Dr. Richard Grimmett

Foreword by:
Jason Kridner
Co-founder of BeagleBoard.org

BeagleBone Robotic Projects

Second Edition

Create complex and exciting robotic projects with the BeagleBone Blue

Packt

BeagleBone Robotic Projects

Second Edition

Create complex and exciting robotic projects with the BeagleBone Blue

Dr. Richard Grimmett

BIRMINGHAM - MUMBAI

BeagleBone Robotic Projects

Second Edition

First published: December 2013

Second edition: June 2017

Production reference: 1090617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-313-6

www.packtpub.com

Credits

Author
Dr. Richard Grimmett

Reviewers
Shantanu Bhadoria
Marcelo Boá
Jason Kridner

Commissioning Editor
Vijin Boricha

Acquisition Editor
Heramb Bhavsar

Content Development Editor
Sharon Raj

Technical Editor
Prashant Chaudhari

Copy Editor
Stuti Srivastava

Project Coordinator
Virginia Dias

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Kirk D'Penha

Production Coordinator
Aparna Bhagat

Foreword

10 years ago now, Gerald and I envisioned the original BeagleBoard, which would democratize access to computers that were small enough, low-power enough, capable enough, open enough, understandable enough, and affordable enough to encourage hundreds of thousands of new developers to build electronics systems that they controlled, rather than simply creating an application that merely runs on someone else's platform. From open hardware DNA-analysis machines to advanced transportation systems and prototypes of Mars rovers, over a million BeagleBoards and BeagleBones have now gone into developers' projects, and often, units built around the open source hardware designs have gone into production systems, enabling entrepreneurs and visionaries to realize their dreams. Enabling the `http://beagleboard.org/` community has remained my passion, and I support it everyday. Traveling to trade shows, chatting on the live IRC channel, maintaining the website, answering technical queries on the mailing list, updating documentation, supporting innovations with our suppliers and distributors, and ultimately creating new designs all pay off when I see individuals succeeding in learning about and creating with programming and electronics.

Robotics has always been a popular application of the BeagleBoard, but the new BeagleBone Blue provides a unique set of features, taking the exploration and implementation of robotics to another level of simplicity, completeness, and community activity. Robotics provides a compelling opportunity to build and go beyond a basic understanding of mechanics, electronics, programming, and networking technologies that impact nearly all of our daily lives. Moving beyond the initial experience, the open nature enables you to collaborate, build understanding from history, and eliminate any barriers to where you can take your learning, all the way to making your own product--and that makes you part of a compelling community.

Richard has already written the book on BeagleBone robotics with his titles *BeagleBone Robotic Projects* and *Mastering BeagleBone Robotics*. These are excellent books, providing practical introductions to rewarding creations. With the introduction of the BeagleBone Blue, it was natural for me to reach out to Richard with early access to the board. In this edition, Richard fast-tracks you into robotics and the BeagleBoard.org community with a practical set of hands-on experiences that get you started, and he further gives you the tools to help bring in others to this amazing world. I hope you'll join us as we make this world a better place by mastering robotics and sharing the joy of creation and creativity it offers.

Jason Kridner

Co-founder of BeagleBoard.org

About the Author

Dr. Richard Grimmett has been fascinated by computers and electronics from his very first programming project, which used Fortran on punch cards. He has bachelor's and master's degrees in electrical engineering and a PhD in leadership studies. He also has 26 years of experience in the radar and telecommunications industries, and 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 robotics projects.

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

About the Reviewers

Shantanu Bhadoria is an avid traveler and an author of several popular open source projects in Perl, Python, Golang, and NodeJS, including many IoT projects. When in Singapore, he works on paging and building control systems for skyscrapers and large campuses in Singapore, Hong Kong, and Macau. He has authored and contributed to public projects dealing with control over gyroscopes, accelerometers, magnetometers, altimeters, PWM generators, and other sensors and controllers, as well as sensor fusion algorithms such as Kalman filters.

His work in IoT and other fields can be accessed from his GitHub account at `https://github.com/shantanubhadoria`.

He is also the author of Device::SMBus, a popular Perl library used to control devices over the I2C bus.

Marcelo Boá is an electronics technician who has a bachelor's degree in information systems. He has worked for 10 years in the field of electronic maintenance. He has also worked in Java development, Oracle PL/SQL, PHP, ZK framework, shell scripts, HTML, JavaScript, Ajax, NodeJS, AngularJS, Linux, Arduino, and BeagleBone.

He started as a PL/SQL trainee at the Federal Technological University of Paraná, Brazil. He worked for several companies on many different kinds of electronic circuits and hardware, gaining technical experience at Sony, Aiwa, and Gradiente. 10 years later, he returned to Java development with the ZK framework, developing software for call centers in Curitiba's Software Park. He worked as a systems analyst in the warehouse management systems and industrial automation department at SSI SCHAEFER and provided support to large companies in the distribution sector, such as Boticário, Posigraf, Sadia BRF, GTFoods, Cotriguaçú, Unifrango, and Cocari.

He also reviewed Mastering Beaglebone Robotics.

I would like to thank my wife, Marcela Contador, for giving me all her support.

Jason Kridner has over 25 years of experience in developing embedded electronics, from digital circuits and digital signal processing to high-level systems integration around RTOS environments and Linux. As an applications engineer at Texas Instruments, Jason has taken joy in helping others solve both simple and complex embedded systems problems. Seeking to share his passion with others, he co-founded `http://beagleboard.org/` in 2008, creating platforms that hundreds of thousands of users have now enjoyed using, advancing their programming and electronics skills. He has co-authored two books on BeagleBone, *Bad to the Bone* and *BeagleBone Cookbook*.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1788293134`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface 1

Chapter 1: Getting Started with the BeagleBone Blue 8

- **Powering up and connecting to the BeagleBone Blue** 9
- **Accessing the operating system** 18
- **Accessing the BeagleBone Blue remotely via WLAN** 20
- **Summary** 36

Chapter 2: Programming the BeagleBone Blue 37

- **Basic Linux commands showing how to navigate around the filesystem on the BeagleBone Blue** 38
- **Creating, editing, and saving files on the BeagleBone Blue** 43
- **Creating and running Python programs on the BeagleBone Blue** 45
- **Some basic programming constructs on the BeagleBone Blue** 49
- **A brief introduction to the C programming language** 59
- **Summary** 63

Chapter 3: Making the Unit Mobile - Controlling Wheeled Movement 64

- **Getting started** 65
- **Controlling your mobile platform programmatically using the BeagleBone Blue** 69
  - Connecting the DC motors to the BeagleBone Blue 69
  - Controlling the DC motors programmatically 71
- **Accessing the compass on the BeagleBone Blue** 77
- **Summary** 83

Chapter 4: Avoiding Obstacles Using Sensors 84

- **Different types of sensors used** 84
  - The sonar sensor 85
  - The infrared sensor 86
  - The LiDAR sensor 86
- **Connecting a sonar sensor to an Arduino** 87
  - Accessing the sonar sensor from the Arduino IDE 88
- **Creating an array of sensors** 93
- **Dynamic path planning with your mobile platform** 96
  - Basic path planning 97
  - Avoiding obstacles 100

Summary 103

Chapter 5: Allowing Our BeagleBone Blue to See 104

Connecting your USB camera to your BeagleBone Blue and viewing the images 105
Downloading and installing OpenCV, a fully featured vision library 108
Using the vision library to detect colored objects 111
Summary 116

Chapter 6: Providing Speech Input and Output 117

Hardware prerequisites 118
Connecting the hardware and making an input sound 120
Using eSpeak to allow our projects to respond in a robot voice 126
Using Pocketsphinx to interpret your voice commands 127
Providing the capability to interpret your commands and have your robot initiate action 135
Summary 139

Chapter 7: Making the Unit Very Mobile - Controlling Legged Movement 140

Connecting the BeagleBone Blue to your mobile platform 143
Creating a Linux program to control your mobile platform 145
Making your mobile platform truly mobile by issuing voice commands 150
Summary 151

Chapter 8: Using a GPS Receiver to Locate Your Robot 152

Connecting the BeagleBone Blue to a GPS device 153
Accessing the GPS programmatically and determining how to move to a location 166
Summary 171

Chapter 9: By Land, By Sea, By Air 172

The BeagleBone Blue and robots that can sail 173
Connecting an analog airspeed sensor 175
Getting sensor data from the airspeed sensor 178
Long range control of the BeagleBone Blue 178
BeagleBone Blue and robots that can fly 180
The BeagleBone Blue in robots that can go under the water 186
Summary 190

Chapter 10: System Dynamics 191

Controlling your robot via a game pad controller 191
Controlling your robot via a web interface 196

Summary 205
Index 206

Preface

The world we live in today is bursting with new possibilities, all made possible by new technology. Cell phones and personal computers, once the cutting edge of technology, are now a standard part of our lives. Self-driving cars, robotic vacuum cleaners, and software that can predict our shopping patterns are moving from the world of science fiction to the world of our everyday lives.

Much of this new technology is fueled by small and inexpensive but powerful processors that are not only easy to program, but are surrounded by a universe of inexpensive hardware that expands their capabilities to areas that only a few years ago weren't even in the realm of imagination. This book covers one flavor of this technology, the BeagleBone Blue. This processor embodies the next generation of do-it-yourself processors: it not only has the processing capability, but also incorporates much of the necessary surrounding support hardware in a single board.

This book will take you through a number of different projects that will show you how to take full advantage of the BeagleBone Blue. These projects include robots that roll, walk, fly, and sail. In each case, you'll learn how to use the full power of the BeagleBone Blue to create projects that would have required thousands of dollars of hardware just a few years ago.

So grab your BeagleBone Blue, and let's go!

What this book covers

`Chapter 1`, *Getting Started with the BeagleBone Blue,* is designed to help the novice be successful in their first few moments with the unit. The chapter begins with a discussion of how to connect power and ends with a full system, configured and ready to begin connecting any of the amazing devices and software capabilities to fulfill almost any project dream.

`Chapter 2`, *Programming the BeagleBone Blue,* introduces, or reviews for those who are already familiar, basic Linux, editing, and programming techniques that will be useful through the rest of the book. We'll cover how to interact from the command line, how to create and edit a file using an editor, and basic Python and C programming.

`Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement,* is based on how one of the first things you might want to do is create a robot that can move around and explore its environment. Perhaps the easiest way to do this is by adding a wheeled or tracked platform. This chapter details how to control a DC motor so that the unit can drive wheels or tracks.

`Chapter 4`, *Avoiding Obstacles Using Sensors,* explores the different types of sensors that can help you complete your projects. These sensors can help you know when you are approaching an obstacle, which direction you are moving in, or how to get from here to there.

`Chapter 5`, *Allowing Our BeagleBone Blue to See,* shows how with speech, computer vision has moved forward in amazing ways with the introduction of the webcam and the integrated camera for cell phones and laptops. This chapter provides the details of how to connect a webcam, both the hardware and the software, so that we can use it to input visual data into our system.

`Chapter 6`, *Providing Speech Input and Output*, explains how a few years ago, the concept of a computer that can talk and listen was science fiction, but today, it is becoming a standard part of new cell phones. This chapter introduces how the BeagleBone Blue system can both listen to speech and also respond in kind. This is not as easy as it sounds (pun intended), and we'll expose some basic capabilities while also understanding some of the key limitations.

`Chapter 7`, *Making the Unit Very Mobile - Controlling Legged Movement*, discusses how one of the impressive capabilities that really sets a robotic project apart is the ability to control arms and legs. This is done using servos, whose position can be controlled using our system. We'll also introduce the capability of external dedicated servo controllers that can make this job much easier.

`Chapter 8`, *Using a GPS Receiver to Locate Your Robot*, explains how knowing where we are and whether to communicate it to others or to find a path to a different location can add significant possibilities to our project. GPS has become ubiquitous in our world, and its use is now taken for granted. In this chapter, we'll show how to enable it in your own project.

`Chapter 9`, *By Land, By Sea, By Air*, goes through how now that we have a powerful toolkit, we can expand our horizons to even more possibilities.

`Chapter 10`, *System Dynamics,* discusses how we've added lots of amazing capabilities to our project. At this point, we might want to integrate several of these together in order to build complex machines. This chapter covers this process in more detail, including offering some help in the form of open source software that can make this even easier.

What you need for this book

The hardware required is introduced at the start of each chapter.

Software list:

Chapter 1	
Xfce	`sudo apt-get install xfce4`
WinScp	`https://winscp.net/eng/index.php`
Putty	`http://www.putty.org/`
VNC server	`sudo apt-get install tightvncserver`
Real VNC	`https://www.realvnc.com/`
Chapter 2	
Emacs	`sudo apt-get install emacs`
build-essential	`sudo apt-get install build-essential`
Chapter 5	
guvcview	`sudo apt-get install guvcview`
libavformat	`sudo apt-get install libavformat`
ffmpeg	`sudo apt-get install ffmpeg`
libcv2.3	`sudo apt-get install libcv2.3`
libcvaux2.3	`sudo apt-get install libcvaux2.3`
libhighgui2.3	`sudo apt-get install libhighgui2.3`
python-opencv	`sudo apt-get install python-opencv`
python-opencv-doc	`sudo apt-get install python-opencv-doc`
libcv-dev	`sudo apt-get install libcv-dev`
libcvaux-dev	`sudo apt-get install libcvaux-dev`
libhighgui-dev	`sudo apt-get install libhighgui-dev`
python-numpy	`sudo apt-get install python-numpy`
Chapter 8	

gpsd	`sudo apt-get install gpsd`
gpsd-clients	`sudo apt-get install gpsd-clients`
ND-100S Application	On CD with HW or `http://www.usglobalsat.com/store/download/590/nd-100_gps_test_setup.zip`

Who this book is for

This book is for anyone who has been curious about using new, low-cost hardware to create robotics projects that have previously been the domain of research labs of major universities or defense departments. Some programming background is useful, but if you know how to use a personal computer, you can, with the aid of the step-by-step instructions in this book, construct complex robotics projects.

