfaceKit.setOnAttachHandler(interfaceKitAttached)
interfaceKit.setOnDetachHandler(interfaceKitDetached)
interfaceKit.setOnInputChangeHandler(interfaceKitInputChanged)
interfaceKit.setOnOutputChangeHandler(interfaceKitOutputChanged)
interfaceKit.setOnSensorChangeHandler(interfaceKitSensorChanged)
```

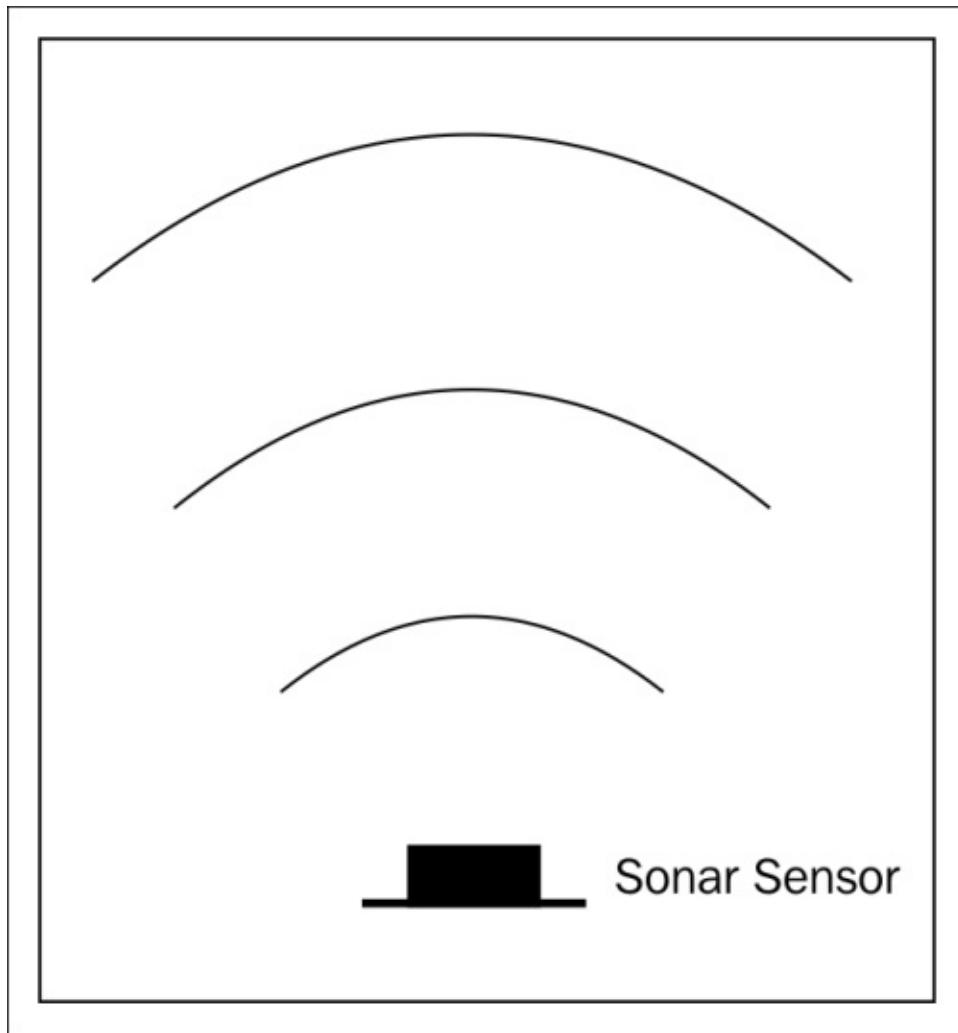
- `interfaceKit.openPhidget()` – This opens the device and attaches all the callbacks.
- `interfaceKit.waitForAttach(10000)` – This tells the program to wait to make sure that the device attaches.
- `for i in range(interfaceKit.getSensorCount()):` – This creates a loop for all the available sensors.
- `interfaceKit.setDataRate(i, 4)` – This sets the update rate to 4 ms.
- `print("Press Enter to quit....")` – This tells the user to hit *Enter* in order to quit the program.
- `chr = sys.stdin.read(1)` – This tells the program to get ready to read a character. When a character comes, this will close the program.
- `interfaceKit.closePhidget()` – This closes the interface to the object.
- `exit(0)` – This exits the program.

Now your project can sense objects!

Connecting Raspberry Pi to a USB sonar sensor

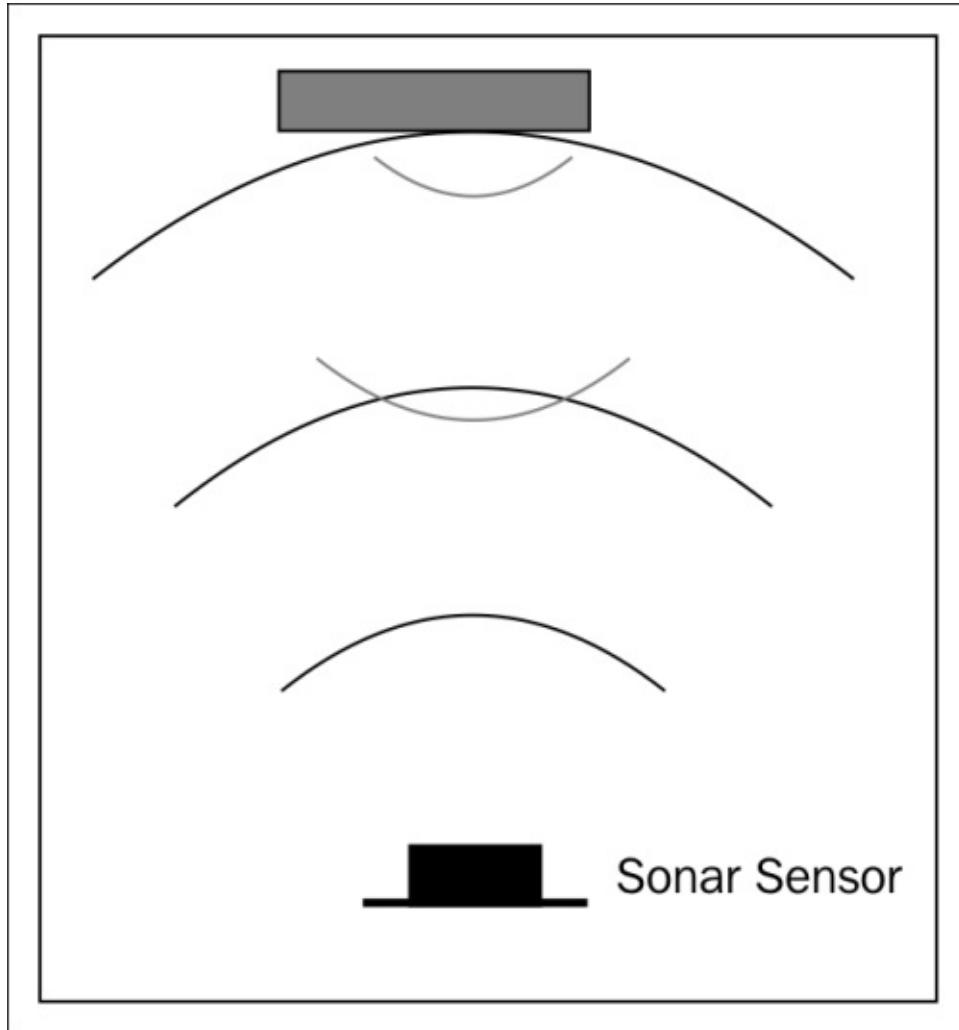
There is yet another way to sense the presence of objects: using a sonar sensor. But, before you add this capability to our system, here's a little tutorial on sonar sensors. This type of sensor uses ultrasonic sound to calculate the distance from an object.

The sound wave travels out from the sensor as illustrated in the following diagram:



The device sends out a sound wave at the rate of 10 times per second. If

an object is in the path of these waves, then the waves are reflected off the object, sending waves that return to the sensor, as shown in the following figure:



The sensor then measures any return waves. It uses the time difference between the sound wave that was sent out and the wave that was returned to measure the distance of the object.

Connecting the hardware

The first thing you'll want to do is connect the USB sonar sensor to your PC, just to make sure everything works well. Perform the following steps to do so:

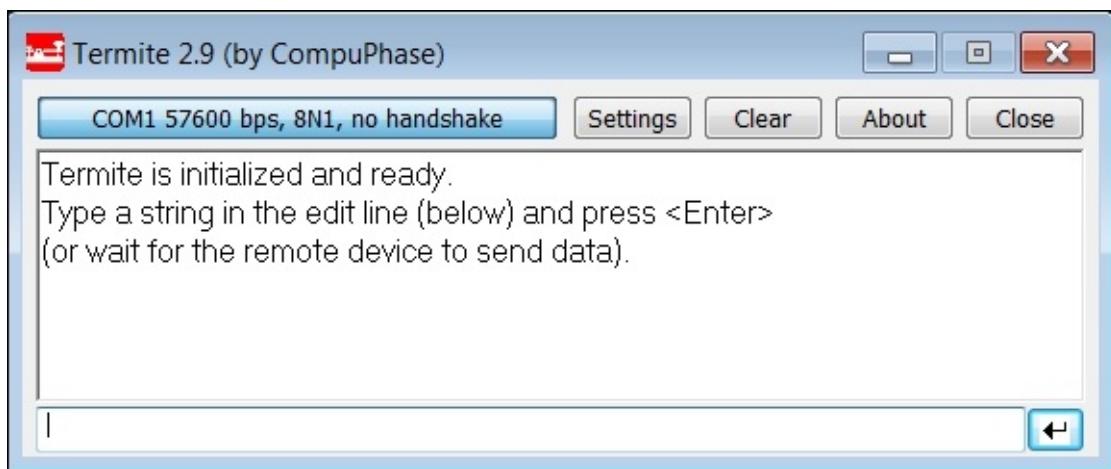
1. First, download the terminal emulator software from <http://www.maxbotix.com/articles/059.htm> and click on the **Windows Download** button.

The screenshot shows a website for MaxBotix, Inc., featuring a navigation menu on the left with links like Home, Products / Buy Now, Documents & Downloads, Performance Data, Tutorials & Application Notes, Contact, and News. On the right, there's a banner for "High performance ultrasonic rangefinders" with a search bar and social media links. The main content area displays an article titled "Terminal Program Setup Guide" written by Tom Bonar on 10-25-2012, updated on 04-03-2013. The article includes a screenshot of a terminal window showing a list of ports (P015_P1 through P015_P15) and download links for Windows, Linux, and Apple OS. Below the article, there are logos for Sensors Portal Magazine Top 10 Products of 2012 and FIRST Robotics Competition, along with a note about preferred operating system instruction sets.

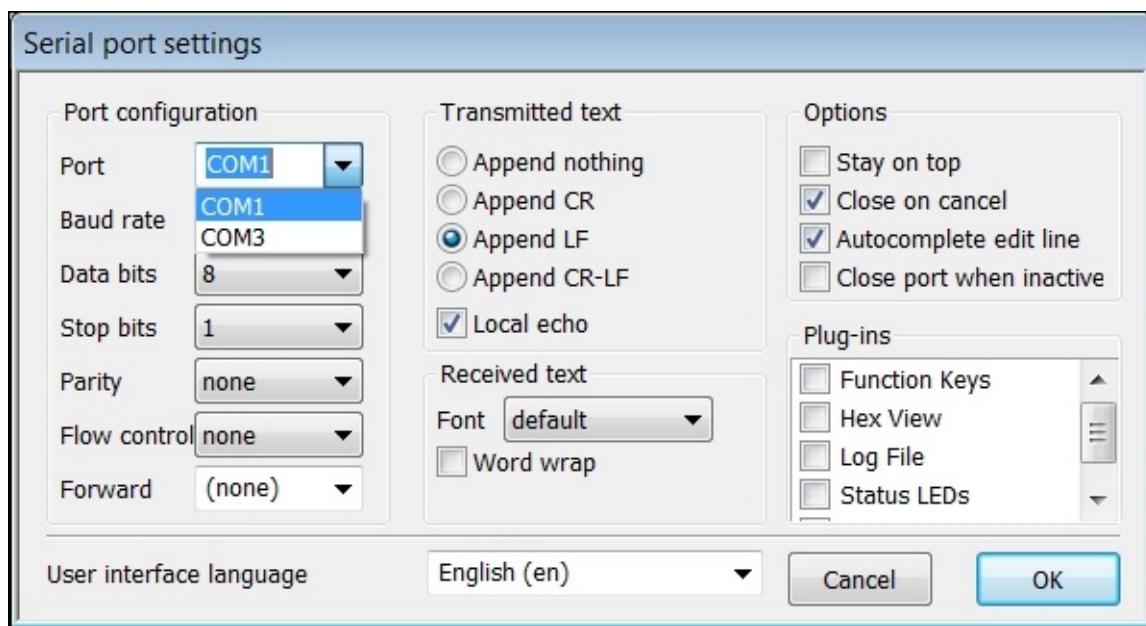
2. Unzip this file. Then, plug the sensor into a USB port on your PC and open the terminal emulator file by selecting this file from the directory.

Name	Date modified	Type	Size
Function_Keys.flt	9/16/2013 5:53 PM	FLT File	36 KB
Hex_View.flt	9/16/2013 5:53 PM	FLT File	36 KB
Log_File.flt	9/16/2013 5:53 PM	FLT File	47 KB
setup.log	9/16/2013 5:53 PM	Text Document	2 KB
Status_LEDs.flt	9/16/2013 5:53 PM	FLT File	33 KB
Termite.exe	9/16/2013 5:53 PM	Application	116 KB
Timestamp.flt	9/16/2013 5:53 PM	FLT File	39 KB
WritingFilters.pdf	9/16/2013 5:53 PM	Adobe Acrobat D...	66 KB

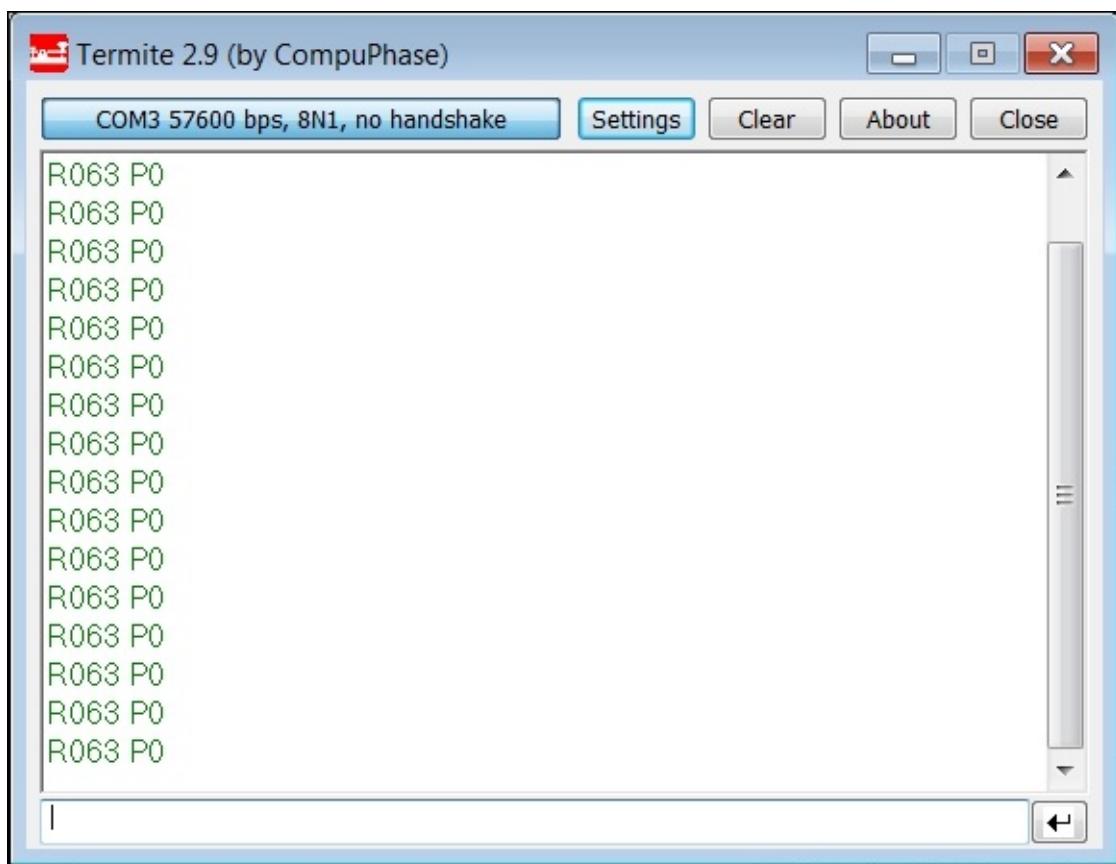
3. The following application window should pop up:



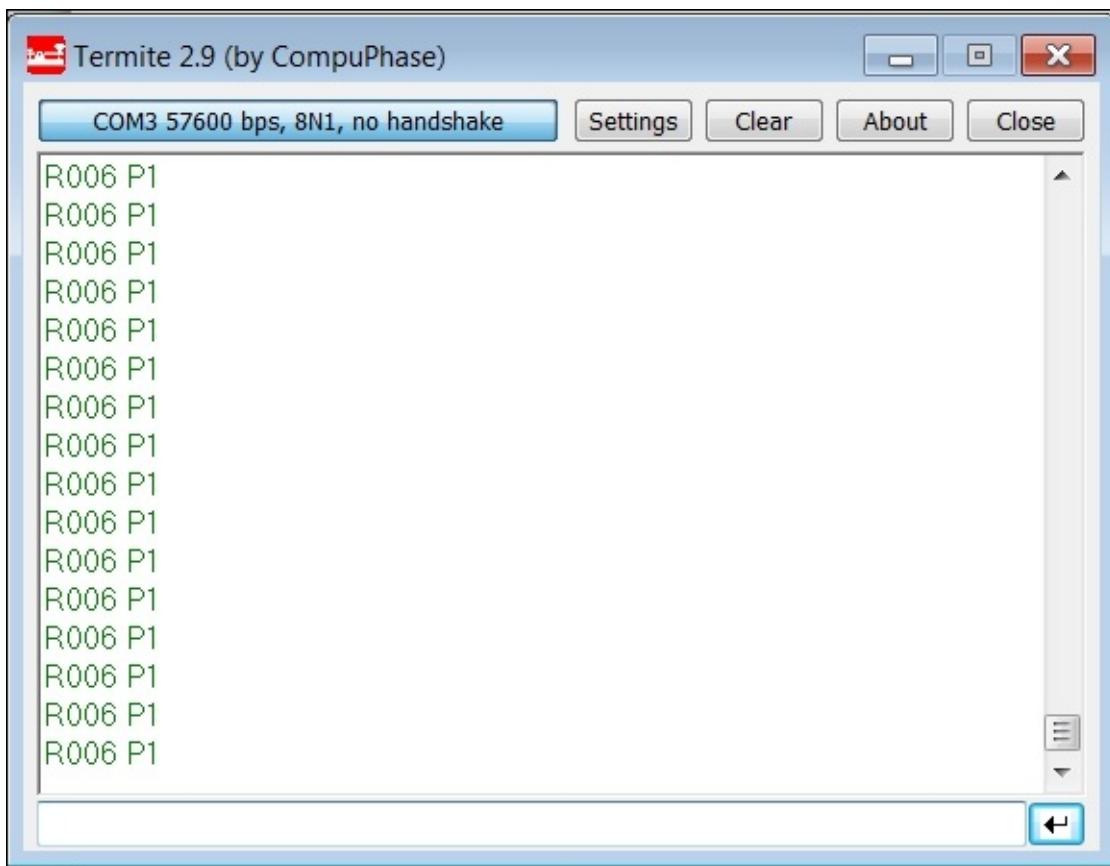
4. We'll need to change the setting in order to find the sensor. So, click on the **Settings** button and you should see the following screenshot:



5. Click on the **Port** menu and select the port that is connected to your sensor. In my case, I selected **COM3**, clicked on **OK**, and I saw the following screenshot on the main screen:



6. Note the sensor readings. Now place an object in front of the sensor. You should now see something as shown in the following screenshot:



The readings have changed, specifically the values after **R** and **P**, indicating that an object is in front of the sensor. You'll need to read these values into your program and then you can avoid colliding with the object.

Now that you know how the unit works, you'll want to mount the USB sensor on your mobile platform. In this case, I am going to mount the USB sonar sensor on my quadruped robot.

Make sure you plug one end of the USB cable into the sensor and the other end into the USB hub connected to Raspberry Pi.

With the hardware all constructed and the sensor working, you can start talking to your USB sensor using Raspberry Pi. You are going to create a simple Python program that will read the values from the sensor. To do this using Emacs as an editor, type `emacs sonar.py`. A new file called `sonar.py` will be created. Then, type the code as shown in the following screenshot:

A screenshot of a terminal window titled "sonar.py". The window has a menu bar with "File", "Edit", "Options", "Buffers", "Tools", "Python", and "Help". The code in the terminal is:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial

ser = serial.Serial('/dev/ttyUSB0', 57600, timeout = 1)
x = ser.read(100)
print(x)

-UU-----F1  sonar.py      All L2      (Python)-----
Wrote /home/pi/sonar.py
```

Let's go through the code to see what is happening:

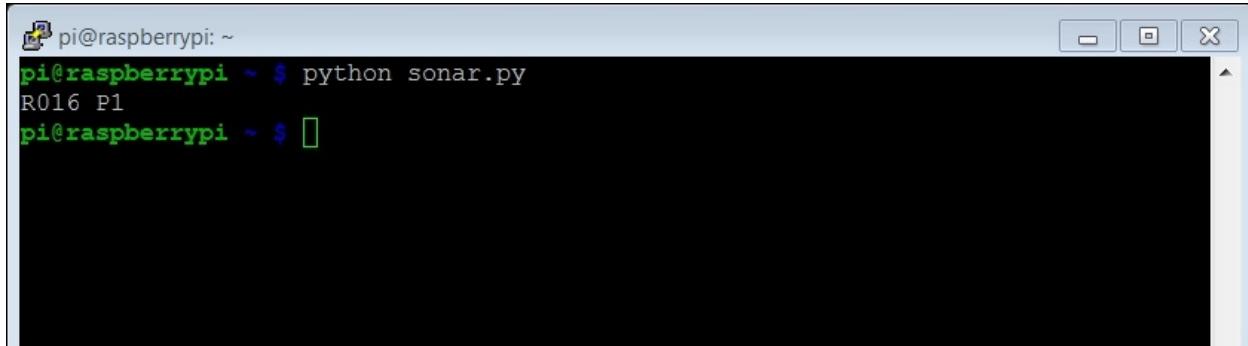
- `#!/usr/bin/python` – As explained earlier, the first line simply makes this file available for us to execute from the command line.
- `import serial` – Again, we import the `serial` library. This will allow us to interface the USB sonar sensor.
- `ser=serial.Serial('devttyUSB0', 57600, timeout = 1)` – This command sets up the serial port to use the `devttyUSB0` device, which is the sonar sensor using a baud rate of `57600` and a timeout of `1`.
- `x = ser.read(100)` – The next command reads the next 100 values from the USB port.
- `print(x)` – The final command then prints out the value.

Once you have created this file, you can run the program and talk to the device. You can do this by typing `./sonar.py` and the program will run. I have found that sometimes, the device returns no data for the first time, so don't be surprised if you print out no values the first time you run your program. The second time, you should receive a valid return string. The following screenshot is the result of running my program:

A screenshot of a terminal window showing the output of the `sonar.py` program. The terminal shows:

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ python sonar.py
R110 P0
pi@raspberrypi ~ $
```

The sensor returns [110](#), which indicates the relative distance to a barrier in millimeters. If you place a good reflector at just a few inches in front of the sensor and run the program, you will get the following result:

A screenshot of a terminal window titled "pi@raspberrypi: ~". The window contains the following text:

```
pi@raspberrypi ~ $ python sonar.py
R016 P1
pi@raspberrypi ~ $
```

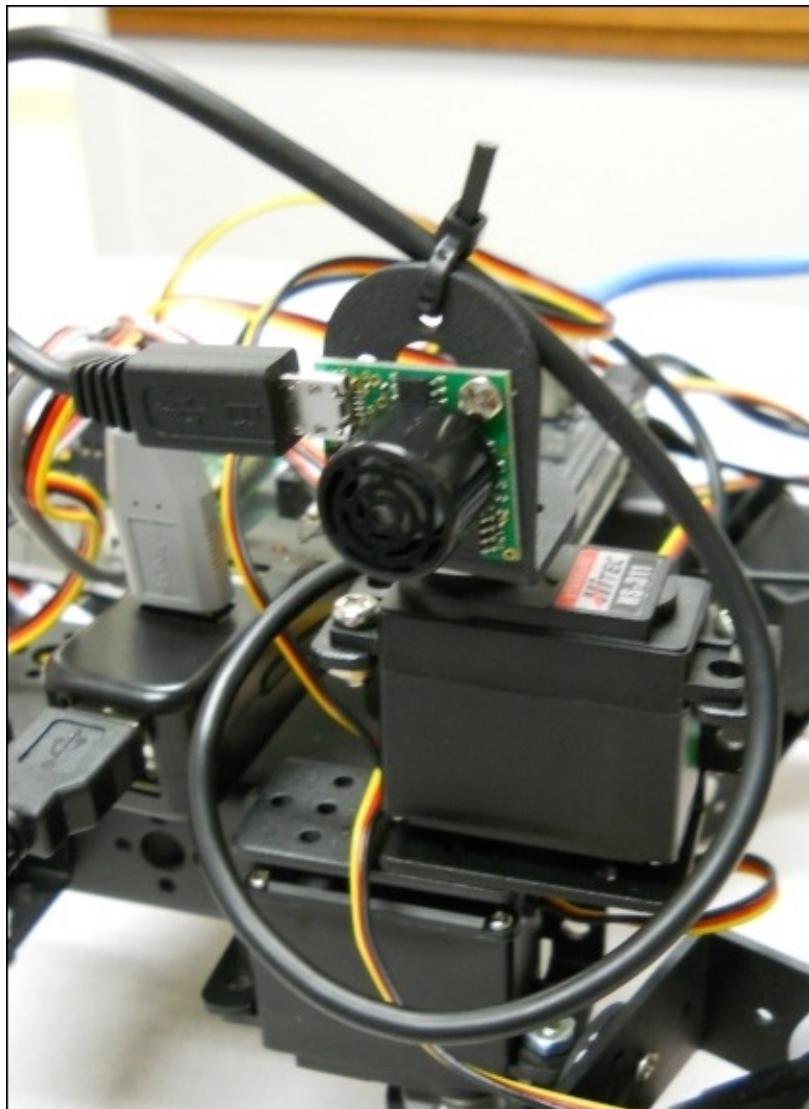
The window has a blue header bar with the title and standard window control buttons (minimize, maximize, close) in the top right corner.

Now the robot can sense its environment so it can avoid bumping into walls and other barriers!

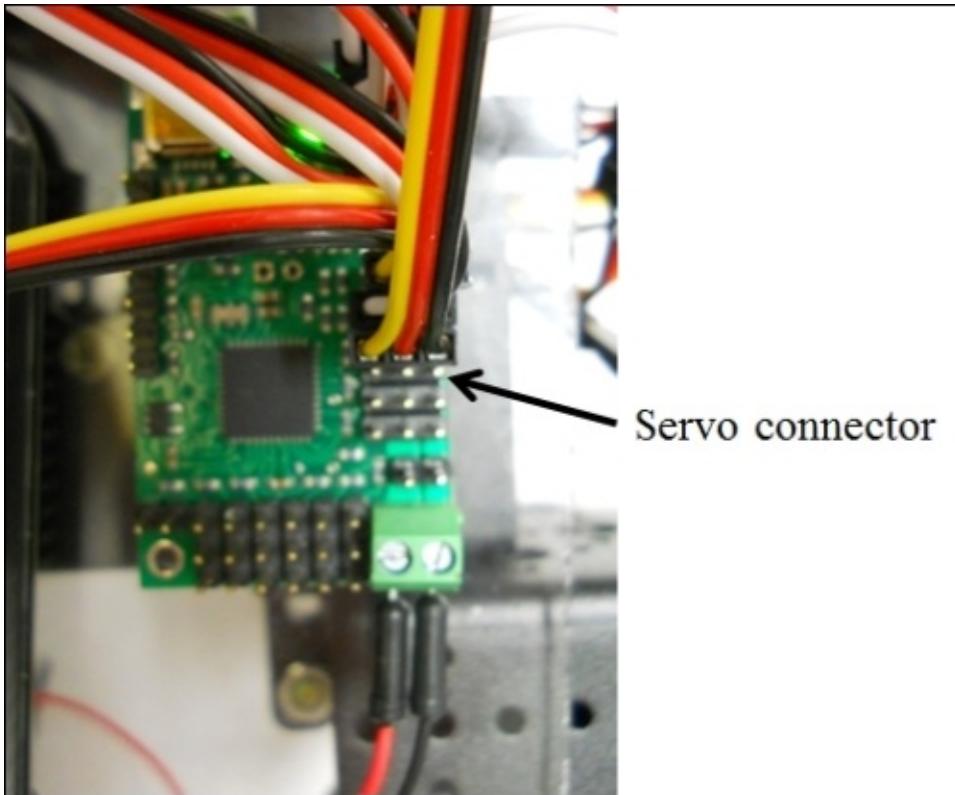
Using a servo to move a single sensor

You now have your sensors in place. If we want to sense more than just one direction, we could use several sensors, each mounted to a different side of the robot. However, there is a way to use servos to move your sensor, which allows you to use a single sensor to sense in several directions.

The simplest way to avoid having to purchase and configure several sensors is to mount the sensor on a single servo and then use a servo bracket to connect this assembly to the platform. Using the sonar sensor, the assembly will look something as shown in the following image:



Make sure you connect your servo to the servo controller; it can fit into any open connection. I am connecting mine to my quadruped robot that has eight servos to control, so I have connected mine to the eighth connection on the servo controller board, as follows:



I'll assume that you already have your sensor up and working and know how to read data. In this section, you will add the ability to move the sensor by communicating with the servo through the servo controller we configured in the previous chapter.

For the program, you will begin with the `robot.py` program you created in [Chapter 6, Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs](#), as you are going to need to access the servo controller. However, you may want to keep a copy of this program, just in case you want to use it later. First, go to the directory that contains the `robot.py` program; in my case, I placed it in the `maestro_linux` directory, so I would type `cd ./maestro_linux` from my login or home directory. Now, let's create a copy of this program by typing `cp robot.py sense.py`.

You'll want to edit this program. So if you are using the Emacs editor, type `emacs sense.py`. The program you want to create will look as shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~" with a menu bar including File, Edit, Options, Buffers, Tools, Python, and Help. The window contains Python code for controlling a servo motor and reading from a sonar sensor. The code uses the serial module to communicate with a Pololu Micro Maestro servo controller and a USB sonar sensor. It includes methods for setting servo angles and reading sensor data. The code ends with a main block that initializes the robot, sets servo angles, and reads sensor data twice.

```
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time
class PololuMicroMaestro(object):
    def __init__(self, port= "/dev/ttyACM0"):
        self.ser = serial.Serial(port = port)
    def setAngle(self, channel, angle):
        minAngle = 0.0
        maxAngle = 180.0
        minTarget = 256.0
        maxTarget = 13120.0
        scaledValue = int((angle / ((maxAngle - minAngle) / (maxTarget - minTarget))) + minTarget)
        commandByte = chr(0x84)
        channelByte = chr(channel)
        lowTargetByte = chr(scaledValue & 0x7F)
        highTargetByte = chr((scaledValue >> 7) & 0x7F)
        command = commandByte + channelByte + lowTargetByte + highTargetByte
        self.ser.write(command)
        self.ser.flush()
    def close(self):
        self.ser.close()
if __name__=="__main__":
    robot = PololuMicroMaestro()
    sensor=serial.Serial('/dev/ttyUSB0', 57600, timeout = 1)
    robot.setAngle(8,65)
    time.sleep(2.5)
    range = sensor.read(100)
    print(range)
    robot.setAngle(8,90)
    time.sleep(2.5)
    range = sensor.read(100)
    print(range)
    robot.setAngle(8,115)
    time.sleep(2.5)
    range = sensor.read(100)
    print(range)
```

-UUU:**--F1 sense.py All L16 (Python)-----

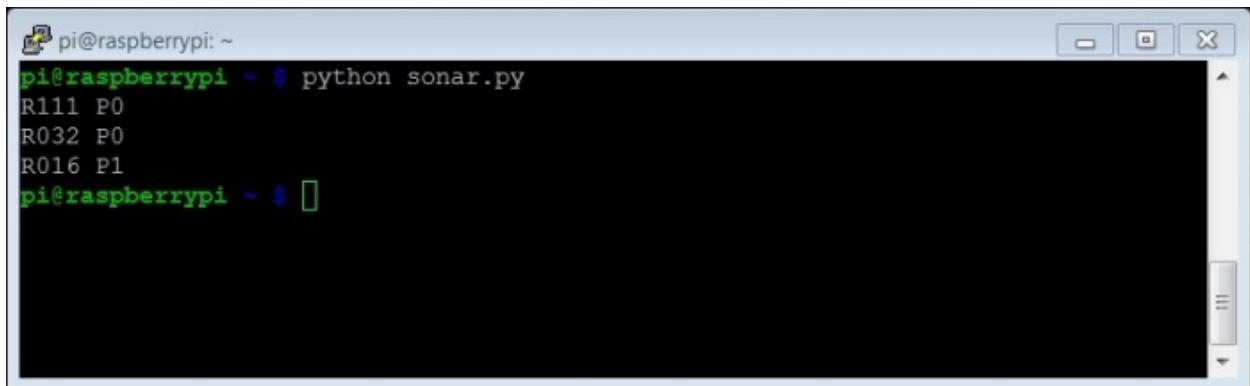
Let's walk through the code to see what it does. I will begin with the section that begins with `if __name__=="__main__":` as everything above this comes to us from the `robot.py` code and is covered in the previous chapter.

- The `robot=PololuMicroMaestro()` line initializes the servo motor controller and connects it to the proper USB port.
- The next line opens a serial port, which we will call `sensor`, that connects you to the USB sonar sensor at the `/dev/ttyUSB0` port and

sets its parameters.

- You can now ask the servo to go to a specific position and take a reading. In this case, I am doing this for the servo positions at 65 degrees, 90 degrees, and 115 degrees. At each of these locations, you can ask for a range reading. Note that you need to wait 2.5 seconds for the sensor to respond, based on the specifications of the manufacturer for the device to deliver a stable reading.

That's it! Now, you can sense objects in front of you and on either side. The following screenshot is an example of what might be displayed as a result of running the program:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ python sonar.py
R111 P0
R032 P0
R016 P1
pi@raspberrypi ~ $
```

If you are adding the sensor-servo combination to your wheeled vehicle, you'll need to add the servo motor controller as well. The motor controllers, servo controllers, and USB sonar or IR sensor can all coexist on the same Raspberry Pi. You'll need to merge the [dcmotor.py](#) and [sense.py](#) programs so that you can access each individual capability.

You can also use sensors to find an object by having two sensors that determine the distance of each sensor from the object and then triangulate its position. This can help your robot actually find the position of specific obstacles. How this is accomplished is detailed on the MaxBotix website at

http://www.maxbotix.com/documents/MaxBotix_Ultrasonic_Sensors_Find
You now have all the knowledge you need to add this type of capability to your robot.

For more details on using IR and sonar sensors for obstacle avoidance, you can visit several good places on the web. Try
<http://www.intorobotics.com/interfacing-programming-ultrasonic-sensors->

[tutorials-resources](#)/ and <http://www.geology.smu.edu/~dpa-www/robo/challenge/obstacles.html> as well as
http://www.societyofrobots.com/member_tutorials/book/export/html/71.

Summary

Congratulations! Your robot can now detect and avoid walls and other barriers. You can also use these sensors to detect objects that you might want to find. In the next chapter, you'll learn how to disconnect your robot from all its wires and control it wirelessly.

Chapter 8. Going Truly Mobile – The Remote Control of Your Robot

Based on the previous chapters, you now have mobile robots that can move around, accept commands, see, and even avoid obstacles. This chapter will teach you how to electronically communicate with your robot without using any wires.

As you send your device out into the world, you may still want to communicate with it electronically without connecting a cable. If you add this capability, you can change what your mobile robot is doing without any physical contact but still remain in complete control of your project.

In this chapter, we will cover the following points:

- Connecting Raspberry Pi to a wireless USB input device
- Using the wireless USB input device in order to issue commands to your project
- Connecting to your robot over wireless LAN
- Connecting to your robot over Zigbee

Gathering the hardware

In this chapter, you'll learn how to connect to your device wirelessly. There are several ways to accomplish this. The first way that we'll cover in this chapter is to do this with a standard USB wireless input device. This is, perhaps, the easiest way to control your robot but only provides basic functionality with a limited range. The second way to connect to your robot that you will learn is via a wireless LAN device; this provides an excellent bandwidth and good range but requires a bit more hardware. This will provide you with the opportunity to both control your robot and see what it is seeing.

Finally, in this chapter, we will cover communicating with your robot

wirelessly via a dedicated wireless link called ZigBee. It will provide the same sort of control as with the wireless USB device but with a much greater range.

No matter what you choose, you will probably want to purchase a small LCD display first for your Raspberry Pi. This will allow you to monitor what is going on with your project. In the previous chapters, you used a separate computer monitor for this. However, the monitor is just too big and not really designed for mobile use. Fortunately, there are several inexpensive choices for small LCDs with **SVideo input** that connect right to Raspberry Pi. The following is an image of the device I have used for some of my projects:



This type of LCD is available on Amazon and other online electronics stores, so you should be able to get it at almost any place. One of the challenges, however, is that most of these are designed for auto applications and may require 12 volts to operate. I have been lucky; each one of the devices that I have ordered has worked just fine at 5 volts. However, if you want to make sure that you get one that is compatible, you can order it from adafruit.com; they have a set of devices that is compatible with Raspberry Pi. Also, there are several versions of LCDs that are made for Raspberry Pi as a plugin cape, but I like the standalone versions better, only because they give me more flexibility in mounting the unit. However, there is one challenge. You will need to power the LCD, which is one of the reasons why I like to choose a dual-output USB battery for my projects. The following is an image of such a battery:



You'll also need to order a power adapter for your LCD screen, depending on how it accepts power. The LCD shown in the previous image comes with an adapter that has a red and a black wire connector output. If the unit can work at 5 volts, you can connect the LCD to your cell phone battery USB connector using a USB-to-TTL Serial cable. In this case, you'll plug the red and black connectors out of your LCD into the red and black connectors on the USB-to-TTL serial cable. You can also plug the LCD into the battery connector that is set up for your robot if it supplies between 5 to 12 volts.

Just a quick note on battery selection. You'll need to keep in mind two key characteristics when you buy your battery. The first is the size of the battery, normally noted in **mAh** or **milliAmpHours**. This is a measure of the capacity of the battery, which will tell you how long a battery might last while drawing a certain current. For example, if you purchased a 2000 mAh battery and drew an average of 100 mAmps, this battery would theoretically last for 20 hours. However, it won't last that long. For, as the battery discharges, the voltage will start to drop as well, and eventually, you won't be able to power your system. The voltage drop depends on the quality of the battery.

The second key characteristic is the amount of current you can draw at

any given time. Most batteries give a C rating, and if you divide the mAh rating by this value, you will get the amount of instantaneous current that you can draw. This value is important because this will need to estimate the amount of power you need to draw from your electronics. For example, Raspberry Pi can draw as much as 500 mAmps, so you'll need to make sure you get a battery that can supply this kind of current.

Now that you have a screen, you can display the results on the robotic platform itself. No extra programming is needed; the Raspbian release will automatically send signals to the LCD screen and boot with the screen acting as a display. After the system boots, it will look something like what is shown in the following screenshot:



Now that you can display what is going on inside Raspberry Pi, you need to choose the wireless connection that you'd like to use. If you just need to send basic control signals to your device and you are always going to be close, a 2.4 GHz wireless keyboard is the best choice.

The following image is of a standard 2.4 GHz wireless keyboard:



This is a Logitech keyboard. Logitech generally makes very reliable keyboards, and these connect well to Raspberry Pi. This keyboard is available online on Amazon and at most electronics or computer stores. You'll notice that this version has a built-in mouse pad.