Conventions

In this book, you will find a number of text styles 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 is set as follows:

```
void loop() {
delay(500);
unsigned int uS1 = sonar1.ping();
unsigned int uS2 = sonar2.ping();
unsigned int uS3 = sonar3.ping();
Serial.print(uS1 / US_ROUNDTRIP_CM);
Serial.print(“,”);
Serial.print(uS2 / US_ROUNDTRIP_CM);
Serial.print(“,”);
Serial.println(uS3 / US_ROUNDTRIP_CM);
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
void loop() {
delay(500);
unsigned int uS1 = sonar1.ping();
unsigned int uS2 = sonar2.ping();
unsigned int uS3 = sonar3.ping();
Serial.print(uS1 / US_ROUNDTRIP_CM);
Serial.print(“,”);
Serial.print(uS2 / US_ROUNDTRIP_CM);
Serial.print(“,”);
Serial.println(uS3 / US_ROUNDTRIP_CM);
}
```

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

```
sudo rc_test_motors
```

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."

Warnings or important notes appear in a box like this.

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

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

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

Customer support

Now that you are the proud owner of a Packt 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 this book 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.

You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/BeagleBone-Robotic-Projects-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from

`https://www.packtpub.com/sites/default/files/downloads/BeagleBoneRoboticProjectsSecondEdition_ColorImages.pdf`

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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

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

Piracy

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

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

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

Questions

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

1
Getting Started with the BeagleBone Blue

Welcome to the beginning of an exciting adventure! Hopefully, you've ordered and received your BeagleBone Blue, an exciting platform that combines some excellent computing resources with the capability of accessing a wide range of hardware power.

Ordering the hardware (HW) is the exciting part of any project. You have wonderful dreams of all that you might accomplish once this amazing piece of technology is delivered. Unfortunately, the frustration of the first few attempts at accessing the capabilities of the unit can leave many developers, especially those with little experience with this type of dedicated system, so discouraged that the board ends up on the shelf, gathering dust next to the pet rock and cassette tape recorder.

In this chapter, we will cover the following topics:

- Powering up and connecting to the BeagleBone Blue
- Accessing the operating system
- Accessing the board remotely via WLAN
- Downloading the example code and colored images

There is rarely anything as exciting as ordering the latest new technology and anticipating its arrival. You daydream of the projects you'll build, the amazing things you can do, and the accolades you'll receive from family, friends, and colleagues. However, reality rarely meets your fantasies. This chapter will hopefully help you avoid the pitfalls that normally accompany unboxing and configuring a dedicated processor system such as the BeagleBone Blue. You'll step through the process and get answers to all kinds of specific questions so that you can understand what is going on. If you don't get yourself through this chapter, then you'll not be successful at any of the others, so buckle up and get ready for an exciting ride.

The most challenging aspect of accomplishing this for me as your guide is trying to decide to what level I should describe each step. Some of you are beginners, others have some limited experience, while still others will know significantly more than I in some of these areas. I'll try and keep it brief, but also try to be thorough so that at least you'll know what steps to take in order to be successful. I'll also try and point out some of the different ways you can get help if you are encountering problems.

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

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

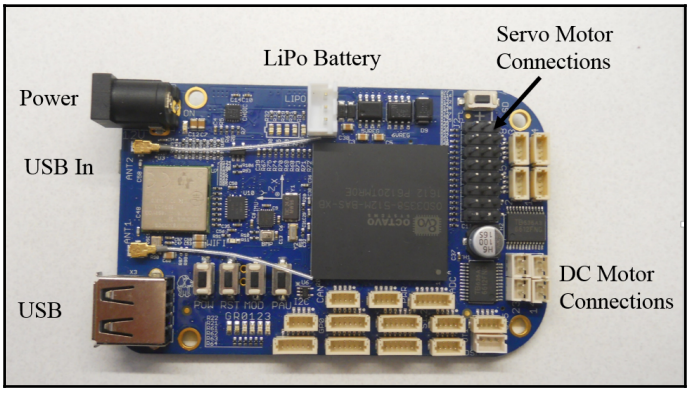
1. A BeagleBoard Blue
2. The USB cable provided with the board
3. An external power supply that can provide 7.4 to 18 volts or a standard 2S LiPo battery for the BeagleBone Blue
4. A wireless LAN connection for the BeagleBone Blue
5. A separate computer that is connected to the Internet

Powering up and connecting to the BeagleBone Blue

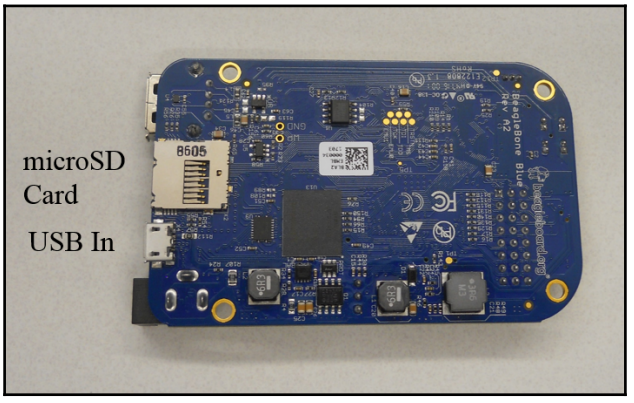
The board has finally arrived. The box should contain the board and a USB cable that can power the board. Let's just look at a few notes on the Beagle Bone Blue first:

This board is an evolved version of the BeagleBone Black, which was an evolved version of the BeagleBone. They are all the same size, but the BeagleBone Blue adds a lot of new capabilities that, in the past, you'd need to add using external hardware. Specifically, the Blue adds the ability to control servos and DC motors directly, has built-in wireless LAN and Bluetooth, and also a built-in IMU and barometer. But more about all of this impressive capability later.

Before plugging anything in, inspect the board for any issues that might 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. Here is the board, with some of the more important connections labelled for your information:

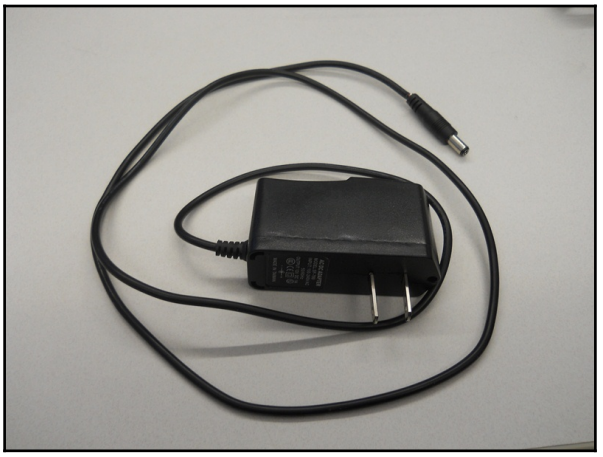


And here is the other side:

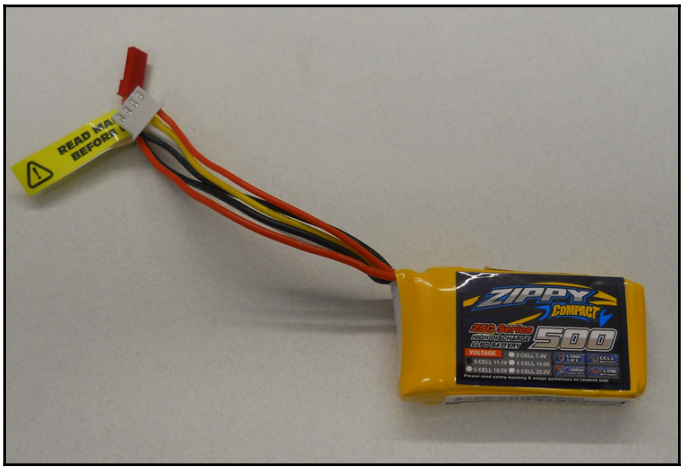


So, let's get started. First, you'll need to power the board.

There are several ways to power the BeagleBone Blue. There is a standard DC power connector on the board. This power connector accepts voltages from 8 to 18 VDC in a 5.5 x 2.1 mm connector. This is the connection you would use if you wanted to plug the unit into a wall adapter--one that can provide from 8 to 18 volts and at least 1 amp, such as this one:

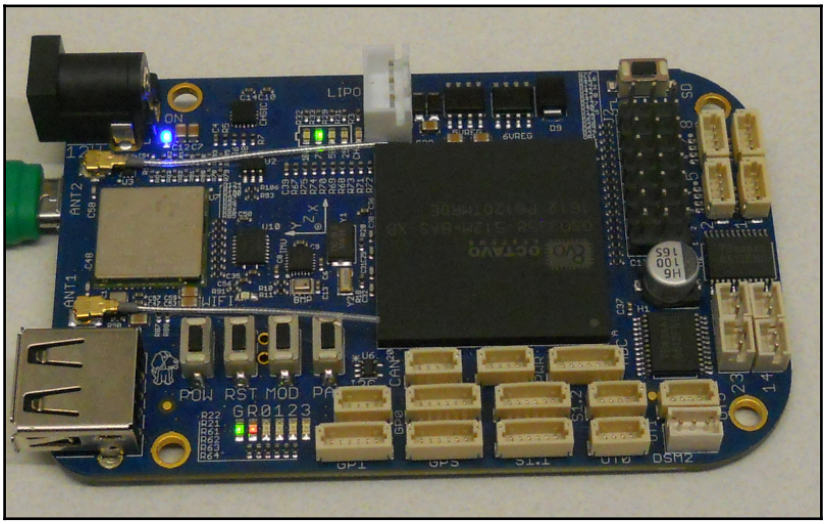


There is also a LiPo battery connector that connects to a two-cell, 7.4V LiPo battery, such as the one pictured here:



You can also connect the BeagleBone Blue to a host computer using a USB cable, with a standard-sized USB connector to connect to the host computer on one end and a micro-USB connector to connect to the BeagleBone Blue. No matter which power source you use, you'll need to make sure the unit can supply the power to the BeagleBone Blue as well as to any additional hardware that is attached to the board. If you have nothing else attached, the board can draw up to 1 amp. You may need several amps more if you are connecting to servo or DC motors, but let's start with a simple connection for your host computer that not only will provide power but will also allow you to make your first contact with the board. So plug the standard USB cable end into the host USB port and the microUSB connector in the USB In connection on the back side of the BeagleBone Blue.

When you plug the board in, the PWR LED should glow blue on the board. There are two sets of LEDs that will give you additional indications of activity. Here is a close-up of the board, just so you're certain which one to look for:

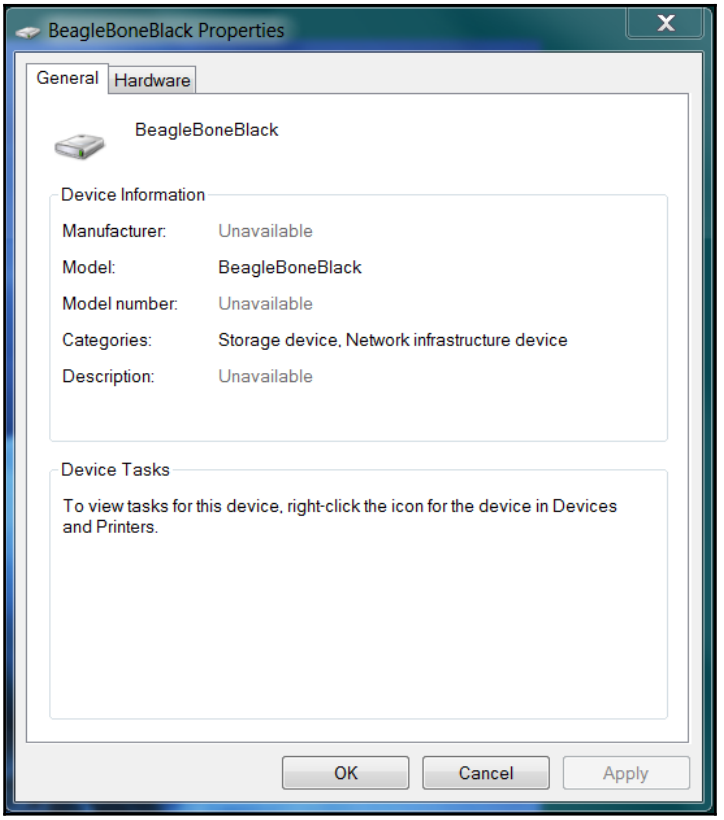


The LEDs labelled GR0123 are the indicators that are useful for indicating the state of the board. They will all flash on and then, eventually, the LED labelled 0 will flash on and off as a heartbeat indicator: this lets you know that your processor is working.

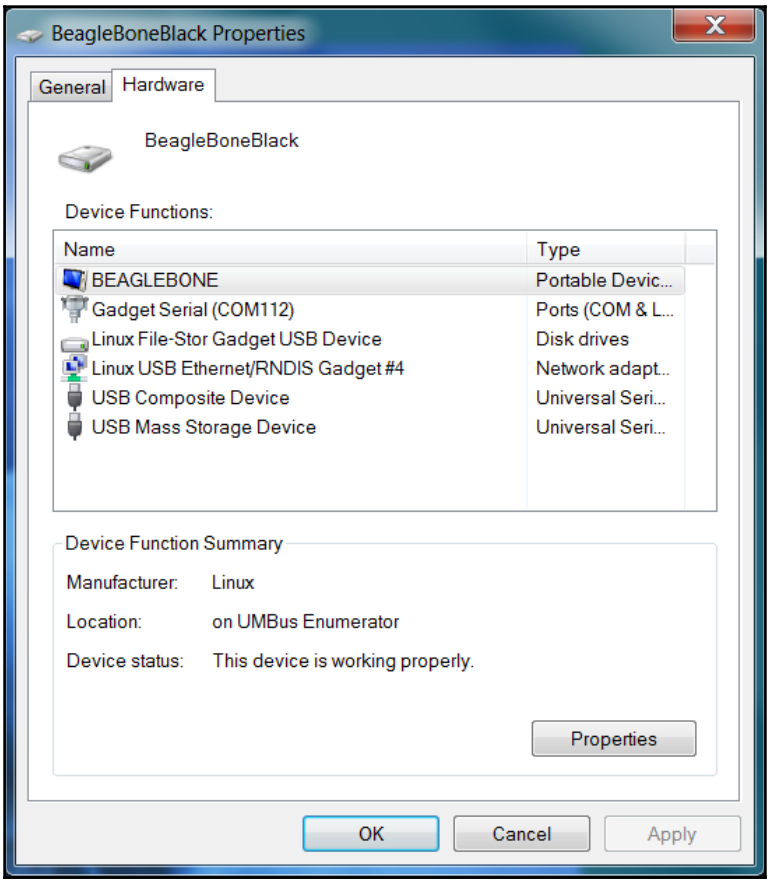
Now we can use some computer SW to make sure our board is operating correctly. When you first plug the board into a Windows PC, you'll see the indicator in the lower right indicating that new HW is being installed. Eventually--and this may take a while--you'll get the indication that your device is ready to use. If you are using Windows 7, you can view the device in your **Devices and Printers** display (select this from the **Start** menu).

You should see the following:

If you see this, and LED 0 is flashing in a heartbeat fashion, you've successfully connected your board. Once you've connected, you can actually talk with your board via the USB connection. To do this, you'll first connect via serial port, but you'll need to know which port. To discover this, click on the **BeagleBoneBlack** connection, and you should see this selection:

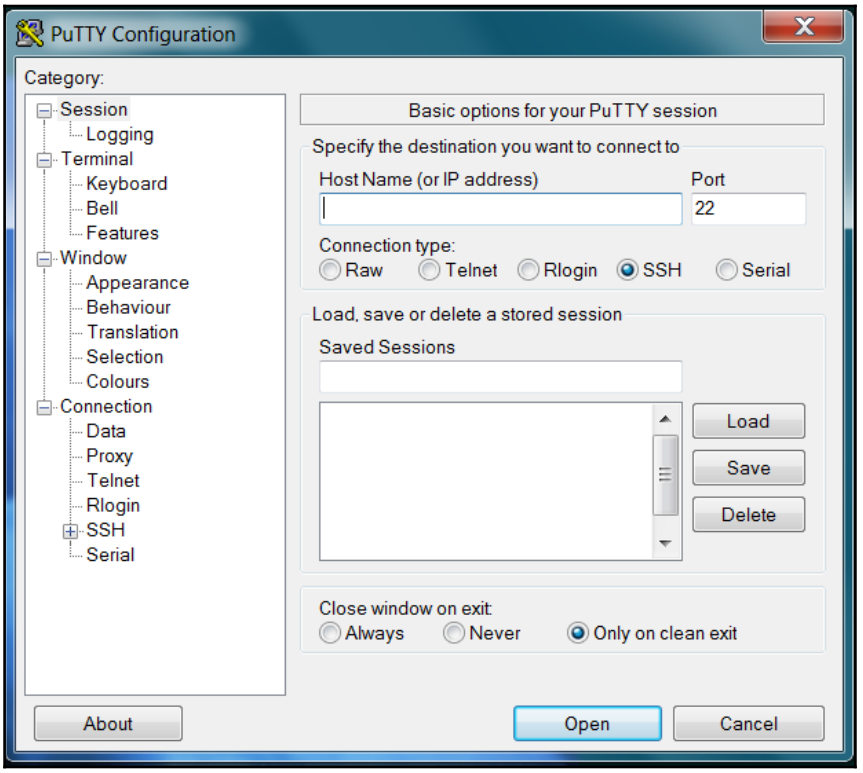


Now, click on the **Hardware** tab, and you should see this:

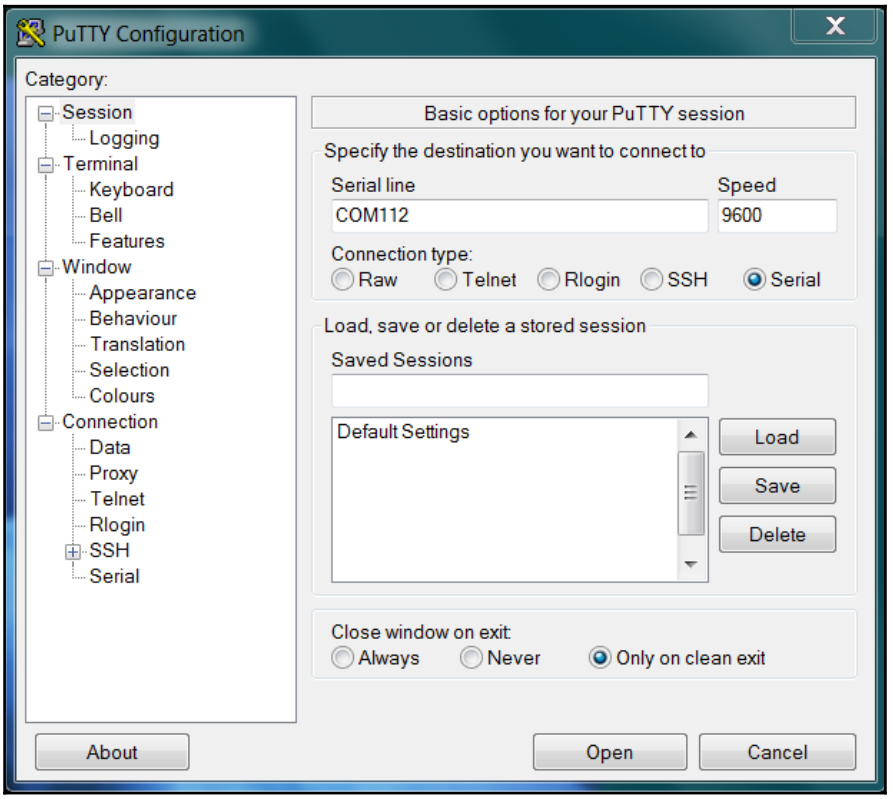


In this case, the unit is connected to serial port COM112. You'll need a terminal emulator program to connect to the board--PuTTY is one that is readily available and free. Go to `http://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`, and download and install the latest version.

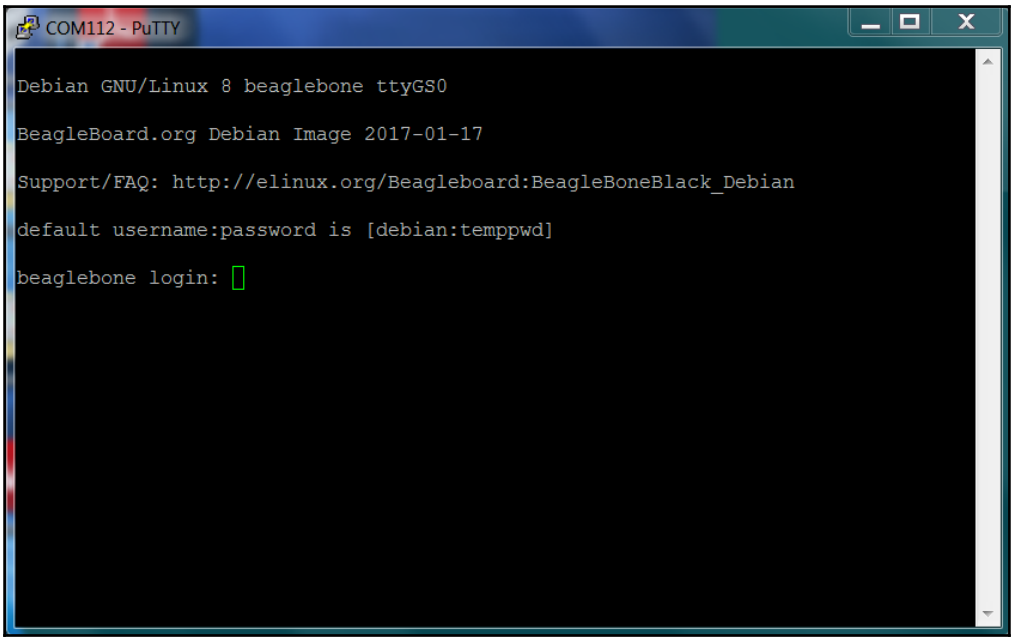
Now open the PuTTY application. You should see this:



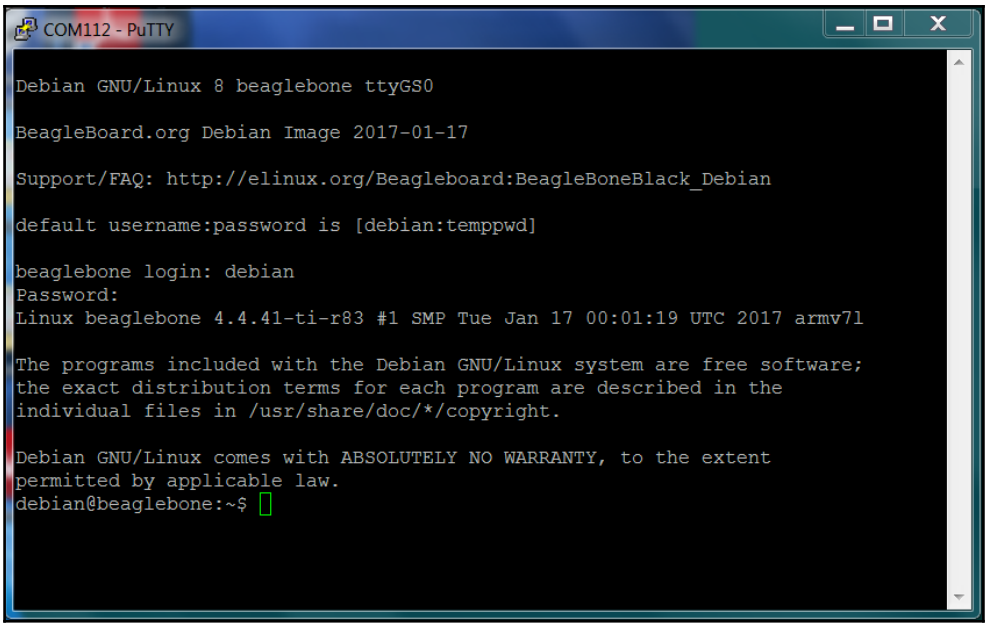
You'll need to establish a serial connection to the BeagleBone Blue. To do this, fill out the following on PuTTY:



In this case, I've also saved these settings under the **Default Settings** label. That way, you won't need to re-enter these each time. Now select **Open**, and you should see the terminal window open to the BeagleBone Blue. When the window opens, place your mouse in the window, select it, and then hit *Enter*; you should see this:



You can log in by typing the default username and password, `debian` and `temppwd`, respectively. Now you should see this:



If you've reached this point, congratulations! You are logged in and ready to interact with your BeagleBone Blue. By the way, you'll notice that the board does not echo the characters as you enter your password. It does not want someone to look over your shoulder and see your password as you type. Despite no characters appearing on the screen, your password is being entered.

If you're having problems, the folks at `beagleboard.org` have a rich set of forums that can help you work through any of the problems you might be having unpacking and powering on the board.