Another option is a small keyboard that looks more like a game controller. It will make your projects look amazing and will make it easier to control. The following is an image of such a keyboard:



This 2.4 GHz wireless keyboard by HausBell is small, about the size of a game controller, relatively inexpensive and is sold online, again by Amazon.

For this application, there are several choices that you could make for wireless technology to communicate with Raspberry Pi. Bluetooth is quite popular and works well. However, it comes with the added complexity of having to pair the device with the Bluetooth USB dongle and the system. The 2.4 GHz wireless technology comes with the keyboard and wireless USB receiver already paired. So, the device only works with the USB dongle that is shipped with the device, and the system automatically recognizes the device as long as the USB receiver is plugged into the USB port of Raspberry Pi.

The 2.4 GHz wireless devices work with the same frequency range as many 2.4 GHz wireless LAN devices, although they do not use the same modulation or protocol that is used by the standard 2.4 GHz wireless LAN. Rather, they use a proprietary modulation and protocol that is specific to the device and company that manufactures the device. There are more details on the 2.4 GHz wireless keyboards at http://www.logitech.com/images/pdf/emea_business/2.4ghz_white_paper.pdf.

While each device is different, most use the same overall approach where they define a number of different channels or small frequency ranges inside the overall range of 2.4 GHz. The keyboard communicates with the USB receiver on one of these frequencies. However, if either the keyboard or the USB receiver senses that some other device is transmitting on that frequency, the device will move to a different channel to try and avoid the interference.

The transmissions between the wireless keyboard and USB receiver are encrypted, so no device except the paired keyboard and USB receiver can understand the messages that are being sent between the two devices. The range of the keyboard and receiver pair is dependent upon the amount of power both use for transmission; the higher the power, the longer the range. Unfortunately, the higher the power, the less is the time for which the batteries in the wireless device last. Most wireless keyboards are designed to work for up to 10 meters or around 30 feet.

Sometimes, you may want to have a physical connection with your Raspberry Pi, and want not only to control your robot but also connect via VNC Viewer, so you can access the display of Raspberry Pi; in such a case, wireless LAN is the best choice. For this, you'll need to purchase a

supported wireless LAN device. The following is an image of such a device:



It is best to choose a device that is known to be supported by your Raspbian release. Check the http://elinux.org/RPi_USB_Wi-Fi_Adapters link for a list of these devices. Note that you'll need access to a wireless LAN signal, so you'll need to supply this by connecting to your own wireless LAN. This can come from a wireless LAN router that you have already set up, a spare wireless LAN router that can set up an ad hoc network, or many of today's smart cell phones, which also allow you to turn your device into a hotspot that can provide this signal.

If you'd like to have basic communication with your device but want to do it at a significant distance, ZigBee provides a possible solution. There are a number of different types of devices; one is a USB stick-type ZigBee device (www.zigbee.org), and the following is an image of such a device:



Also, there are devices made to connect with Linux systems such as Raspberry Pi. The following is an image of one of these devices:



This is the device we will use in this chapter. Make sure you purchase an XBee Series 1 device as it is the easiest device to configure and use, and there is a great open source community support for the device too. If you choose a different device, you'll need to follow the directions for that device from the manufacturer. Also, if you want to use this type of point-to-point communications, you'll need two units, one for Raspberry Pi and the other for the host computer.

Connecting Raspberry Pi to a wireless USB keyboard

You've been able to control your projects using a LAN connection, but you don't always want to have your projects tethered in this manner. In this section, I'll show you how to connect via a wireless keyboard.

Find your USB keyboard. It should come with a USB dongle. Plug the USB dongle into the Raspberry Pi USB port. After some time, the unit should power on to the windowing system. Now, if you move your finger around on the mouse pad, you should see the mouse moving on the screen. You can also select a terminal and type some text into the terminal. You now have keyboard and mouse inputs. Next, you will learn how to accept the key strokes into a program in order to control the robot.

Using the keyboard to control your project

Now that the keyboard is connected, let's figure out how to accept commands on Raspberry Pi. Now that you can enter commands wirelessly, the next step is to create a program that can take these commands and then have your project execute them. There are a couple of options here; you'll see examples of both. The first is to simply include the command interface in your program. Let's take an example of the program you wrote to move your wheeled robot, `dcmotor.py`. If you want, you can copy that program using `cp dcmotor.py remote.py`.

In order to add user control, you need two new programming constructs, the `while` loop and the `if` statement. Let's add them to the program and then we will learn what they do. The following is a listing of the area of code you are going to change:

```

#!/usr/bin/python
import serial
import time
def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)
    ser.write(sendByte)
    ser.write(chr(speed))
ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
var = 'n'
while var != 'q':
    var = raw_input(">")
    if var == '<':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == '>':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'f':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'r':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
ser.close()

```

-UU-:***-F1 remote.py All L43 (Python)-----

You will edit your program by making some changes. Add the code in the preceding screenshot just below the `ser = serial.Serial('devttyUBS0', 19200, timeout = 1)` statement. The code can be explained as follows:

- The `var = 'n'` statement will define a variable named `var` and it will be of the type `character`, which you will use in your program to get the input from the user.
- The `while var != 'q':` statement will place your program in a loop.

This loop will keep repeating until you or the user enters the letter `q`.

- The `var = raw_input(">")` statement will get the `character` value from the user. The `>` text is simply the character that will be displayed for the user to enter something.
- The `if var == '<':` statement checks the value that you get from the user. If it is a `<` character, the robot will turn left by running the right DC motor for half a second. You will need to determine how much time is required to run the right DC motor for a left turn. The actual time value, `0.5` in this case, may need to be higher or lower.
- The next few lines send a `Speed` command to the motor, wait for 0.5 seconds, and then send a command for the motor to stop.
- The `if var == '>':` statement checks the value that you get from the user. If it is a `>` character, the robot will turn left by running the left DC motor for half a second. You will need to determine how much time is required to run the left DC motor for a right turn. The actual time value, `0.5` in this case, may need to be higher or lower.
- The next few lines send a `Speed` command to the motor, wait for 0.5 seconds, and then send a command for the motor to stop.
- The `if var == 'f':` statement checks the value that you get from the user. If it is a `f` character, the robot will run forward by running the right and left DC motors for half a second. You will need to determine the speed to set each motor to follow a forward path.
- The next few lines send a `Speed` command to both motors, wait for 0.5 seconds, and then send a command for both motors to stop.
- The `if var == 'r':` statement checks the value that you get from the user. If it is a `r` character, the robot will run backward by running the right and left DC motors for half a second. You will need to determine the speed to set each motor to follow a backward path.
- The next few lines send a `Speed` command to both motors, wait for 0.5 seconds, and then send a command for both motors to stop.

Once you have edited the program, save it and make it executable by typing `chmod +x remote.py`. Now you can run the program, but you must run it by typing the command using the wireless keyboard. If you are not yet directly logged into Raspberry Pi, make sure you can see the LCD screen and access it via the wireless keyboard. You can now disconnect the LAN cable; you will be able to communicate with Raspberry Pi via the wireless keyboard. The system should look like the following image:

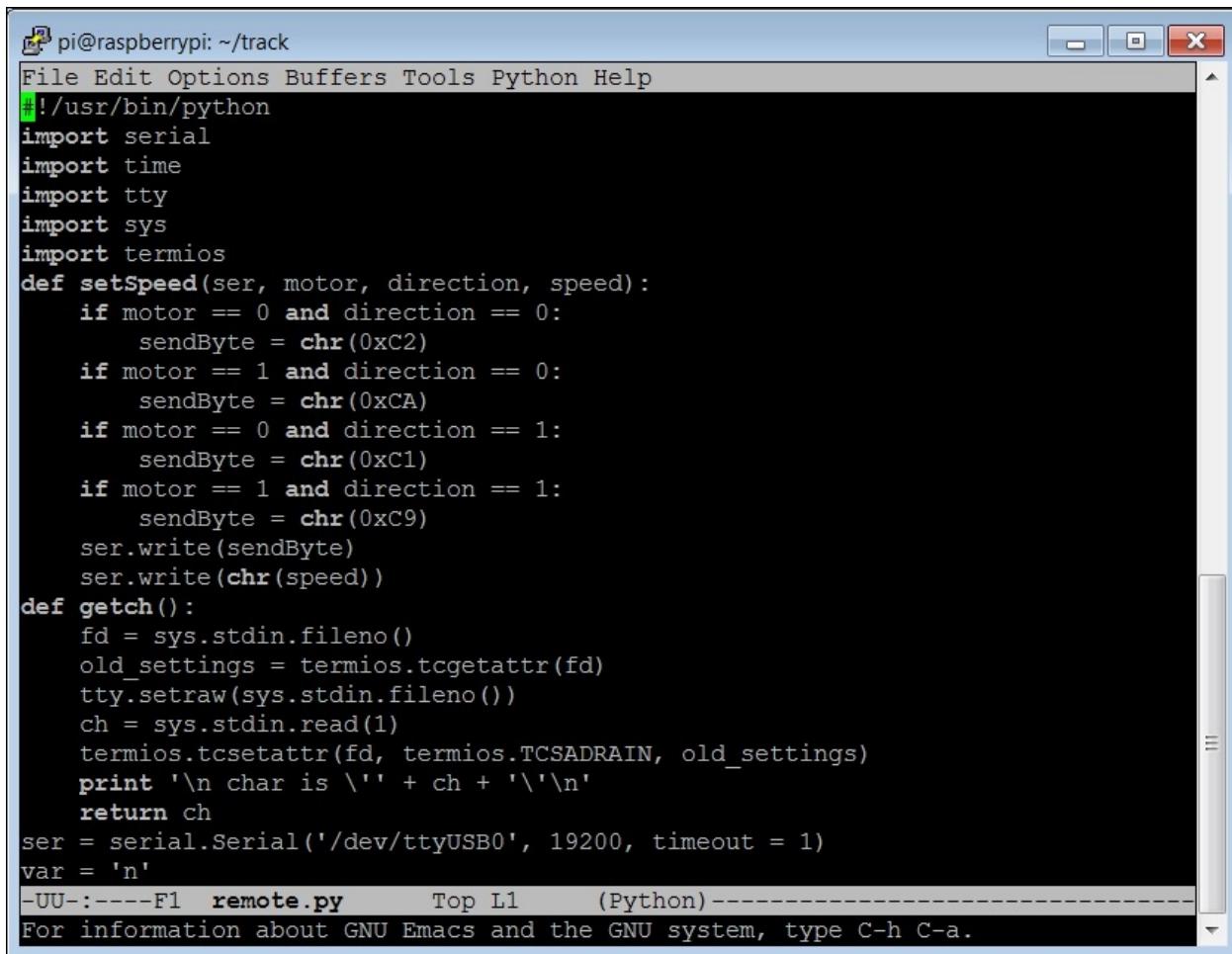


To use this system, type `cd` to go to the directory that holds the `remote.py` program. In my case, this file was in the `homepi/track` directory, so I did a `cd track` from my home directory. Now you can run the program by typing `./remote.py`. The screen will display a prompt, and each time you type the appropriate command (`<`, `>`, `f`, and `r`) and press *Enter*, your robot will move. You need to be advised that the range of this technology is at best

around 30 feet, so don't let your robot get too far away.

Now you can move your robot around using the wireless keyboard! You can even take your robot outside. You don't need the LAN cable to run your programs because you can run them using the LCD display and keyboard.

There is one more change you can make so that you don't have to hit the *Enter* key after each input character is typed. In order to make this work, you'll need to add some inclusions to your program and then add one function that can get a single character without the *Enter* key. The following screenshot shows the first set of changes you'll need to make:



The screenshot shows a terminal window titled "pi@raspberrypi: ~/track". The window contains Python code for controlling a robot. The code includes imports for serial, time, tty, sys, and termios. It defines two functions: `setSpeed` and `getch`. The `setSpeed` function takes parameters `ser`, `motor`, `direction`, and `speed`. It uses if statements to determine the `sendByte` value based on the motor number and direction. The `getch` function reads a single character from standard input using `sys.stdin.read(1)` and prints it to the screen. The script then creates a `Serial` object named `ser` pointing to `/dev/ttyUSB0` at 19200 baud. The file is saved as `remote.py`.

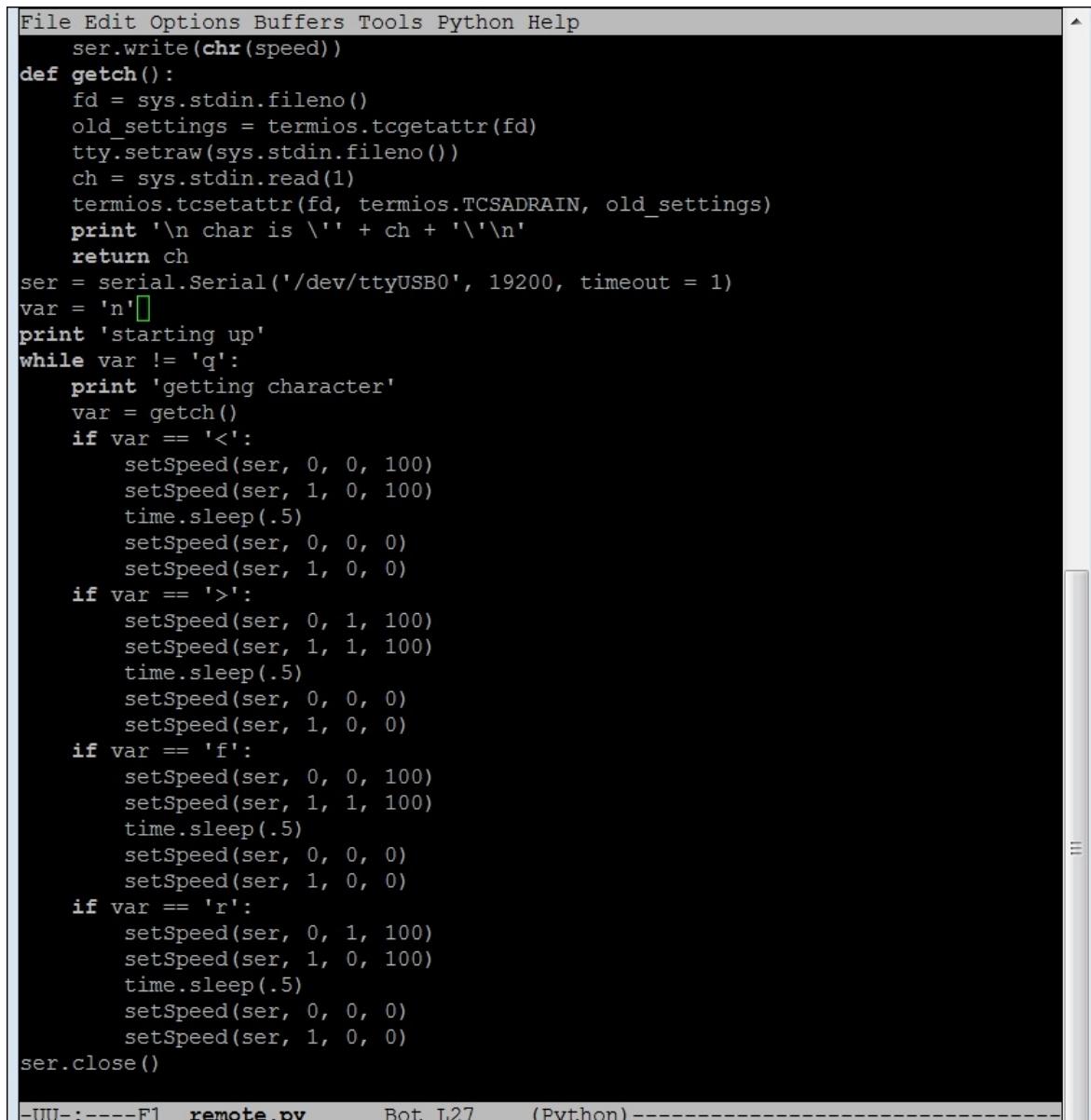
```
pi@raspberrypi: ~/track
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time
import tty
import sys
import termios
def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)
    ser.write(sendByte)
    ser.write(chr(speed))
def getch():
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    tty.setraw(sys.stdin.fileno())
    ch = sys.stdin.read(1)
    termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    print '\n char is \'' + ch + '\' \n'
    return ch
ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
var = 'n'
-UU-----F1  remote.py      Top L1      (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

The specifics of the preceding part of the file are listed as follows:

- You'll need to add `import tty`, `import sys`, and `import termios`. All these are the libraries you'll need for your function to work. The

`termios` library is the general I/O library used in order to get characters from the keyboard.

- The `setSpeed(ser, motor, direction, speed)`: function is unchanged from the `dcmotor.py` code.
- The `def getch():` function gets a single character without requiring the *Enter* key after each key stroke. The `print` statement in the function is optional; you can use it to map the different keys of the keyboard.
- The next set of changes are shown in the following screenshot:



The screenshot shows a terminal window with the following content:

```
File Edit Options Buffers Tools Python Help
ser.write(chr(speed))
def getch():
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    tty.setraw(sys.stdin.fileno())
    ch = sys.stdin.read(1)
    termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
    print '\n char is \'' + ch + '\'\\n'
    return ch
ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
var = 'n'█
print 'starting up'
while var != 'q':
    print 'getting character'
    var = getch()
    if var == '<':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == '>':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'f':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'r':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
ser.close()

-UU-----F1  remote.py      Bot L27      (Python)-----
```

- The `var = getch()` statement calls the function that returns the

character, without having to type the *Enter* key. Now, it is important to note that the program changes the terminal settings. So, when you run your program, you can no longer stop the program by typing *Ctrl + C*. You'll have to type `q` to restore your terminal settings.

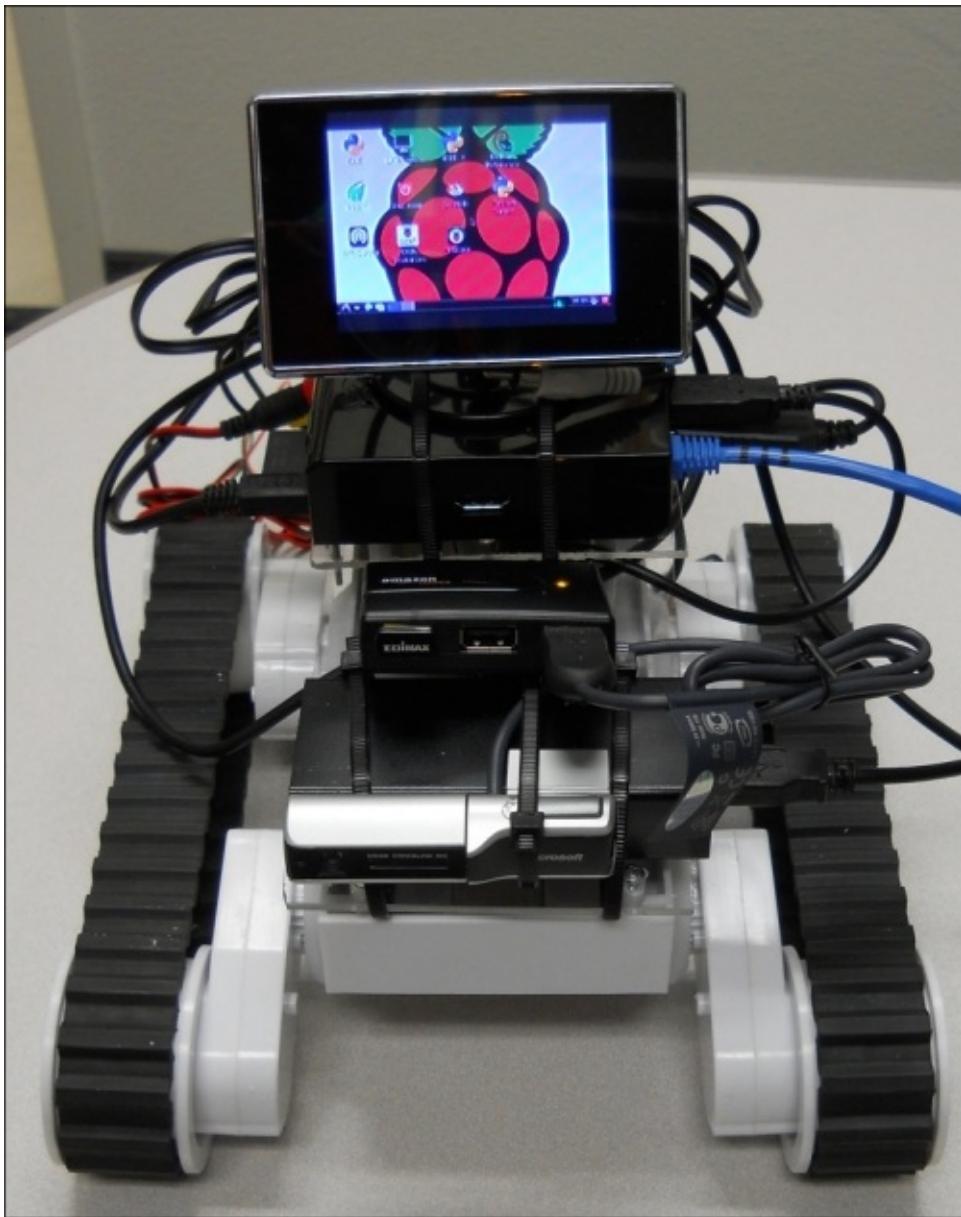
Many users are comfortable using gaming keypads. There are several that come with a wireless connection. You could try connecting one of these to your robot if you want it to seem more like a real video game. In my system, I used the wireless keypad by HausBell and mapped the arrow keys at the top of the keyboard to tell my robot to go forward, backward, left, and right. I figured out which key strokes these translated to by simply running my program and looking at the `print` statement in the program. You can also add additional functionality to the program to stop the car if the program senses if it has been a long time since a key stroke has come in, in case you've lost connection to the keyboard.

Working remotely with your Raspberry Pi through a wireless LAN

The last section showed you how to control your robot projects via a wireless USB keyboard. But, what if you want more flexibility? You may not only want to control your robot but also monitor it, see what it is picturing, and so on. The easiest way to do this is using a wireless LAN connection.

To configure a wireless LAN connection, start by inserting the wireless LAN card into Raspberry Pi. Here is where it gets a bit tricky; as you'll need a wireless LAN USB connection, a USB connection to your motor controller, and you'll also want to connect your USB webcam or perhaps a distance sensor. You'll need to use a USB hub in order to connect all of these. So connect the hub to Raspberry Pi and then connect the devices to the hub. You may want to choose a powered USB hub as it will give better power supply to the USB wireless LAN device.

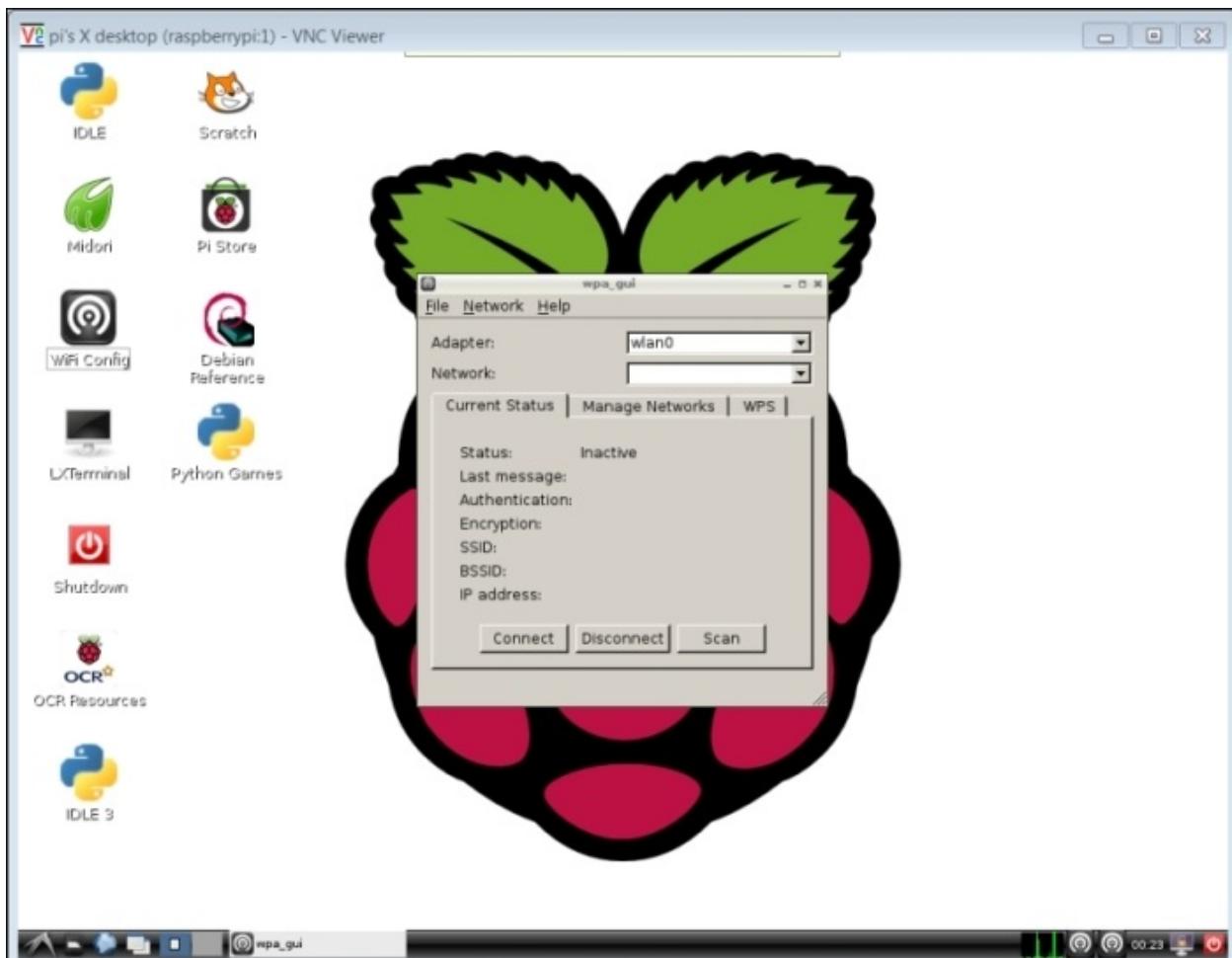
The following is an image of the platform with the hub and the devices connected:



Make sure you use a powered USB hub as you'll be talking to a USB camera and a wireless LAN device that can take a bit of power. Additionally, you'll need to provide a network so that your robot can connect. You can use an available network if there is one in the area you are working in and you have access to any passwords. I like to create a dedicated wireless LAN by connecting a wireless router to my laptop, as I can take this configuration anywhere. The following is an image of this configuration:

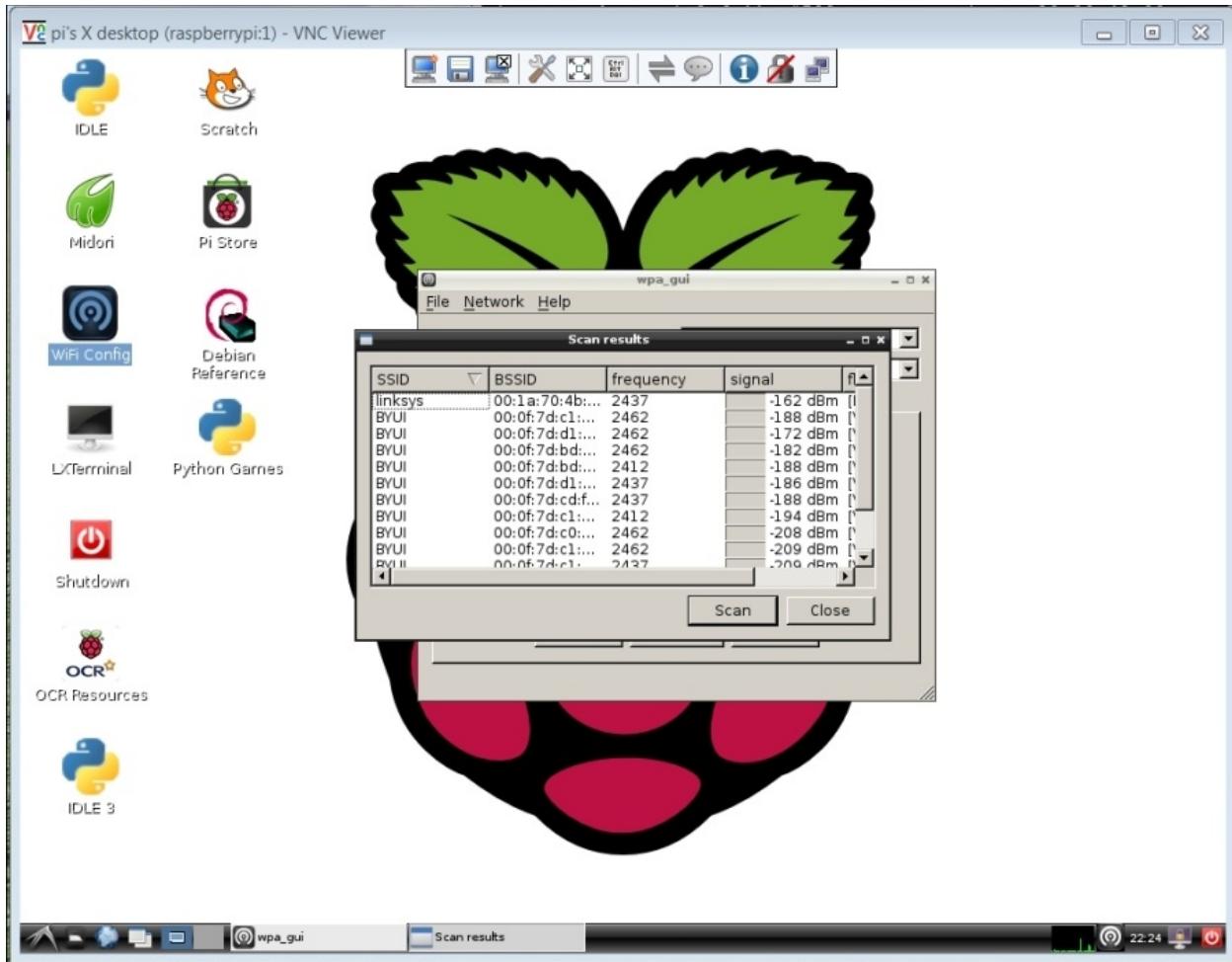


We are not going to walk through the mentioned configuration. It will be heavily dependent on your router, but we can just set up a simple, unsecured network since we don't use it all the time. Once you have connected all the devices, turn on the power to the robot. You'll need your screen and wireless keyboard to set up the wireless LAN network. Once the system has been booted, you should see the **WiFi Config** icon on the desktop of Raspberry Pi; click on it and you should see the following screenshot:

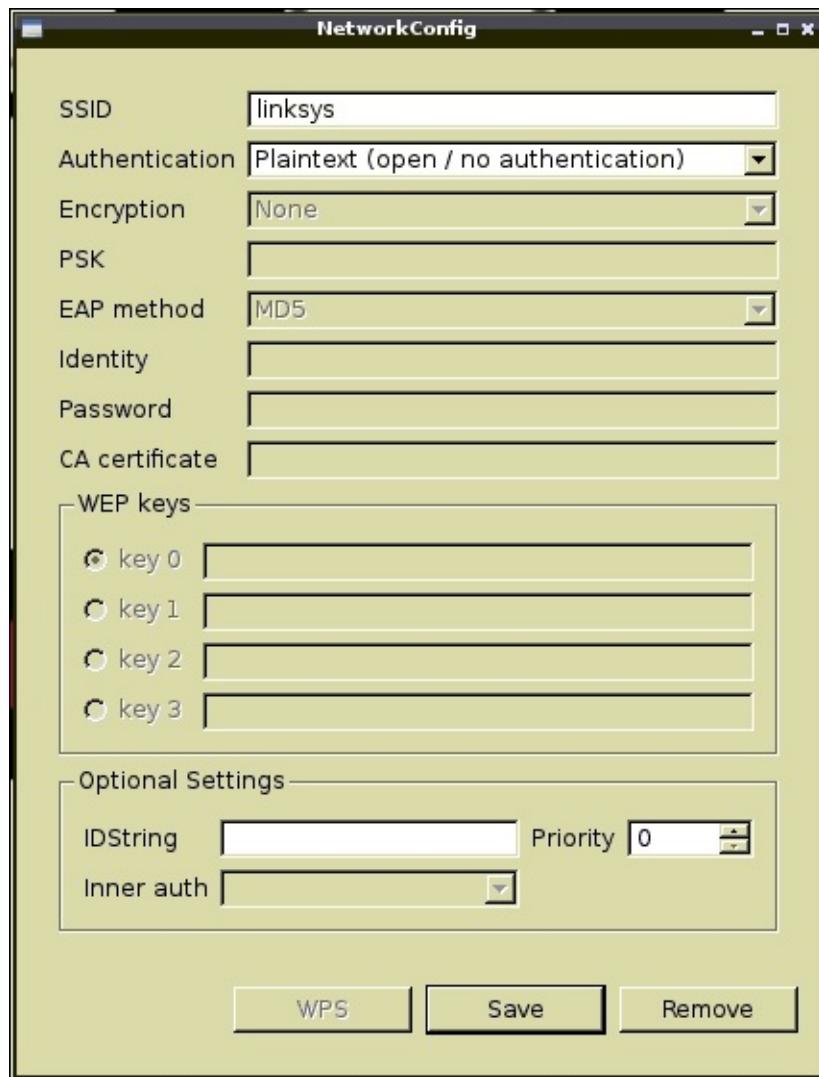


Now you are close to establishing a connection to the network. If no adapter appears in the **Adapter:** option, check to make sure that the devices are plugged into the USB hub and the hub is plugged into Raspberry Pi.

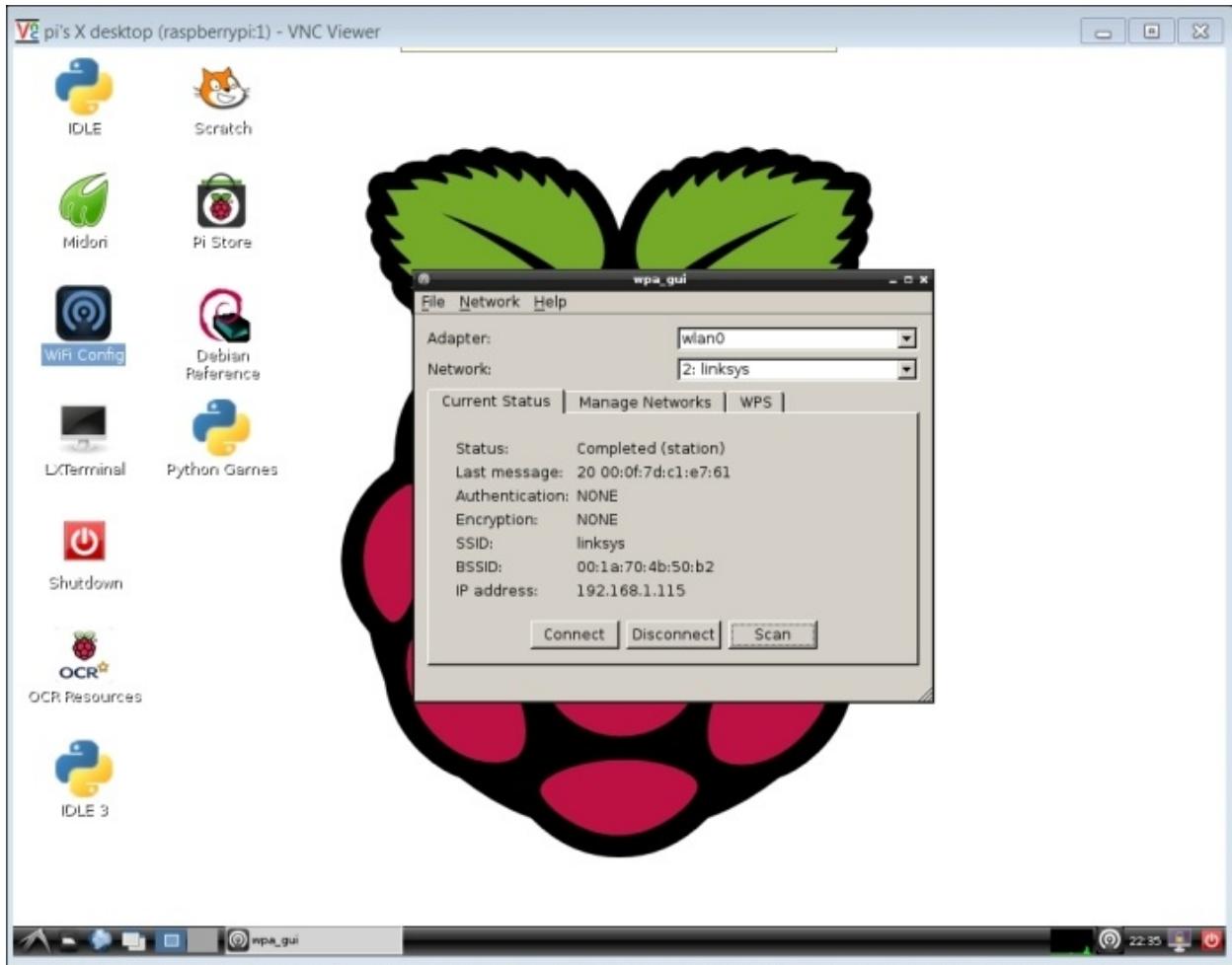
Click on the **Scan** button as shown in the following screenshot of the pane:



As you can note in the preceding screenshot, the Linksys network is available, as are a number of secure networks. Now, if you select [linksys](#), it will ask you to enter the specific network configuration parameters. I use the defaults for my open network as follows:

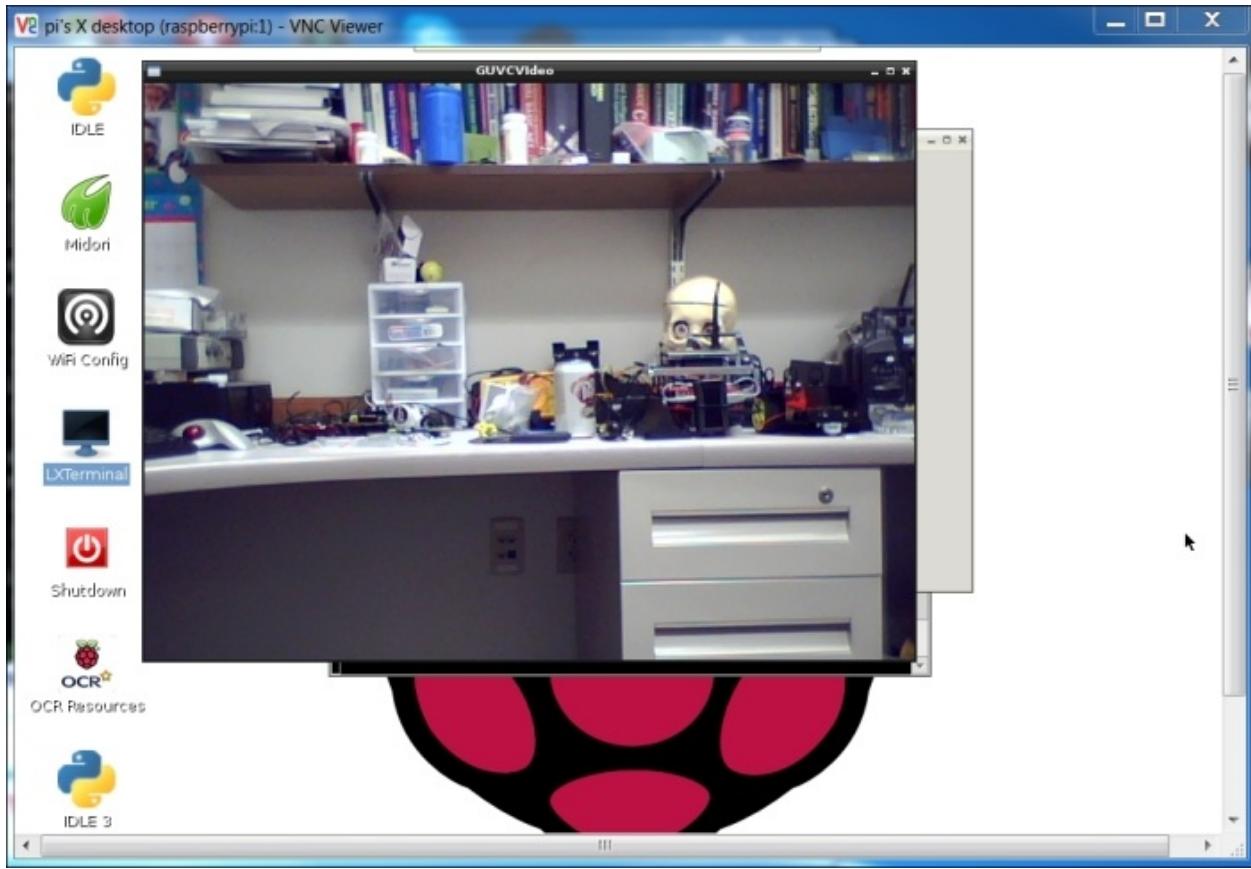


If you go back and click on **Connect**, it will connect to the network and give you an IP address as follows:



You can also configure the system using terminal commands. There are several tutorials that show you how to configure; try pingbin.com/2012/12/setup-wifi-raspberry-pi/ or <http://learn.adafruit.com/adafruits-raspberry-pi-lesson-3-network-setup/setting-up-wifi-with-occidentalis>.

Now you can use this tool from your laptop using PuTTY or the VNC server. The following is an image showing a **VNC Viewer** image of the webcam on Raspberry Pi communicated via the wireless LAN connection:



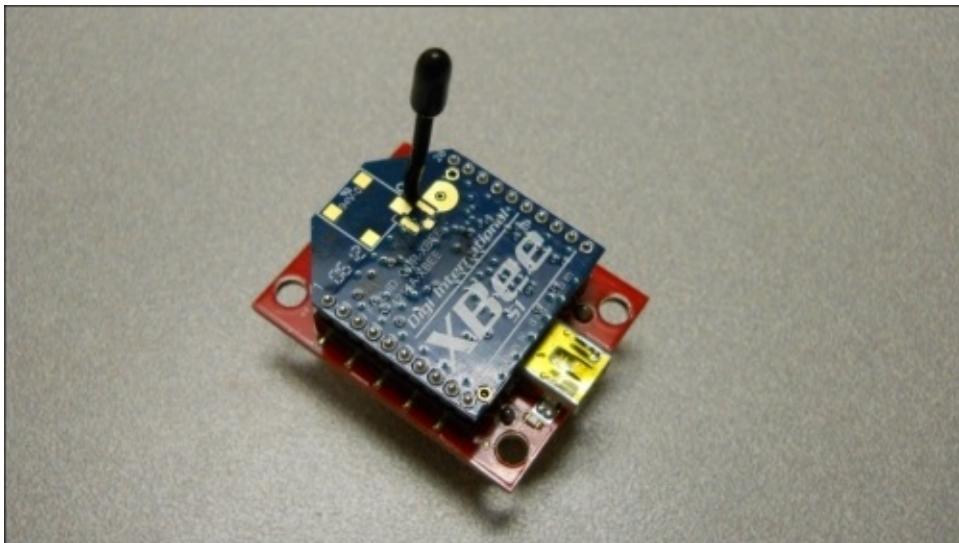
You can also do a similar sort of configuration using your smartphone in the hotspot mode. In this case, the cell phone is providing the wireless LAN connection. Connect both the laptop and Raspberry Pi to the hotspot using the information given to you on your cell phone. Now you can communicate via the SSH and VNC servers wirelessly.

Working remotely with your Raspberry Pi through ZigBee

Now you can remote to your device and control it via a wireless USB device as well as a wireless LAN connection. Now, let's look at a technology that can extend the wireless connection much further. The technology is ZigBee, and it is made for longer range wireless communications.

The ZigBee standard is built upon the IEEE 802.15.4 standard: a standard that was created to allow a set of devices to communicate with each other in order to enable a low data rate coordination of multiple devices. The ZigBee part of the standard ensures interoperability between the vendors of these low-rate devices. The IEEE 802.15.4 part of the standard specifies the physical interface, and the ZigBee part of the standard defines the network and applications interface. To find out more about ZigBee, try www.zigbee.org. Since we are only interested in the physical interface working together, you can buy IEEE 802.15.4 devices. But, ZigBee devices are a bit more prevalent, are supersets of IEEE 802.15.4, and are also quite inexpensive.

The other standard that you might hear as you try to purchase or use devices like these is XBee. This is a specific company's implementation, Digi, of several different wireless standards with standard hardware modules that can connect in many different ways to different embedded systems. They make some devices that support ZigBee; the following is an image of this type of device, which supports ZigBee attached to a shield that provides a USB port:



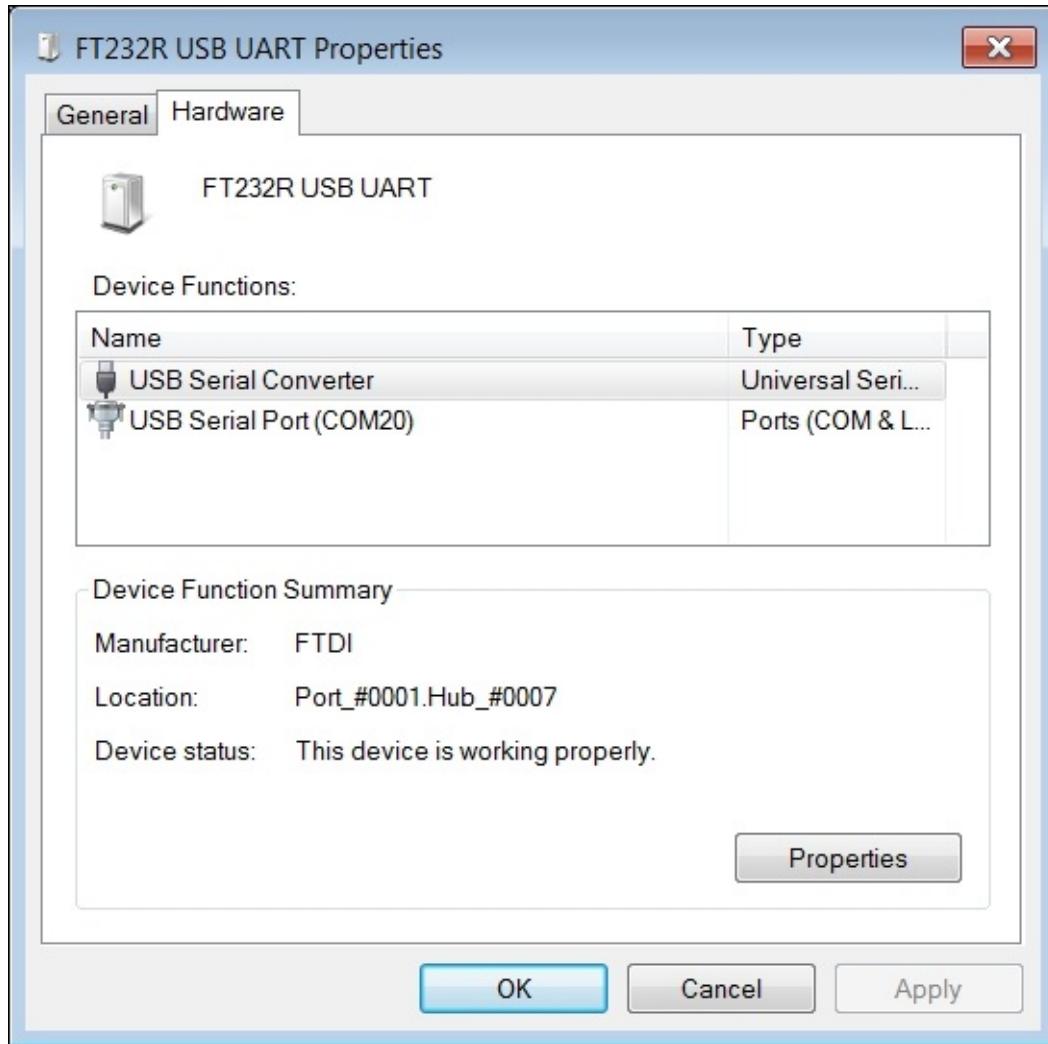
As noted at the beginning of this chapter, you will be learning how to use this specific device. The advantage of using this device is that it is configured to make it very easy to create and manage a simple link between two XBee Series 1 devices. Make sure you have an XBee device that supports ZigBee Series 1. You'll also need to purchase a shield that provides a USB port connection to the device.

Now, let's get started with configuring your two devices to make them talk. I'll give an example here using Windows and a PC. A Linux user can do something similar but using a Linux terminal program. An excellent tutorial is available at <http://web.univ-pau.fr/~cpham/WSN/XBee.html>.

If you are using Windows, plug one of the devices into your personal computer. Your computer should find the latest drivers for the device. You should see your device when you click on the **Devices and Printers** option from the **Start** menu as follows:



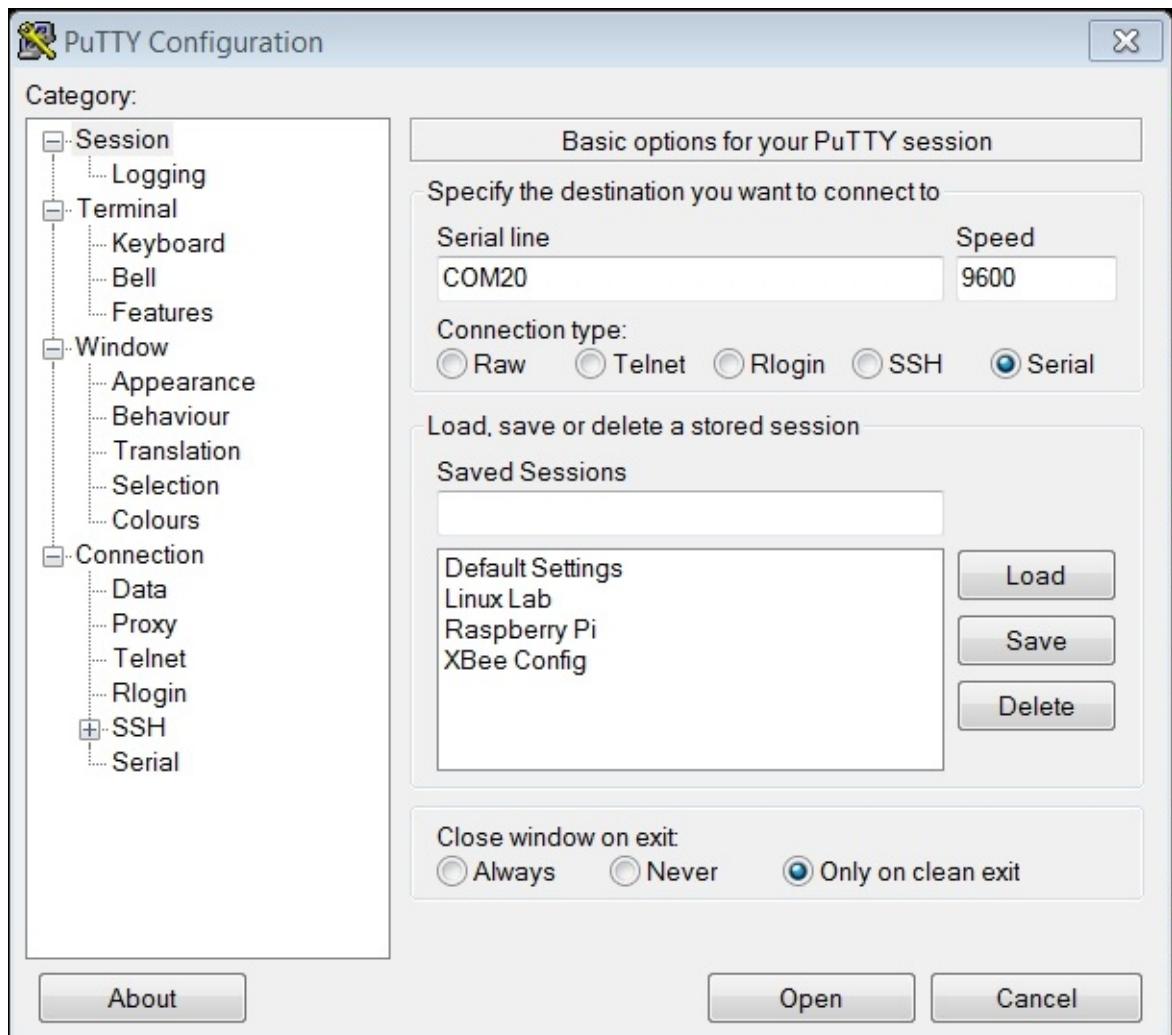
The device is now available to communicate with via the IEEE 802.15.4 wireless interface. We could set up a full ZigBee compliant network, but we're just going to communicate from one device to another directly, so we'll just use the device as a serial port connection. Double-click on the device icon and then select the **Hardware** tab; you should see the following screenshot:



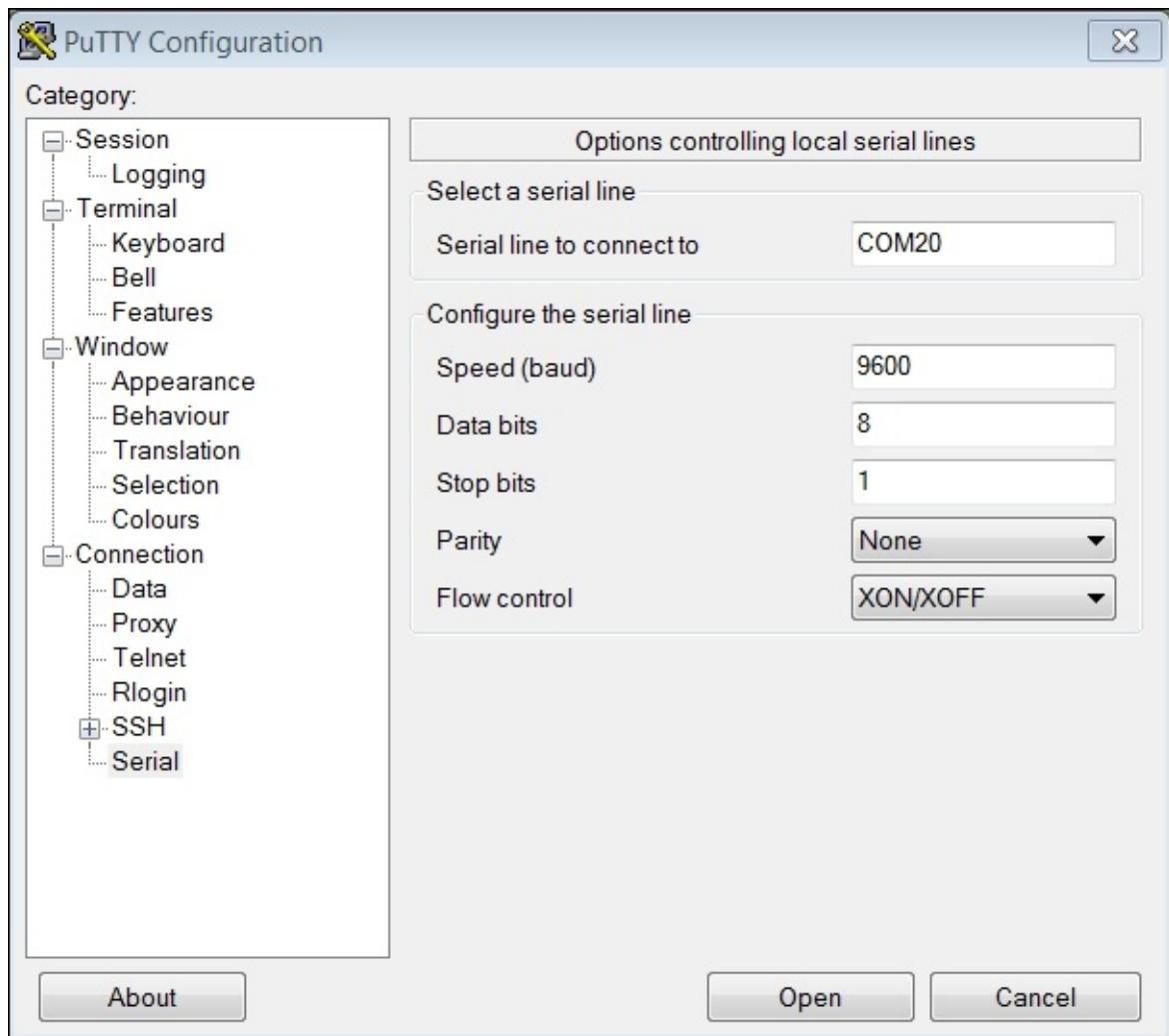
Note that the device is connected to the **COM20** serial port. We'll use this to communicate with the device and configure it. You can use any terminal emulator program; I like to use PuTTY, which is already on my computer.

Perform the following steps to configure the device:

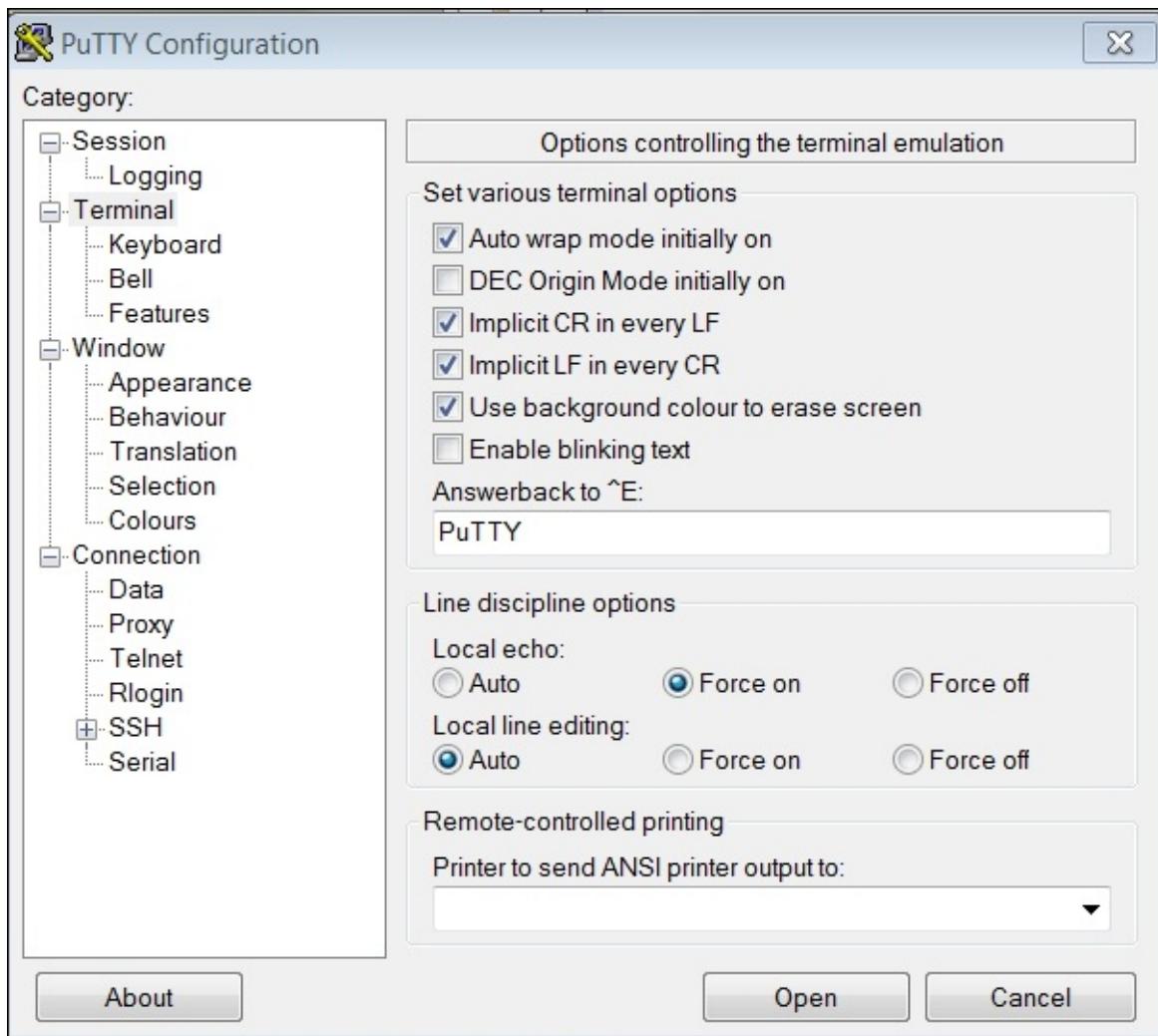
1. Open PuTTY and select the **Serial** option and (in this case) the **COM20** port. The following screenshot shows how to fill in the PuTTY window to configure the device:



- Configure the following parameters in terminal window (the **Serial** option in the **Category:** selection set): Baudrate, **9600**; the **Data bits** option, **8**; **Parity, None**, and the **Stop bits** option, **1** as follows:



3. Make sure you also select **Force on** for the **Local echo** option and check the **Implicit CR in every LF** and **Implicit LF in every CR** options available under the **Terminal** tab of the **Category:** selection set as follows:



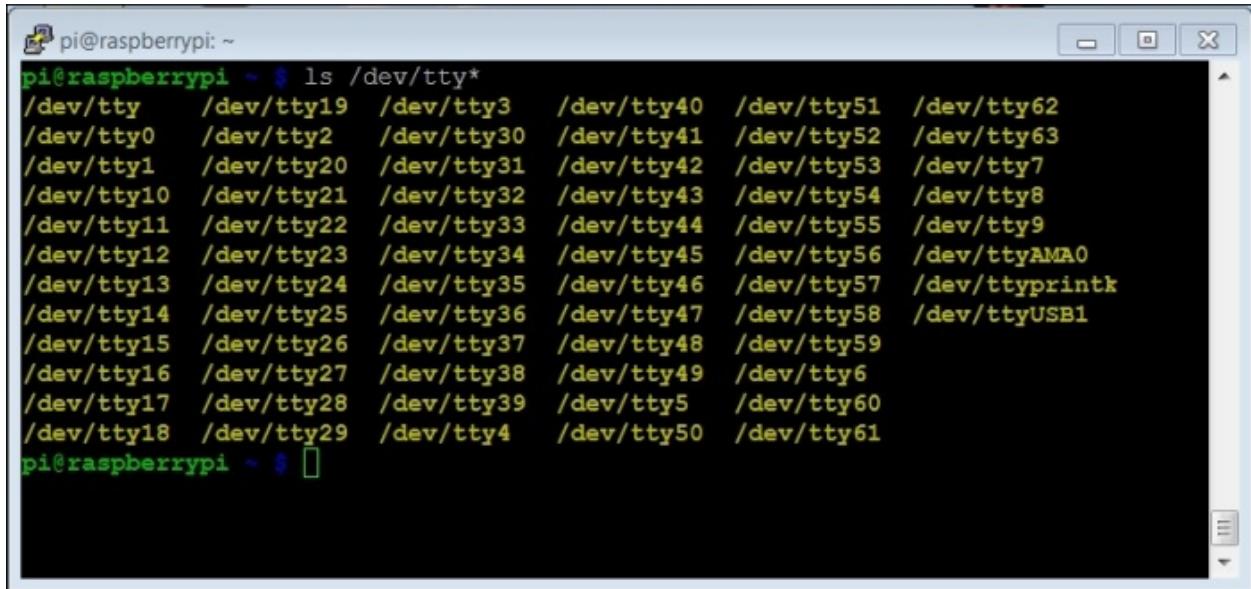
4. Connect to the device by clicking on **Open**.
5. Enter the commands to the device through the terminal window, as shown in the following screenshot:

```
COM20 - PuTTY
+++OK
ATID3001
OK
ATMY1
OK
ATDH0
OK
ATDL2
OK
ATID
3001
ATMY
1
ATDH
0
ATDL
2
ATWR
OK
[green square]
```

6. The response **OK** comes from the device as you enter each command. Now plug the other device into the PC. Note that it might choose a different **COM** port; click on the **Devices and Printers** option, double-click on the device's icon, and select the **Hardware** tab to find the **COM** port. Follow the same steps to configure the second device, except there are two changes. The following is a screenshot of the terminal window for these commands:

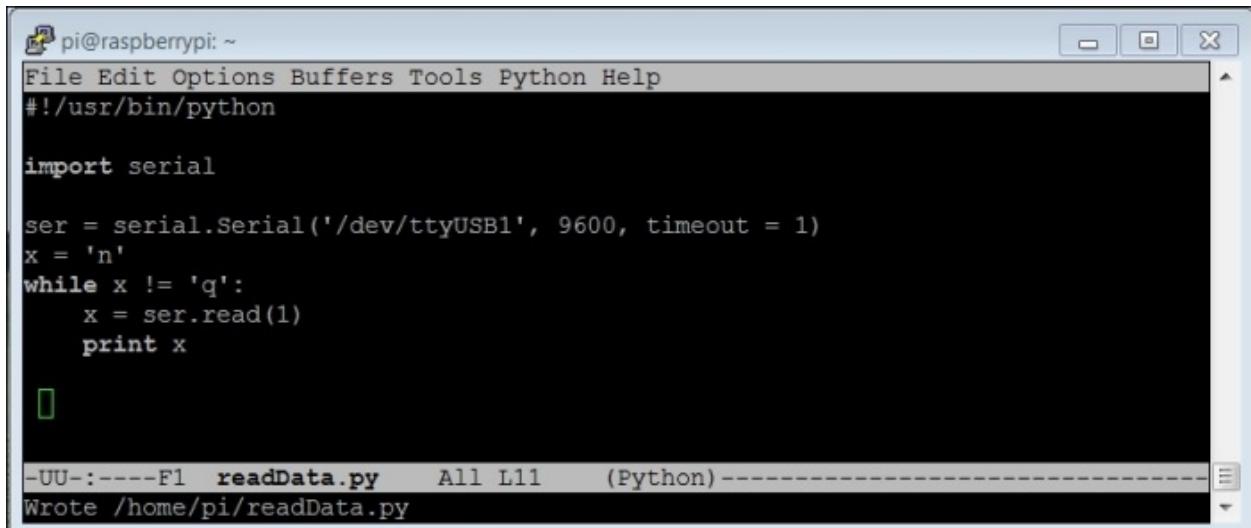
```
COM21 - PuTTY
+++OK
ATID3001
OK
ATMY2
OK
ATDH0
OK
ATDL1
OK
ATID
3001
ATMY
2
ATDH
0
ATDL
1
ATWR
OK
[green square]
```

The devices are now ready to talk to one another. Plug one of the devices into the Raspberry Pi USB port. Using a terminal window, show the devices that are connected by typing `ls /dev/tty*`. It will look something like what is shown in the following screenshot:



```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ls /dev/tty*
/dev/tty      /dev/tty19  /dev/tty3    /dev/tty40  /dev/tty51  /dev/tty62
/dev/tty0     /dev/tty2   /dev/tty30   /dev/tty41  /dev/tty52  /dev/tty63
/dev/tty1     /dev/tty20  /dev/tty31   /dev/tty42  /dev/tty53  /dev/tty7
/dev/tty10    /dev/tty21  /dev/tty32   /dev/tty43  /dev/tty54  /dev/tty8
/dev/tty11    /dev/tty22  /dev/tty33   /dev/tty44  /dev/tty55  /dev/tty9
/dev/tty12    /dev/tty23  /dev/tty34   /dev/tty45  /dev/tty56  /dev/ttyAMA0
/dev/tty13    /dev/tty24  /dev/tty35   /dev/tty46  /dev/tty57  /dev/ttprintfk
/dev/tty14    /dev/tty25  /dev/tty36   /dev/tty47  /dev/tty58  /dev/ttyUSB1
/dev/tty15    /dev/tty26  /dev/tty37   /dev/tty48  /dev/tty59
/dev/tty16    /dev/tty27  /dev/tty38   /dev/tty49  /dev/tty6
/dev/tty17    /dev/tty28  /dev/tty39   /dev/tty5   /dev/tty60
/dev/tty18    /dev/tty29  /dev/tty4    /dev/tty50  /dev/tty61
pi@raspberrypi ~ $
```

Note that the device appears at `devttyUSB1`. Now, you'll need to create a Python program that will read the preceding input. The following screenshot depicts a listing of such a program:



```
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

import serial

ser = serial.Serial('/dev/ttyUSB1', 9600, timeout = 1)
x = 'n'
while x != 'q':
    x = ser.read(1)
    print x


```

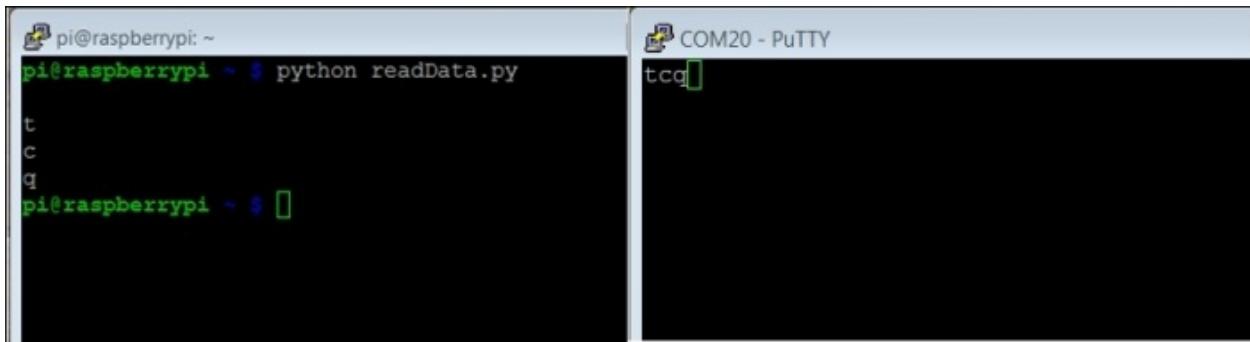
The following points explain the functionality of the code:

- The `#!/usr/bin/python` statement allows your program to run without

invoking Python on the command line

- The `import serial` statement imports the `Serial` port library
- The `ser = serial.Serial('devttyUSB1', 9600, timeout = 1)` statement opens a serial port pointing to the `devttyUSB1` port with a baud rate of `9600` and a timeout of `1`
- The `x = 'n'` statement defines a character variable and initializes it to `'n'`; so, we go through the loop at least once
- You will enter the `while` loop, `while x != 'q':`, until the user enters the character `q`
- The `x = ser.read(1)` statement reads 1 byte from the serial port
- The `print x` statement prints out the value

Now if you run `readData.py` in a terminal window and you have the PuTTY program on your personal computer connected to the other XBee module, you should see the characters, which you type on the personal computer terminal windows, come out on the terminal windows running on Raspberry Pi. The following are the two screenshots given side by side:



Connecting this functionality to your robot is very easy. Start with the `remote.py` program that you created earlier in the chapter. Copy this into a new program by typing `cp remote.py xbee.py`. Now, let's remove some of the code, parts that you don't need, and add a bit that will accept the character input from the XBee module. The following screenshot shows a listing of the code:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/track". The window contains a Python script named "xbee.py". The script defines a function "setSpeed" which takes four parameters: "ser", "motor", "direction", and "speed". It uses an if-elif-else ladder to determine the byte value based on the motor number and direction. The script then writes these bytes to the serial port "ser". It also defines two serial ports: "serInput" (9600 baud) and "ser" (19200 baud). A loop reads input from "serInput" and sends commands to "ser" based on the input. The input characters correspond to movement commands: '<' for left, '>' for right, 'f' for forward, and 'r' for reverse. After each movement command, the script performs a short sleep. Finally, it closes both serial ports.

```
pi@raspberrypi: ~/track
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial, time
def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)
    ser.write(sendByte)
    ser.write(chr(speed))
serInput = serial.Serial('/dev/ttyUSB0', 9600, timeout = 1)
ser = serial.Serial('/dev/ttyUSB1', 19200, timeout = 1)
var = 'n'
while var != 'q':
    var = serInput.read(1)
    if var == '<':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == '>':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'f':
        setSpeed(ser, 0, 0, 100)
        setSpeed(ser, 1, 1, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
    if var == 'r':
        setSpeed(ser, 0, 1, 100)
        setSpeed(ser, 1, 0, 100)
        time.sleep(.5)
        setSpeed(ser, 0, 0, 0)
        setSpeed(ser, 1, 0, 0)
ser.close()
serInput.close()
-UU-----F1  xbee.py      All L13  (Python)-----
Wrote /home/pi/track/xbee.py
```

There are only two meaningful changes as follows:

- `serInput = serial.Serial('devttyUSB0', 9600, timeout = 1)` – This statement sets up a serial port, getting an input from the XBee

device. It is important to note that the `USB0` and `USB1` settings might be different in your specific configuration based on whether the XBee serial device or the motor controller serial device configures first.

- `var = serInput.read(1)` – This statement, instead of getting the input from the user via the keyboard, you will be reading the characters from the XBee device.

That's it! Now your robot should respond to commands sent from your terminal window on our personal computer. You could also create an application on your personal computer that could turn mouse movements or other inputs into the proper commands for your robot.

Summary

Congratulations! Now you can take your robot out into the big wide world. You can even use the LCD and keyboard to make changes to your program, although the smaller screen size makes this a bit difficult. However, now that your robot is truly mobile, you may want to give it a sense of position and direction. The next chapter will show you how to add GPS to your robotic project.

Chapter 9. Using a GPS Receiver to Locate Your Robot

Assuming that you have completed the tasks from the previous chapters, you should now have mobile robots that can move around, accept commands, see, and even avoid obstacles. This chapter will help you locate your robot if it moves, which can be useful for a robot that is fully autonomous. Your robot is mobile, but let's not let it get lost. You're going to add a GPS receiver so that you can always—well, almost always—know where your robot is.

As you let your device loose, you may want it to not only know where it is, but also have a way of finding out if it has made it to the desired location. One of the coolest things to connect to your robot is a GPS location device. In this chapter, I'll show you how to connect a GPS receiver to your project and then use it to move in the correct direction.

In this chapter, we will cover the following topics:

- Connecting Raspberry Pi to a GPS device so that it can locate itself in relation to the world, at least where you can receive a GPS signal
- Accessing the GPS programmatically and using its position information to move the robot to a specific location

To complete the tasks in this chapter, you'll need a GPS device. There are a lot of options and they come with many different interfaces, but because we want to avoid using a soldering iron or other complex connection processes, we're going to choose one with a USB interface. For example, the following is an image of the device I have used for some of my projects:



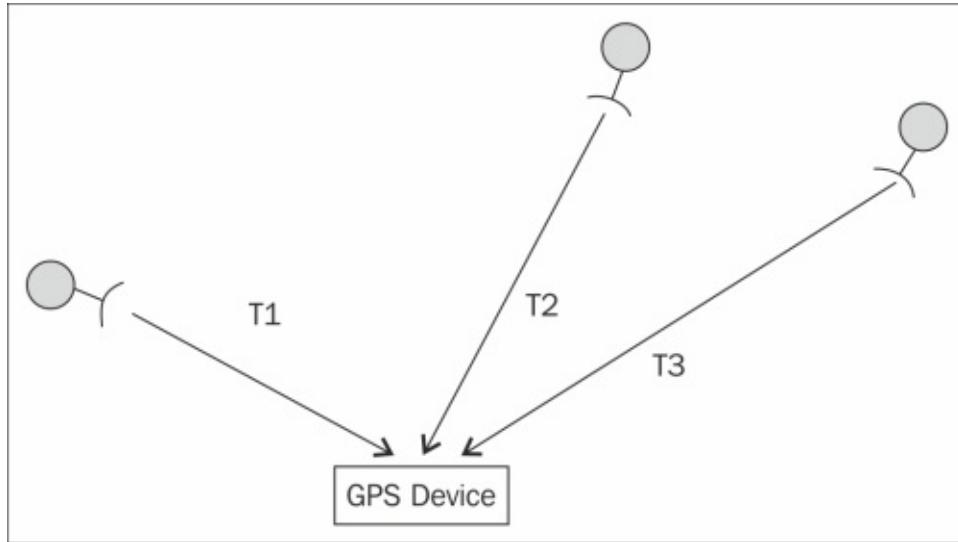
The model number of the device in the previous image is **ND-100S** from GlobalSat. It is small, inexpensive, and supports Windows, Mac OS X, and Linux, so our system should be able to interface with it. It is available on Amazon and other online electronics stores, so you should be able to get it almost anywhere. However, it does not have the sensitivity of some other GPS devices. So, if you will be using your robot in buildings or other locations that might stifle GPS signals, you should look for devices that are more sensitive to the signals from the GPS satellites.

Connecting Raspberry Pi to a GPS device

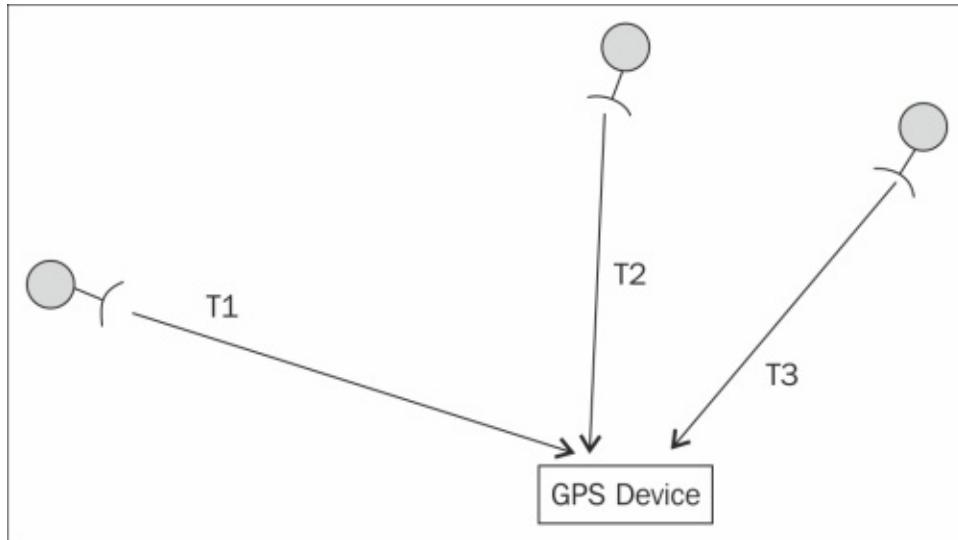
Before we get started, let me first give you a brief tutorial on GPS. **GPS**, which stands for **Global Positioning System**, is a system of satellites that transmit signals. GPS devices use these signals to calculate a position of an object. There are a total of 24 satellites transmitting signals all around the earth at any given moment, but your device can only "see" the signal from a much smaller set of satellites.

Each of these satellites transmits a very accurate time signal that your device can receive and interpret. It receives the time signal from each of these satellites and then based on the delay—the time it takes the signal to reach the device—it calculates the receiver's position based on a

procedure called triangulation. The following two diagrams illustrate how a device uses the difference between the delay data from three satellites to calculate its position:

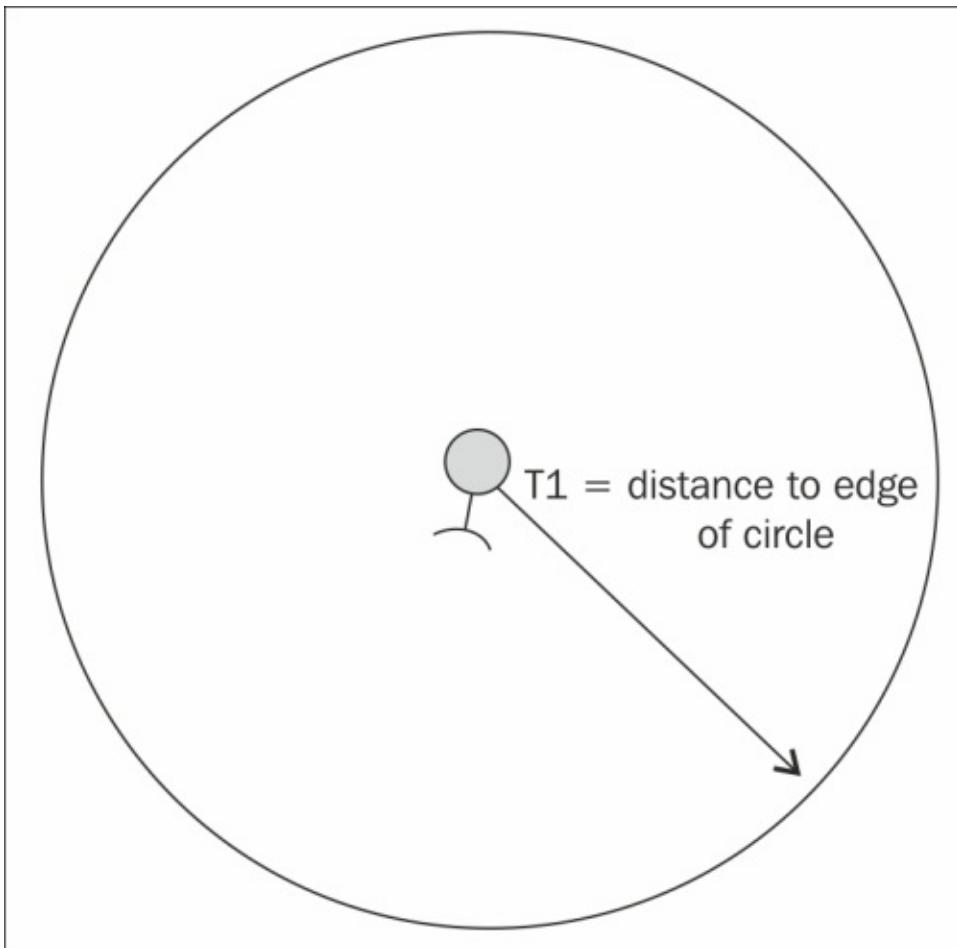


The GPS device is able to detect the three signals and the time delays associated with receiving them. In the following diagram, the device is at a different location and the time delays associated with the three signals have changed from those in the previous diagram:

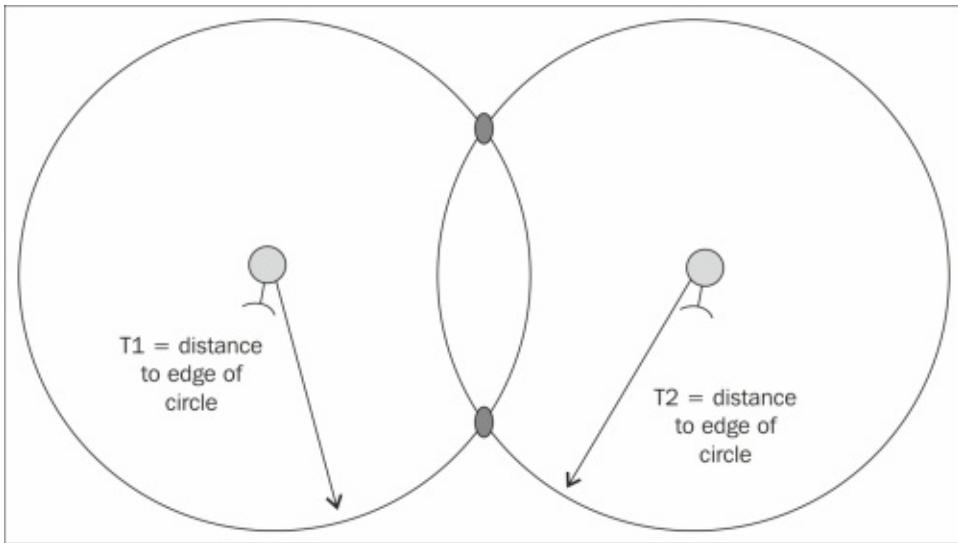


The time delays of the signals **T1**, **T2**, and **T3** can provide the GPS with an absolute position using triangulation. Since the positions of the satellites are known, the amount of time that the signal takes to reach the GPS device is also a measure of the distance between that satellite and

the GPS device. To simplify this concept, let's see an example in two dimensions. If the GPS device knows its distance from one satellite based on the amount of time delay, we could draw a circle around the satellite at that distance and know that our GPS device is on the boundary of that sphere, as shown in the following diagram:



If we have two satellites' signals and know the distance between them, we can draw two circles as shown in the following diagram:

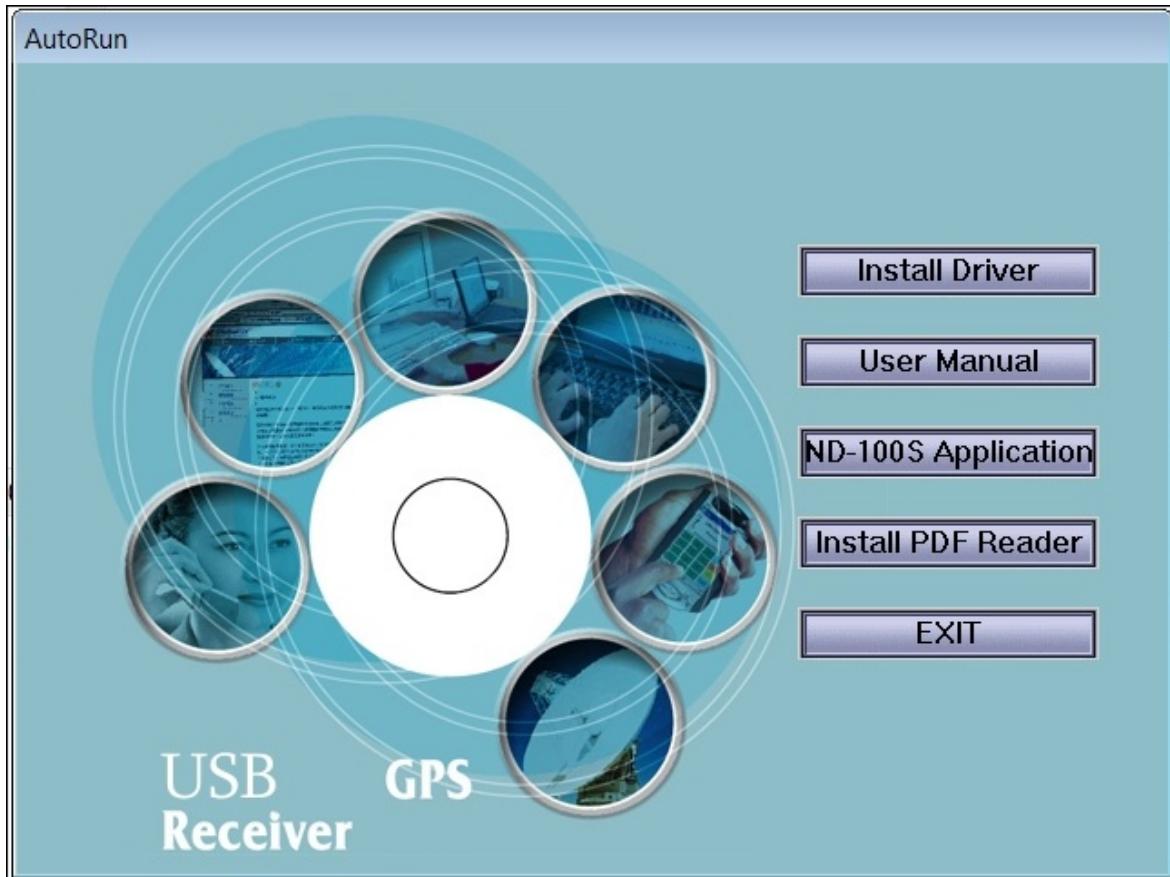


However, we know that since we can only be at points on the boundary of the circle, we must be at one of the two points that are on the boundary of both circles. Adding an additional satellite would eliminate one of these two points, providing us our exact location. We need more satellites if we are going to do this in all three dimensions.

Now it's time to connect the device. While this is optional, I suggest that you connect a dongle to your PC. This will let you know whether or not the unit works and help you understand the device a little better. Then, you'll connect it to Raspberry Pi.

In order to install the GPS system on your PC, perform the following steps:

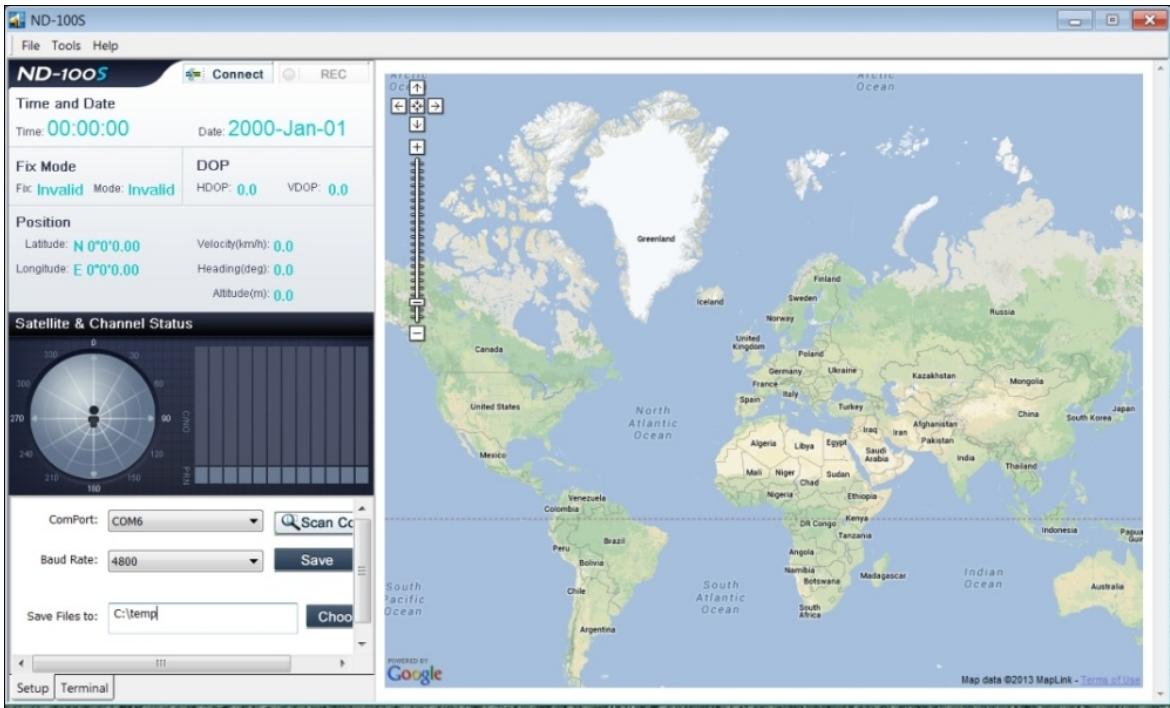
1. Insert the CD and run the setup program. You should see a window as shown in the following screenshot:



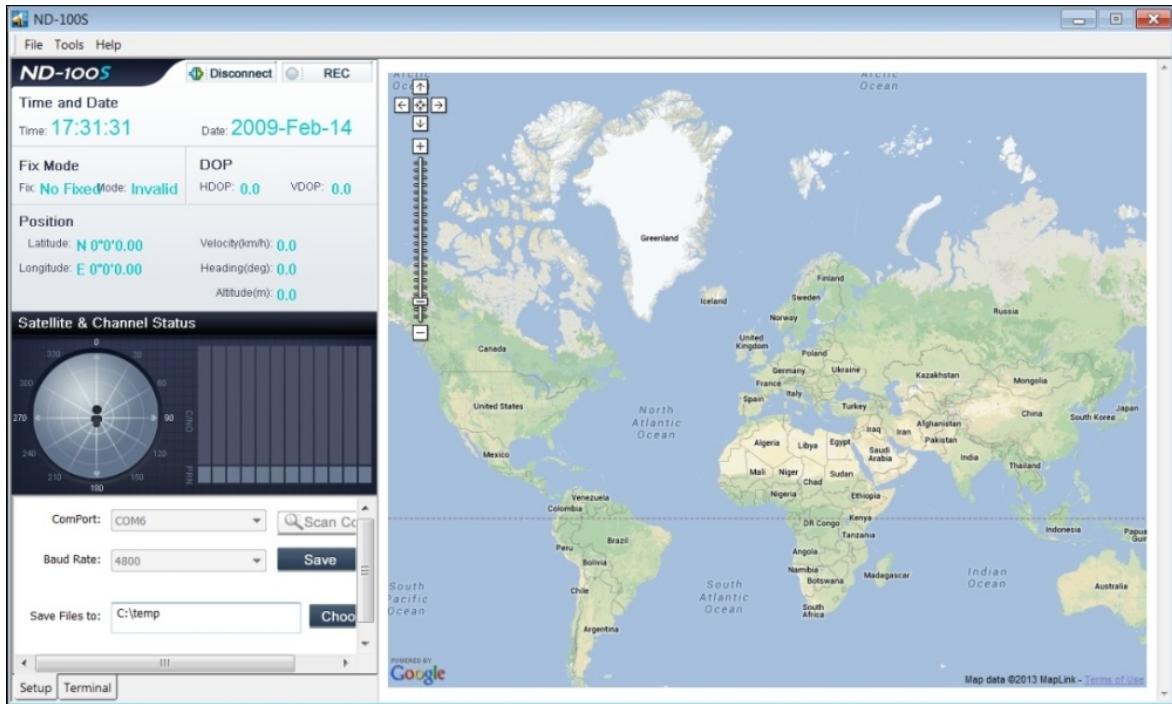
2. Click on both the **Install Driver** and **ND-100S Application** buttons and follow the default instruction procedures. If you have installed both the drivers and the application, you should be able to plug the GPS device into the USB port on your PC. A blue light at the end of the device should indicate that the device has been plugged in. The system will recognize the device, install the appropriate drivers, and give you access to it (this may take a few minutes). To ensure that the device has been installed, check your **Devices and Printers** start menu selection (if you're running Windows 7). You should see the following screenshot:



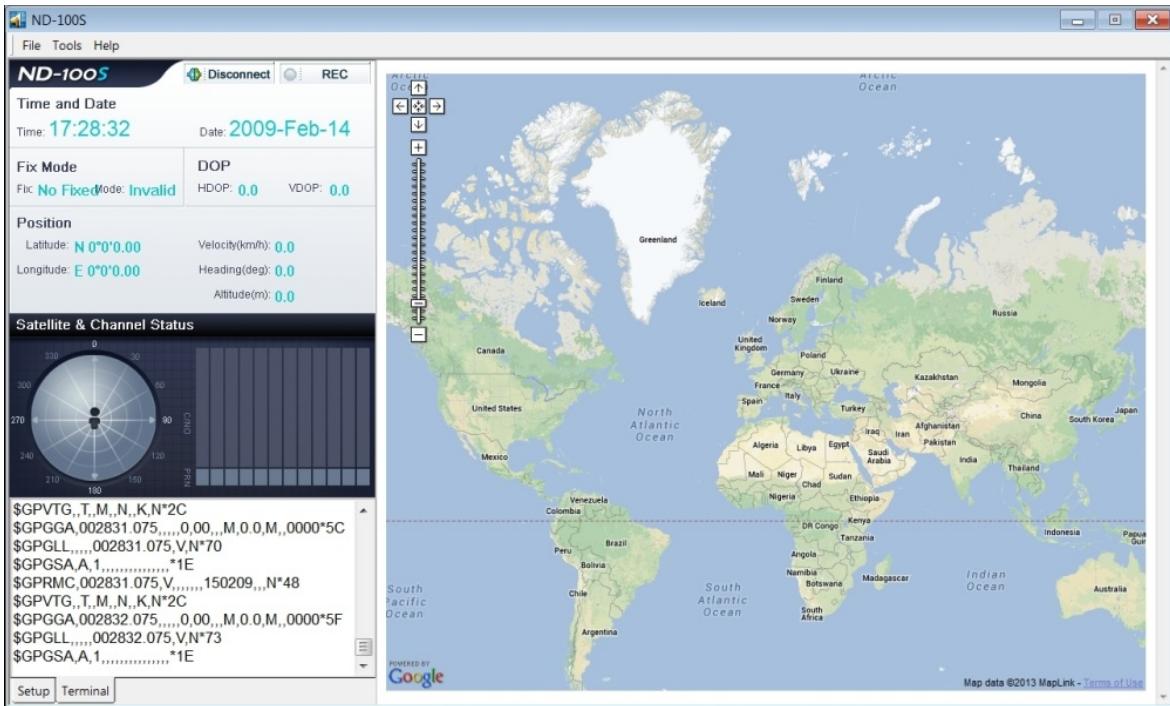
- Once the device is installed, you can also run the application that is available on the CD-ROM. At startup, it should look as shown in the following screenshot:



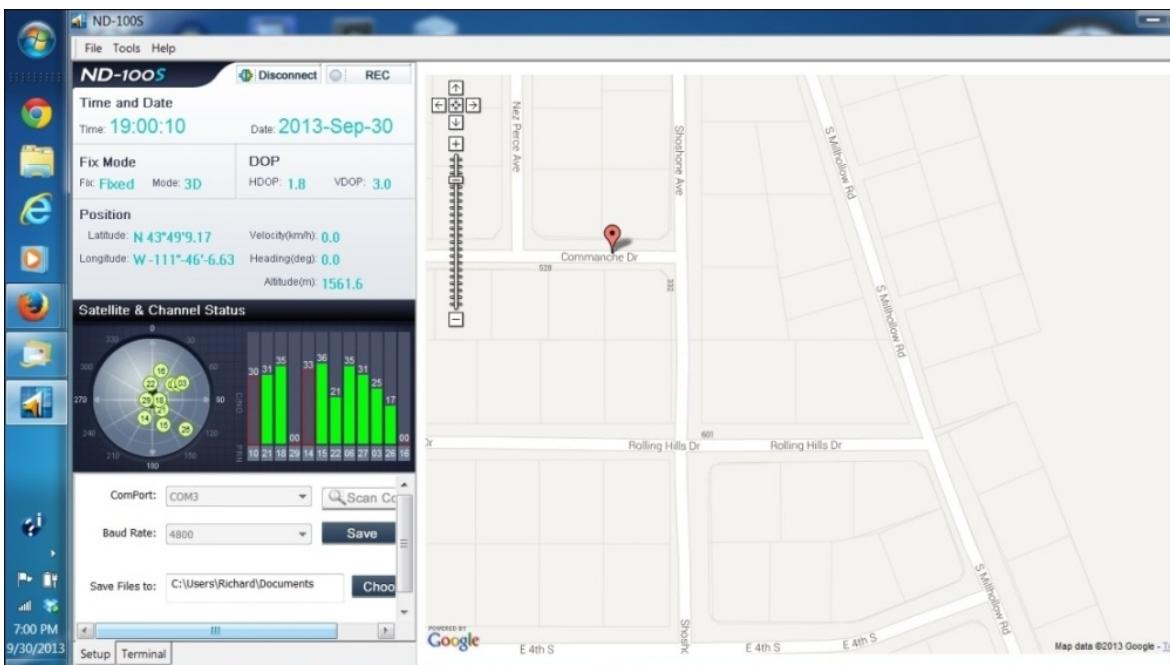
4. Now click on the **Connect** button to the top-left of the screen. It should now look as shown in the following screenshot:



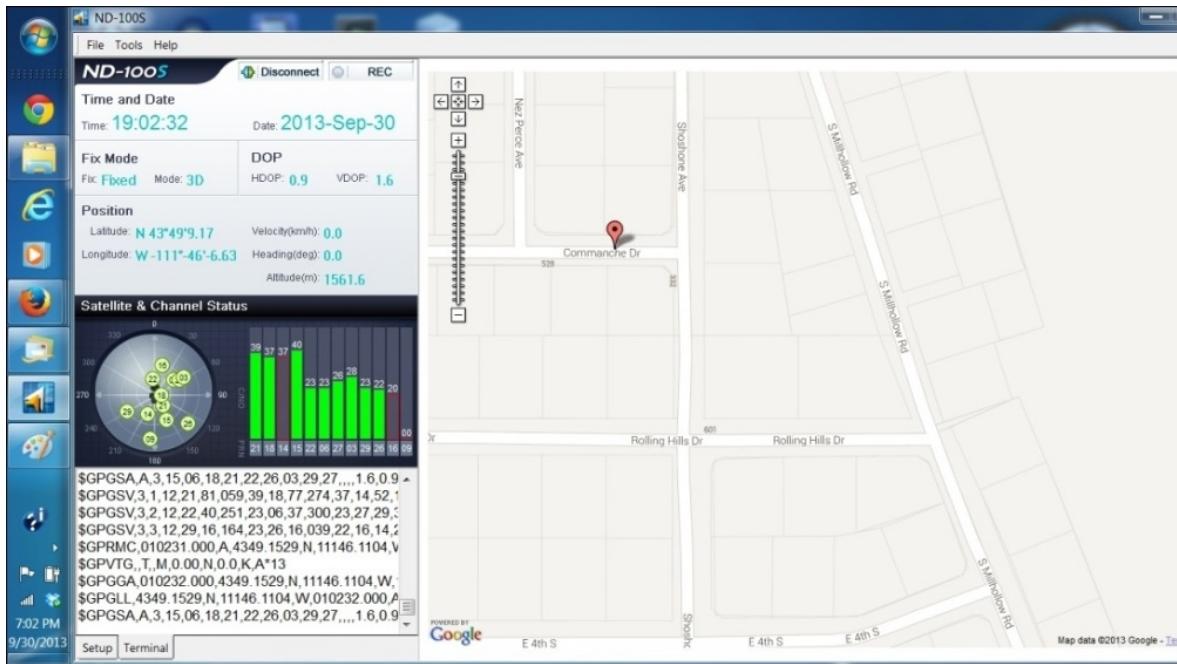
5. Unfortunately, if you are in a building or a place where receiving information from the GPS satellites is difficult, the device may struggle to find its position. If you want to find out whether or not the system is working even though it may struggle to find signals, select the **Terminal** tab selection in the lower-left corner of the screen. You should see what is shown in the following screenshot:



6. Notice that the lower-left window indicates that the device is trying to find its location. Initially, the unit in my office was unable to locate the satellites, which is not surprising when you're in a building designed to restrict the transmission of signals in and out of the building. Following the same procedure on my laptop shows the following screenshot as a result:



7. You'll notice that the blue LED on the end of the GPS device is flashing. Now we have our position. When I select the **Terminal** tab, it shows the raw data returned by the GPS device:



We'll use that raw data from the previous screenshot in our next section to plan our path to other positions. So, in an environment where GPS data is available, the unit is able to sync up with it and show your position. The next step will be to hook it to your Raspberry Pi robot.

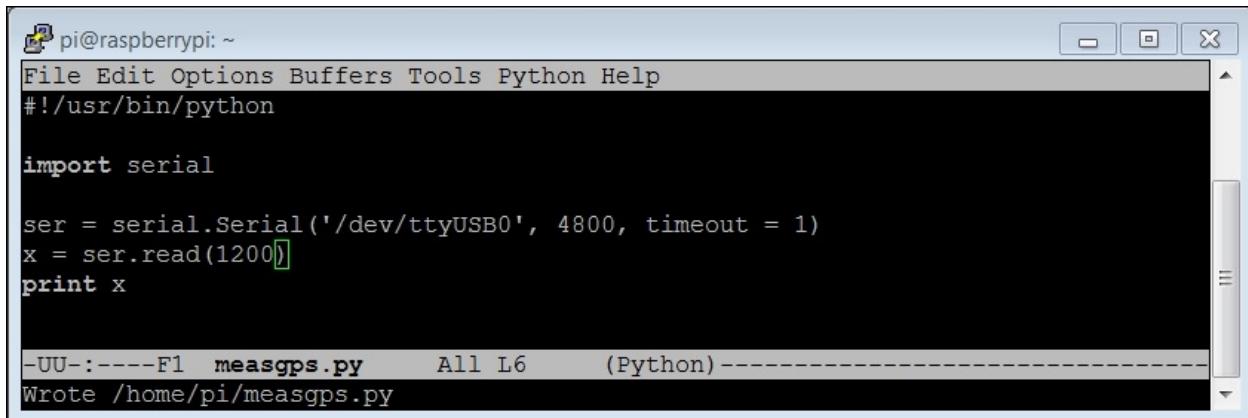
First, connect the GPS unit by plugging it into one of the free USB ports on the USB hub. Once it is plugged in and the unit is rebooted, type `lsusb` and you should see the output shown in the following screenshot:

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ lsusb
Bus 001 Device 002: ID 0424:9512 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 018: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port
pi@raspberrypi ~ $
```

The device is shown as [Prolific Technology, Inc. PL2303 Serial Port](#).

Your device is now connected to your Raspberry Pi.

Now create a simple Python program that will read the value from the GPS device. If you are using Emacs as an editor, type `emacs measgps.py`. A new file will be created called `measgps.py`. Then, type the code shown in the following screenshot:



The screenshot shows an Emacs window titled "pi@raspberrypi: ~". The menu bar includes File, Edit, Options, Buffers, Tools, Python, and Help. The buffer contains the following Python code:

```
#!/usr/bin/python

import serial

ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)
x = ser.read(1200)
print x
```

The status bar at the bottom indicates "-UU-----F1 measgps.py All L6 (Python)-----" and "wrote /home/pi/measgps.py".

Let's go through the code to see what is happening:

- `#!/usr/bin/python` – As discussed earlier, this line simply makes this file available for us to execute from the command line.
- `import serial` – We import the `serial` library. This will allow us to interface the USB GPS sensor with the GPS system.
- `ser = serial.Serial('devttyUSB0', 4800, timeout = 1)` – The first command sets up the serial port to use the `devttyUSB0` device, which is our GPS sensor using a baud rate of 4800 and a timeout of one second.
- `x = ser.read(1200)` – The next command then reads a set of values from the USB port. In this case, we read 1200 bytes; this includes a fairly complete set of our GPS data.
- `print x` – The final command then prints out the value obtained from the previous command.

Once you have created this file, you can run the program and talk to the device. Do this by typing `python measgps.py` and the program will run. You should see the output as shown in the following screenshot:

```

$GPGSA,A,1,,,,,,,,,,,*1E
$GPRMC,001712.037,V,,,,,,150209,,,N*43
$GPVTG,,T,,M,,N,,K,N*2C
$GPGGA,001713.037,,,,0,00,,,M,0.0,M,,0000*56
$GPGLL,,,,,001713.037,V,N*7A
$GPGSA,A,1,,,,,,,,,,,*1E
$GPRMC,001713.037,V,,,,,,150209,,,N*42
$GPVTG,,T,,M,,N,,K,N*2C
$GPGGA,001714.037,,,,0,00,,,M,0.0,M,,0000*51
$GPGLL,,,,,001714.037,V,N*7D
$GPGSA,A,1,,,,,,,,,,,*1E
$GPRMC,001714.037,V,,,,,,150209,,,N*45
$GPVTG,,T,,M,,N,,K,N*2C
$GPGGA,001715.037,,,,0,00,,,M,0.0,M,,0000*50
$GPGLL,,,,,001715.037,V,N*7C
$GPGSA,A,1,,,,,,,,,,,*1E
$GPGSV,1,1,00*79
$GPRMC,001715.037,V,,,,,,150209,,,N*44
$GPVTG,,T,,M,,N,,K,N*2C
$GPGGA,001716.037,,,,0,00,,,M,0.0,M,,0000*53
$GPGLL,,,,,001716.037,V,N*7F
$GPGSA,A,1,,,,,,,,,,,*1E
$GPRMC,001716.037,V
pi@raspberrypi ~ $

```

The device returns raw readings to you, which is a good sign. Unfortunately, there isn't much good data here, as the robot is again indoors. How do we know this? Look at one of the lines that start with **\$GPRMC**; this line should tell us our current latitude and longitude values. The GPRS reports the following code:

\$GPRMC,001714.037,V,,,,,,150209,,,N*45

The previous line of data should take the form shown in the following table, with each field separated by a comma:

0	1	2	3	4	5	6	7	8	9	10	11	12
\$GPRMC	220516	A	5133.82	N	00042.24	W	173.8	231.8	130694	004.2	W	*7

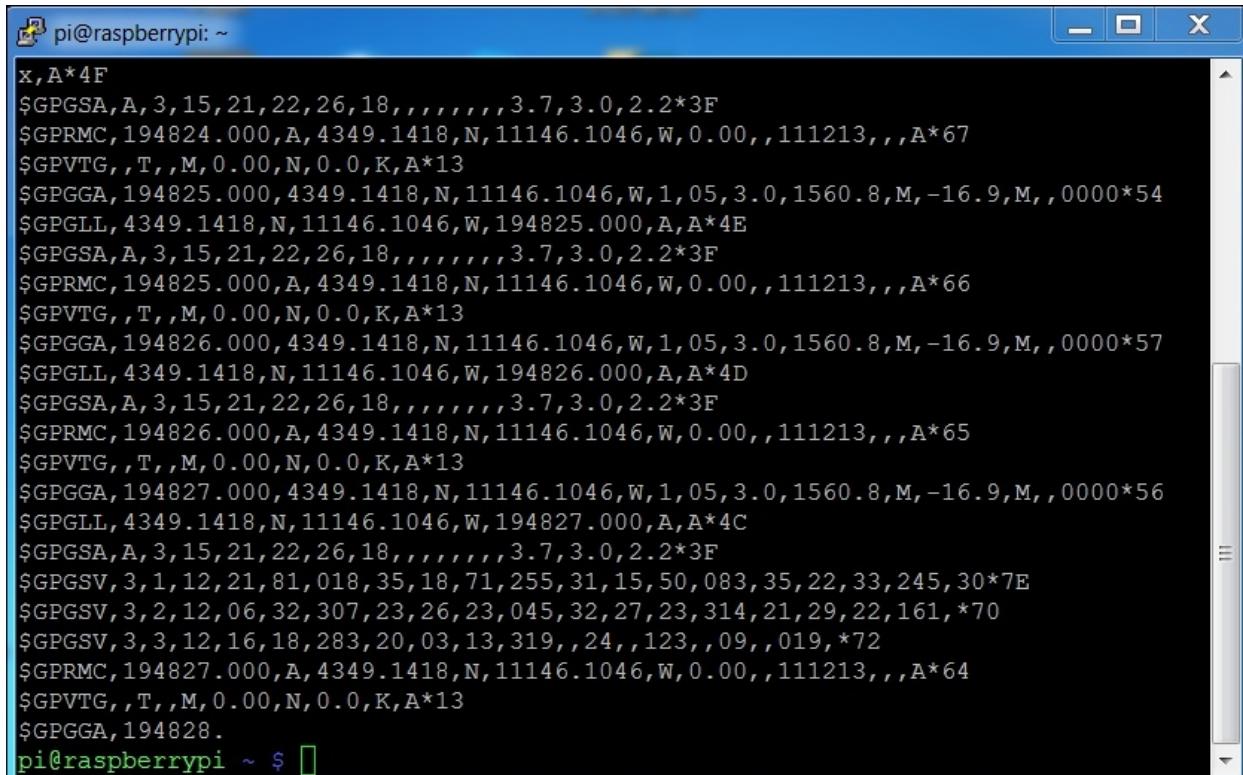
The following table offers an explanation of each of the fields shown in

the previous table:

Field	Value	Explanation
1	220516	Timestamp
2	A	Validity: A (OK), V (invalid)
3	5133.82	Current latitude
4	N	North or south
5	00042.24	Current longitude
6	W	East or west
7	173.8	Speed in knots at which you are moving
8	3	Course: The angular direction in which you are moving
9	130694	Date stamp
10	0004.2	Magnetic variation: The variation between magnetic and true north
	W	

11		East or west
12	*70	Checksum

In our case, the field #2 either reports v or that the unit cannot find enough satellites to get a position. Take the unit outdoors and we can get the result shown in the following screenshot from our [measgps.py](#) program:



```

pi@raspberrypi: ~
x,A*4F
$GPGSA,A,3,15,21,22,26,18,,,,,,3.7,3.0,2.2*3F
$GPRMC,194824.000,A,4349.1418,N,11146.1046,W,0.00,,111213,,,A*67
$GPVTG,,T,,M,0.00,N,0.0,K,A*13
$GPGGA,194825.000,4349.1418,N,11146.1046,W,1,05,3.0,1560.8,M,-16.9,M,,0000*54
$GPGLL,4349.1418,N,11146.1046,W,194825.000,A,A*4E
$GPGSA,A,3,15,21,22,26,18,,,,,,3.7,3.0,2.2*3F
$GPRMC,194825.000,A,4349.1418,N,11146.1046,W,0.00,,111213,,,A*66
$GPVTG,,T,,M,0.00,N,0.0,K,A*13
$GPGGA,194826.000,4349.1418,N,11146.1046,W,1,05,3.0,1560.8,M,-16.9,M,,0000*57
$GPGLL,4349.1418,N,11146.1046,W,194826.000,A,A*4D
$GPGSA,A,3,15,21,22,26,18,,,,,,3.7,3.0,2.2*3F
$GPRMC,194826.000,A,4349.1418,N,11146.1046,W,0.00,,111213,,,A*65
$GPVTG,,T,,M,0.00,N,0.0,K,A*13
$GPGGA,194827.000,4349.1418,N,11146.1046,W,1,05,3.0,1560.8,M,-16.9,M,,0000*56
$GPGLL,4349.1418,N,11146.1046,W,194827.000,A,A*4C
$GPGSA,A,3,15,21,22,26,18,,,,,,3.7,3.0,2.2*3F
$GPGSV,3,1,12,21,81,018,35,18,71,255,31,15,50,083,35,22,33,245,30*7E
$GPGSV,3,2,12,06,32,307,23,26,23,045,32,27,23,314,21,29,22,161,*70
$GPGSV,3,3,12,16,18,283,20,03,13,319,,24,,123,,09,,019,*72
$GPRMC,194827.000,A,4349.1418,N,11146.1046,W,0.00,,111213,,,A*64
$GPVTG,,T,,M,0.00,N,0.0,K,A*13
$GPGGA,194828.
pi@raspberrypi ~ $ 
```

Notice that the \$GPRMC line now reads as follows:

\$GPRMC,194827.000,A,4349.1418,N,11146.1046,W,0.00,,111213,,,A*64

Our values will now be as shown in the following table:

Field	Value	Explanation

1	020740.000	Timestamp
2	A	Validity: A (OK), V (invalid)
3	4349.1426	Current latitude
4	N	North or south
5	11146.1064	Current longitude
6	W	East or west
7	1.82	Speed in knots at which you are moving
8	214.11	Course: The angular direction in which you are moving
9	021013	Date stamp
10		Magnetic variation: The variation between magnetic and true north
11		East or west
12	*7B	Checksum

Now you have some indication of where you are; however, the GPS data is in raw form that may not mean much. In the next section, we will figure out how to do something with these readings.

Accessing the GPS programmatically

Now that you can access your GPS device, let's work on accessing the data programmatically. Your project should now have the GPS device connected and have access to query the data via the serial port. In this section, you will create a program to use this data to discover where you are and then you can determine what to do with that information.

If you've completed the previous section, you should be able to receive the raw data from the GPS unit. Now you want to be able to do something with this data, for example, find your current location and altitude and then decide whether your target location is to the west, east, north, or south.

First, get the information from the raw data. As noted previously, our position and speed is in the [\\$GPMRC](#) output of our GPS device. We will first write a program to simply parse out a couple of pieces of information from that data. So open a new file (you can name it [location.py](#)) and edit it as shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window contains Python code for reading GPS data from a USB port. The code uses the serial library to read data from "/dev/ttyUSB0" at 4800 baud, timeout of 1 second. It then finds the first occurrence of the "\$GPRMC" string and isolates the line. The line is split by commas, and if the third element is 'V', it prints "No location found". Otherwise, it prints the latitude, longitude, speed, and course. The terminal window also shows the file path "/home/pi/location.py" and the command "(Python)".

```
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

import serial

ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)
x = ser.read(1200)
pos1 = x.find("$GPRMC")
pos2 = x.find("\n", pos1)
loc = x[pos1:pos2]
data = loc.split(',')
if data[2] == 'V':
    print 'No location found'
else:
    print "Latitude = " + data[3] + data[4]
    print "Longitude = " + data[5] + data[6]
    print "Speed = " + data[7]
    print "Course = " + data[8]

-UU:----F1  location.py   All L17  (Python)-----
wrote /home/pi/location.py
```

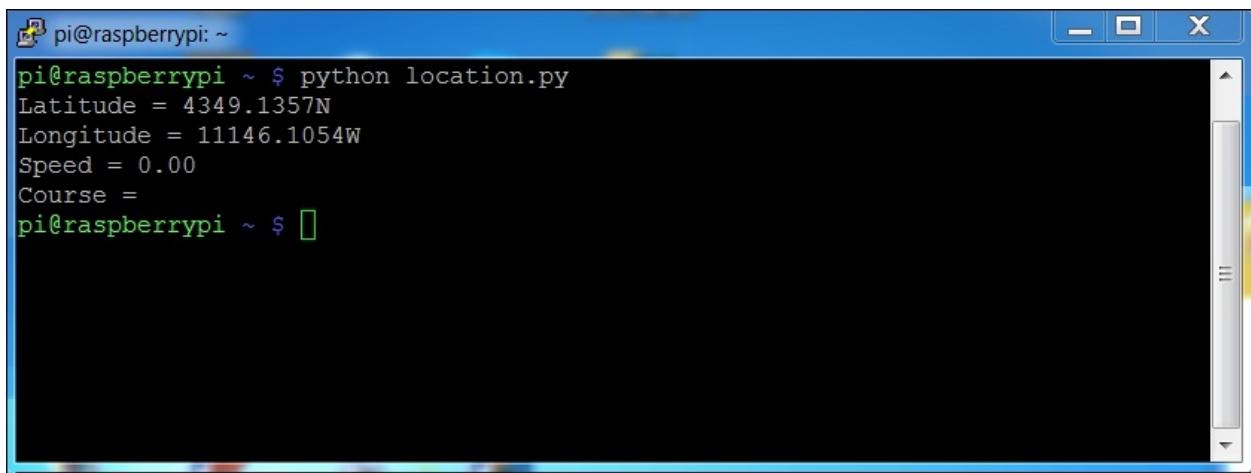
The code lines are explained as follows:

- `#!/usr/bin/Python` – As always, this line simply makes this file available for you to execute from the command line.
- `import serial` – You again import the `serial` library. This will allow you to interface the USB GPS sensor with the GPS system.
- `if __name__=="__main__":` – The main part of your program is then defined using this line.
- `ser = serial.Serial('devttyUSB0', 4800, timeout = 1)` – The first command sets up the serial port to use the `devttyUSB0` device, which is your GPS sensor using a baud rate of 4800 and a timeout value of one second.
- `x = ser.read(500)` – The next command then reads a set of values from the USB port. In this case, you read 500 values, which includes a fairly complete set of your GPS data.
- `pos1 = x.find("$GPRMC")` – This will find the first occurrence of `$GPRMC` and set the value `pos1` to that position. In this case, you want to isolate the `$GPRMC` response line.
- `pos2 = x.find("\n", pos1)` – This will find the end of this line.
- `loc = x[pos1:pos2]` – The variable `loc` will now hold the path

including all the information you are interested in.

- `data = loc.split(',') – This will break your comma-separated line into an array of values.`
- `if data[2] == 'V': – You now check to see whether or not the data is valid. If not, the next line simply prints out that you did not find a valid location.`
- `else: – If the data is valid, the next few lines print out the various pieces of data.`

The following screenshot is an example showing the result that appeared when my device was able to find its location:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window contains the following text output:

```
pi@raspberrypi ~ $ python location.py
Latitude = 4349.1357N
Longitude = 11146.1054W
Speed = 0.00
Course =
pi@raspberrypi ~ $
```

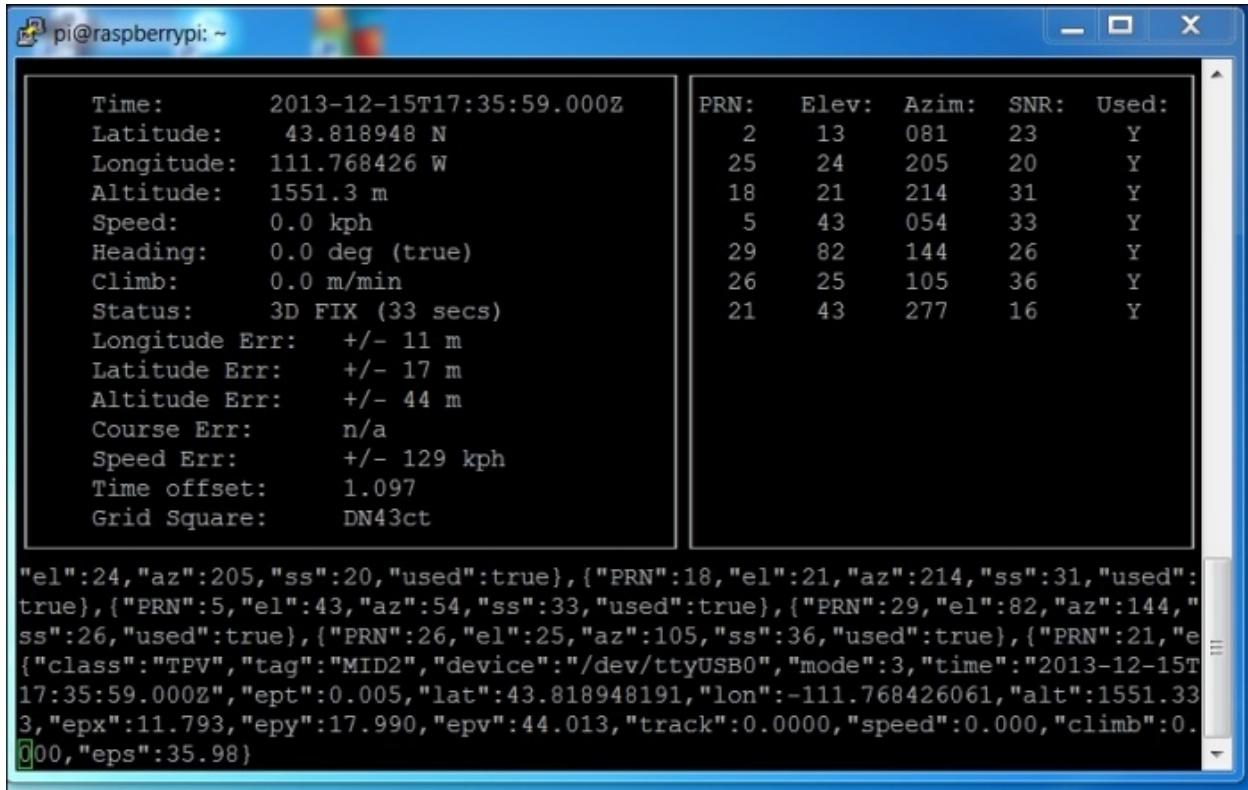
Once you have the data, you can do some interesting things with it. For example, you might want to figure out the distance from and direction to another waypoint. There is a piece of code at

<http://code.activestate.com/recipes/577594-GPS-distance-and-bearing-between-two-GPS-points/> that you can use to find the distances from and bearings to other waypoints based on your current location. You can easily add this code to your `location.py` program to update your robot on the distances and bearings to other waypoints.