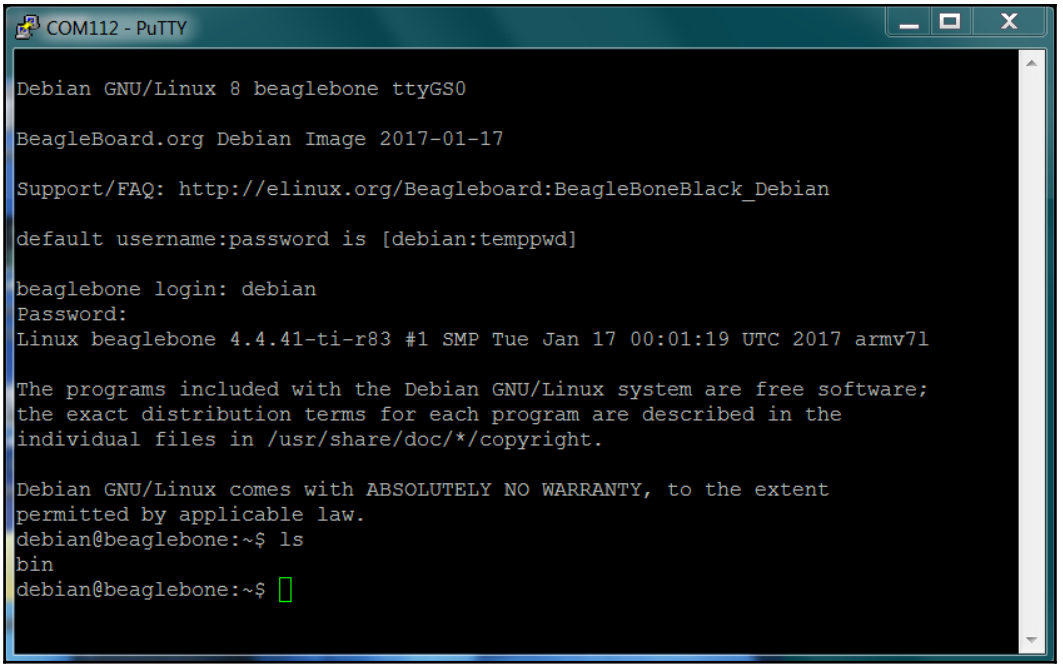
Accessing the operating system

Now that you have your system all up and working, you can access the operating system. The default operating system on the board is Debian, fortunately just the version you're interested in using. Linux, unlike Windows, or Android or iOS, is not tightly controlled by a single company. It is a group effort, mostly open source; it is available for free and 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 is one of these versions. This particular distribution is a popular one among DIYers, and it will allow you to use a number of different freeware software packages. Debian also has excellent support for new HW, and this can be very important for our projects.

The BeagleBone Blue also supports using other operating systems by installing them on a micro SD card and inserting the card into the micro SD slot on the board. To find out more about this capability, refer to the details on `https://beagleboard.org`.

So, you are going to use a version of Linux called Debian on your BeagleBone Blue. Once you are logged in to your BeagleBone Blue, you can issue some simple commands. If you type `ls`, you should see something like this:

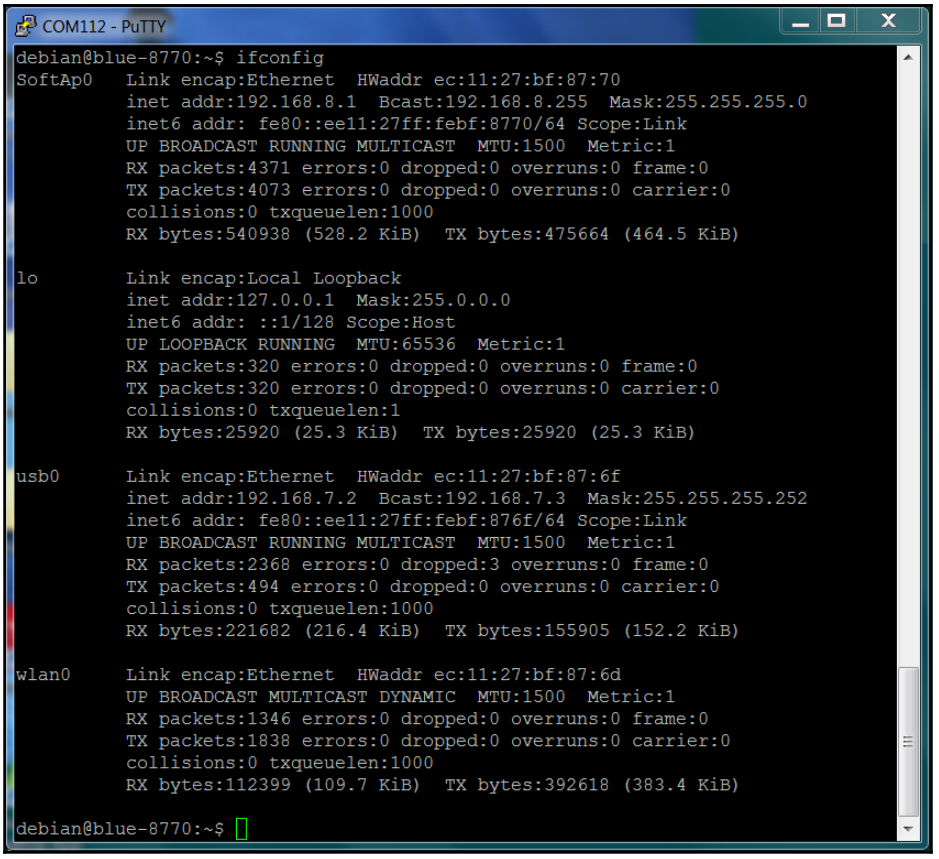


The `ls` command simply lists the files and directories in the current directory. In this case, there is only one: the `bin` directory. In the next chapter, you'll learn more about the commands you'll need to access the Linux capability of the BeagleBone Blue as well as how to program the unit.

Accessing the BeagleBone Blue remotely via WLAN

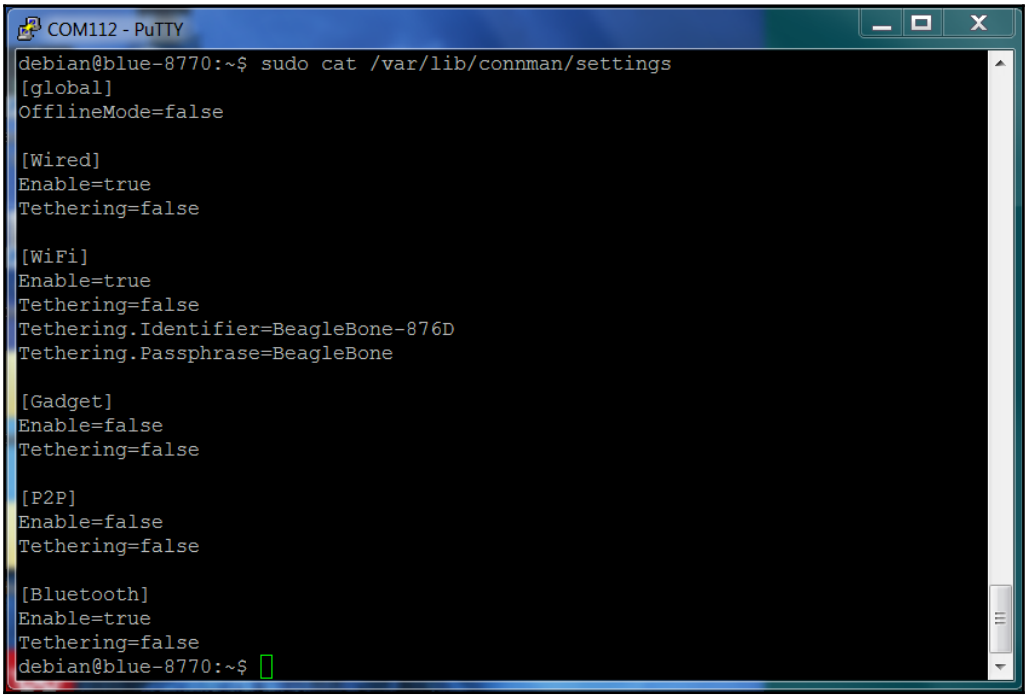
It is important that you can access the BeagleBone Blue through the serial connection. However, you're going to want a connection that is of higher bandwidth. Also, it will be very helpful to connect to the board without having a physical connection. Fortunately, the BeagleBone Blue has wireless LAN capability on board to make this easy. You now have access to the Debian Linux system and can type in commands and see their result in a terminal window. You'll use this capability to understand the WLAN connection on the BeagleBone Blue.

Type in `ifconfig` at the prompt. You should get a display like the following:



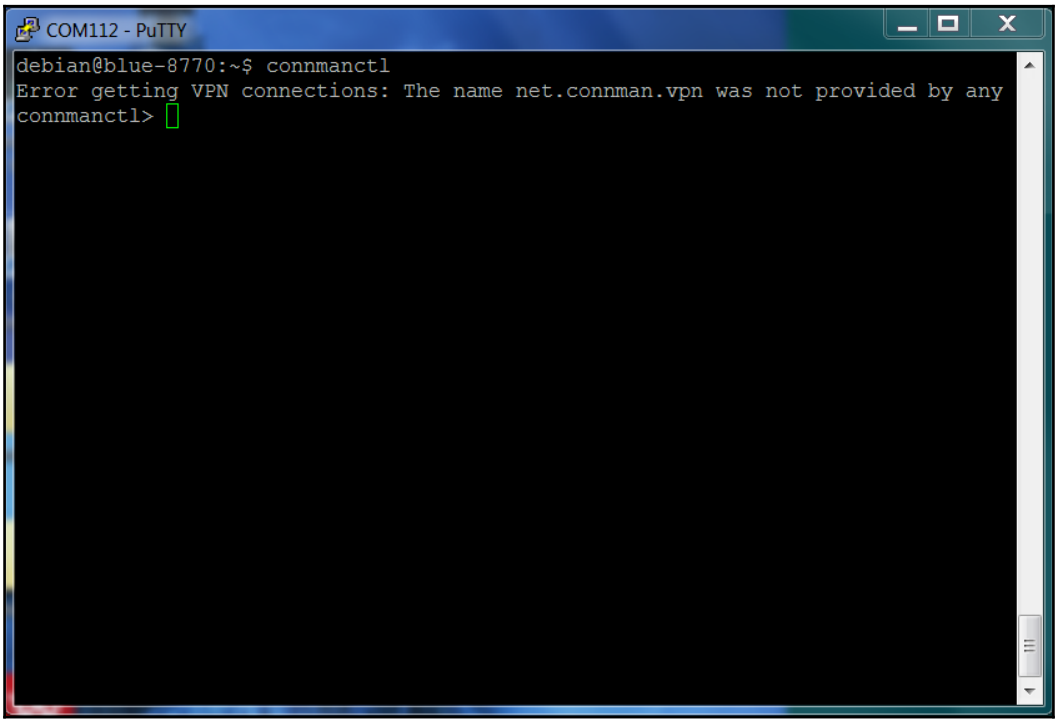
This information gives you an idea about how you can connect to the Internet, and have a valid Internet address. In this case, the `eth0` connection, normally a cabled connection to the board, is not connected to the Internet. That makes sense because the BeagleBone Blue doesn't have a cabled Internet connection. The `lo` connection is a local loopback connection, you'll not need to use it. The `usb0` connection is the connection that is available over the USB cable that you have plugged in from the host computer, and it has the address `192.168.7.2`. You can access this connection from your host computer, but your BeagleBone Blue can't access the Internet from this connection.

The `wlan0` connection is the connection for your WLAN device on board the BeagleBone Blue. This particular connection is not yet connected to the Internet: as you can see, there is no inet addr value on the second line. Your WLAN is, however, broadcasting as a wireless access point. To know the specifics of the wireless access point, you can type `cat/var/lib/connman/settings`, and you should see something like this:



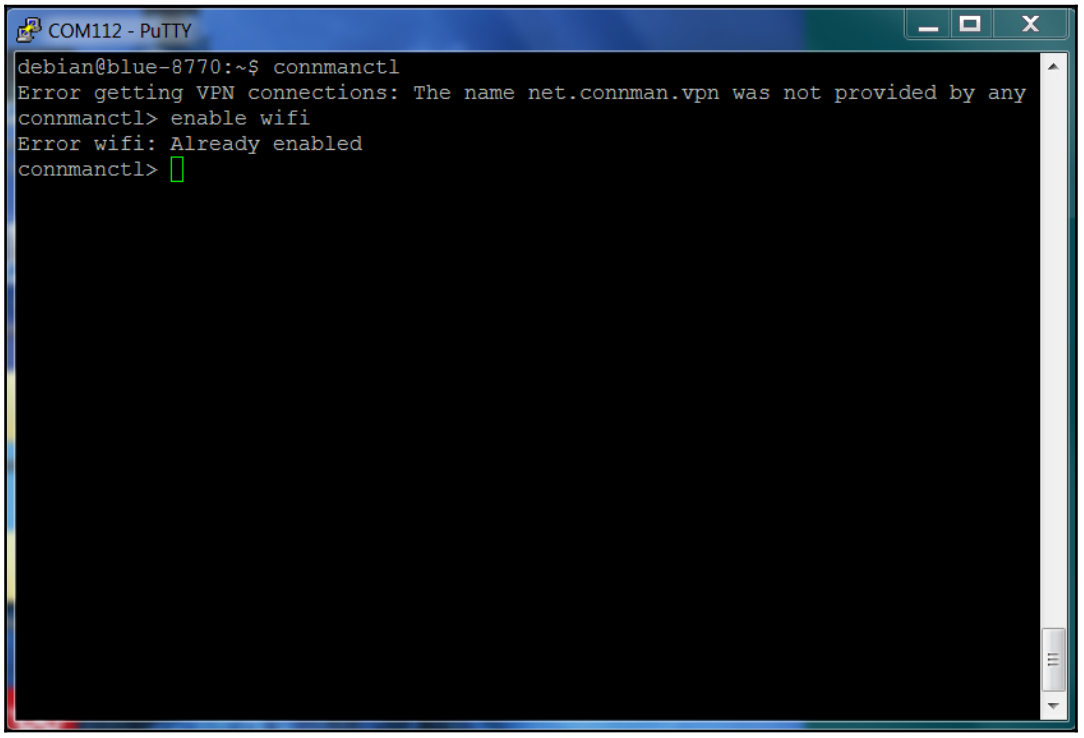
The specific part we are interested in is the `[WiFi]` set, which shows the broadcast SSID as the `Tethering.Identifier` and the password as the `Tethering.Passphrase`.