Now your robot knows where it is and the direction it needs to go to get to other locations! There is another way to configure your GPS device that may make it a bit easier to access the data from other programs; it is using a set of functionality held in the `gpsd` library. To install this capability, type `sudo apt-get install gpsd gpsd-clients` and this will install the `gpsd` software. For a tutorial on this software, use

http://wiki.ros.org/gpsd_client/Tutorials/Getting%20Started%20with%20gp

This software works by starting a background program (called a daemon) that communicates with your GPS device. We can then just query that program to get the GPS data. To start the process, type `sudo gpsd devttyUSB0 -F varrun/gpsd.sock`. You can run the program by typing `cgps`. The following screenshot shows a sample result:

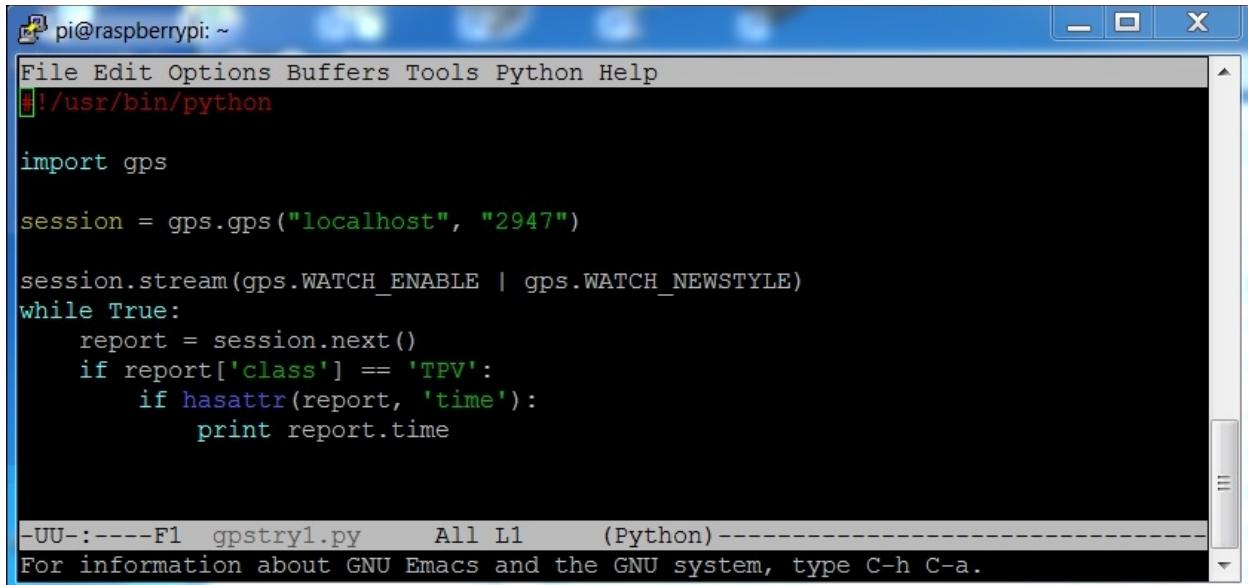


The screenshot shows a terminal window titled "pi@raspberrypi: ~". The window displays two columns of GPS data. The left column contains formatted data, and the right column contains raw data. The raw data at the bottom is a JSON object representing the GPS fix.

Time:	2013-12-15T17:35:59.000Z	PRN:	Elev:	Azim:	SNR:	Used:
Latitude:	43.818948 N	2	13	081	23	Y
Longitude:	111.768426 W	25	24	205	20	Y
Altitude:	1551.3 m	18	21	214	31	Y
Speed:	0.0 kph	5	43	054	33	Y
Heading:	0.0 deg (true)	29	82	144	26	Y
Climb:	0.0 m/min	26	25	105	36	Y
Status:	3D FIX (33 secs)	21	43	277	16	Y
Longitude Err:	+/- 11 m					
Latitude Err:	+/- 17 m					
Altitude Err:	+/- 44 m					
Course Err:	n/a					
Speed Err:	+/- 129 kph					
Time offset:	1.097					
Grid Square:	DN43ct					

```
{"el":24,"az":205,"ss":20,"used":true}, {"PRN":18,"el":21,"az":214,"ss":31,"used":true}, {"PRN":5,"el":43,"az":54,"ss":33,"used":true}, {"PRN":29,"el":82,"az":144,"ss":26,"used":true}, {"PRN":26,"el":25,"az":105,"ss":36,"used":true}, {"PRN":21,"el":105,"az":144,"ss":36,"used":true}, {"class": "TPV", "tag": "MID2", "device": "/dev/ttyUSB0", "mode": 3, "time": "2013-12-15T17:35:59.000Z", "ept": 0.005, "lat": 43.818948191, "lon": -111.768426061, "alt": 1551.33, "epx": 11.793, "epy": 17.990, "epv": 44.013, "track": 0.0000, "speed": 0.000, "climb": 0.000, "eps": 35.98}
```

The previous screenshot displays both the formatted and some of the raw data that is being received from the GPS sensor. If you get a timeout error when attempting to run this program, type `sudo killall gpsd` to kill all running instances of the daemon and then type `sudo gpsd devttyUSB0 -F varrun/gpsd.sock` again. You can also access this information from a program. To do this, edit a new file called `gpstry1.py`. The code will look as shown in the following screenshot:



```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

import gps

session = gps.gps("localhost", "2947")

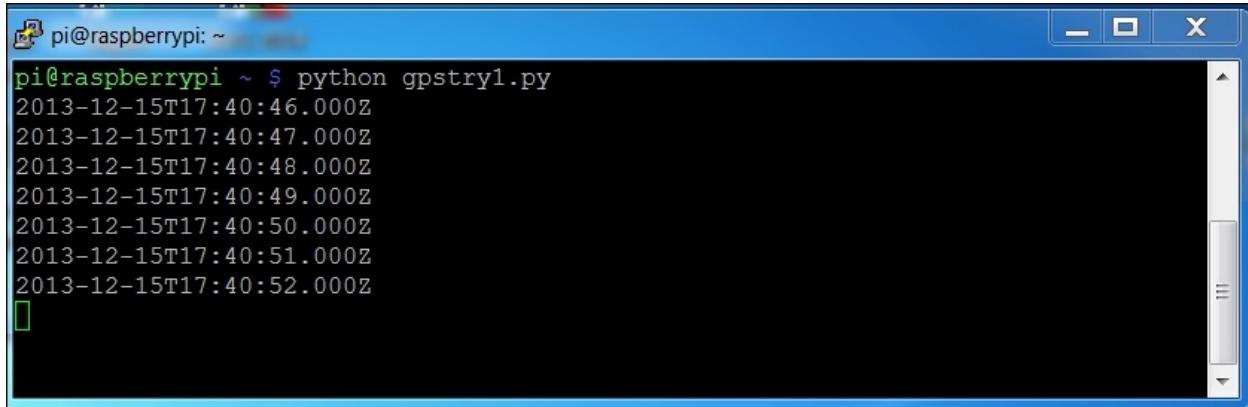
session.stream(gps.WATCH_ENABLE | gps.WATCH_NEWSTYLE)
while True:
    report = session.next()
    if report['class'] == 'TPV':
        if hasattr(report, 'time'):
            print report.time

-UU-----F1 gpstry1.py All L1 (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

The following are the details of your code:

- `#!/usr/bin/Python` – As always, this line simply makes this file available for you to execute from the command line.
- `import gps` – In this case, you import the `gps` library. This will allow you to access the `gpsd` functionality.
- `session = gps.gps("localhost", "2947")` – This opens a communication path between the `gpsd` functionality and our program. It also opens port 2947, which is assigned to the `gpsd` functionality, on the localhost.
- `session.stream(GPS.WATCH_ENABLE | GPS.WATCH_NEWSTYLE)` – This tells the system to look for new GPS data as it becomes available.
- `while True:` – This simply loops and processes information until you ask the system to stop (it can be stopped by pressing `Ctrl + C`).
- `report = session.next()` – When a report is ready, it is saved in the `report` variable.
- `if report['class'] == 'TPV':` – This line checks to see if the report will give you the type of report that you need.
- `if hasattr(report, 'time'):` – This line makes sure that the report holds time data.
- `print report.time` – This prints the time data. I use this in my example because the time data is always returned, even if the GPS is not able to see enough satellites to return position data. To see other possible attributes, visit www.catb.org/gpsd/gpsd_json.html for details.

Once you have created the program, you can run it by typing `python gpstry1.py`. The following screenshot shows what the output you get after running the program should look like:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window contains the command "python gpstry1.py" followed by a series of timestamp outputs: "2013-12-15T17:40:46.000z", "2013-12-15T17:40:47.000z", "2013-12-15T17:40:48.000z", "2013-12-15T17:40:49.000z", "2013-12-15T17:40:50.000z", "2013-12-15T17:40:51.000z", and "2013-12-15T17:40:52.000z". The terminal has a blue header bar and a black background.

```
pi@raspberrypi ~ $ python gpstry1.py
2013-12-15T17:40:46.000z
2013-12-15T17:40:47.000z
2013-12-15T17:40:48.000z
2013-12-15T17:40:49.000z
2013-12-15T17:40:50.000z
2013-12-15T17:40:51.000z
2013-12-15T17:40:52.000z
```

One cool way to display positional information is using a graphical display including a map of your current position. There are several map applications that can interface with your GPS to indicate your location on a map.

One map application that works well is GpsPrune. To get this application, type `sudo apt-get install gpsprune`. To run this command, you'll need to be in a graphical environment, so you'll need to run it either with a display and keyboard attached or using vncserver. You'll also need to store your data so that the program can import it. To do this, let's amend our `location.py` program to save the data to a file. First, copy the program by typing `cp location.py gpsdata.py`. Now edit the program to make it look as shown in the following screenshot:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Python Help
#!/usr/bin/python

import serial

f = open('data.txt', 'w')
ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)
x = ser.read(1200)
pos1 = x.find("$GPRMC")
pos2 = x.find("\n", pos1)
loc = x[pos1:pos2]
print loc
f.write(loc)
data = loc.split(',')
if data[2] == 'V':
    print 'No location found'
else:
    print "Latitude = " + data[3] + data[4]
    print "Longitude = " + data[5] + data[6]
    print "Speed = " + data[7]
    print "Course = " + data[8]

-UU-----F1  gspdata.py      All L1      (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

There are the following two changes you need to make:

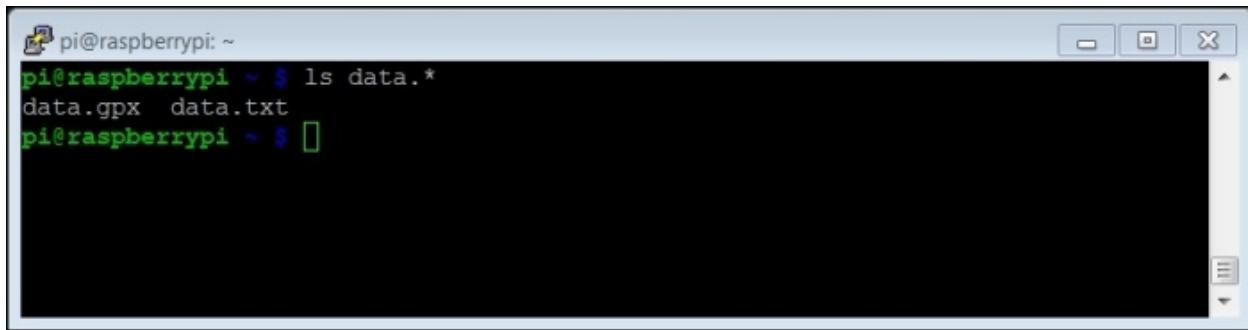
- `f = open('data.txt', 'w')` – This line opens the file `data.txt` for writing the data in it
- `f.write(loc)` – This will write the line `loc` in the file which will hold the entire data set

Now run the program by typing `python gspdata.py`. After the program is run, you should see a new data file called `data.txt`. You can view the contents of the file by typing `emacs data.txt`. You should see the result as shown in the following screenshot:

```
pi@raspberrypi: ~
File Edit Options Buffers Tools Help
$GPRMC,161413.000,A,4349.1340,N,11146.1110,W,0.00,,141213,,,A*68

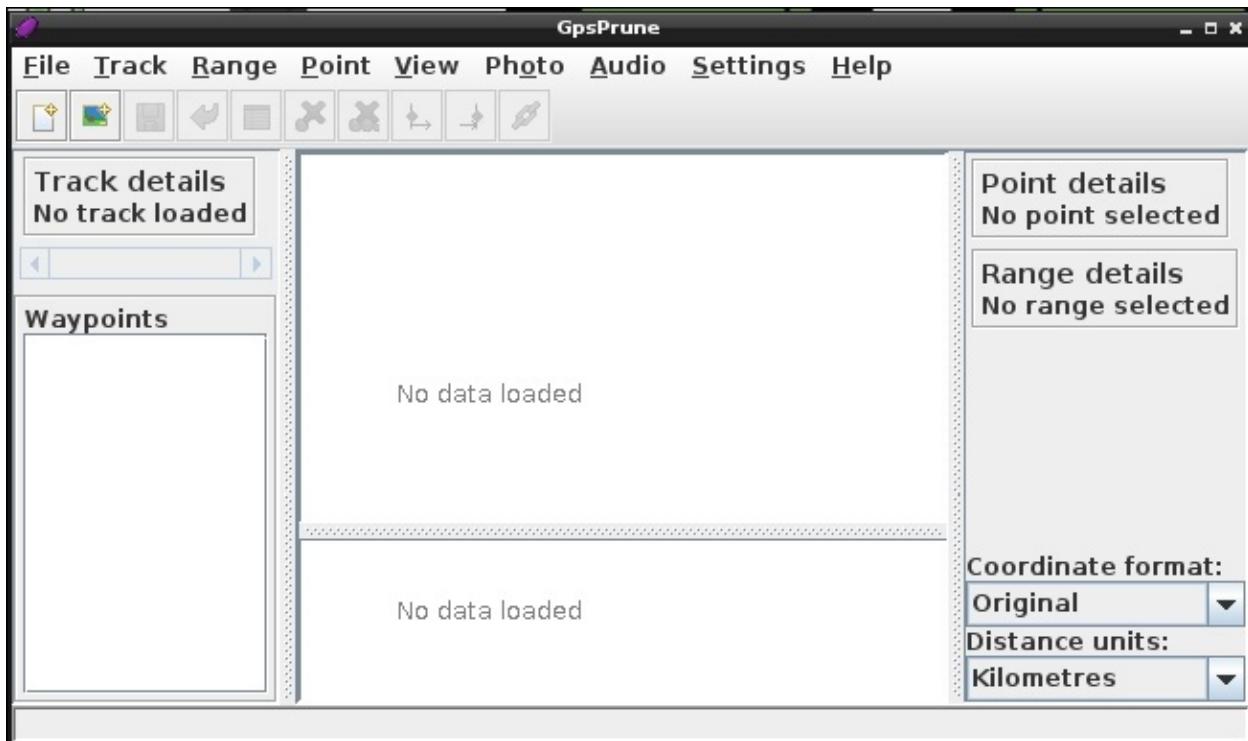
-UU- (Mac)----F1  data.txt      All L1      (Text)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

You can look at the data in the GpsPrune application. First, you need to convert your NMEA-formatted data to a data format that GpsPrune can understand. To do this, type `gpsbabel -i NMEA -f data.txt -o GPX -F data.gpx`. Type an `ls data.*` command and you should see the files as shown in the following screenshot:

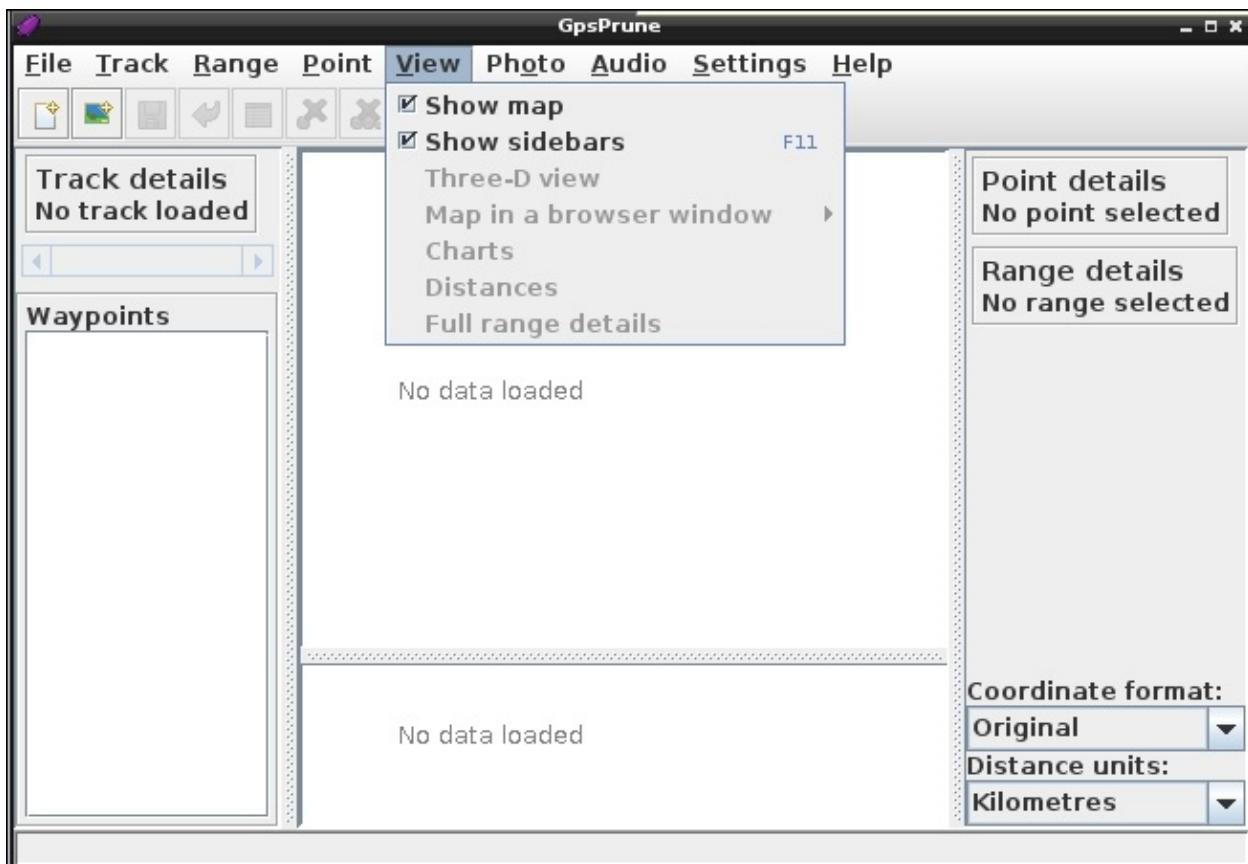


```
pi@raspberrypi: ~
pi@raspberrypi ~ $ ls data.*
data.gpx  data.txt
pi@raspberrypi ~ $
```

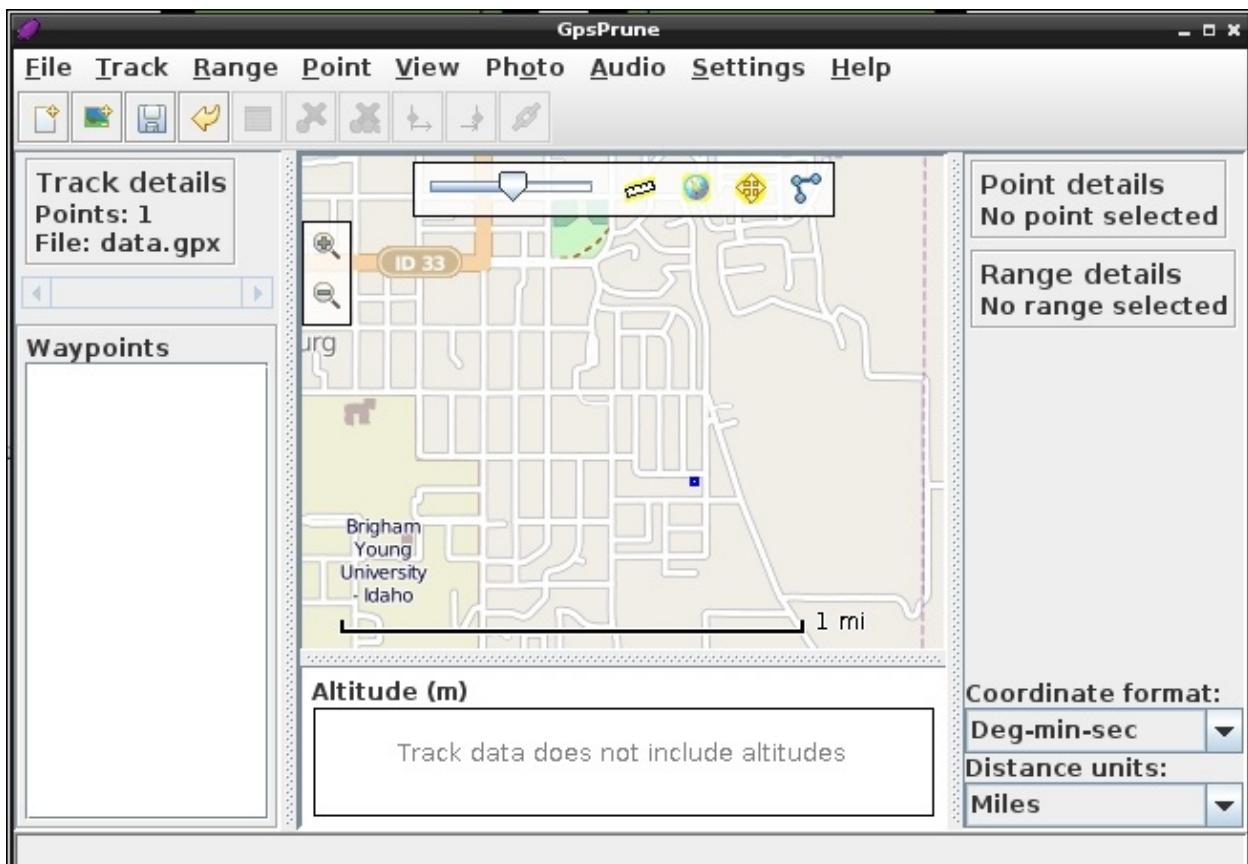
Now run the program by typing `gpsprune` in a terminal window. The application will open as shown in the following screenshot:



Turn on the map function by navigating to **View | Show map** as shown in the following screenshot:

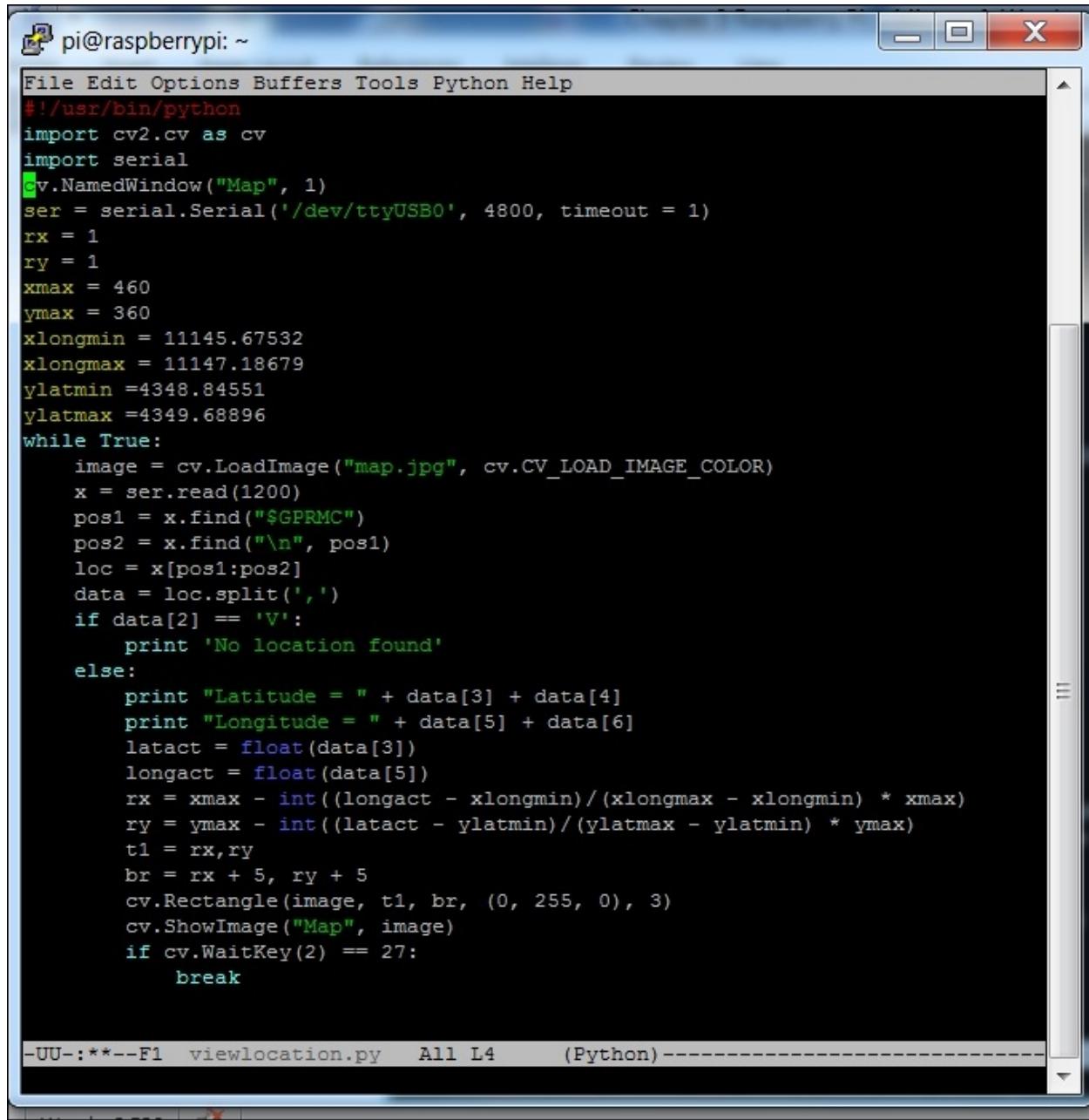


Now open your file by navigating to **File | Open** and then selecting your filename with the [.gpx](#) file extension. The window shown in the following screenshot should open:



You may need to zoom out to see your location, but it should be there on the map. You could also capture multiple locations and then display them as routes. But you might want something a bit simpler; a program that allows you to enter a waypoint and display a map that shows both the waypoint and your current location. You can do this with the skills you learned in [Chapter 4, Adding Vision to Raspberry Pi](#).

Let's start with the code in `location.py`. Make a copy by typing `cp location.py viewlocation.py`. Now, make the changes shown in the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window contains Python code for displaying location data on a map. The code imports cv2 and serial libraries, defines variables for window creation and serial port, and enters a loop to read GPRMC data from the serial port. It then prints latitude and longitude, converts them to pixel coordinates, and draws a rectangle on a map image. A status bar at the bottom shows "-UU-:***-F1 viewlocation.py All L4 (Python)-----".

```
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import cv2.cv as cv
import serial
cv.NamedWindow("Map", 1)
ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)
rx = 1
ry = 1
xmax = 460
ymax = 360
xlongmin = 11145.67532
xlongmax = 11147.18679
ylatmin = 4348.84551
ylatmax = 4349.68896
while True:
    image = cv.LoadImage("map.jpg", cv.CV_LOAD_IMAGE_COLOR)
    x = ser.read(1200)
    pos1 = x.find("$GPRMC")
    pos2 = x.find("\n", pos1)
    loc = x[pos1:pos2]
    data = loc.split(',')
    if data[2] == 'V':
        print 'No location found'
    else:
        print "Latitude = " + data[3] + data[4]
        print "Longitude = " + data[5] + data[6]
        latact = float(data[3])
        longact = float(data[5])
        rx = xmax - int((longact - xlongmin)/(xlongmax - xlongmin) * xmax)
        ry = ymax - int((latact - ylatmin)/(ylatmax - ylatmin) * ymax)
        t1 = rx, ry
        br = rx + 5, ry + 5
        cv.Rectangle(image, t1, br, (0, 255, 0), 3)
        cv.ShowImage("Map", image)
    if cv.WaitKey(2) == 27:
        break

-UU-:***-F1 viewlocation.py All L4 (Python)-----
```

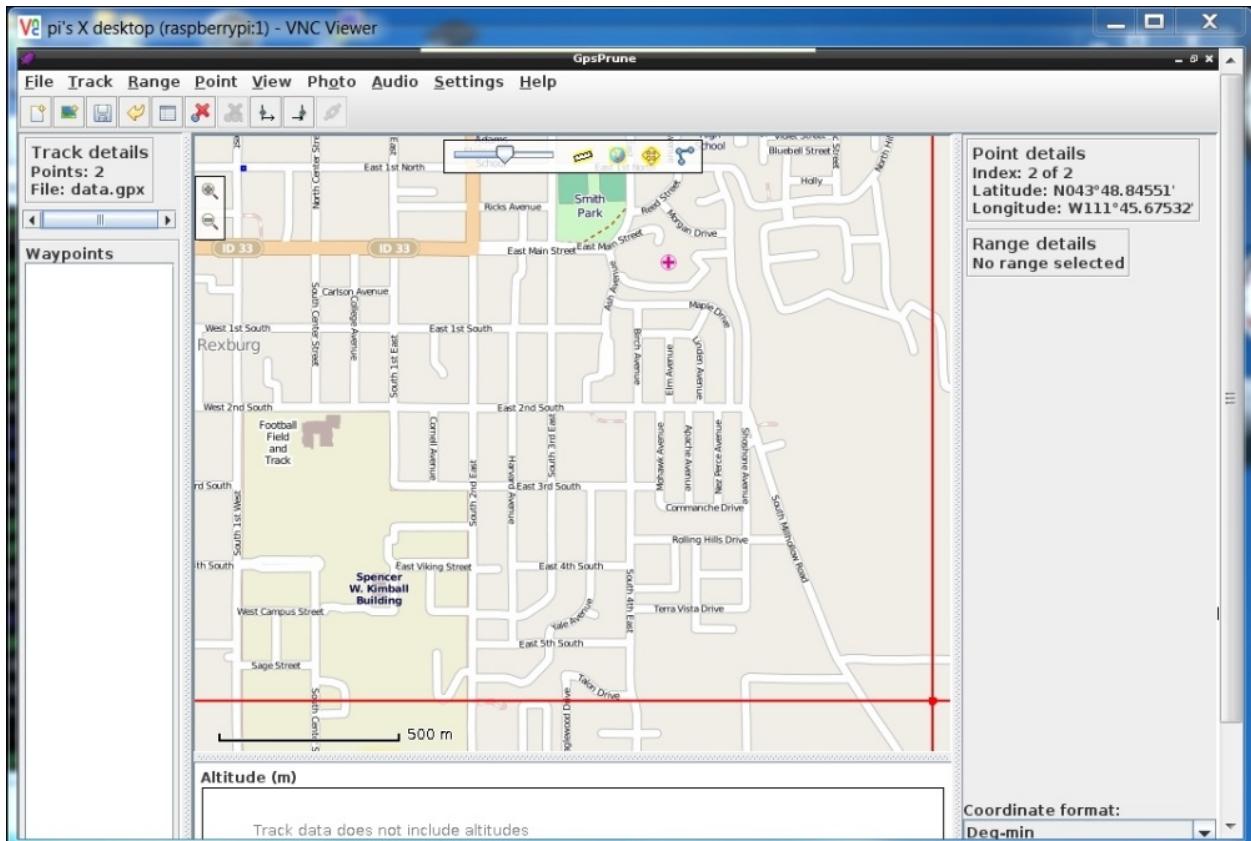
The following are the details of the changes:

- `import cv2 as cv` – You'll be using the OpenCV library to draw the images, so you need to import it using this command.
- `rx = 1` – This variable will hold the x pixel value of your location.
- `ry = 1` – This variable will hold the y pixel value of your location.
- `xmax = 460` – This is the maximum x pixel value in the map you created.
- `ymax = 360` – This is the maximum y pixel value in the map you

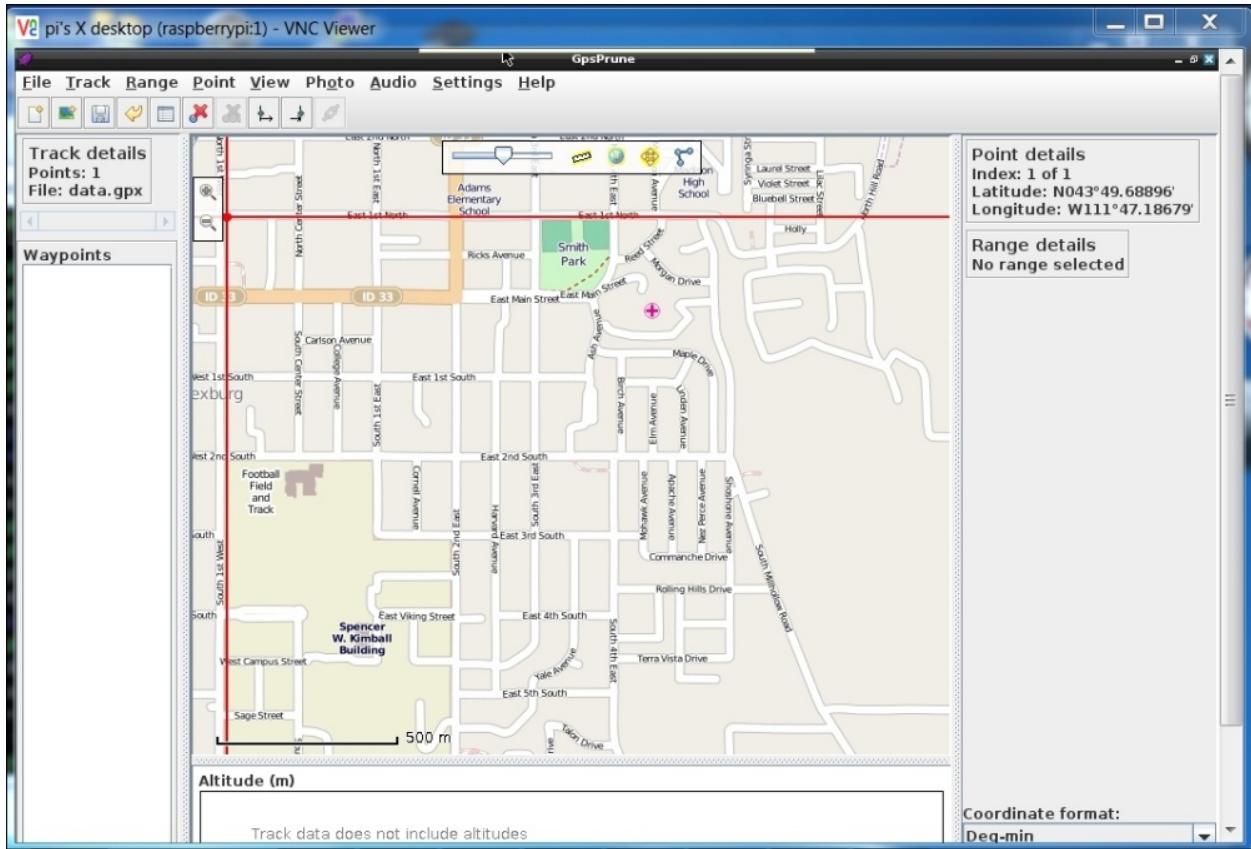
created.

- `xlongmin = 11145.67532` – This is the minimum longitude value in the map you created.
- `xlongmax = 11147.18679` – This is the maximum longitude value in the map you created.
- `ylatmin = 4348.84551` – This is the minimum latitude value in the map you created.
- `ylatmax = 4349.68896` – This is the maximum latitude value in the map you created.
- `while True:` – This loops through the program, drawing the map and the position each time you get a new position from the GPS system.
- `image = cv.LoadImage("map.jpg", cv.CV_LOAD_IMAGE_COLOR)` – This creates a new data structure called `image`, and initializes it with the map image.
- `latact = float(data[3])` – This command turns the string that is the latitude into a float value.
- `longact = float(data[5])` – This command turns the string that is the longitude into a float value.
- `rx = xmax - int((longact - xlongmin)/(xlongmax - xlongmin) * xmax)` – This calculates the x value in pixels by taking the ratio of the range from the actual longitude value to the minimum longitude value divided by the total longitude range.
- `ry = ymax - int((latact - ylatmin)/(ylatmax - ylatmin) * ymax)` – This calculates the y value in pixels by taking the ratio of the range from the actual latitude value to the minimum latitude value divided by the total latitude range.
- `t1 = rx, ry` – This creates a point value that holds the x and y values of your position.
- `br = rx + 5, ry + 5` – You'll want to draw a rectangle so you can see the created point; this draws a rectangle value that is 5 x 5 in size.
- `cv.Rectangle(image, t1, br, (0, 255, 0), 3)` – This adds the drawn rectangle to the map image.
- `cv.ShowImage("Map", image)` – This shows the map with the rectangle in a window.
- `if cv.WaitKey(2) == 27:` – The `WaitKey` object displays the image and also checks to see whether or not the *Esc* key has been pressed. If it has been pressed, it will execute the next statement.
- `break` – This closes the loop and the program.

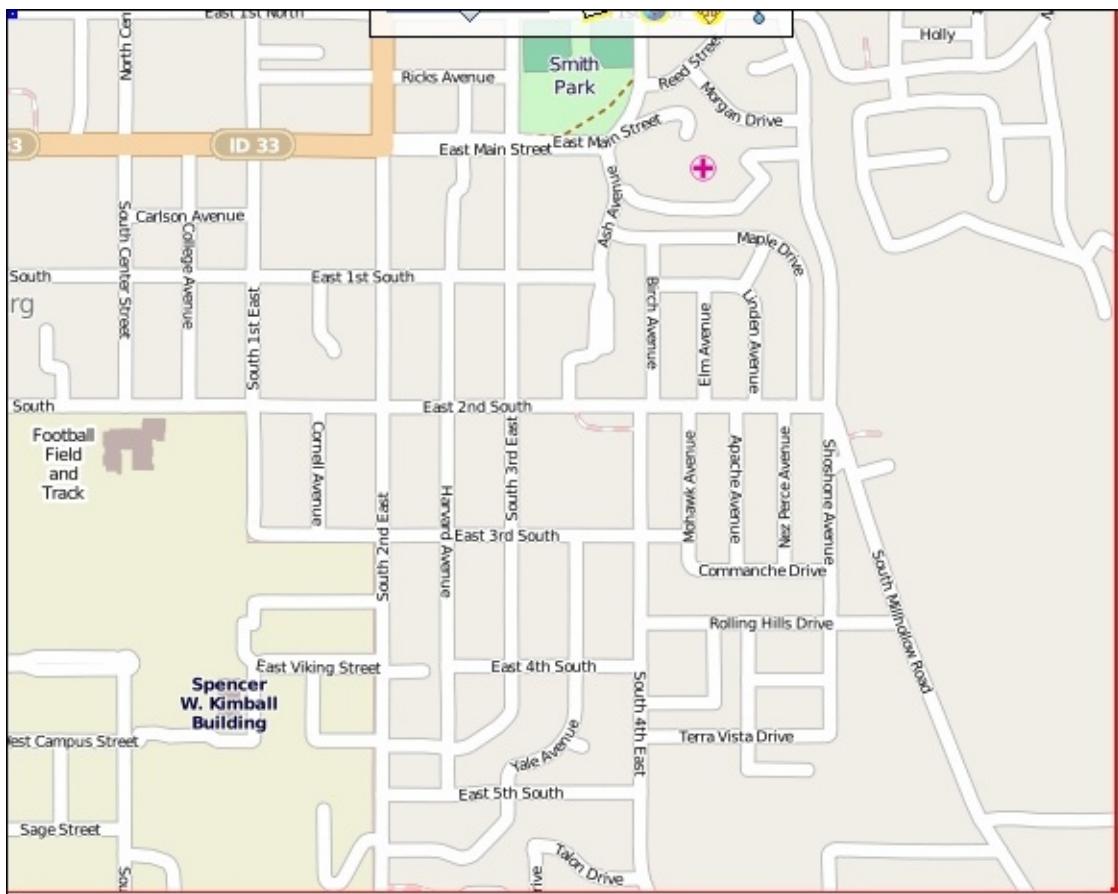
You'll also need to build a map to be displayed. You'll need to know the coordinates of the corners of your map. I used GpsPrune to build a map. I first removed the point that was plotted by opening the file. Then, I selected a point in the bottom-right corner. The following is a screenshot of the application:



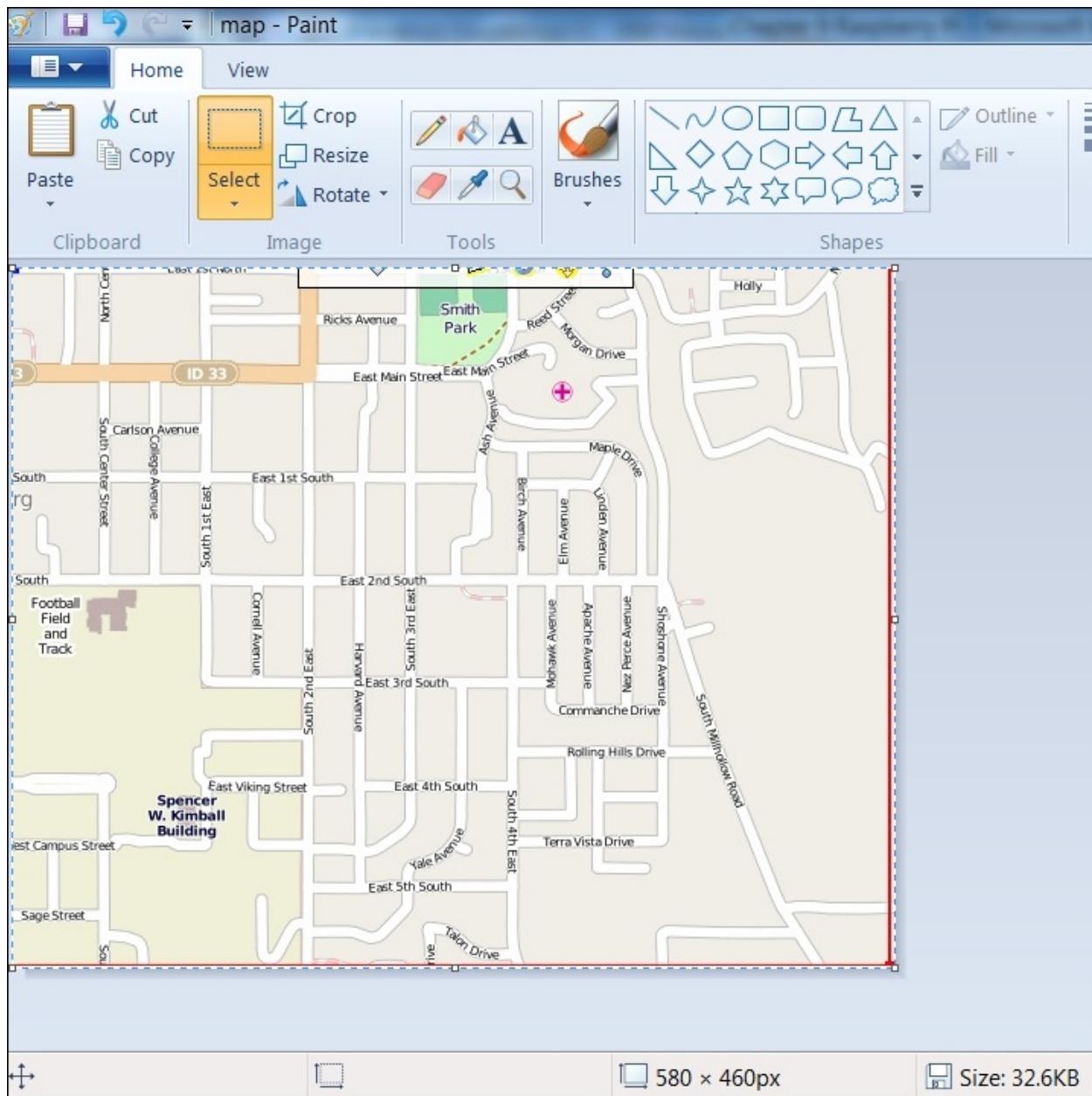
Make sure the **Coordinate format** dropdown is set to **Deg-min**. Notice the **Latitude** and **Longitude** values in the upper-right corner, which in this case are **43°48.84551'** and **111°45.67532'**. Now add another point in the upper-left corner, as shown in the following screenshot:



In this case, the **Latitude** value is **43°49.68896'** and the **Longitude** value is **111°47.18679'**. These are the corners of your map. Now take a screenshot of the map and use a program to crop the map at those two corners, as shown in the following screenshot:

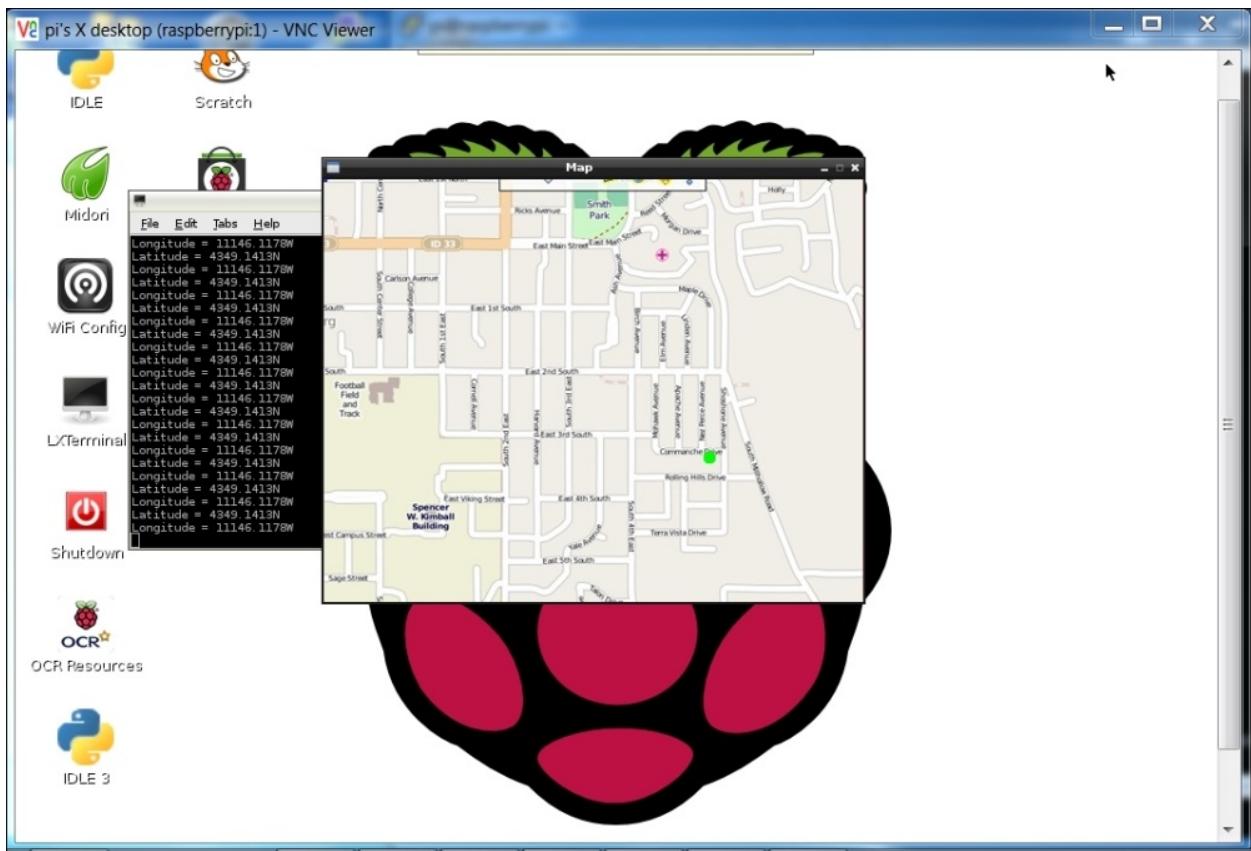


For this example, I was using the **VNC Viewer** application on my PC to run GpsPrune on Raspberry Pi. This made it easy to take a screenshot by pressing *Ctrl + Print Screen* and then import the image to the Paint application. I then cropped the image and saved it as [map.png](#). The following screenshot shows the view of the image as seen in the Paint application:



Note the resolution of the picture shown at the bottom of the page in the previous screenshot; this is important for your program. I then moved the resolution of the picture to Raspberry Pi using WinSCP, which we detailed in [Chapter 1, Getting Started with Raspberry Pi](#).

Now that I have my file and Python code ready, I can now run the program using `python viewlocation.py`. The output of this operation appears as shown in the following screenshot:



Now you can view your location via a map; it is marked by the green dot. As you move around, so will the green dot move on the screen.

Summary

Congratulations! Your robot can now move around without getting lost. This capability is more useful for robots that can go far from home. You can use the information to identify routes to different waypoints and track where your robot has been.

In the next chapter, we'll cover how to bring all of the various functionality you have been working on together in a single, integrated system.

Chapter 10. System Dynamics

In the previous chapters, we've spent time learning a lot about individual functionalities that we can add to our robotic projects. In this chapter, we'll learn how to integrate these different parts into a single system.

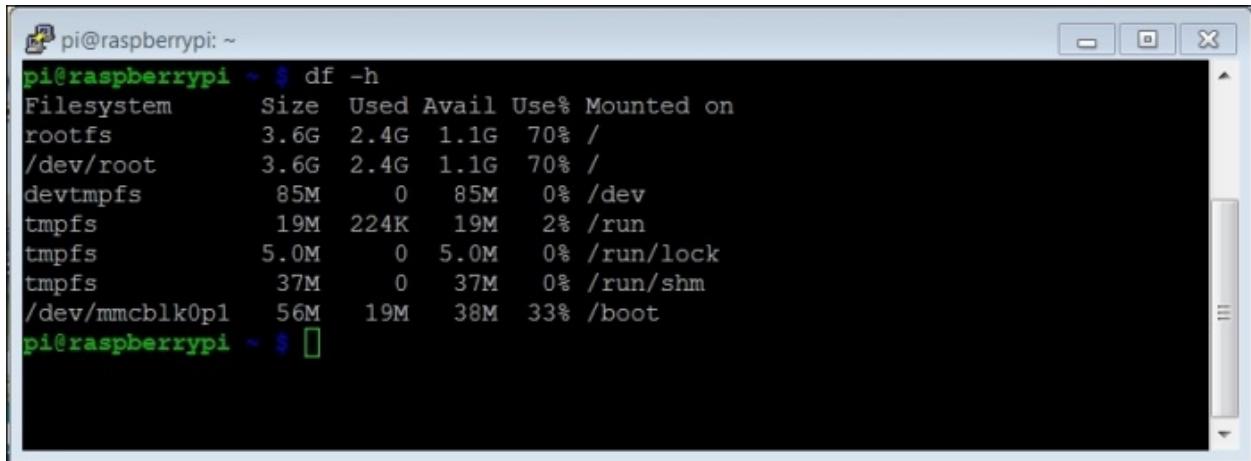
Getting started

We've spent a large amount of time on individual functionality, and your robotic projects now have many functionalities that we can add to them. This chapter will bring all of these parts together into a framework that allows the different parts to work together. You don't want the robot to just walk, talk, or see. You want it to perform all of these in a coordinated package. In this chapter, you'll learn how to programmatically connect all of these individual capabilities and make your projects seem intelligent.

In this chapter, we will cover the following points:

- Creating a general control structure so that different capabilities can work together through system calls
- Introducing the Robot Operating System as a supported framework for robotic capabilities

We're finally done with purchasing hardware! In this chapter, we'll be adding the functionality via software. You'll need ample storage space for an array of the new software. First, let's check how much space you have in your memory card. You can also use the `df -h` command to see this information. You should see something like the following screenshot when you type the `df -h` command:



A screenshot of a terminal window titled "pi@raspberrypi: ~". The window displays the output of the command "df -h", which shows the following disk usage:

Filesystem	Size	Used	Avail	Use%	Mounted on
rootfs	3.6G	2.4G	1.1G	70%	/
/dev/root	3.6G	2.4G	1.1G	70%	/
devtmpfs	85M	0	85M	0%	/dev
tmpfs	19M	224K	19M	2%	/run
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	37M	0	37M	0%	/run/shm
/dev/mmcblk0p1	56M	19M	38M	33%	/boot

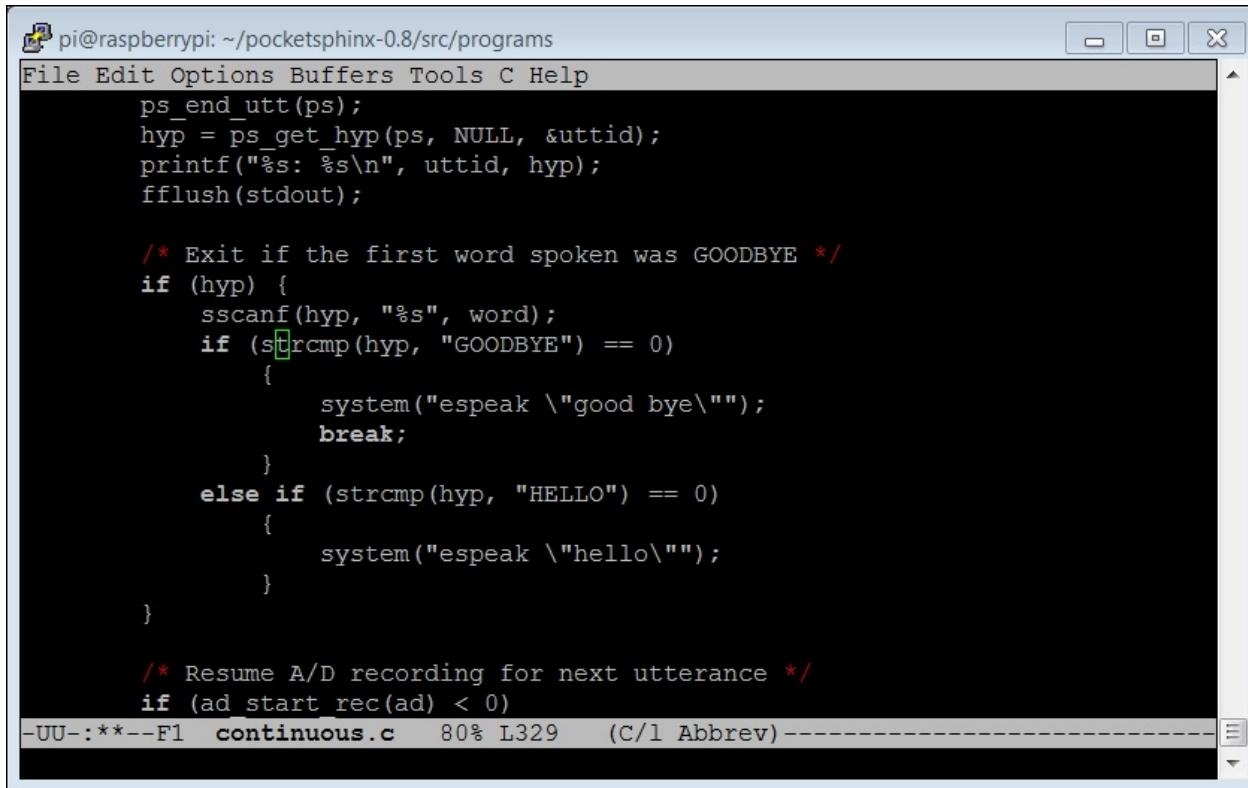
The terminal prompt "pi@raspberrypi ~ \$" is visible at the bottom.

In this case, I have 1.1 GB, which should be enough to add the capability I need.

Creating a general control structure

Now that you have a mobile robot, you want to coordinate all of its different abilities. Let's start with the simplest approach using a single control program that can call other programs and enable all the capabilities.

You've already done this once in [Chapter 3, Providing Speech Input and Output](#), where you edited the `continuous.c` code to allow it to call other programs to execute functionality. The following is a screenshot of the code that we used from the `homeubuntu/pocketsphinx-0.8/programs/src/` directory:



The screenshot shows a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains the source code for the `continuous.c` file. The code is written in C and includes logic to handle speech input. It uses the `sscanf` function to read a word from the input, and the `strcmp` function to check if it's "GOODBYE" or "HELLO". If "GOODBYE" is detected, it calls the `system("espeak \"good bye\"")` command to speak the words. If "HELLO" is detected, it calls the `system("espeak \"hello\"")` command to speak the words. The code also includes a section for resuming A/D recording for the next utterance.

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
File Edit Options Buffers Tools C Help
ps_end_utt(ps);
hyp = ps_get_hyp(ps, NULL, &uttid);
printf("%s: %s\n", uttid, hyp);
fflush(stdout);

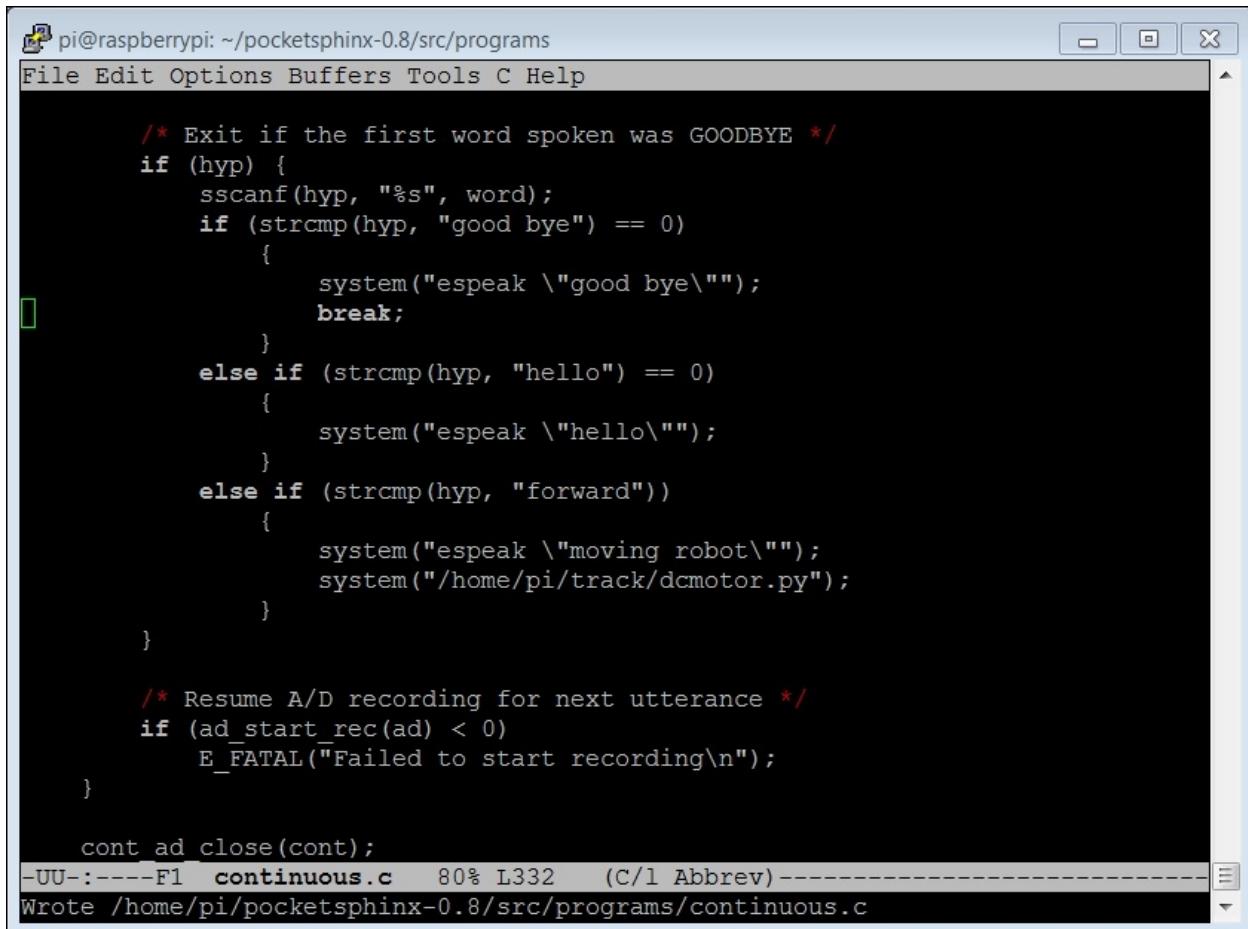
/* Exit if the first word spoken was GOODBYE */
if (hyp) {
    sscanf(hyp, "%s", word);
    if (strcmp(hyp, "GOODBYE") == 0)
    {
        system("espeak \"good bye\"");
        break;
    }
    else if (strcmp(hyp, "HELLO") == 0)
    {
        system("espeak \"hello\"");
    }
}

/* Resume A/D recording for next utterance */
if (ad_start_rec(ad) < 0)
-UU-:***-F1  continuous.c  80% L329  (C/l Abbrev) -----
```

The functionality that is important to us is the `system("espeak \"good bye\"")`; line of code. When you use the `system` function call, the program actually calls a different program, in this case the `espeak` program, and passes to it the `"good bye"` parameter so that the words

good and bye come out of the speaker.

The following screenshot is another example from [Chapter 5, Creating Mobile Robots on Wheels](#), where you wanted to command your robot to move:



A screenshot of a terminal window titled "pi@raspberrypi: ~/pocketsphinx-0.8/src/programs". The window contains C code for a speech recognition program. The code includes logic to handle commands like "good bye", "hello", and "forward". It uses the espeak library to speak the commands and calls a Python script named "dcmotor.py" to move the robot forward. The terminal shows the code being typed and then executed, with the message "Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c" at the bottom.

```
pi@raspberrypi: ~/pocketsphinx-0.8/src/programs
File Edit Options Buffers Tools C Help

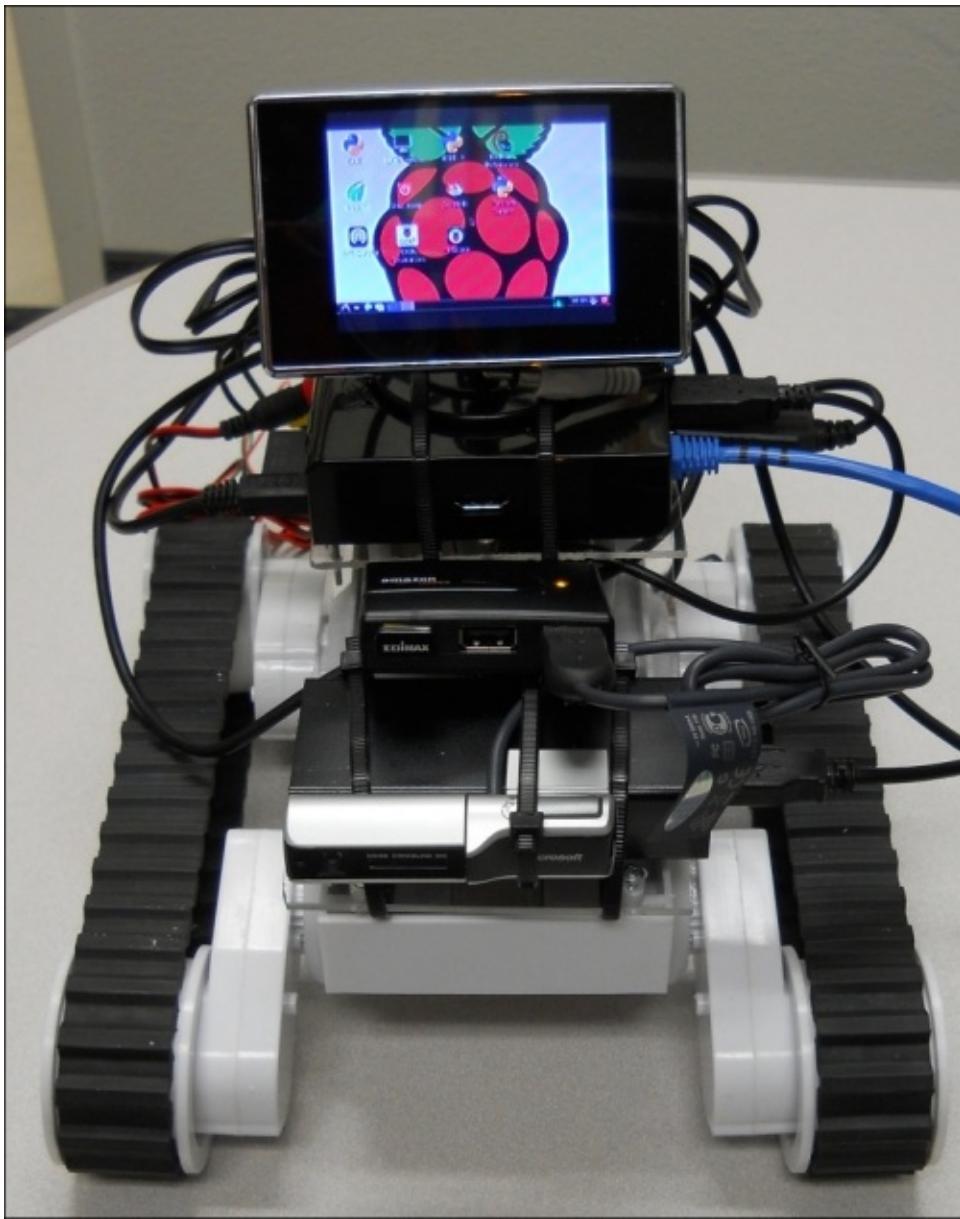
    /* Exit if the first word spoken was GOODBYE */
    if (hyp) {
        sscanf(hyp, "%s", word);
        if (strcmp(hyp, "good bye") == 0)
        {
            system("espeak \"good bye\"");
            break;
        }
        else if (strcmp(hyp, "hello") == 0)
        {
            system("espeak \"hello\"");
        }
        else if (strcmp(hyp, "forward"))
        {
            system("espeak \"moving robot\"");
            system("/home/pi/track/dcmotor.py");
        }
    }

    /* Resume A/D recording for next utterance */
    if (ad_start_rec(ad) < 0)
        E_FATAL("Failed to start recording\n");
}

cont ad close(cont);
-UU-----F1 continuous.c 80% L332 (C/l Abbrev)-----
Wrote /home/pi/pocketsphinx-0.8/src/programs/continuous.c
```

In this case, if you say `forward` to your robot, it will execute two programs. The first program you call is the `espeak` program with the parameter `"moving robot"`. These words should then come out from the speaker on the robot. The second program is the `dcmotor.py` program, which should include the commands to move the robot forward.

I am now going to include an example in Python; it is my preferred language. I am going to use my tracked robot, which is shown in the following image:



This robot has a camera and is also able to communicate via a speaker. You can also control it via a wireless keyboard. You will now add the functionality to follow a colored ball, turn as the ball goes right or left, and tell you when it is turning.

You also need to make sure all of your devices are available to your programs. For this, you'll need to make sure your USB camera as well as the two DC motor controllers are connected. To connect the camera, follow the steps in [Chapter 4, Adding Vision to Raspberry Pi](#), in the *Connecting the USB camera to Raspberry Pi and viewing the images*

section. It works best to connect the USB camera first before connecting any other USB devices.

When the camera is up and running, you'll want to connect and check the DC motor controllers as described in [Chapter 5](#), *Creating Mobile Robots on Wheels*. You may want to run the `dcmotor.py` program just to make sure you are connected and both the motors work.

In this project, you are going to involve three different programs. First, you are going to create a program that will find out whether the ball is on the right-hand side or the left-hand side. This is going to be your main control program. You are also going to create a program that moves your robot approximately 45 degrees to the right and another program that moves it 45 degrees to the left. You are going to keep these programs very simple, and you may just want to put them all in the same source file. But as the complexity of each of these programs grows, it will make more sense for them to be separate, so this is a good starting point for your robotic code. Also, if you want to use the code in another project, or want to share it, this sort of separation helps.

You are going to create three programs for this project. In order to keep this situation organized, I created a new directory in my home directory by typing `mkdir robot` in my home directory. I will now put all my files in this directory.

The next step is to create two files that can move your robot: one to the left-hand side and the other to the right-hand side. For this, you will have to create two copies of the `dcmotor.py` code as you did in [Chapter 5](#), *Creating Mobile Robots on Wheels*, in your `robot` directory. If you have created that file in your home directory, type `cp dcmotor.py ./robot/move_left.py` `cp dcmotor.py ./robot/move_right.py`. Now you'll edit the files, changing two numbers in the program. Edit the code, shown in the following screenshot, in the `move_left.py` file:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/robot". The window contains Python code for controlling a robot. The code includes imports for serial and time, defines a setSpeed function, and initializes a serial port. It then calls setSpeed for both motors at 100 speed, sleeps for half a second, and finally closes the serial port. The terminal window has a standard Linux-style interface with a title bar, menu bar, and status bar.

```
pi@raspberrypi: ~/robot
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time

def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)
    ser.write(sendByte)
    ser.write(chr(speed))

ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
setSpeed(ser, 0, 0, 100)
setSpeed(ser, 1, 0, 100)
time.sleep(.5)
setSpeed(ser, 0, 0, 0)
setSpeed(ser, 1, 0, 0)
ser.close()

-UU-----F1 move_left.py All L1 (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

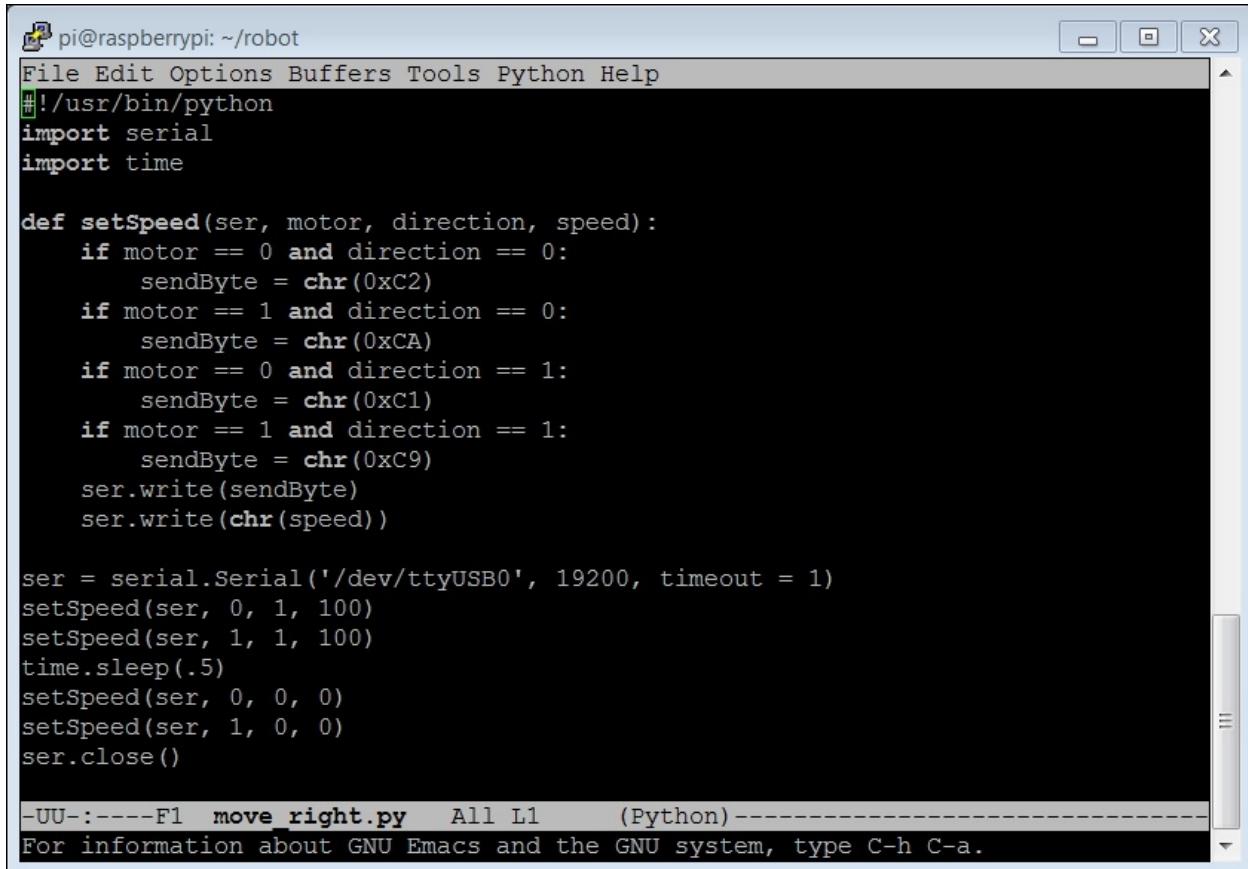
The following are the details for this code:

- `#!/usr/bin/python`—This statement sets the program so it can be run directly from the program line
- `import serial`—This statement imports the `serial` library so you can talk to the motor controller
- `import time`—This statement imports the `time` library so you can use the `time.sleep()` function to add a fixed delay
- `def setSpeed(ser, motor, direction, speed):`—This statement has the `setSpeed` function you will call in your program that sets the speed and direction for a given motor
- `if motor == 0 and direction == 0:`—This statement sets Motor 1 in the forward direction
- `sendByte = chr(0xC2)`—This statement is the actual `byte` command to the motor controller
- `if motor == 1 and direction == 0:`—This statement sets Motor 2 to go in the backward direction
- `sendByte = chr(0xCA)`—This statement is the actual `byte` command

to the motor controller

- `if motor == 0 and direction == 1:` – This statement sets Motor 1 to go in the backward direction
- `sendByte = chr(0xC1)`—This statement is the actual `byte` command for the motor controller
- `if motor == 1 and direction == 1:` – This statement sets Motor 2 to go in the forward direction
- `sendByte = chr(0xC9)`—This statement is the actual `byte` command for the motor controller
- `ser.write(sendByte)`—This statement sends the `byte` command out of the `serial` port
- `ser.write(chr(speed))`—This statement sends the `speed` byte out of the `serial` port
- `ser = serial.Serial('devttyUSB0', 19200, timeout = 1)`—This statement initializes and opens the `serial` port
- `setSpeed(ser, 0, 0, 100)`—This statement sends the `forward` command to Motor 1 at a speed of 100 units
- `setSpeed(ser, 1, 0, 100)`—This statement sends the `reverse` command to Motor 2 at a speed of 100 units
- `time.sleep(.5)`—This statement causes the motor to wait for 0.5 seconds
- `setSpeed(ser, 0, 0, 0)`—This statement sets the speed of Motor 1 to 0 units
- `setSpeed(ser, 1, 0, 0)`—This statement sets the speed of Motor 2 to 0 units
- `ser.close()`—This statement closes the `serial` port

Similarly, you will also need to edit `moveright.py`, as shown in the following screenshot:



A screenshot of a terminal window titled "pi@raspberrypi: ~/robot". The window shows Python code for controlling a robot. The code includes a function `setSpeed` that takes parameters `ser`, `motor`, `direction`, and `speed`. It uses `chr` to convert integers to bytes and `ser.write` to send them over a serial connection. The script then initializes the serial port and calls `setSpeed` twice with different parameters to move the motors in opposite directions. The terminal window has a menu bar with "File", "Edit", "Options", "Buffers", "Tools", "Python", and "Help". The status bar at the bottom indicates "-UU-----F1 move right.py All L1 (Python)-----" and "For information about GNU Emacs and the GNU system, type C-h C-a."

```
pi@raspberrypi: ~/robot
File Edit Options Buffers Tools Python Help
#!/usr/bin/python
import serial
import time

def setSpeed(ser, motor, direction, speed):
    if motor == 0 and direction == 0:
        sendByte = chr(0xC2)
    if motor == 1 and direction == 0:
        sendByte = chr(0xCA)
    if motor == 0 and direction == 1:
        sendByte = chr(0xC1)
    if motor == 1 and direction == 1:
        sendByte = chr(0xC9)
    ser.write(sendByte)
    ser.write(chr(speed))

ser = serial.Serial('/dev/ttyUSB0', 19200, timeout = 1)
setSpeed(ser, 0, 1, 100)
setSpeed(ser, 1, 1, 100)
time.sleep(.5)
setSpeed(ser, 0, 0, 0)
setSpeed(ser, 1, 0, 0)
ser.close()

-UU-----F1 move right.py All L1 (Python)-----
For information about GNU Emacs and the GNU system, type C-h C-a.
```

This time, the `setSpeed` numbers are changed to run the motors in the opposite direction, turning the robot to the right-hand side.

The final step is to create the main control program. You're going to start with the `camera.py` program you edited in [Chapter 4, Adding Vision to Raspberry Pi](#). Execute this by typing `cp homepi/example/python/camera.py ./follow.py` while you are in your `robot` directory. Open this file with your editor; if you are using `emacs`, type `emacs follow.py` and then edit the code as shown in the following screenshot:

The screenshot shows a terminal window titled "pi@raspberrypi: ~/robot". The window contains a Python script named "follow.py". The script uses the OpenCV library to capture frames from a camera, process them for color, and track a specific color. It also uses the subprocess module to call an external program "move_right.py" or "move_left.py" based on the tracked object's position. The script includes logic to move the object towards the center if it is off-center and to stop if a key is pressed.

```
File Edit Options Buffers Tools Python Help
import cv2.cv as cv
import time
import subprocess

capture = cv.CaptureFromCAM(0)
cv.SetCaptureProperty(capture, 3, 360)
cv.SetCaptureProperty(capture, 4, 240)

while True:
    img = cv.QueryFrame(capture)
    cv.Smooth(img, img, cv.CV_BLUR, 3)
    hue_img = cv.CreateImage(cv.GetSize(img), 8, 3)
    cv.CvtColor(img, hue_img, cv.CV_BGR2HSV)
    threshold_img = cv.CreateImage(cv.GetSize(hue_img), 8, 1)
    cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)
    storage = cv.CreateMemStorage(0)
    contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP, cv.CV_C\b
HAIN_APPROX_SIMPLE)
    points = []
    cx = 0
    cy = 0
    while contour:
        rect = cv.BoundingRect(list(contour))
        contour = contour.h_next()
        size = (rect[2] * rect[3])
        if size > 100:
            pt1 = (rect[0], rect[1])
            pt2 = (rect[0] + rect[2], rect[1] + rect[3])
            cx = rect[0]
            cy = rect[1]
            cv.Rectangle(img, pt1, pt2, (38, 160, 60))
        cv.ShowImage("Colour Tracking", img)
        if cx > 280:
            text = "moving right"
            subprocess.call('espeak '+text, shell = True)
            subprocess.call('./move_right.py')
        if cx < 20 and cx > 0:
            text = "moving left"
            subprocess.call('espeak '+text, shell = True)
            subprocess.call('./move_left.py')
        if cv.WaitKey(10) == 27:
            break

-UU-:**--F1  follow.py      All L8      (Python)
```

Let's look at the following code statements along with their functions:

- `import cv2.cv as cv`—This statement imports the `cv` library.
- `import time`—This statement imports the `time` library.
- `import subprocess`—This statement imports the `subprocess` library;

this will allow you to call other programs within your Python program.