You will now want to access an available WLAN to update your system, and perhaps download additional software. To set up your connection, you'll use the `connmanctl` command. The **Connmanctl** application is set up to configure your device. Perhaps one of the first things you would like to do is to connect to an available WLAN connection. You can then use this to update your device, and also download other software. To do this, type `connmanctl` at the prompt. You should see this:

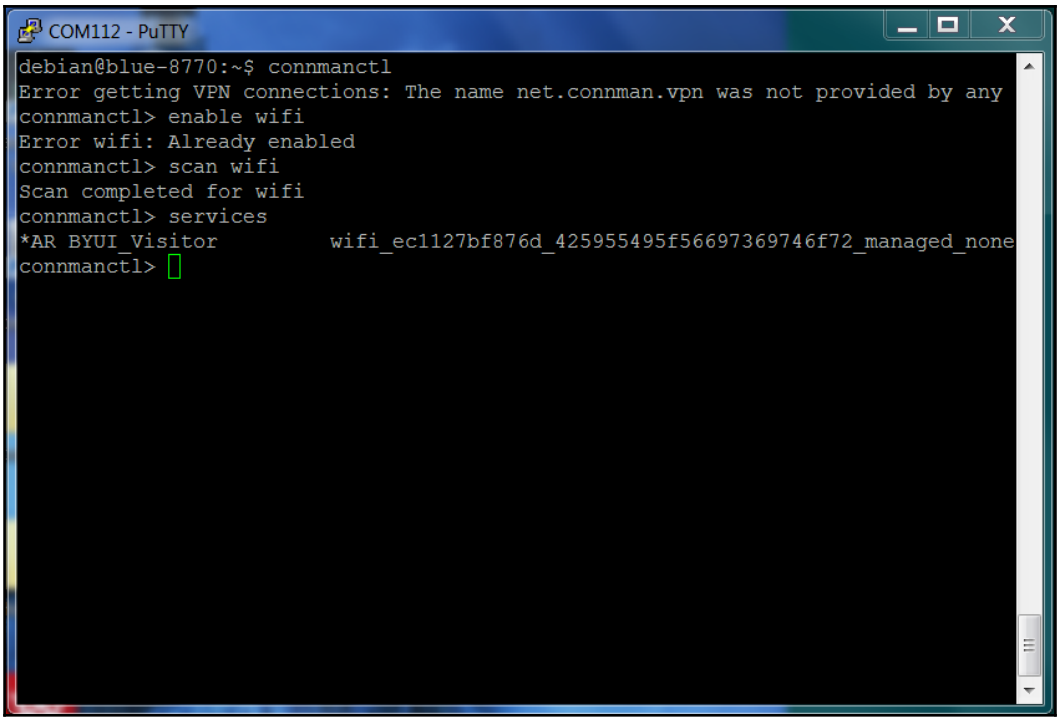


Now, follow these steps:

1. Type `enable wifi`. It should indicate that the Wi-Fi is already enabled:

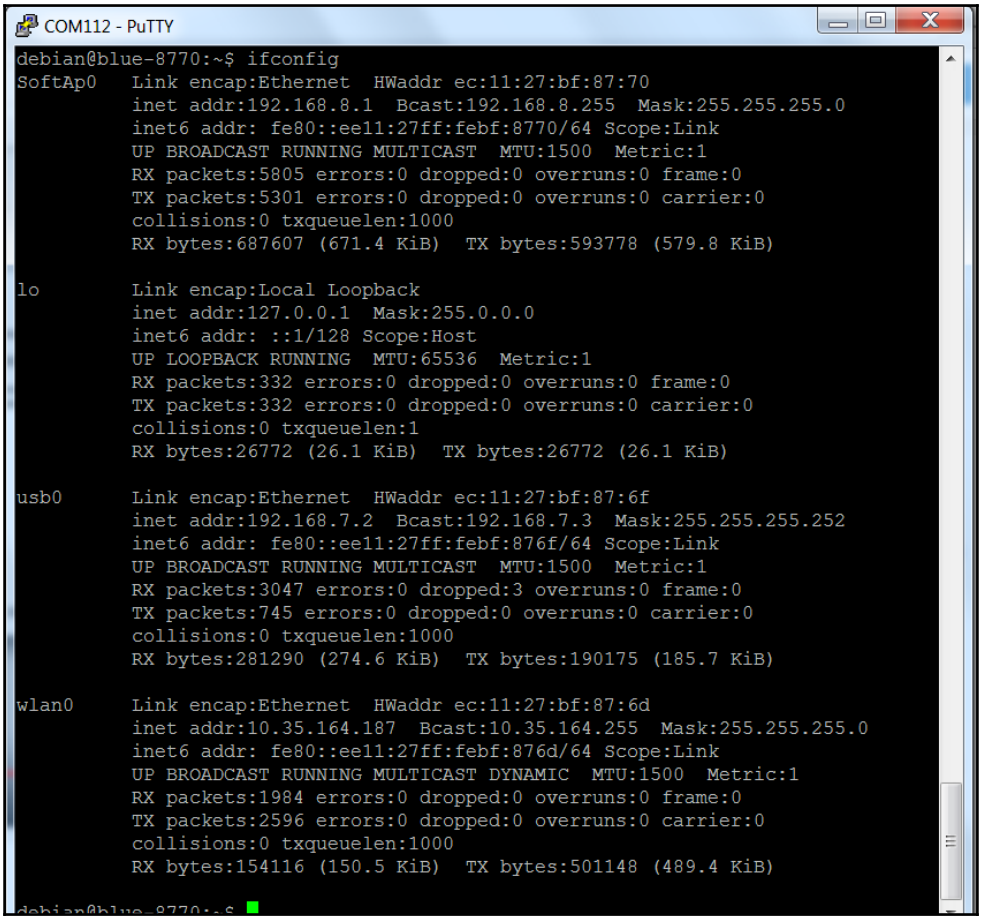


2. Type `scan wifi`. This will scan for possible Wi-Fi connections. It will return when the scan is complete.
3. The `scan wifi` command, however, does not return the scan data directly. You'll need to type `services`, and it will return a list of possible Wi-Fi connections, like this:



4. Now type `agent on`. This will register these different Wi-Fi locations so that you can connect.
5. Then type `connect`, followed by the identifier of the Wi-Fi connection you wish to connect to. In this case, the command will be `connect wifi_ec1127bf876d_425955495f56697369746f72_managed_none`. If the Wi-Fi connection requires a password, it will prompt you for that password.

6. Now you should be connected to the network. Type `quit`, and then type `ifconfig` at the prompt, and you should see something like this:



The `wlan0` connection now has a valid address at `10.35.164.187`. This address has been assigned by the system.

Generally, there are two types of IP addresses that our board can be assigned; one is called static and the other dynamic. In the static case, the same address will always be used. In the dynamic case, the address may change each time the system boots since it asks the system for an address, which it then uses for that session. This is called DHCP. Most systems are configured for the dynamic case; however, if your system isn't changing, you will most likely get the same address each time you power on and log in to the system. However, you may want to ensure you get the same address or to force a specific address if there is no wifi connection. To do this, enter the following command inside the connmatctl application: `confi [wifi_identifier] -ipv4 manual [desired address] [mask] [router address] -nameservers 8.8.8.8`. Here, `[wifi_identifier]` is your wifi identifier, `[desired address]` is the desired address of the device, `[mask]` is the mask associated with the address, which is almost always `255.255.255.0`, and `[router address]` is the address of the router.

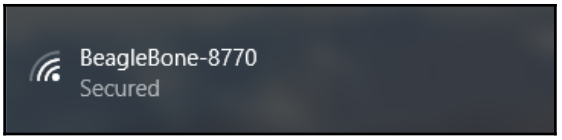
Now that you are connected to wifi, you'll want to update your system. Type in `sudo apt-get update` and then `sudo apt-get upgrade`. The system will prompt you for the sudo password, which is the same as the password you used to log in. Once you enter this, the system will automatically go out and find all the upgrades associated with the system and applications that you have installed. This may take a very long time, depending on how out-of-date your system has become.

Now that your BeagleBone Blue is connected to wifi and you have updated your Debian system, you can connect to the board via the wifi connection. You'll be installing some additional SW on your BeagleBone Blue. You will be using `apt-get` to install SW throughout this book. This is the command that Debian uses to go out and find SW and then install it on your machine. The nice thing about this is that it will also normally go out and find dependencies and download them as well; thus, not only the package you want, but the packages that are needed for that package are installed as well. However, a bit of caution: this is not fool-proof! You may find times when the SW you have installed will not function because of a dependency that the system does not know about.

There are four ways you can access your system from your external PC. The first is the one you have been using, a simple terminal interface using PuTTY through a serial connection. Now that you can connect to your BeagleBone Blue via wifi, you can access it through the PuTTY SSH protocol over the wifi connection.

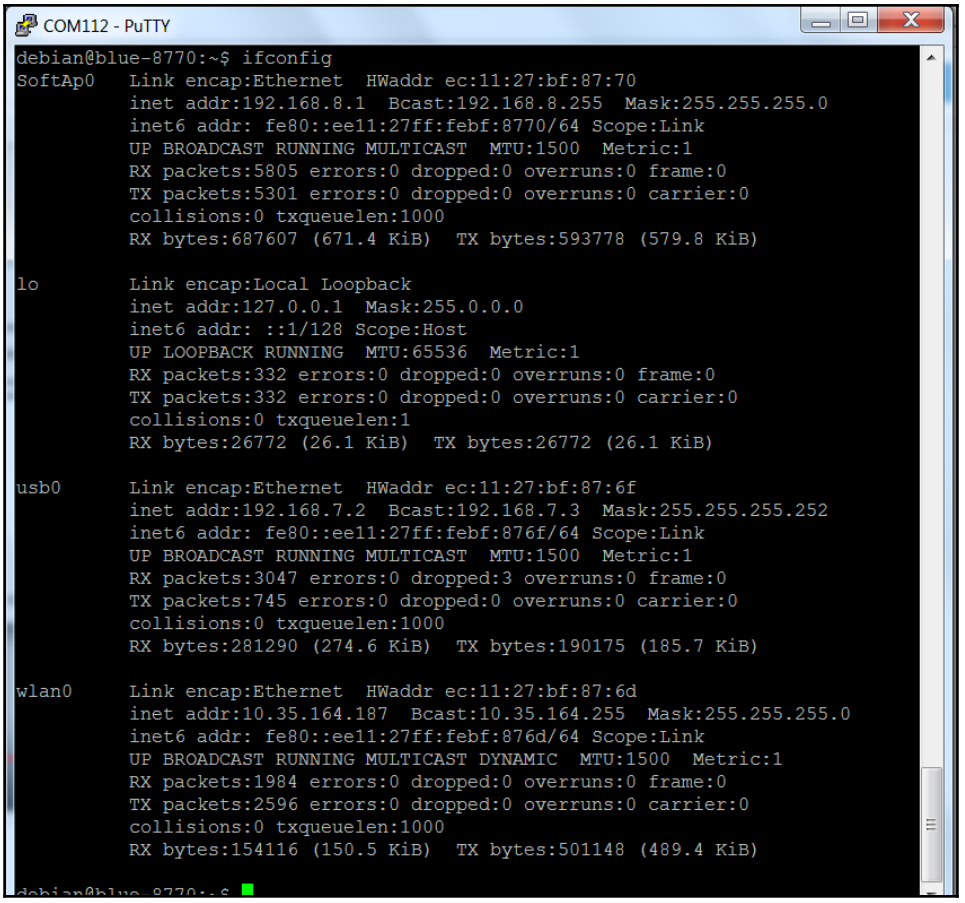
The second way to access your board via wifi is using a program called vncserver, which will allow you to open a graphical "window" on your PC that will show you what the embedded system would be displaying on its display. Finally, there is a way to simply transfer files via a program called WinScp, which is custom-made for this purpose.

The first step to enable any sort of wifi connection is to connect to your board from your host computer. On your host computer, you should be able to scan for available connections. If you are using a Windows host computer, you should see something that looks like this in your network settings indicator:



This is the wifi signal from your BeagleBone Blue. Connect to this Wi-Fi network by selecting it and entering the password which, if you haven't changed it, is BeagleBone. You should see that your host computer has now connected to this network. Now you can use this wireless connection with your device.