- `capture = cv.CaptureFromCAM(0)`—This statement sets up the program to capture your images from the webcam.
- `cv.SetCaptureProperty(capture, 3, 360)`—This statement sets the `x` resolution on the image to 360.
- `cv.SetCaptureProperty(capture, 4, 240)`—This statement sets the `y` resolution of the image to 240.
- `while True:`—This statement executes the loop over and over until the `Esc` key is pressed.
- `img = cv.QueryFrame(capture)`—This statement brings in the image.
- `cv.Smooth(img, img, cv.CV_BLUR, 3)`—This statement smoothens the image.
- `hue_img = cv.CreateImage(cv.GetSize(img), 8, 3)`—This statement creates a new image that will hold the hue-based image.
- `cv.CvtColor(img, hue_img, cv.CV_BGR2HSV)`—This statement moves a copy of the image using the `hue` values in `hue_img`.
- `threshold_img = cv.CreateImage(cv.GetSize(hue_img), 8, 1)`—This statement creates a new image that will hold all the blobs of colors.
- `cv.InRangeS(hue_img, (38,120, 60), (75, 255, 255), threshold_img)`—This statement now fills in `hue_img` from `threshold_img`.
- `storage = cv.CreateMemStorage(0)`—This line creates some memory for you to manipulate the images.
- `contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)`—This statement finds all the areas on your image that are within the threshold. There may be more than one so you want to capture them all.
- `points = []`—This statement creates an array to hold all the different possible points of color.
- `cx = 0`—This statement gives a variable to hold the `x` location of `color`.
- `cy = 0`—This statement gives a variable to hold the `y` location of `color`.
- `while contour:`—This statement now adds a `while` loop that will let you step through all the possible contours. By the way, it is important to note that if there is another larger green blob in the background, you will find that location. Just to keep this simple, we'll assume that your green ball is unique.
- `rect = cv.BoundingRect(list(contour))`—This statement gets a

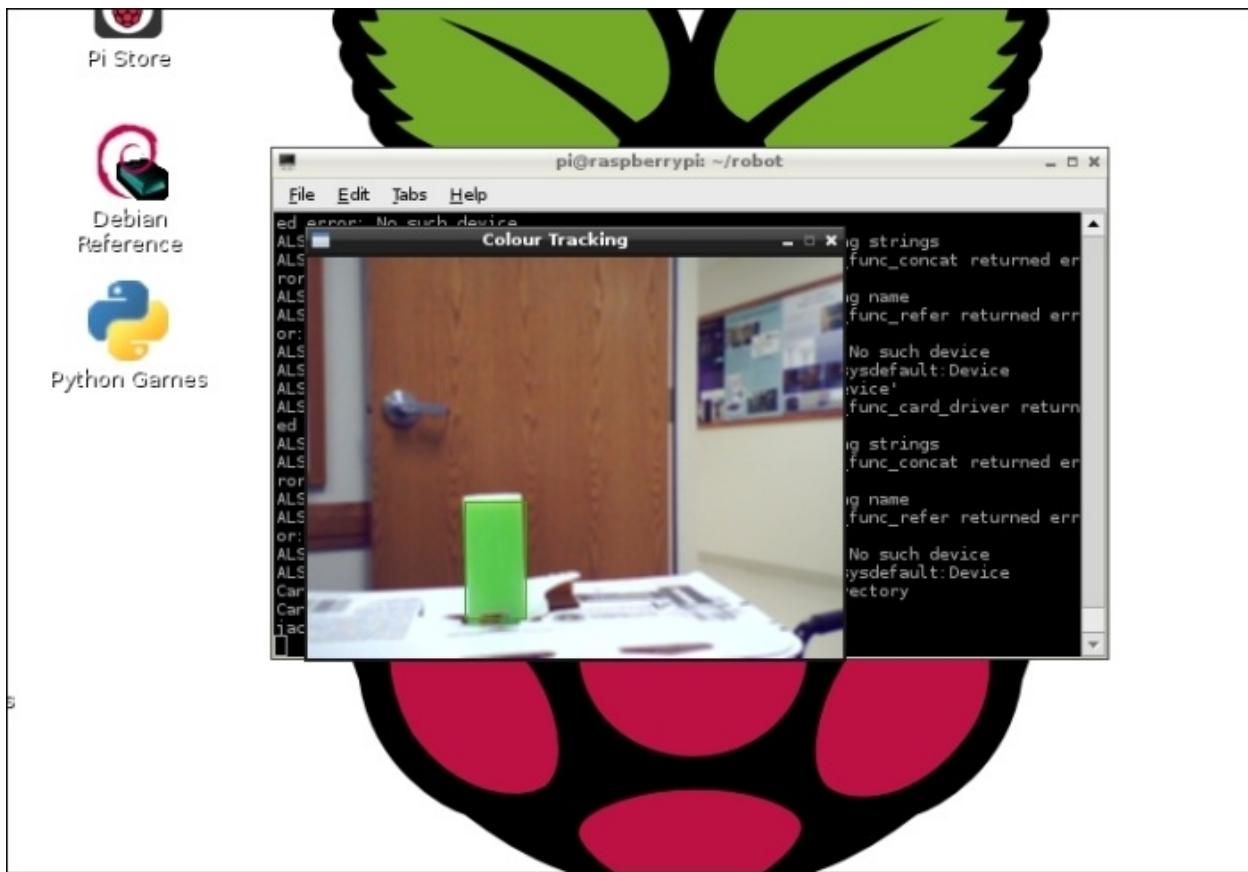
bounding rectangle for each area of `color`. The rectangle is defined by its corners around the blob of `color`.

- `contour = contour.h_next()`—This statement will prepare you for the next `contour` statement (if one exists).
- `size = (rect[2] * rect[3])`—This statement calculates the diagonal length of the rectangle you are evaluating.
- `if size > 100:`—This checks to see whether the area is big enough for our purpose.
- `pt1 = (rect[0], rect[1])`—This statement defines a `pt` variable at the `x` and `y` coordinates on the left-hand side of the blob's rectangular location.
- `pt2 = (rect[0] + rect[2], rect[1] + rect[3])`—This statement defines a `pt` variable at the `x` and `y` coordinates on the right-hand side of the blob's rectangular location.
- `cx = rect[0]`—This statement sets the value of `cx` to the `x` location of `color`.
- `cy = rect[1]`—This statement sets the value of `cy` to the `y` location of the color.
- `cv.Rectangle(img, pt1, pt2, (38, 160, 60))`—Here, you add a rectangle to your original image, identifying where you think it is located.
- `cv.ShowImage("Colour Tracking", img)`—This statement displays the image on the screen.
- `if cx > 280:`—This statement checks to see whether the object is too far to the right.
- `text = '"moving right"`—This statement gets you ready to call the user.
- `subprocess.call('espeak '+text, shell = True)`—This statement calls `espeak` with a message.
- `subprocess.call('./move_right.py')`—This statement calls the Python program, which will move the unit to the right-hand side.
- `if cx < 20 and cx > 0:`—This statement checks to see whether the object is too far to the left. Make sure you exclude `0`, which will be the case initially if there is no object.
- `text = '"moving left"`—This statement gets you ready to call the user.
- `subprocess.call('espeak '+text, shell = True)`—This statement calls `espeak` with a message.
- `subprocess.call('./move_left.py')`—This statement calls the

Python program, which will move the unit to the left-hand side.

- `if cv.waitKey(10) == 27`—This statement kills the program if you press the *Esc* key.
- `break`—This statement stops the program.

Now, you can run the program by typing `python ./follow.py`. You can also type `chmod +x follow.py` and then run the program by typing `./follow.py`. The window should be displayed as shown in the following screenshot:



The green rectangle indicates that the program is following the color green. As the green color is moved towards the edge on the left-hand side, the robot should rotate slightly to the left-hand side. As the green color is moved towards the edge on the right-hand side, the robot should rotate slightly towards the right-hand side.

With OpenCV, it is also possible to perform motion detection. There are a couple of good tutorials on how to do this with OpenCV. One simple example is at <http://www.steinm.com/blog/motion-detection-webcam->

[python-opencv-differential-images/](#). Another example, a bit more complex but more elegant, can be found at <http://stackoverflow.com/questions/3374828/how-do-i-track-motion-using-opencv-in-python>.

While using motion detection, if you put your wind-up walker toy in front of the camera, you will see the output on the webcam (using the code from the second tutorial) as follows:



You can then use it to move the robot to follow the motion.

Using the structure of the Robot Operating System to enable complex functionalities

As you can see, communicating between different aspects of our project can be challenging. In this section, I'm going to introduce you to a special operating system that is designed specifically for use with robotic projects, the **Robot Operating System (ROS)**. This operating system works on top of Linux and provides some interesting functionality.

The operating system is available at www.ros.org. However, the most useful link to the Wiki for the ROS is wiki.ros.org. If you visit this link, you will find a complete set of documentation and downloads. There are also a number of resources that can be useful if you'd like to learn more about the ROS in depth. One of the better resources is the book *Learning ROS for Robotics Programming*, Martinez and Fernandez, Packt Publishing.

This section will not cover the ROS in detail but will introduce you to some of the basics. Start by going to the www.ros.org website. If you select **Install**, you'll note that there is a wide range of Linux operating systems and hardware support. You'll also note that there are several different versions or releases of the ROS. Some of the later releases are Hydro, Fuerte, and Groovy. One of the challenges of using an operating system like this is to decide which release to use. The most recent release will have the largest number of features. It may also have a significant number of issues that may cause problems.

I often prefer using a past release that has been used by a larger number of people; that way, I run into fewer problems. I also don't like to build large packages like this myself. It can take a great deal of time, and often, you'll run into cryptic error messages that can take days to resolve. So, for this tutorial, I am going to use an older version that I have been successful with in the past. It is called **Groovy**, and while it is not the latest version, it is very stable and easy to use.

Note

The ROS brings with it quite a bit of code. If you are going to work with the ROS, I would recommend that you have at least an 8 GB card installed on Raspberry Pi.

I normally follow the installation instructions at <http://wiki.ros.org/groovy/Installation/Raspbian>. The following are the instructions in a step-by-step form:

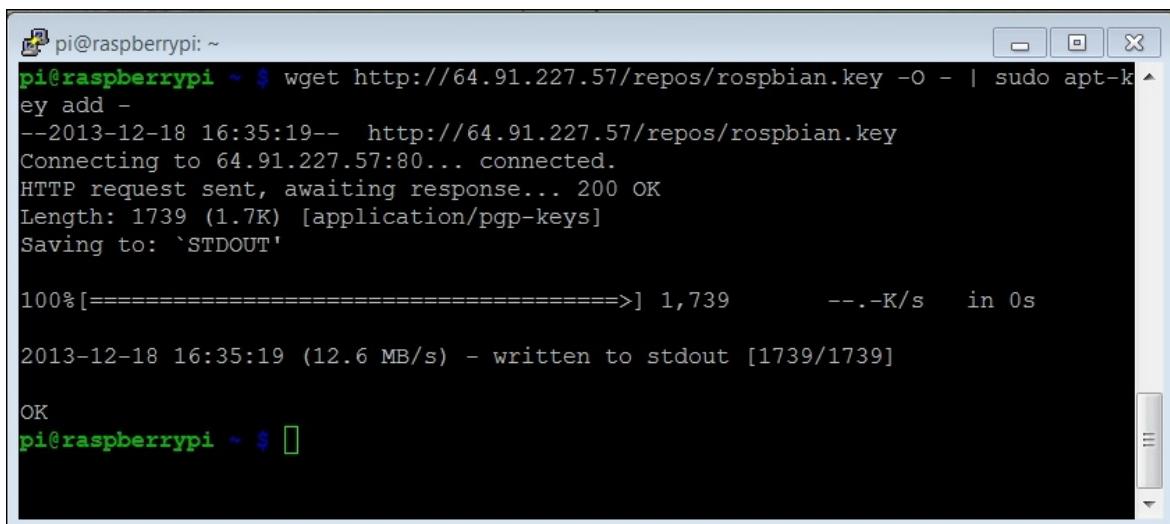
1. Add the repository to your `apt` sources. This is the place where your `apt-get` appears when it is trying to find packages to install. The following is the command:

```
sudo sh -c 'echo "deb  
http://64.91.227.57/repos/rospbian wheezy main" >  
etc/apt/sources.list.d/rospbian.list'
```

2. Next, add the following `apt` key:

```
wget http://64.91.227.57/repos/rospbian.key -O - |  
sudo apt-key add -
```

3. When you add the command, you should get the following screenshot:



A screenshot of a terminal window titled 'pi@raspberrypi: ~'. The window shows the command 'wget http://64.91.227.57/repos/rospbian.key -O - | sudo apt-key add -' being run. The output of the command is displayed, showing the download of the key file from the specified URL and its successful addition to the apt keyring. The terminal window has a standard Linux-style interface with a title bar and scroll bars.

```
pi@raspberrypi: ~
pi@raspberrypi ~ $ wget http://64.91.227.57/repos/rospbian.key -O - | sudo apt-key add -
--2013-12-18 16:35:19--  http://64.91.227.57/repos/rospbian.key
Connecting to 64.91.227.57:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1739 (1.7K) [application/pgp-keys]
Saving to: `STDOUT'

100%[=====] 1,739      --.-K/s   in 0s

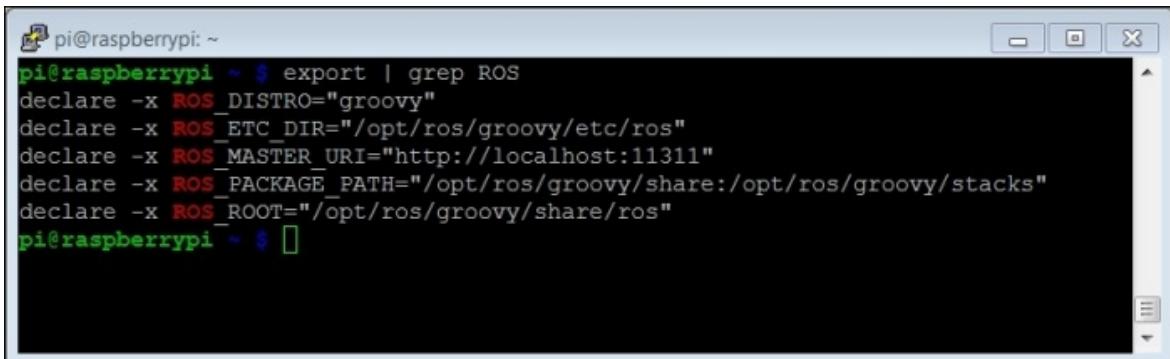
2013-12-18 16:35:19 (12.6 MB/s) - written to stdout [1739/1739]

OK
pi@raspberrypi ~ $ []
```

4. Now, reload the `apt` sources so that your Raspberry Pi will know

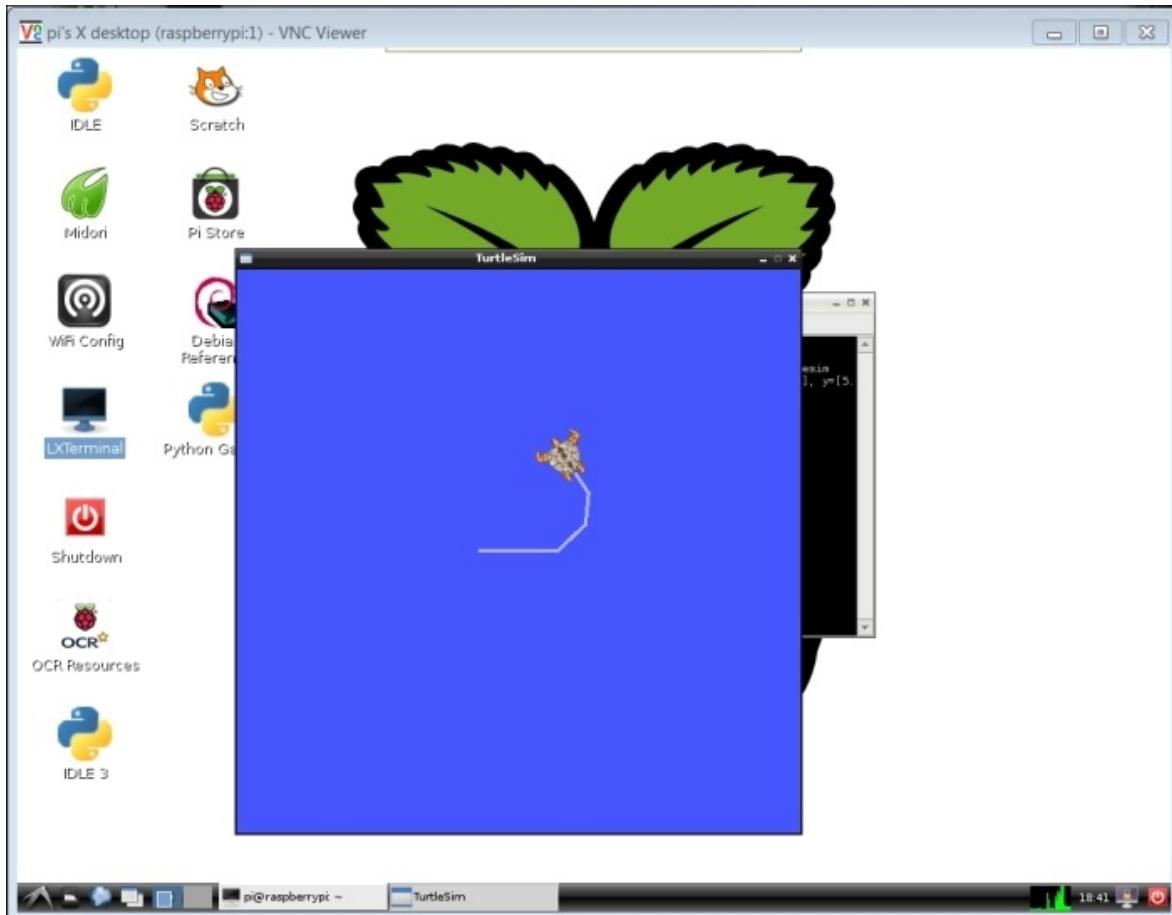
where the files are by typing `sudo apt-get update`.

5. Now, install the ROS packages. Installing `ros_comm` will install all the significant packages you'll need by typing `sudo apt-get install ros-groovy-ros-comm`. The package is quite large and will take some time.
6. Before you can use the ROS, you'll need to install and initialize `rosdep` to let you track dependencies and run some core features. Type `sudo rosdep init` and then `rosdep update`.
7. You'll also want to set up the ROS environment variables so that they are automatically added to your session every time you launch a terminal window. To perform this, type `echo "source optros/groovy/setup.bash" >> ~/.bashrc` and then `source ~/.bashrc`.
8. To add one more tool, `python-rosinstall`, which can help in installing the ROS packages, type `sudo apt-get install python-rosinstall`.
9. As a check to make sure the ROS is set up correctly, type `export | grep ROS`. You should see the following screenshot:

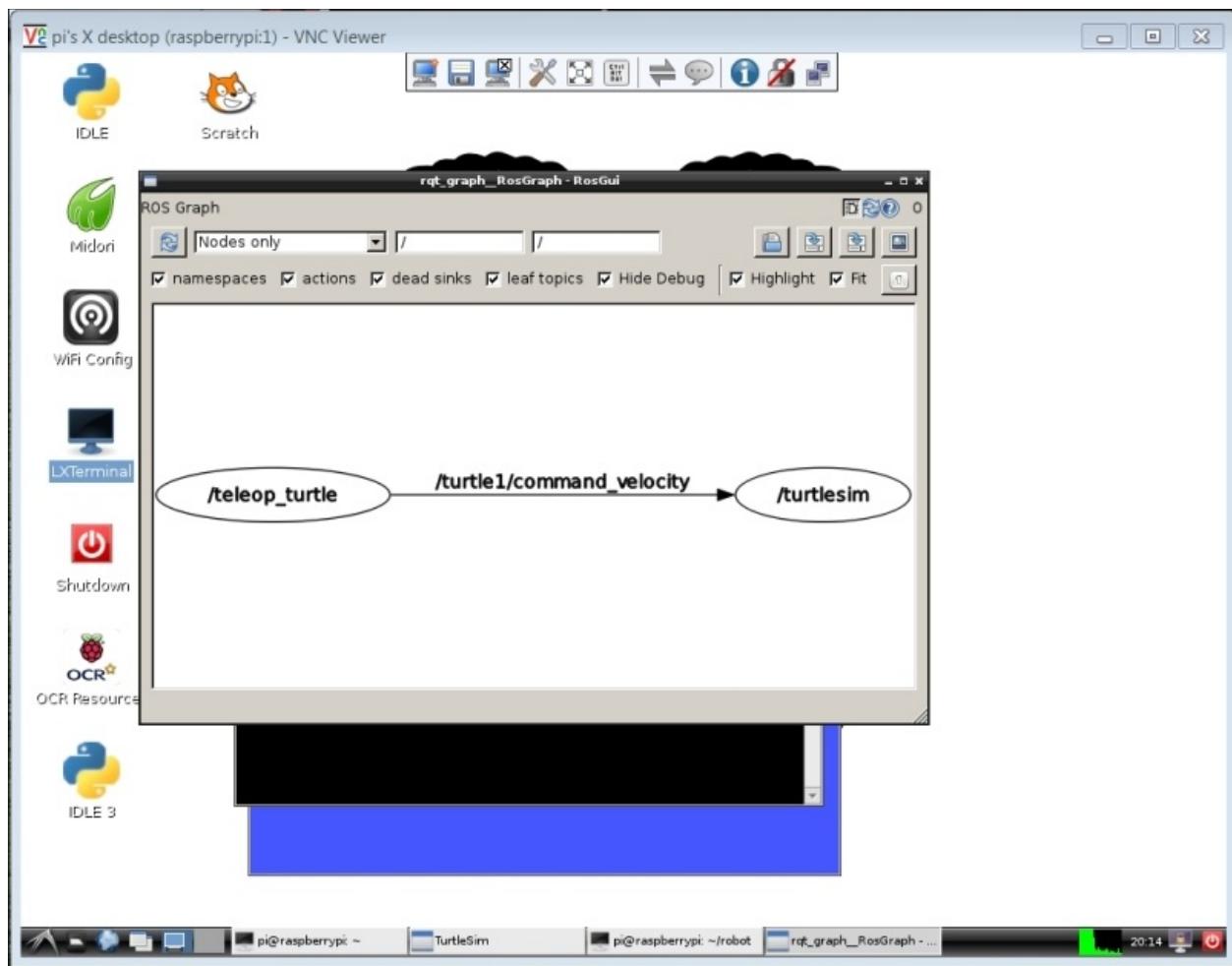


```
pi@raspberrypi: ~
pi@raspberrypi ~ $ export | grep ROS
declare -x ROS_DISTRO="groovy"
declare -x ROS_ETC_DIR="/opt/ros/groovy/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/opt/ros/groovy/share:/opt/ros/groovy/stacks"
declare -x ROS_ROOT="/opt/ros/groovy/share/ROS"
pi@raspberrypi ~ $
```

10. Once installed, you should go through the tutorials at wiki.ros.org/ROS/Tutorials; they will introduce you to the features of the ROS and how to use it in our robotic projects. You will learn how it can provide a systematic way of configuring and communicating between multiple features running in different programs. It even comes with some programs that implement some interesting vision and motor control capabilities. The following is the **TurtleSim** tutorial running on Raspberry Pi:



One of the really powerful features of the ROS is its ability to show you how your information is flowing between applications. As you follow the tutorial, you will end up with two applications running, [teleop_turtle](#) and [turtlesim](#). In this example, you can use an application called [rqt_graph](#) to record the flow of information between the two applications, as shown in the output in the following screenshot:



It will be very difficult to illustrate all of the functionalities of the ROS here. It may take a bit of time, but you can learn to use the ROS to give your robot more functionality.

Summary

Now you can coordinate complex functionalities for your robot. Your robot can walk, talk, see, hear, and even sense its environment, all at the same time. In the next chapter, you'll learn how to construct robots that can fly, sail, and even go under water.

The ROS is fortunately free and open source. It has a very complex set of functionalities, but if you spend some time learning it, you can start using some of the most comprehensive functionality being developed in robotics research today.

Chapter 11. By Land, Sea, and Air

You've built robots that can navigate on land; now let's look at some possibilities for utilizing the tools you have used so far to build some robots that dazzle the imagination. By now, I hope you are comfortable accessing the USB control channels and with talking to servo controllers and other devices that can communicate over USB. Instead of leading you through each step, in this chapter, I'm going to point you in the right direction and then allow you to explore a bit. I'll try to give you some examples using some of the projects that are going on around the Internet.

You don't want to limit your robotic possibilities to just walking or rolling. You'll want your robot to fly, or sail, or swim. In this chapter, you'll see how you can use the capabilities you have already mastered in projects that defy gravity, explore the open sea, or navigate below the open sea.

In this chapter, we will do the following:

- Use Raspberry Pi in robots that can sail
- Use Raspberry Pi in robots that can fly
- Use Raspberry Pi in robots that can go underwater

We need to add hardware to our robotics in order to complete these projects. Since the hardware is different for each of these projects, I'll introduce them in each individual section.

Using Raspberry Pi to sail

Now that you've created platforms that can move on land, let's turn to a completely different type of mobile platform, that is, one that can sail. In this section, you'll discover how to use Raspberry Pi to control your sailboat.

Getting started

Fortunately, sailing on water is about as simple as walking on land. First, however, you need a sailing platform. The following is an image of an RC sailing platform that can be modified to accept control from Raspberry Pi:



In fact, many RC-controlled boats can be modified to add Raspberry Pi. All you need is space to put the processor, the battery, and any additional control circuitry. In this case, the sailing platform has two controls: a rudder, which is controlled by a servo, and a second servo, which controls the position of the sail, as shown in the following image:



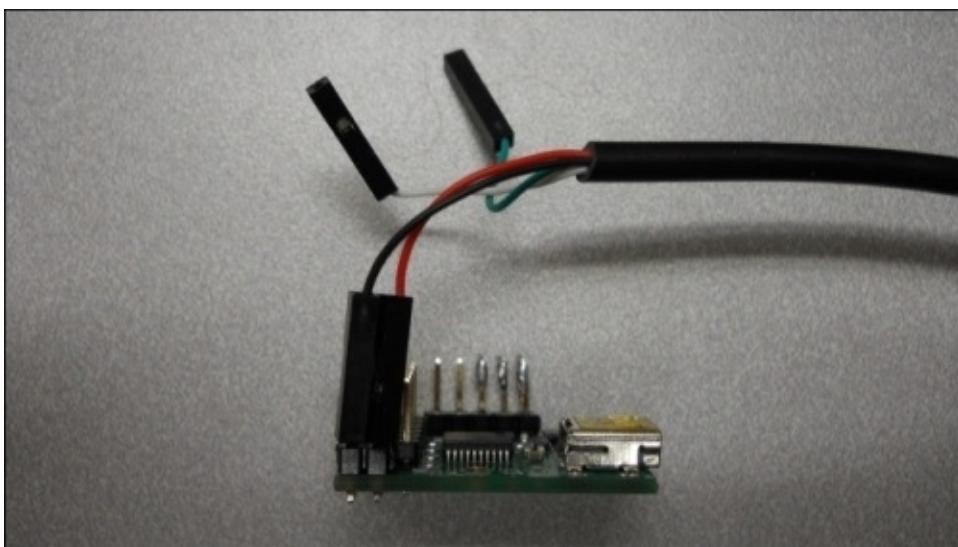
To automate the control of the sailboat, you'll need Raspberry Pi, a battery, and a servo controller. The servo controller I would advise for this project is the one that you used in [Chapter 6, Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs](#). It is a six-servo controller made by Pololu (available at www.pololu.com), and it looks as follows:



The advantage is that this servo controller is very small and fits in limited space. The only challenge is getting a power connection for the device. Fortunately, there is a cable that you can purchase that makes these power connections available from a standard cable. The cable you want is a USB-to-TTL serial/RS232 adapter cable. Make sure that the TTL end of the cable has individual female connectors. You can get this cable at www.amazon.com and also at www.adafruit.com. The following is an image of the cable:

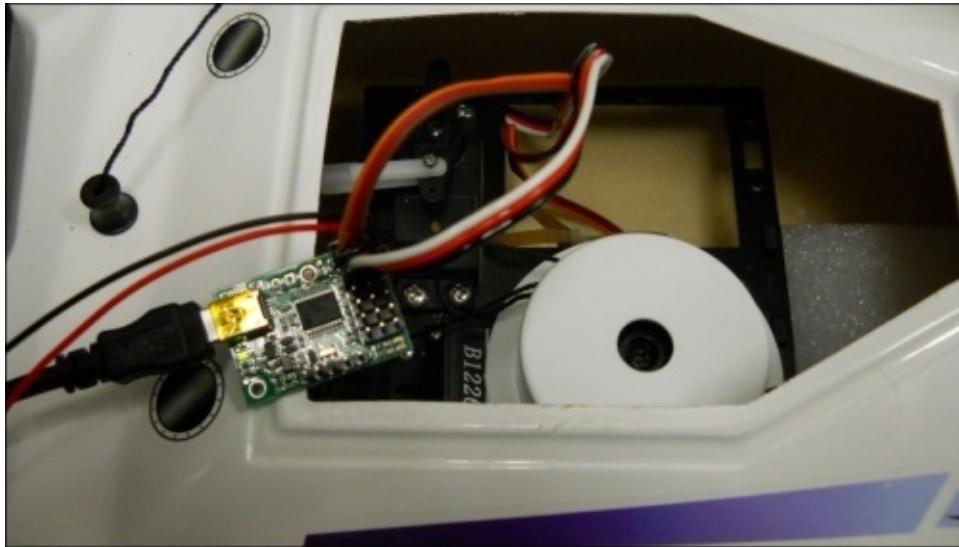


The red and black wires are for connection to the power. These can be connected to the servo controller in the following manner:



Once you have assembled your sailboat, you will first need to hook up

the servo controller to the servos on the boat. You should try to control the servos before installing all the electronics inside the boat, as shown in the following image:



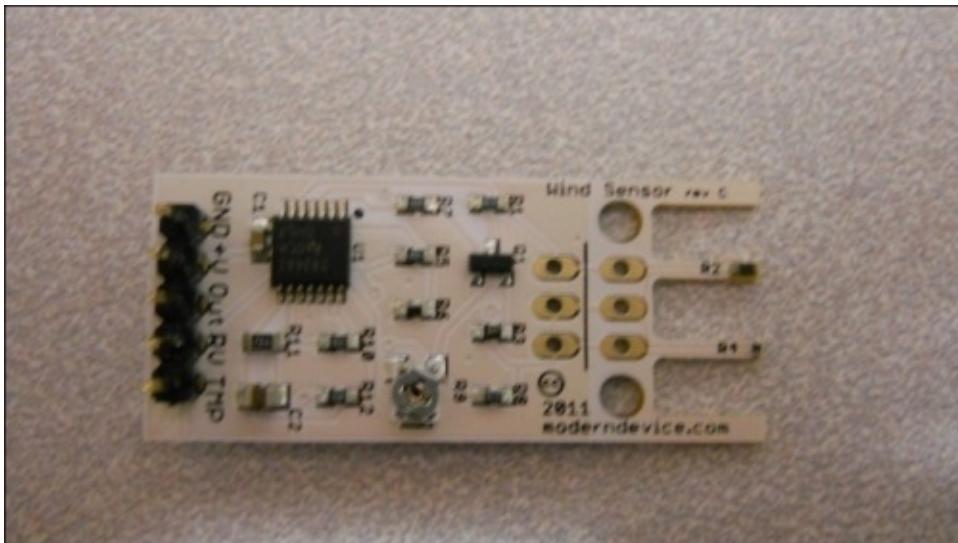
Just as in [Chapter 6, Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs](#), you can use the Maestro Servo Controller software to control the servo controller from your PC. When you are ready to hook it up to Raspberry Pi, you can start with the same Python program you used in [Chapter 6, Making the Unit Very Mobile – Controlling the Movement of a Robot with Legs](#). You will probably want to control the system without a wired connection, so you can use the principles that you learned in [Chapter 8, Going Truly Mobile – The Remote Control of Your Robot](#).

It may be a bit challenging if you are using the standard 2.4 GHz keyboard or a smaller 2.4 GHz controller. One possible solution is wireless LAN, where you can set up your own ad hoc wireless network using a router connected to a laptop. Also, as noted in [Chapter 8, Going Truly Mobile –The Remote Control of Your Robot](#), many cell phones have the ability to set up a wireless hot spot, which can create a wireless network so that you can communicate remotely with your sailboat.

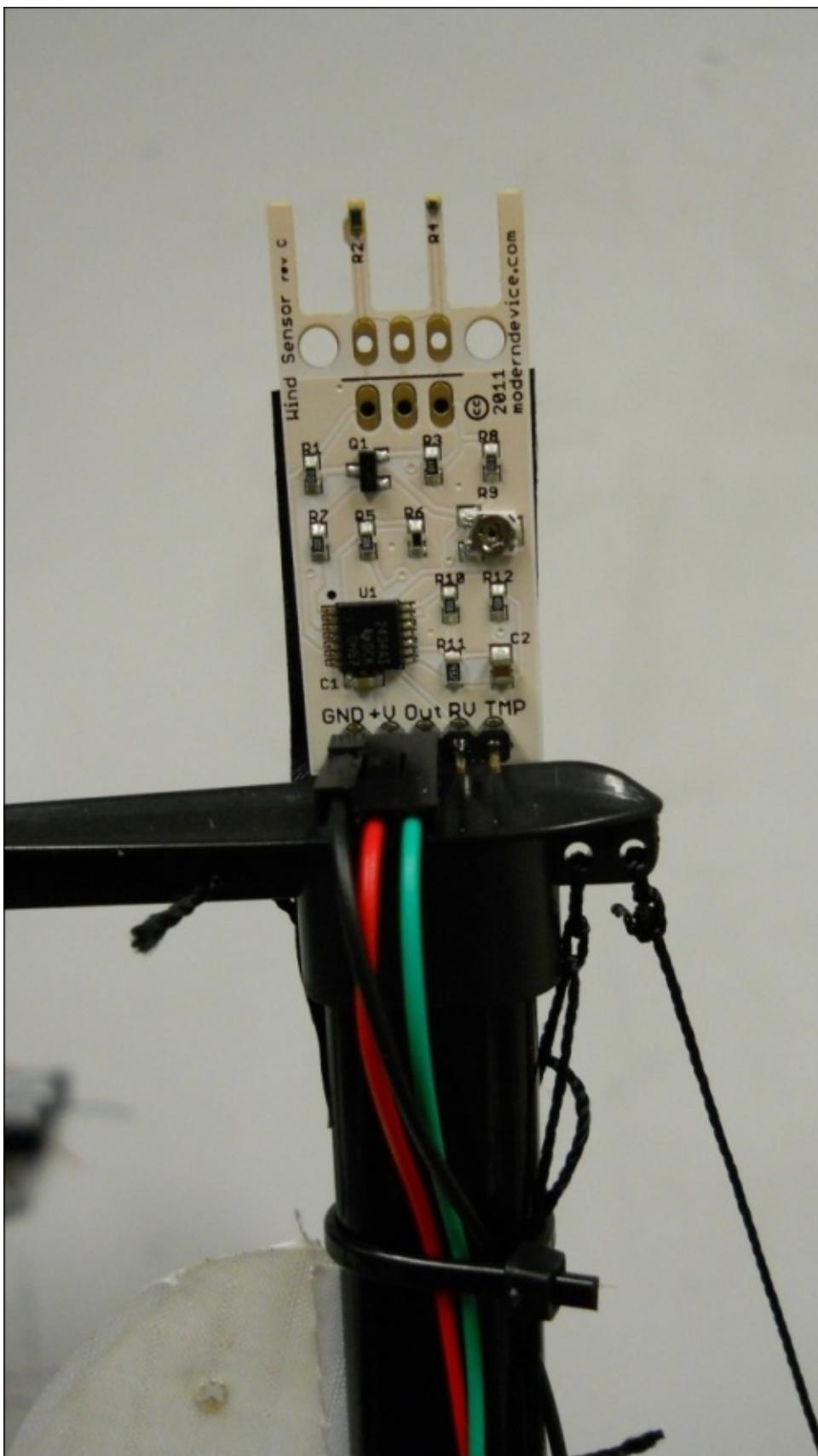
Another possible solution is to use ZigBee wireless devices to connect your sailboat to a computer. We already covered the details in [Chapter 8, Going Truly Mobile – The Remote Control of Your Robot](#). You'll need two of them and a USB shield for each. You can get these at a number of

places, including www.adafruit.com. If you can connect your computer and Raspberry Pi via this wireless network, the advantage is that it carries communications to and from your sailboat and can have a range of up to a mile using the right devices.

Now you can sail your boat, controlling it all through an external keyboard or through a ZigBee wireless network from your computer. If you want to fully automate your system, you could add your GPS and then have your sailboat sail to each of the programmed positions. One additional item you may want to add to make the system fully automated is a wind sensor. The following is an image of a wind sensor that is fairly inexpensive if you buy it from www.moderndevice.com:



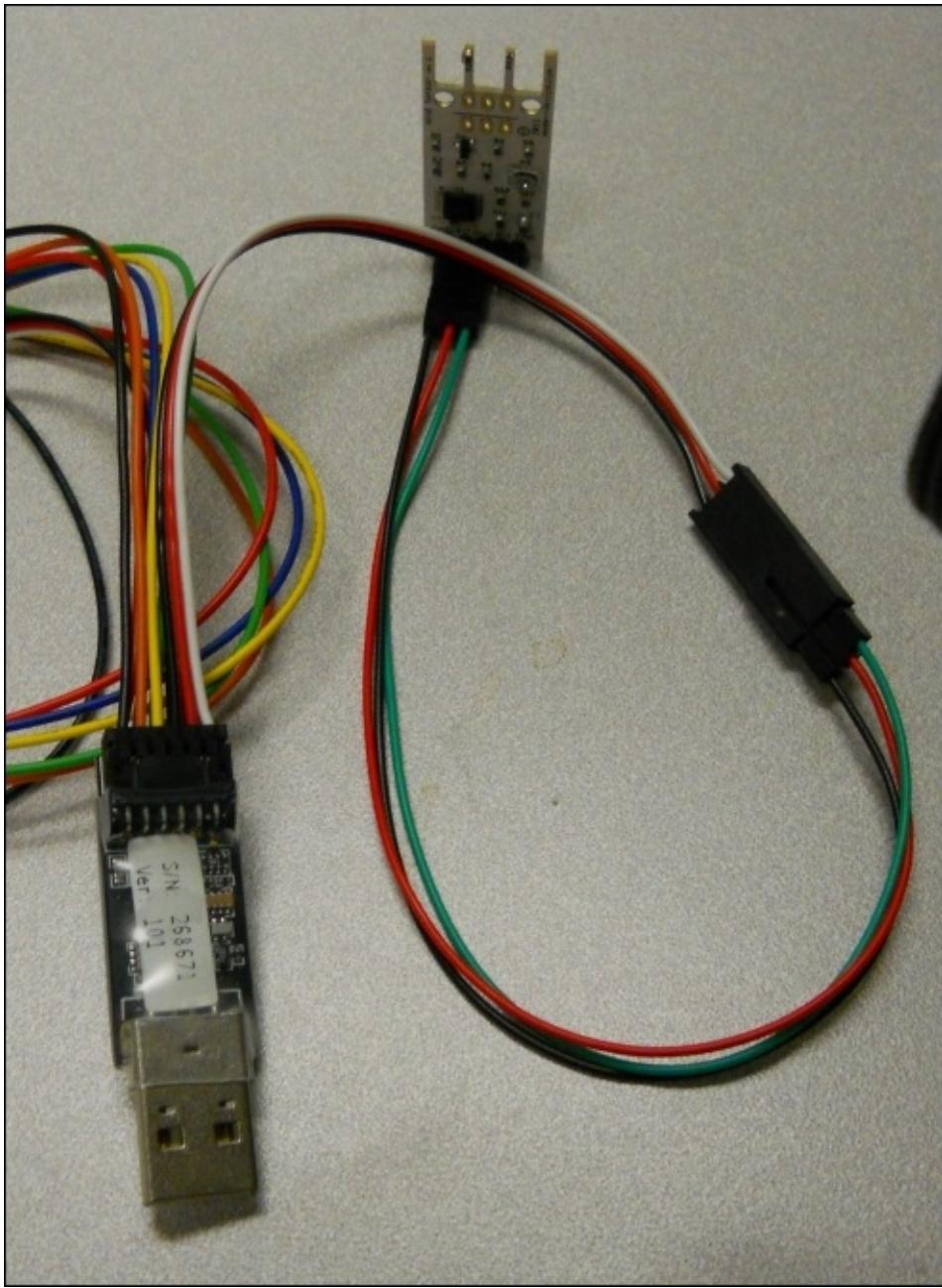
You can mount it to the mast if you'd like; I used a small piece of heavy-duty tape and mounted it to the top of the mast, as shown in the following image:



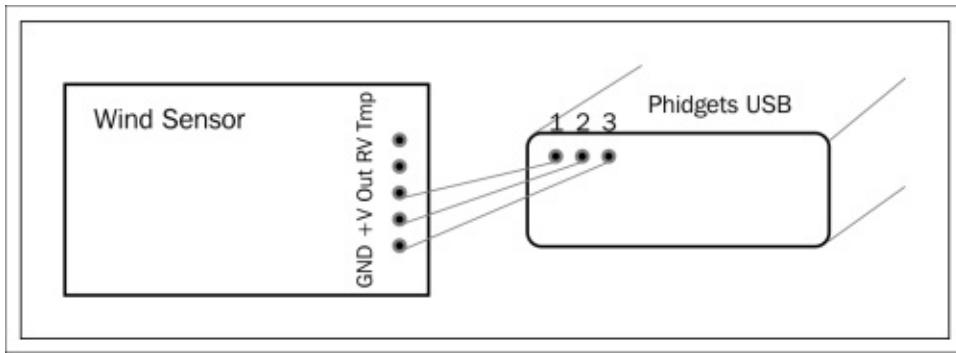
To add this to your system, you'll also need a way to take the analog input from the sensor and send it to Raspberry Pi. The easiest way to do this is to add a USB device that samples the analog signal and reports the measurements over the USB bus. The device I like to use is the PhidgetInterfaceKit 2/2/2 from www.phidgets.com. The following is an image of this device:



The following is an image of the wind sensor connected to the converter:



The following wiring diagram shows how the wind sensor interfaces with the Phidgets USB device:



Now you can access the wind speed from the USB connection in the same way that you received data from the other USB devices that you have already used. The Phidgets website will lead you through the download process; I chose Python as my language and downloaded the appropriate libraries and sample code. When I run this sample code, I get the following output while blowing on the sensor:

```
ubuntu@ubuntu-armhf: ~/Python_2.1.8.20130926/Python
InterfaceKit 268671: Sensor 0: 495
InterfaceKit 268671: Sensor 0: 505
InterfaceKit 268671: Sensor 0: 515
InterfaceKit 268671: Sensor 0: 526
InterfaceKit 268671: Sensor 0: 536
InterfaceKit 268671: Sensor 0: 545
InterfaceKit 268671: Sensor 0: 555
InterfaceKit 268671: Sensor 0: 547
InterfaceKit 268671: Sensor 0: 537
InterfaceKit 268671: Sensor 0: 526
InterfaceKit 268671: Sensor 0: 516
InterfaceKit 268671: Sensor 0: 507
InterfaceKit 268671: Sensor 0: 497
InterfaceKit 268671: Sensor 0: 487
InterfaceKit 268671: Sensor 0: 478
InterfaceKit 268671: Sensor 0: 468
InterfaceKit 268671: Sensor 0: 458
InterfaceKit 268671: Sensor 0: 448
InterfaceKit 268671: Sensor 0: 438
InterfaceKit 268671: Sensor 0: 428
InterfaceKit 268671: Sensor 0: 418
InterfaceKit 268671: Sensor 0: 408
InterfaceKit 268671: Sensor 0: 399
```

Now that you have a way to measure your location and the wind, you can use your Raspberry Pi to sail your boat all by itself. I am not a sailor myself, but you'll want to use the wind sensor to get a sense of the direction of the wind and then use classic sailing techniques such as

tacking (moving back and forth to use the wind effectively) to sail.

While constructing this, you'll need to be careful with waterproofing, especially while sailing in heavy wind. Think about attaching a hatch that covers the electronics securely. I added small screws and tabs and also some waterproof sealant to hold the hatch.

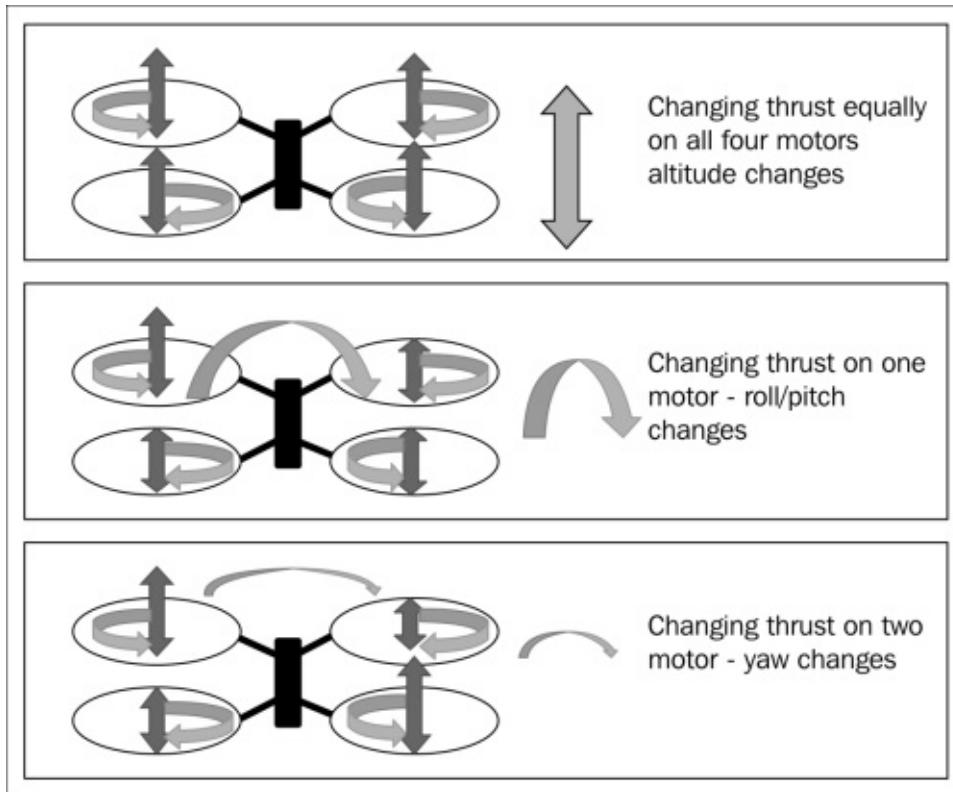
Using Raspberry Pi to fly robots

You've now built robots that can move around on a wheeled structure, robots that have legs, and robots that can sail. You can also build robots that can fly by relying on Raspberry Pi to control their flight. There are several possible ways to incorporate Raspberry Pi into a flying robotic project, but the most straightforward way is to add it to a quadcopter project.

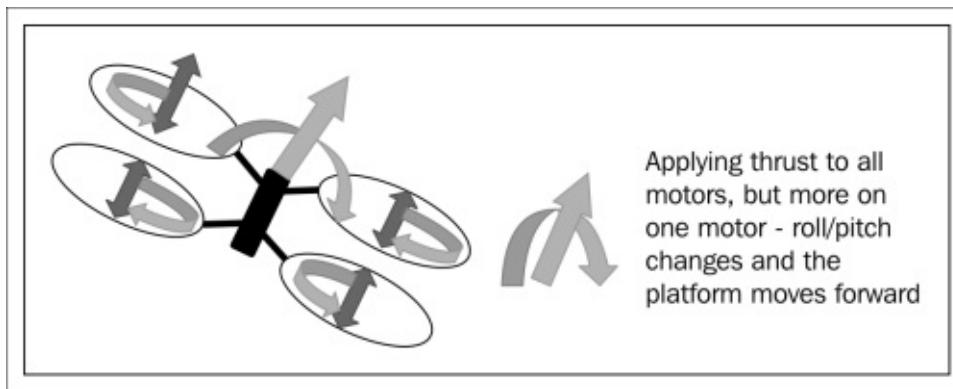
Quadcopters are a unique subset of flying platforms that have become very popular in the last few years. They are a flying platform that utilize the same vertical lift concept as helicopters; however, they employ not one, but four motor/propeller combinations to provide an enhanced level of stability. The following is an image of such a platform:



The quadcopter has two sets of counter-rotating propellers; this simply means that two of the propellers rotate one way while the other two rotate the other way to provide thrust in the same direction. This provides a platform that is inherently stable. Controlling the thrust on all four motors allows you to change the pitch, roll, and yaw of the device. The following figure may be helpful to understand the same:

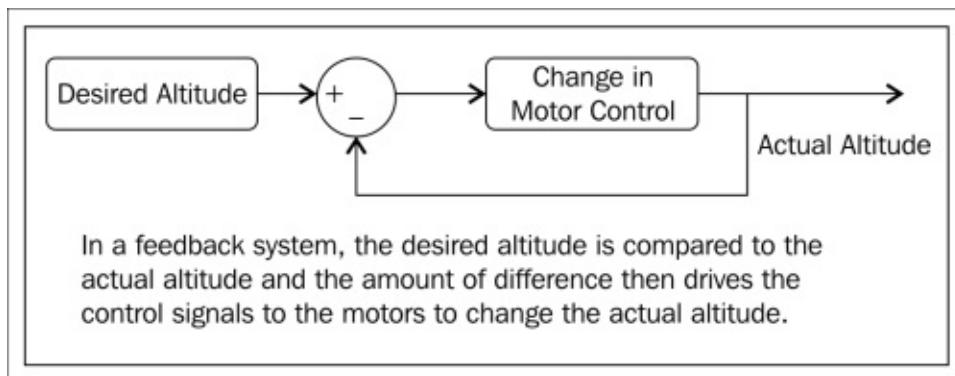


As you can see, controlling the relative speeds of the four motors allows you to control the various ways in which the device can change position. To move forward, or in any direction really, we can combine a change in the roll/pitch with a change in the thrust, so that instead of going up, the device will move forward, as shown in the following figure:



In a perfect world, you might, knowing the components you used to build your quadcopter, know exactly how much control signal to apply to get a certain change in the roll/pitch/yaw or altitude of your quadcopter. You cannot rely on a fixed set of signals as there are simply too many aspects

of your device that can vary. Instead, this platform uses a series of measurements for its position, pitch/roll/yaw, and altitude, and then adjusts the control signals to the motors to achieve the desired result. We call this feedback control. The following figure denotes a feedback system:



As you can see, if your quadcopter is too low, the difference between the desired altitude and the actual altitude will be positive, and the motor control will increase the voltage supplied to the motors, increasing the altitude. If the quadcopter is too high, the difference between the desired altitude and the actual altitude will be negative, and the motor control will decrease the voltage supplied to the motors, decreasing the altitude. If the desired altitude and the actual altitude are equal, then the difference between the two will be zero and the motor control will be held at its current value. Thus the system stabilizes even if the components aren't perfect or if a wind comes along and blows the quadcopter up or down.

One function of Raspberry Pi in this type of robotic project is to actually coordinate the measurement and control of the quadcopter's pitch, roll, yaw, and altitude. This can be done; however, it is a very complex task, and the details of its implementation are beyond the scope of this book. It is unclear whether Raspberry Pi has the horsepower to execute and keep up with this type of application.

However, Raspberry Pi can still be utilized in this type of robotic project by introducing another embedded processor to carry out the low-level control and using Raspberry Pi to manage high-level tasks, such as using the vision system of Raspberry Pi to identify a colored ball and then guiding the platform toward it. Or, as in the sailboat example, you can use Raspberry Pi to coordinate GPS tracking and long range communications

via ZigBee or wireless LAN. I'll cover an example of this in this section.

The first thing you'll need is a quadcopter. There are three approaches to this, which are as follows:

- Purchase an already assembled quadcopter
- Purchase a kit and construct it yourself
- Buy the parts separately and construct the quadcopter

In any case, one of the easiest ways to add Raspberry Pi to your quadcopter is to choose one that uses ArduPilot as its flight control system. This system uses a flight version of Arduino to do the low-level feedback control we talked about earlier. The advantage of this system is that you can talk to the flight control system via USB.

There are a number of assembled quadcopters available that use this flight controller. One place to start is at www.ardupilot.com. This will give you some information on the flight controller, and the store has several preassembled quadcopters. If you are thinking of assembling your own quadcopter, a kit is the right approach. Try www.unmannedtechshop.co.uk/multi-rotor.html or www.buildyourown-drone.co.uk/ArduCopter-Kits-s/33.htm, as these websites not only sell assembled quadcopters, but assembling kits as well.

If you'd like to assemble your own kit, there are several good tutorials about choosing all the right parts and assembling your quadcopter. Try one of the following links:

- blog.tkjelectronics.dk/2012/03/quadcopters-how-to-get-started
- blog.oscarliang.net/build-a-quadcopter-beginners-tutorial-1/
- <http://www.arducopter.co.uk/what-do-i-need.html>

All of these links have excellent instructions.

You may be tempted to purchase one of the very inexpensive quadcopters that are being offered on the market. For this project, you will need the following two key characteristics of the quadcopter:

- The quadcopter flight control will need a USB port so that you can

connect Raspberry Pi to it

- It will need to be large enough and have enough thrust to carry the extra weight of Raspberry Pi, a battery, and perhaps a webcam or other sensing devices

No matter which path you choose, another excellent source for information is <http://code.google.com/p/arducopter>. This gives you some information on how ArduPilot works and also talks about Mission Planner, an open source control software that will be used to control ArduPilot on your quadcopter. This software runs on PC and communicates to the quadcopter in one of two ways, either through a USB connection directly or through a radio connection. It is the USB connection that you will use to communicate between Raspberry Pi and ArduPilot.

The first step for working in this space is to building your quadcopter and getting it to work with an RC radio. When you allow Raspberry Pi to control it later, you may still want to have the RC radio handy, just in case things don't go quite as planned.

When the quadcopter is flying well, based on your ability to control it using the RC radio, you should then begin to use ArduPilot in autopilot mode. To do this, download the software from www.ardupilot.com/downloads. You can then run the software; you should see something like the following screenshot:



You can then connect ArduPilot to the software and click on the **CONNECT** button in the upper-right corner. You should then see something like the following screenshot:



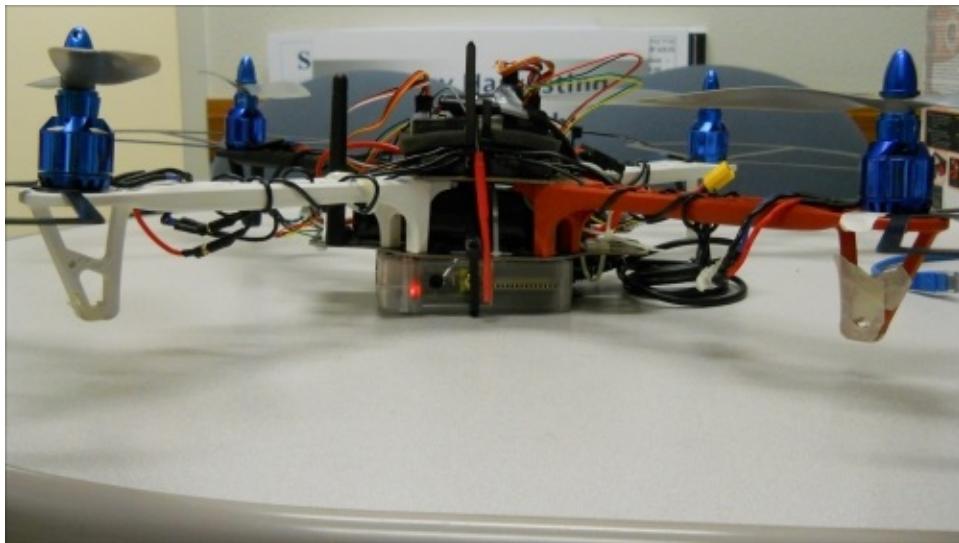
We will not walk through how to use the software to plan an automated

flight path; there is plenty of documentation for that on the website, www.ardupilot.com. Note that in this configuration, you have not connected the GPS on ArduPilot.

What you want to do is hook up Raspberry Pi to ArduPilot on your quadcopter so that it can control the flight of your quadcopter just as Mission Planner does, but at a much lower and more specific level. You will use the USB interface just as Mission Planner does.

To connect the two devices, you'll need to modify the Arduino code, create some Raspberry Pi code, and then simply connect the USB interface of Raspberry Pi to ArduPilot. You can issue the yaw, pitch, and roll commands to the Arduino to guide your quadcopter to wherever you want it to go. The Arduino will take care of keeping the quadcopter stable. An excellent tutorial on how to accomplish this can be found at <http://oweng.myweb.port.ac.uk/buildyour-own-quadcopter-autopilot/>.

The following is an image of my configuration; I put Raspberry Pi in a plastic case and mounted the battery and Raspberry Pi below the quadcopter's main chassis:



Now that you can fly your quadcopter using Raspberry Pi, you can use the same GPS and ZigBee or wireless LAN capabilities mentioned in the last section to make your quadcopter semiautonomous.

Your quadcopter can act completely autonomously as well. Adding a 3G

modem to the project allows you to track your quadcopter no matter where it might go, as long as it can receive a cell signal. The following is an image of such a modem:



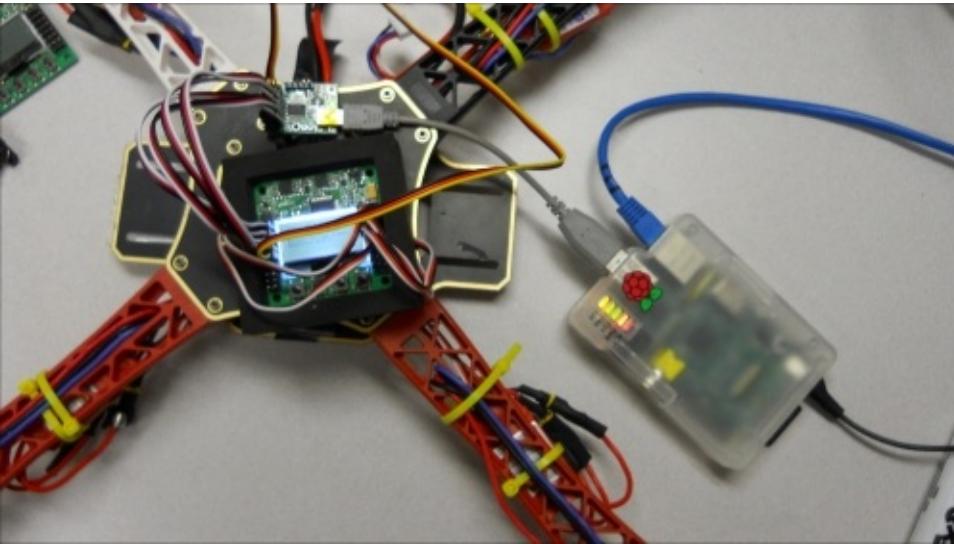
This can be purchased on Amazon or from any cellular service provider. Once you have purchased your modem, simply google instructions on how to configure it in Linux. A sample project that puts it all together can be found at <http://www.skydrone.aero> it is based on BeagleBone Black, a small Linux processor with similar capabilities to Raspberry Pi.

Another possibility for an aerial project is a plane based on ArduPilot and controlled by Raspberry Pi. Look at <http://plane.ardupilot.com/> for information on controlling a fixed wing aircraft with ArduPilot. It would be fairly straightforward to add Raspberry Pi to this configuration.

If you are a bit more confident in your aeronautic capabilities, you can also build a quadcopter using Raspberry Pi and a simpler, less expensive flight control board. The one I like to use is the Hobby King KK2.0 flight control board, shown in the following image:



This flight control board takes its flight inputs via a set of input signals and then sends out control signals to the four motor controllers. The good thing about this controller is that it has built-in flight sensors, and so it can handle the feedback control for the motors. The inputs will come in as electric commands to turn, bank, go forward, or increase the altitude. Normally, these would come from the RC radio receiver. For this project, you can insert Raspberry Pi and the Maestro Servo Controller we covered in [Chapter 6, Making the Unit Very Mobile - Controlling the Movement of a Robot with Legs](#). The following is an image of how you connect Raspberry Pi, the servo controller, and the flight controller:



A close up of the connections between the servo controller and the flight controller board is shown in the following image. Make sure you do not

connect power to the servo controller; it does not need to supply power. Just send the appropriate signals via the servo lines to the receiver input.



You can follow the standard instructions to calibrate and control your quadcopter, with Raspberry Pi creating the commands using the servo controller. It is perhaps best to first use the Pololu Maestro Control Center software to do the calibration; you can then write a program based on the control program you wrote in [Chapter 8, Going Truly Mobile – The Remote Control of Your Robot](#), where key presses from the keyboard can be used to control the quadcopter to make it go up, down, right or left, bank, or turn. An example of this type of system, albeit using an even less expensive controller, is shown at <http://www.instructables.com/id/Autonomous-Cardboard-Raspberry-Pi-Controlled-Quad/>. Another example, this time with a tricopter, can be found at <http://bitoniau.blogspot.com/2013/05/using-raspberry-pi-wifi-as-transmission.html>.

Using Raspberry Pi to make the robot swim underwater

You've explored the possibilities of walking robots, flying robots, and sailing robots. The final frontier is robots that can actually maneuver under the water. It only makes sense to use the same techniques that you've mastered to explore the undersea world. In this section, I'll explain how to use the capabilities that you have already developed in a **Remote Operated Vehicle (ROV)** robot. There are, of course, some interesting challenges that come with this type of project, so get ready to get wet!

As with the other projects in this chapter, there are possibilities of either buying an assembled robot or assembling one yourself. If you'd like to buy an assembled ROV, try <http://openrov.com>. This project, funded by Kickstarter, provides a complete package, albeit with electronics based on BeagleBone Black. If you are looking to build your own robot, there are several websites that document possible instructions for you to follow, such as <http://dzlsevilgeniuslair.blogspot.dk/search/label/ROV>. Additionally, <http://www.mbari.org/education/rov/> and <http://www.engadget.com/2007/09/04/build-your-own-underwater-rov-for-250/> show platforms to which you can add Raspberry Pi.

Whether you have purchased a platform or designed your own, the first step is to engage Raspberry Pi to control the motors. Fortunately, you should have a good idea of how to do this, as *Chapter 5, Creating Mobile Robots on Wheels*, covers how to use a set of DC motor controllers to control DC motors. In this case, you will need to control three or four motors based on which kind of platform you build. Interestingly, the problem of control is quite similar to the quadcopter control problem. If you use four motors, the problem is almost exactly the same; except instead of focusing on up and down, you are focusing on moving the ROV forward.

There is one significant difference: the ROV is inherently more stable. In the quadcopter, your platform needed to hover in the air, a challenging control problem because the resistance of air is so small and the platform responds very quickly to changes. Because the system is so dynamic, a

microprocessor is needed to respond to the real-time measurements and to individually control the four motors to achieve stable flight.

This is not the case underwater, where our platform does not want to move dramatically. In fact, it takes a good bit of power to make the platform move through the water. You, as the operator, can control the motors with enough precision to get the ROV moving in the direction you want.

Another difference is that wireless communication is not available to you underwater, so you'll be tethering your device and running controls from the surface to the ROV through wires. You'll need to send control signals and video so you can control the ROV in real time.

You already have all the tools at your disposal for this project. As noted from [Chapter 5, Creating Mobile Robots on Wheels](#), you know how to hook up DC motor controllers; you'll need one for each motor on your platform. [Chapter 4, Adding Vision to Raspberry Pi](#), shows you how to set up a webcam so that you can see what is around you. All of this can be controlled from a laptop at the surface, connected via a LAN cable and running vncserver.

Creating the basic ROV platform should open the possibility of exploring the undersea world. An ROV platform has some significant advantages. It is very difficult to lose (you have a cable attached), and because the device tends to move quite slowly, the chances for catastrophic collisions are significantly less than many other projects. The biggest problem, however, is keeping everything dry!

Summary

Now you have access to a wide array of different robotics projects that can take you over land, on the sea, or in the air. Be prepared for some challenges and always plan on a bit of rework. Well, we've reached the end of the book, which hopefully is only a beginning to your robotic adventures. We've covered how to construct robots that can see, talk, listen, and move in a wonderful variety of ways. Feel free to experiment; there are so many additional capabilities that can be added to make your robot even more amazing.

Adding infrastructure such as the Robot Operating System opens even more opportunities for complex robotic systems. Perhaps one day you'll even build a robot that passes the Turing test; that is one robot that may be mistaken for a human being!

Index

A

- Advanced Linux Sound Architecture (ALSA) libraries / [Hooking up the hardware to make and input sound](#)
- aplay / [Hooking up the hardware to make and input sound](#)
- ArduPilot
 - URL / [Using Raspberry Pi to fly robots](#)

B

- Bison / [Using PocketSphinx to accept your voice commands](#)
- board
 - about / [The unveiling](#)
 - powering / [The unveiling](#)
 - accessing, remotely / [Accessing the board remotely](#)

C

- C/C++ programming language
 - about / [Introduction to the C/C++ programming language](#)
- Carnegie Mellon University (CMU)
 - URL / [Using PocketSphinx to accept your voice commands](#)
- cd (change directory) command / [Basic Linux commands on Raspberry Pi](#)
- clear command / [Basic Linux commands on Raspberry Pi](#)
- colored objects
 - detecting, vision library used / [Using the vision library to detect](#)

colored objects

- commands
 - interpreting / [Interpreting commands and initiating actions](#)
- copy / [Basic Linux commands on Raspberry Pi](#)
- cp filename1 filename2 command / [Basic Linux commands on Raspberry Pi](#)
- Ctrl + K command / [Creating, editing, and saving files on Raspberry Pi](#)
- Ctrl + X Ctrl + C command / [Creating, editing, and saving files on Raspberry Pi](#)
- Ctrl + X Ctrl + S command / [Creating, editing, and saving files on Raspberry Pi](#)
- Ctrl + _ command / [Creating, editing, and saving files on Raspberry Pi](#)
- cut and paste / [Creating, editing, and saving files on Raspberry Pi](#)

D

- Dagu Rover 5 Tracked Chassis / [Gathering the required hardware](#)
- DC
 - about / [Gathering the required hardware](#)
- Debian distribution / [Installing the operating system](#)
- def getch()\$ function / [Using the keyboard to control your project](#)
- Devices and Printers option / [Working remotely with your Raspberry Pi through ZigBee](#)
- df -h command / [Getting started](#)
- display
 - connecting / [Hooking up a keyboard, mouse, and display](#)
- DOF (degrees of freedom)
 - about / [Gathering the hardware](#)

- DVI input / [Hooking up a keyboard, mouse, and display](#)

E

- Eclipse / [Introduction to the C/C++ programming language](#)
- else statement / [The if statement](#)
- emacs
 - about / [Creating, editing, and saving files on Raspberry Pi](#)
 - Ctrl + X Ctrl + S command / [Creating, editing, and saving files on Raspberry Pi](#)
 - Ctrl + X Ctrl + C command / [Creating, editing, and saving files on Raspberry Pi](#)
 - Ctrl + K command / [Creating, editing, and saving files on Raspberry Pi](#)
 - Ctrl + _ command / [Creating, editing, and saving files on Raspberry Pi](#)
- Enable Boot to Desktop/Scratch option / [Installing the operating system](#)
- Espeak
 - about / [Using Espeak to allow our projects to respond in a robot voice](#)
 - default settings, creating / [Using Espeak to allow our projects to respond in a robot voice](#)
- espeak -f speak.txt command / [Using Espeak to allow our projects to respond in a robot voice](#)

F

- -f option / [Using Espeak to allow our projects to respond in a robot voice](#)
- Frame Rate setting / [Connecting the USB camera to Raspberry Pi and viewing the images](#)
- functions

- about / [Working with functions](#)

G

- general-purpose input/output (GPIO) pins / [Using a motor controller to control the speed of your platform](#)
- general control structure
 - creating / [Creating a general control structure](#)
- GPIO (General Purpose Input/Output)
 - about / [Gathering the hardware](#)
- GPS device
 - Raspberry Pi, connecting to / [Connecting Raspberry Pi to a GPS device](#)
 - accessing, programmatically / [Accessing the GPS programmatically](#)
- Groovy
 - about / [Using the structure of the Robot Operating System to enable complex functionalities](#)

H

- hardware
 - gathering / [Gathering the hardware](#)
- HDMI input / [Hooking up a keyboard, mouse, and display](#)

I

- IDE (Integrated Development Environment) / [Introduction to the C/C++ programming language](#)

- ifconfig command / [Accessing the board remotely](#)
- if statement
 - about / [The if statement](#)
- inet addr / [Accessing the board remotely](#)
- INFO statement / [Using PocketSphinx to accept your voice commands](#)
- infrared sensor
 - Raspberry Pi, connecting to / [Connecting Raspberry Pi to an infrared sensor](#)
- int main() function / [Introduction to the C/C++ programming language](#)

K

- keyboard
 - connecting / [Hooking up a keyboard, mouse, and display](#)
 - using, to control project / [Using the keyboard to control your project](#)
- kill / [Creating, editing, and saving files on Raspberry Pi](#)

L

- LAN router / [Gathering the hardware](#)
- legged mobile platform
 - about / [Gathering the hardware](#)
- legged robot
 - about / [Gathering the hardware](#)
- Linux commands
 - List-short (ls) command / [Basic Linux commands on Raspberry](#)

Pi

- rm filename command / [Basic Linux commands on Raspberry Pi](#)
 - mv filename1 filename2 command / [Basic Linux commands on Raspberry Pi](#)
 - cp filename1 filename2 command / [Basic Linux commands on Raspberry Pi](#)
 - mkdir directoryname command / [Basic Linux commands on Raspberry Pi](#)
 - clear command / [Basic Linux commands on Raspberry Pi](#)
 - sudo command / [Basic Linux commands on Raspberry Pi](#)
-
- List-short (ls) command / [Basic Linux commands on Raspberry Pi](#)
 - list short (ls) / [Hooking up the hardware to make and input sound](#)
 - Logitech keyboard / [Gathering the hardware](#)
 - LXDE Windows system / [Installing the operating system](#)
 - LXTerminal application / [Basic Linux commands on Raspberry Pi](#)
 - LXTerminal selection / [Installing the operating system](#)

M

- make directory / [Basic Linux commands on Raspberry Pi](#)
- makefile program / [Introduction to the C/C++ programming language](#)
- makefile system / [Interpreting commands and initiating actions](#)
- male-male jumper wires
 - about / [Gathering the required hardware](#)
- milliAmpHours (mAh) / [Gathering the hardware](#)
- mkdir directoryname command / [Basic Linux commands on Raspberry Pi](#)
- M key / [Hooking up the hardware to make and input sound](#)
- mobile platform
 - controlling programmatically, Raspberry Pi used / [Controlling your mobile platform programmatically using Raspberry Pi](#)
 - voice commands, issuing by / [Making your mobile platform truly mobile by issuing voice commands, Making your mobile platform truly mobile by issuing voice commands](#)
 - Raspberry Pi connecting to, servo controller used / [Connecting](#)

Raspberry Pi to the mobile platform using a servo controller

- controlling, Linux program created / [Creating a program in Linux to control the mobile platform](#)
- motor controller
 - about / [Gathering the required hardware](#)
 - using, to control platform speed / [Using a motor controller to control the speed of your platform](#)
- mouse
 - connecting / [Hooking up a keyboard, mouse, and display](#)
- move / [Basic Linux commands on Raspberry Pi](#)
- mv filename1 filename2 command / [Basic Linux commands on Raspberry Pi](#)

N

- Nmap
 - running / [Accessing the board remotely](#)

O

- object-oriented code
 - about / [The object-oriented code](#)
- OpenCV
 - about / [Connecting the USB camera to Raspberry Pi and viewing the images, Downloading and installing OpenCV – a fully featured vision library, Creating a general control structure](#)
 - installing / [Downloading and installing OpenCV – a fully featured vision library](#)
- operating system

- installing / [Installing the operating system](#)

P

- Pan and Tilt assembly / [Gathering the hardware](#)
- peripherals
 - adding / [Hooking up a keyboard, mouse, and display](#)
- Phidget Python Module option / [Connecting Raspberry Pi to an infrared sensor](#)
- Phidgets Control Panel application / [Connecting Raspberry Pi to an infrared sensor](#)
- platform speed
 - controlling, motor controller used / [Using a motor controller to control the speed of your platform](#)
- PocketSphinx
 - about / [Using PocketSphinx to accept your voice commands](#)
 - using, to accept voice commands / [Using PocketSphinx to accept your voice commands](#)
 - downloading / [Using PocketSphinx to accept your voice commands](#)
- PocketSphinx directory / [Using PocketSphinx to accept your voice commands](#)
- Pololu / [Gathering the required hardware](#)
 - about / [Using a motor controller to control the speed of your platform](#)
 - URL / [Getting started](#)
- Polulu software
 - URL, for downloading / [Configuring the software](#)
- print command / [The while statement](#), [Working with functions](#), [The](#)

object-oriented code

- PuTTY
 - downloading / [Accessing the board remotely](#)
- PWM
 - about / [Gathering the hardware](#)
- Python
 - about / [Libraries/modules in Python](#)
- Python programs
 - running, on Raspberry Pi / [Creating and running Python programs on Raspberry Pi](#)
 - creating, on Raspberry Pi / [Creating and running Python programs on Raspberry Pi](#)

Q

- Quadcopters
 - about / [Using Raspberry Pi to fly robots](#)
- quit / [Creating, editing, and saving files on Raspberry Pi](#)

R

- Raspberry Pi
 - mobile platform, controlling programmatically / [Controlling your mobile platform programmatically using Raspberry Pi](#)
- Raspberry Pi
 - about / [Getting started](#)
 - operating system, installing / [Installing the operating system](#)

- Linux commands / [Basic Linux commands on Raspberry Pi](#)
- creating / [Creating, editing, and saving files on Raspberry Pi](#)
- editing / [Creating, editing, and saving files on Raspberry Pi](#)
- saving / [Creating, editing, and saving files on Raspberry Pi](#)
- Oython programs, running / [Creating and running Python programs on Raspberry Pi](#)
- Oython programs, creating / [Creating and running Python programs on Raspberry Pi](#)
- USB camera, connecting / [Connecting the USB camera to Raspberry Pi and viewing the images](#)
- connecting to mobile platform, servo controller used / [Connecting Raspberry Pi to the mobile platform using a servo controller](#)
- connecting, to infrared sensor / [Connecting Raspberry Pi to an infrared sensor](#)
- connecting, to USB sonar sensor / [Connecting Raspberry Pi to a USB sonar sensor](#), [Connecting the hardware](#)
- connecting, to wireless USB keyboard / [Connecting Raspberry Pi to a wireless USB keyboard](#)
- working with, wireless LAN used / [Working remotely with your Raspberry Pi through a wireless LAN](#)
- working with, ZigBee used / [Working remotely with your Raspberry Pi through ZigBee](#)
- connecting, to GPS device / [Connecting Raspberry Pi to a GPS device](#)
- using, to sail / [Using Raspberry Pi to sail](#), [Getting started](#)
- using, to fly robots / [Using Raspberry Pi to fly robots](#)
- using, to make robot swim underwater / [Using Raspberry Pi to make the robot swim underwater](#)
- Raspbian / [Installing the operating system](#)
 - about / [Basic Linux commands on Raspberry Pi](#)
- RCA jack input / [Hooking up a keyboard, mouse, and display](#)
- Real VNC / [Accessing the board remotely](#)
- Remote Operated Vehicle (ROV) robot / [Using Raspberry Pi to make the robot swim underwater](#)
- remove / [Basic Linux commands on Raspberry Pi](#)

- rm filename command / [Basic Linux commands on Raspberry Pi](#)
- Robot Operating System
 - about / [Using the structure of the Robot Operating System to enable complex functionalities](#)
 - URL / [Using the structure of the Robot Operating System to enable complex functionalities](#)
- robots
 - controlling, Raspberry Pi used / [Using Raspberry Pi to fly robots](#)
 - swim underwater, Raspberry Pi used / [Using Raspberry Pi to make the robot swim underwater](#)
- rqt_graph / [Using the structure of the Robot Operating System to enable complex functionalities](#)

S

- S-Video / [Hooking up a keyboard, mouse, and display](#)
- sail boat
 - controlling, Raspberry Pi used / [Using Raspberry Pi to sail, Getting started](#)
- Secure Shell Hypterminal (SSH) connection / [Accessing the board remotely](#)
- servo
 - using, to move single sensor / [Using a servo to move a single sensor](#)
- servo controller
 - used, for connecting Raspberry Pi to mobile platform / [Connecting Raspberry Pi to the mobile platform using a servo controller](#)
 - hardware, connecting / [Connecting the hardware](#)
 - software, controlling / [Configuring the software](#)

- setSpeed(ser, motor, direction, speed)\$ function / [Using the keyboard to control your project](#)
- single sensor
 - moving, servo used / [Using a servo to move a single sensor](#)
- sleep function / [Libraries/modules in Python](#)
- sound
 - making / [Hooking up the hardware to make and input sound](#)
 - about / [Using Espeak to allow our projects to respond in a robot voice](#)
- SparkFun
 - URL / [Gathering the required hardware](#)
- sphinxbase module / [Using PocketSphinx to accept your voice commands](#)
- SSH / [Accessing the board remotely](#)
- Stop bits option / [Working remotely with your Raspberry Pi through ZigBee](#)
- sudo apt-get install build-essential command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install ffmpeg command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install libavformat-dev command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install libcv-dev command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install libcv2.3 libcvaux2.3 libhighgui2.3 command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install libcvaux-dev command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install libhighgui-dev command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install opencv-doc command / [Downloading and installing OpenCV – a fully featured vision library](#)
- sudo apt-get install python-opencv command / [Downloading and](#)

installing OpenCV – a fully featured vision library

- sudo apt-get update command
 - about / Downloading and installing OpenCV – a fully featured vision library
- sudo command / Basic Linux commands on Raspberry Pi
- super user / Basic Linux commands on Raspberry Pi
- system function / Creating a general control structure

T

- terminal emulator software
 - URL, for downloading / Connecting the hardware
- TightVNC / Accessing the board remotely
- two-wheeled platforms
 - URL / Gathering the required hardware

U

- undo / Creating, editing, and saving files on Raspberry Pi
- USB-ProxSonar-EZ sensor / Gathering the hardware
- USB camera
 - connecting, to Raspberry Pi / Connecting the USB camera to Raspberry Pi and viewing the images
- USB hub
 - adding / Hooking up a keyboard, mouse, and display
- USB sonar sensor
 - Raspberry Pi, connecting to / Connecting Raspberry Pi to a USB sonar sensor, Connecting the hardware

- USB sound card
 - input volume, controlling / [Hooking up the hardware to make and input sound](#)
 - output volume, controlling / [Hooking up the hardware to make and input sound](#)
- UscCmd command-line application / [Configuring the software](#)

V

- var = getch() statement / [Using the keyboard to control your project](#)
- var = serInput.read(1) statement / [Working remotely with your Raspberry Pi through ZigBee](#)
- VGA / [Hooking up a keyboard, mouse, and display](#)
- vision library
 - using, to detect colored objects / [Using the vision library to detect colored objects](#)
- vncserver / [Accessing the board remotely](#)
- VNC Viewer application / [Accessing the board remotely](#)
- voice commands
 - issuing / [Making your mobile platform truly mobile by issuing voice commands](#)

W

- WaitKey object / [Accessing the GPS programmatically](#)
- Warning statement / [Using PocketSphinx to accept your voice commands](#)
- while statement
 - about / [The while statement](#)
- WinScp / [Hooking up the hardware to make and input sound](#)
- WinSCP / [Accessing the board remotely](#)
- wireless LAN

- used, for working remotely with Raspberry Pi / [Working remotely with your Raspberry Pi through a wireless LAN](#)
- wireless USB keyboard
 - Raspberry Pi, connecting to / [Connecting Raspberry Pi to a wireless USB keyboard](#)

Z

- ZigBee
 - about / [Gathering the hardware](#)
 - used, for working remotely with Raspberry Pi / [Working remotely with your Raspberry Pi through ZigBee](#)