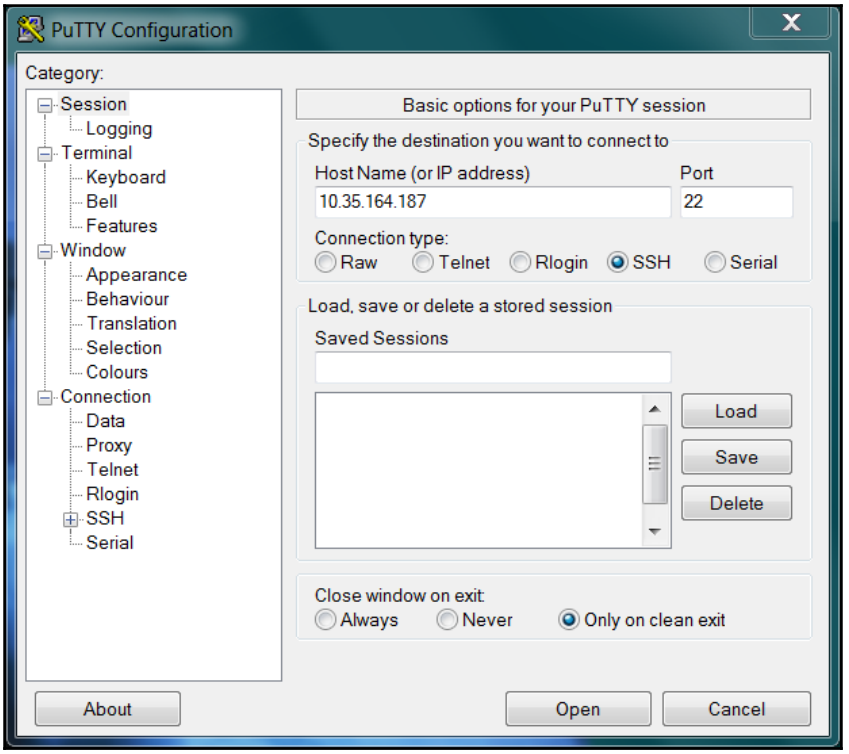
Let's first explore an SSH connection over your Wi-Fi connection. You'll need the IP address of your unit to connect. If you are still connected to the serial interface via PuTTY, you can get this by typing `ifconfig`. You've already done this and seen the inet address. Here is what that looked like:



In this example, under the `wlan0` section, you can see `inet addr:10.35.164.187`. This is the address you'd use if you were connecting to this board.

If you aren't connected to the serial interface, you can still discover the address. If you are using a Windows PC, there are several IP scan utilities that are free and will allow you to find the IP addresses of all the devices attached to your computer. If you are using a Linux machine as a host computer, you can use a Linux application called `nmap`.

Once you know your address, you can connect to your device using the SSH protocol. An SSH terminal is a Secure Shell Hypterminal connection, which simply means you'll be able to access your board and type in commands at the prompt, just like you have done through the serial connection. Fortunately, PuTTY, the terminal application you downloaded earlier, also has this capability. When you start PuTTY, it actually defaults to SSH, so all you'll need to do is type in the inet address of your BeagleBone Blue, like so:



Type the inet address from the previous page in the **Host Name** space, and make sure the **SSH** selection is highlighted. You can also save your configuration so you won't have to type it each time.

When you press **Open**, the system will try and open a terminal window on your BeagleBone Blue through 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 select **OK**, and you should get a terminal with a log in prompt.

Now that you can issue commands to your BeagleBone Blue over wifi, you won't need the serial connection. You can power your BeagleBone either using a power supply or LiPo battery; you will no longer need the serial connection to your host computer. If you'd like to SSH from a Linux or iOS machine, the process is even simpler. Bring up a terminal window, and then type `ssh debian@10.35.164.187 -p 22`. This will then bring you to the log in screen of your BeagleBone Blue, which should look similar to the log in screen you've been using over the serial connection.

SSH is a really useful tool for communicating with your BeagleBone Blue. However, sometimes, you need a graphical look at your system, particularly to debug vision systems. To do this, you'll need to install two sets of software. You'll need to install a graphical user interface package for Debian, and you'll need a way to connect to this graphical user interface remotely.

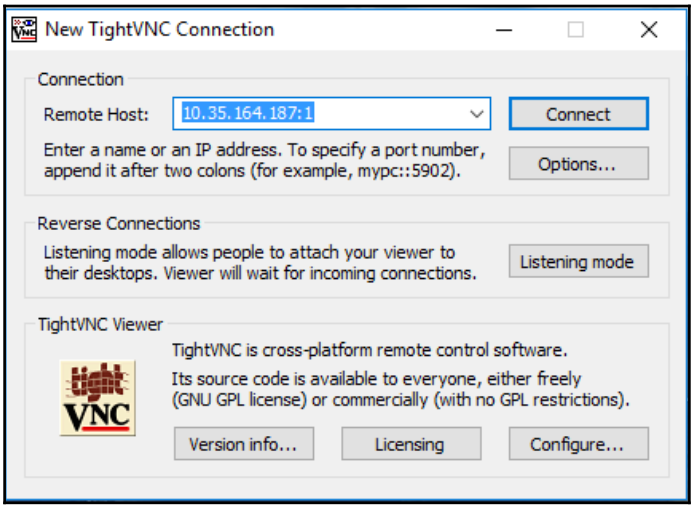
There are several graphical user interface systems available for Debian. One that is simple, lightweight (takes up little memory), and easy to use is **Xfce**. It is also easy to install. Simply type `sudo apt-get install xfce4`. Since the BeagleBone Blue does not have a display connection, now that you have a graphical user interface, you'll need a way to access it remotely.

To get this functionality, you'll install **tightvncserver** on your BeagleBone Blue by typing `sudo apt-get install tightvncserver` in a terminal window. **Tightvncserver** is an application that will allow you to remotely view your complete Windows system. Once you have it installed it on your BeagleBone Blue, you'll need to start the server each time you'd like to use it by typing `vncserver` in a terminal window on the BeagleBone Blue.

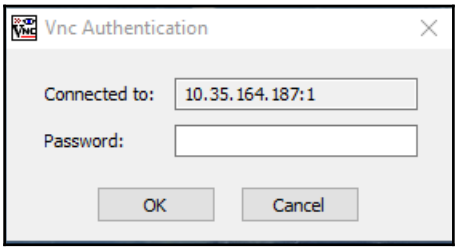
You can set up vncserver to start each time you power on your BeagleBone Blue; however, you'll only need it some of the time, mostly for debugging. If you'd like to set it to start each time, see the instructions at `https://arazblog.wordpress.com/2014/06/22/how-to-setup-vnc-server-on-beagle-bone-black-debian/`.

The first time you type `vncserver`, you'll be prompted for a password. This can, and maybe should, be a different password to your password to access your BeagleBone Blue. This will be the password your remote system will send to access the vncserver running on the board. Select a password--you don't need to set the password for the view-only capability--and then your vncserver will be running.

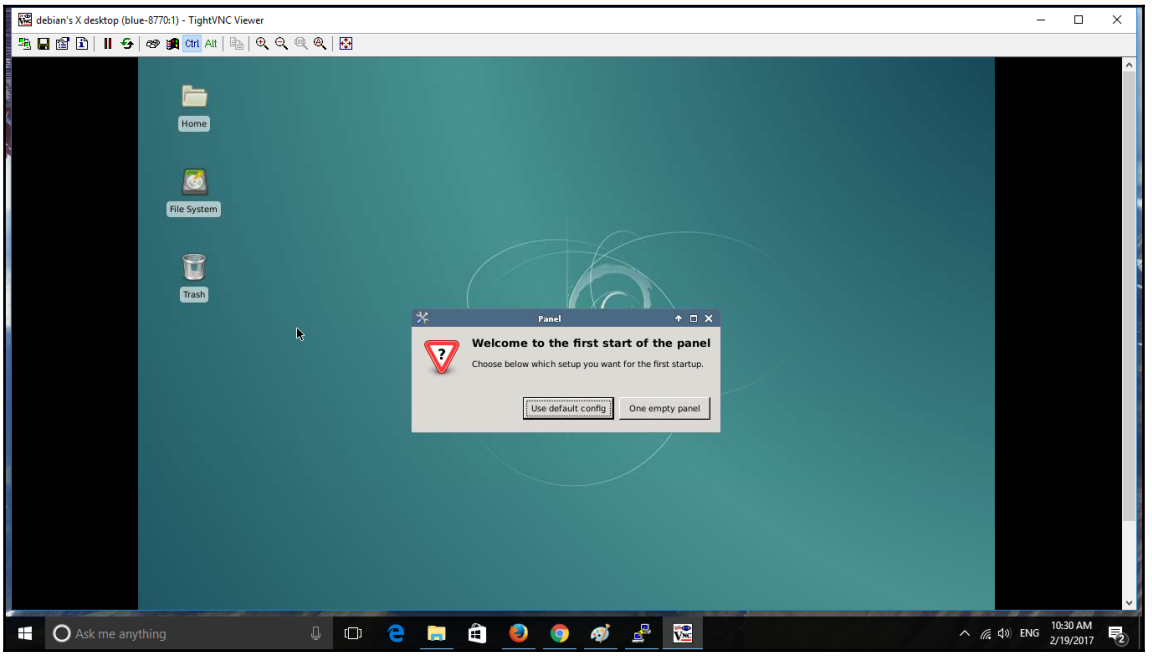
You'll need a VNC viewer application for your remote computer. On a Windows system, or on a Linux System that supports it, you can use the same Tightvncserver application. It is available for free download at `http://www.tightvnc.com/`. Download the file and follow the instructions to install it on your computer. After you install it, you can run it. You should see the following screen:



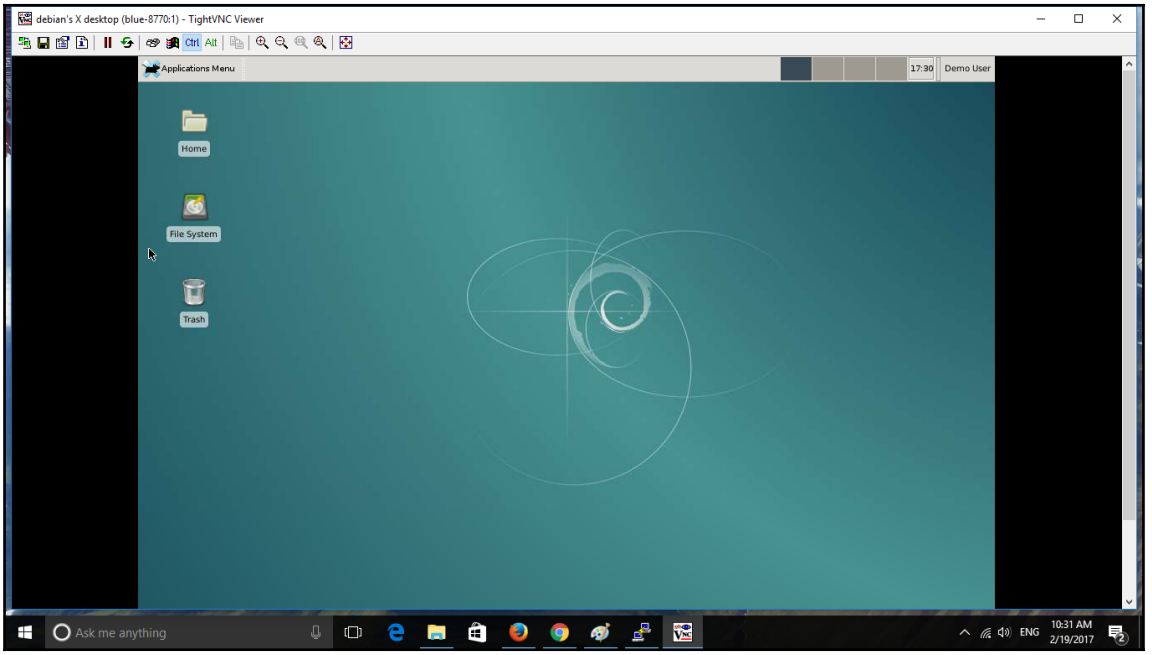
As shown, you'll enter the VNC server address, which is the IP address of your BeagleBone Blue, followed by a `:1`, and select **Connect**. The first time you connect, it may complain about connecting to an unknown computer, but let it know that this is **OK**. You will get this popup:



Type in the password you just entered while starting the vncserver, and you should then get a graphical view of your BeagleBone Blue, which hopefully looks like this:

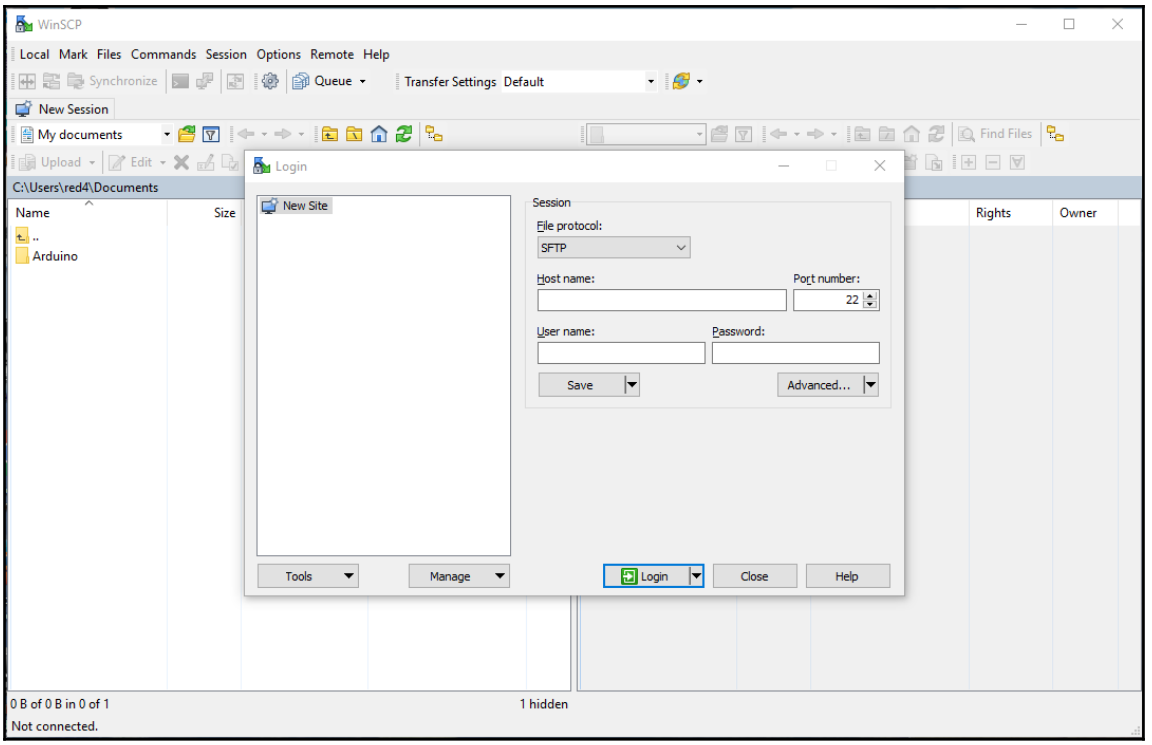


Select the **Use default config**, and you should now see the full graphical display of your BeagleBone Blue:

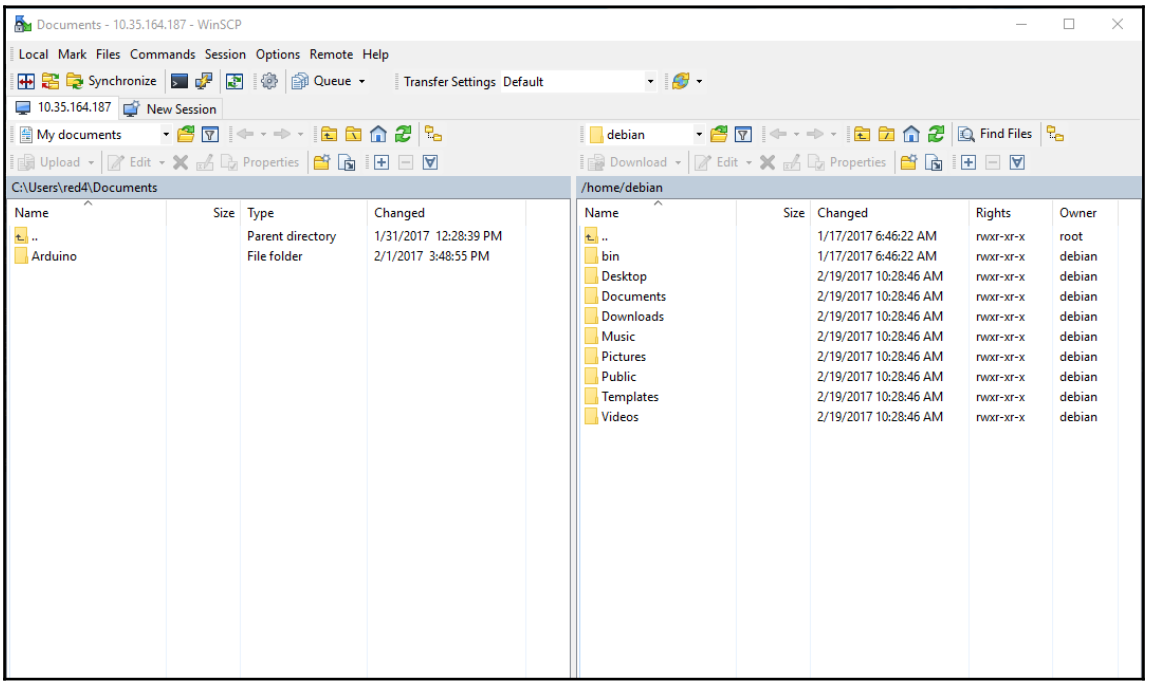


You can now access all the capabilities of your system, albeit possibly slower.

The final piece of SW you may want to use if you are running a Microsoft Windows system is a free application called WinSCP. To download and install this piece of SW, simply search the web for WinSCP and follow the instructions. Once installed, run the program. It will open the following dialog box:



Fill in the host name with your inet address, as shown, and you should come to a dialog box that will ask you for your username and password. After you fill those in and hit *Enter*, you should see the following screen:



Now you can drag and drop files from one system to the other.

Once you've completed this step, you can access your system to the full extent remotely. You need to only connect the power and connect to the board via wifi. If you need to issue simple commands, connect via SSH. If you need a more complete set of graphical functionality, you can access this via vncserver. Finally, if you are using a Windows system and want to transfer files back and forth, you have access to WinScp. Now you are ready to start using the BeagleBone Blue.

Summary

Congratulations! You've completed the first stage of your journey. You have your BeagleBone Blue up and working--no gathering dust in the bin for this piece of HW. It is now ready to start connecting to all sorts of interesting devices in all sorts of interesting ways. One thing that is important to note: the system is not going to be nearly as stable as your PC or Mac. The HW and SW is new, but you'll be successful even though you might have to return and complete some steps again. In the next chapter you'll learn how to program the BeagleBone Blue, an excellent first step in your quest to create amazing projects.

2
Programming the BeagleBone Blue

Before you get started with building your robotic projects, let's take a bit of time to either introduce or review how to program the BeagleBone Blue.

In this chapter, you will:

- Learn some of the basic Linux commands and get to know how to navigate around the file system on the BeagleBone Blue
- Learn how to create, edit, and save files on the BeagleBone Blue
- Learn how to create and run Python programs on the BeagleBone Blue
- Learn how the C programming language is both similar and different from Python so you can both create and edit C code files

Now that things are up and running, you'll want your BeagleBone Blue to start doing something. 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 basic Linux commands and Python and C programming.

While it is fun to build hardware, and you'll spend a good deal of time actually designing and building your robots; without programming, your robots won't get very far. This chapter will introduce you to some important tasks such as file creation and editing as well as some Python and Cd programming concepts so you'll feel comfortable creating some of the fairly simple programs that you'll learn about throughout the book. You'll also learn how to change programs that are already available, making your robot do even more amazing things.

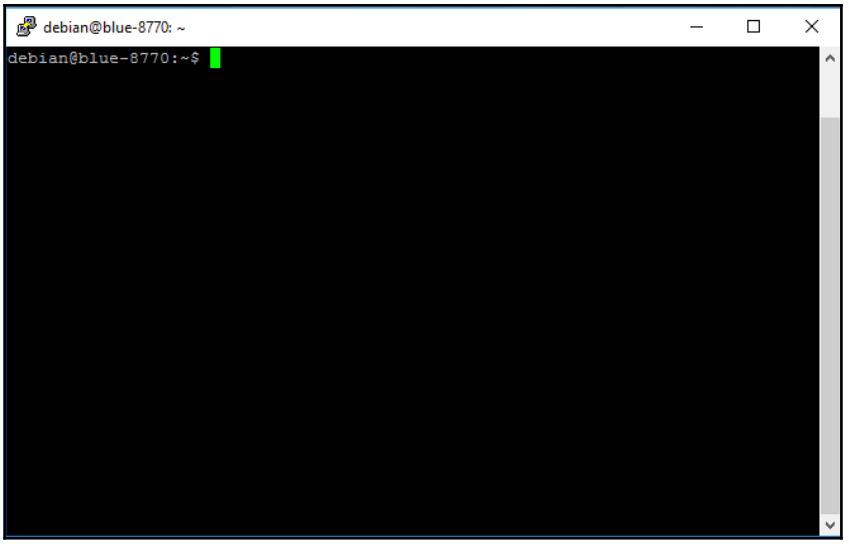
You're going to use the basic configuration that you created in `Chapter 1`, *Getting Started with the BeagleBone Blue*. You can accomplish the tasks in this chapter either by remotely logging into the BeagleBone Blue using VNC server or remotely logging using SSH.

Basic Linux commands showing how to navigate around the filesystem on the BeagleBone Blue

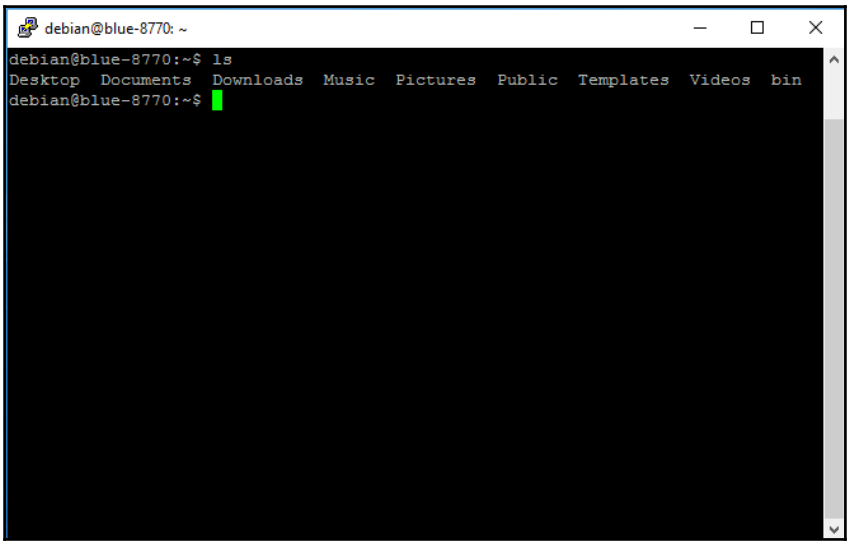
After completing `Chapter 1`, *Getting Started with the BeagleBone Blue* , you should have a working BeagleBone Blue running a version of Linux called Ubuntu. We selected this one because it is the most popular and thus has the largest set of supported hardware and software. The commands I am going to review should also work with other versions of Linux, but I'll be showing examples using Ubuntu.

So, power up your BeagleBone Blue, connect via Wi-Fi, and log in to user SSH with the proper username and password. Now you are ready for a quick tour of Linux. This will not be extensive, but you will learn some of the basic commands and how to access some fundamental functionality.

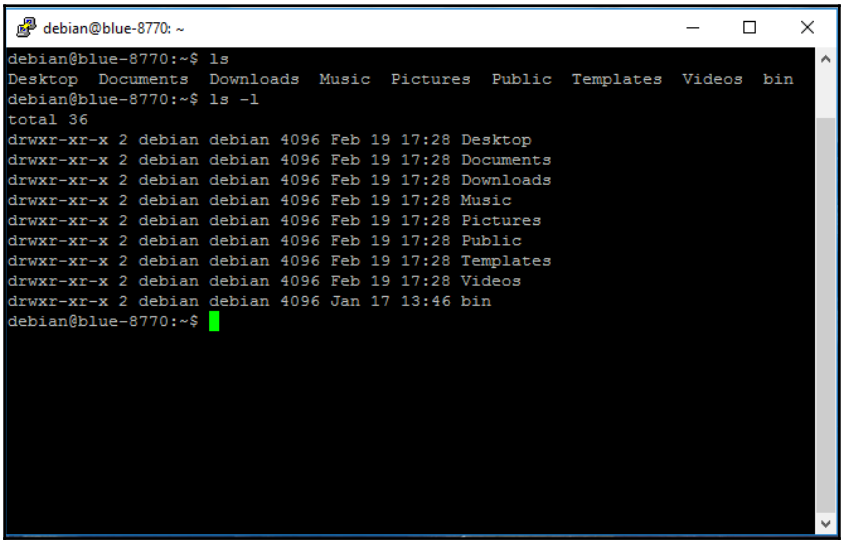
Once you have logged in, you should have an active Terminal window. It should look something like this:



Your cursor is at Command Prompt. Unlike Microsoft Windows or Apple's OS, most of our work will be done by actually typing commands into the command line. So, let's try a few. First, type `ls`, and you should see something like this:

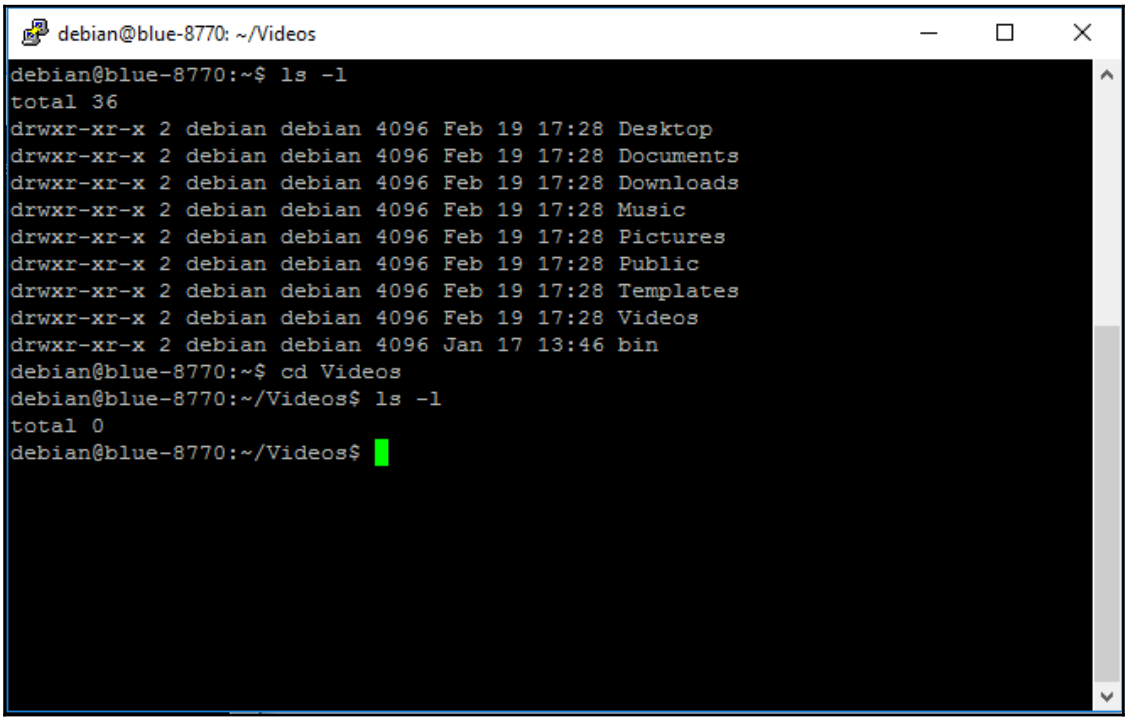


The `ls` command in Linux is the list command, and it lists all the files and directories in the current directory. If you'd like more information on your files, you can type `ls -l`. This should display the following screenshot:

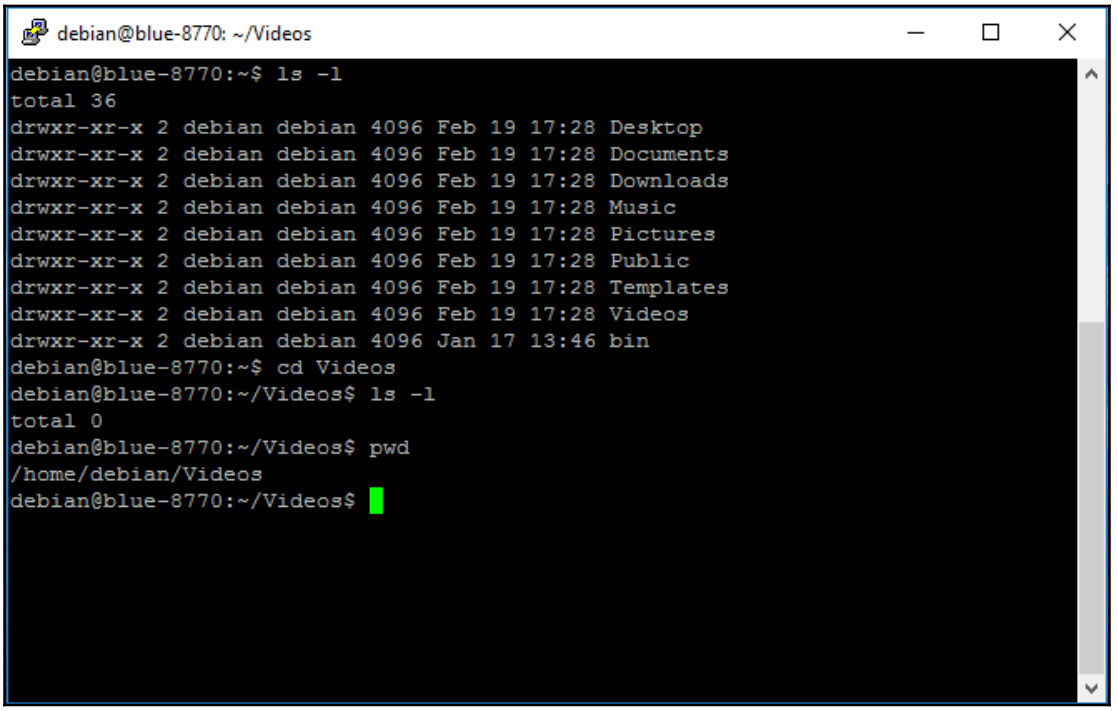


This command provides information about who owns the files, the time they were created, and the various permissions on the files. The files are listed by their names, you can tell the directories because they are normally in a different color, and the letter `d` proceeds the lines for those listing. In this case, `Videos` is a directory. The default installation of `debian` has only a single `bin` directory, and installing the `Xfce` windows manager creates the `Desktop`, `Documents`, `Downloads`, `Music`, `Pictures`, `Public`, `Templates`, and `Videos` directories.

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 `Videos` directory, type `cd Videos`. Now if you issue the `ls -l` command, you should see something like this:

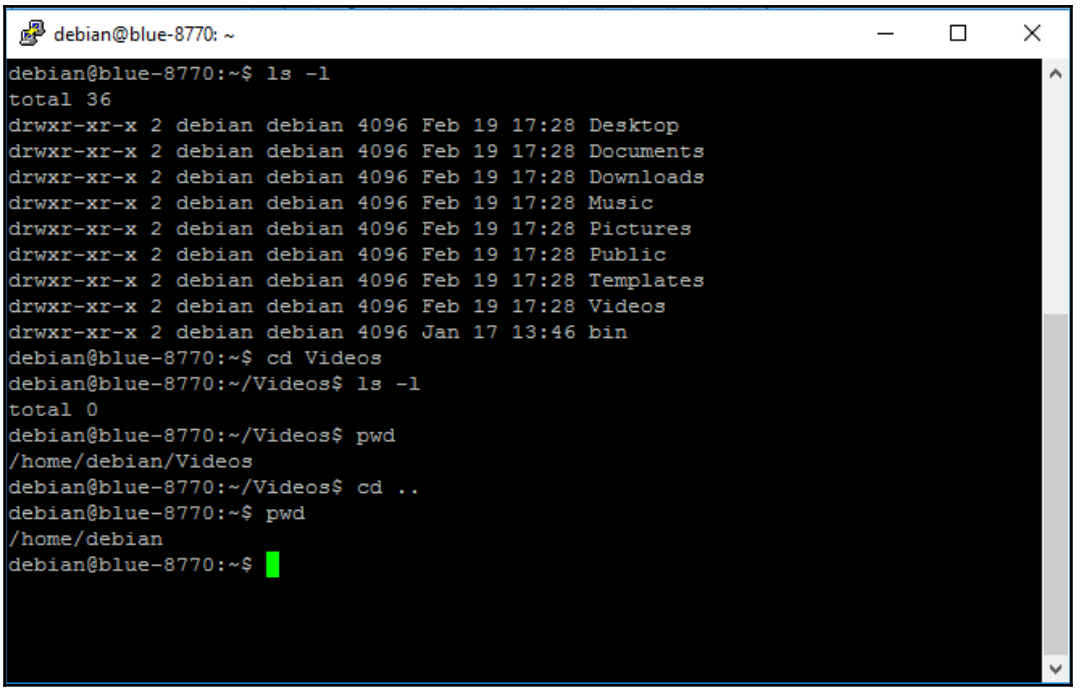


This directory is empty. Now, I should point out that you used a shortcut when you typed `cd Videos`. This command assumed that you wanted to start from the current directory. You can always find the full directory name of your current directory by typing `pwd` (print working directory). If you do that, here is what you should get:



To get to this same directory, you could also have typed `cd /home/debian/Videos` and gotten the exact same result, because you started in the `/home/debian` directory, which is the directory where you always start when you first log into the system.

Now, you can use two different shortcuts to move back to the default directory. The first is to type `cd ..`; this will take you to the directory just above this one in the hierarchy. Do this and then type `pwd`, and you should see the following:



The other way to get back to the home directory is to type `cd ~` as this will always return you to the home directory. You can use these shortcuts, or you can use the entire path name. In this case, if you want to go to the `/home/ubuntu/Video` directory from anywhere in the file system, simply type `cd /home/ubuntu/Video` and you will go to that directory.

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

Linux command:	**What it does:**
`ls -l`	List-long: Lists all the files and directories in the current directory. This includes lots of extra information about the file, including the time at which it was created, permissions, owners, and so on.
`ls`	List-short: Lists all the files and directories in the current directory by just their names.

`rm filename`	Remove: Removes whichever file is specified by the filename.
`mv filename1 filename2`	Move: Renames `filename1` to `filename2`
`cp filename1 filename2`	Copy: Copies `filename1` to `filename2`
`mkdir directoryname`	Make directory: Makes a directory with the name `directoryname`; this will be made in the current directory unless otherwise specified.
`clear`	Clear: Clears the current Terminal window
`sudo`	Super user: If you type the `sudo` command in front of any command, it will execute that command as the super user. This can 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 program you want to run and it seems to suggest that the command does not exist, or permission is denied, try it again with `sudo` in front of the command or the name of the program.

Now you can play around and look at your system and the files that are available to you. Be a bit careful! Linux is not like Windows; it will not warn you if you try to delete a file or copy over a current file. And if you delete the wrong file, your system may no longer be usable and you'll need to reinstall the Debian operating system.

If you need to reinstall the operating system, go to `beagleboard.org`, select your board, and go to the getting started tab. It will give you instructions on how to download and install the latest operating systems.

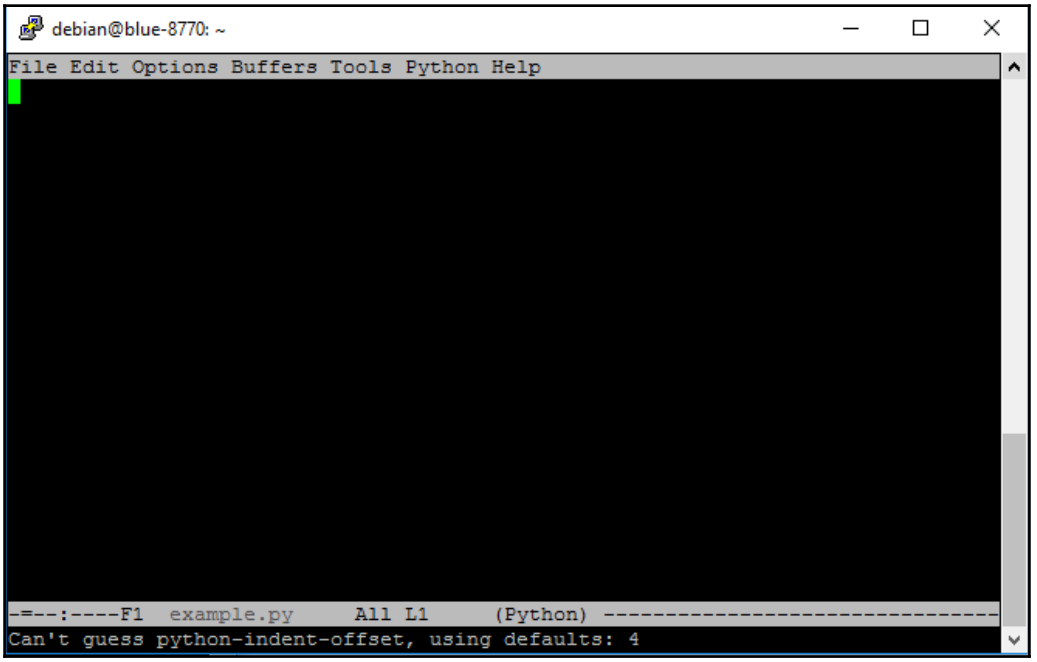
Creating, editing, and saving files on the BeagleBone Blue

Now that you can log in and move easily between directories and see which files are in your directories, 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 have worked in Microsoft Windows, you probably have used a program such as Microsoft Notepad, WordPad, or Word to do this. As you might imagine, these are not available in Linux.

There are several choices, all of which are free. I am going to show you how to use an editor program called **Emacs**. Other possibilities are programs such **nano**, **vi**, **Vim**, and **gedit**. There is nothing more religious than conversation among programmers about which editor to use, so if you have a strong opinion, you are probably already used to using one of these editors. The vi and nano editors are included in the regular Debian distribution; however, I personally find Emacs the most intuitive of the editors, so will show you how to install and use it here. You can use any of the other editors to complete the same task, if you wish.

If you want to use Emacs, then download and install Emacs by typing `sudo apt-get install emacs`.

Once installed, you can run Emacs simply by typing `emacs filename`, where the filename is the name of the file you want to edit. If the file does not exist, then Emacs will create the file. Here is what you will see if you type `emacs example.py` in the prompt:



Note that unlike Windows, Linux doesn't automatically assign file extensions; it is up to us to specify what kind of file we want to create. Also, if you are running Emacs from SSH, you won't have the mouse available, so you'll need to move around 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 you would normally use the mouse to select. For example, when you are ready to save the file, you'll type `ctrl-x ctrl-s`, and that will save the file under the current filename. When you want to quit Emacs, you'll type `ctrl-x ctrl-c`. This will stop Emacs and return you to Command Prompt. Here are a number of keystroke commands you might find useful:

Emacs Command:	**What it does:**
`ctrl-x ctrl-s`	Save: Saves the current file
`ctrl-x ctrl-c`	Quit: Exits Emacs and returns to Command Prompt
`ctrl-k`	Kill: Erases the current line
`Ctrl-_`	Undo: Undoes the last action
Mouse - left mouse button - text selection Cursor - right mouse button	Cut and paste: If you select the text you want to paste with the mouse using the left mouse button, then move the cursor to where you want to paste the code and then hit the right mouse button and the code will be pasted to 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 the BeagleBone Blue

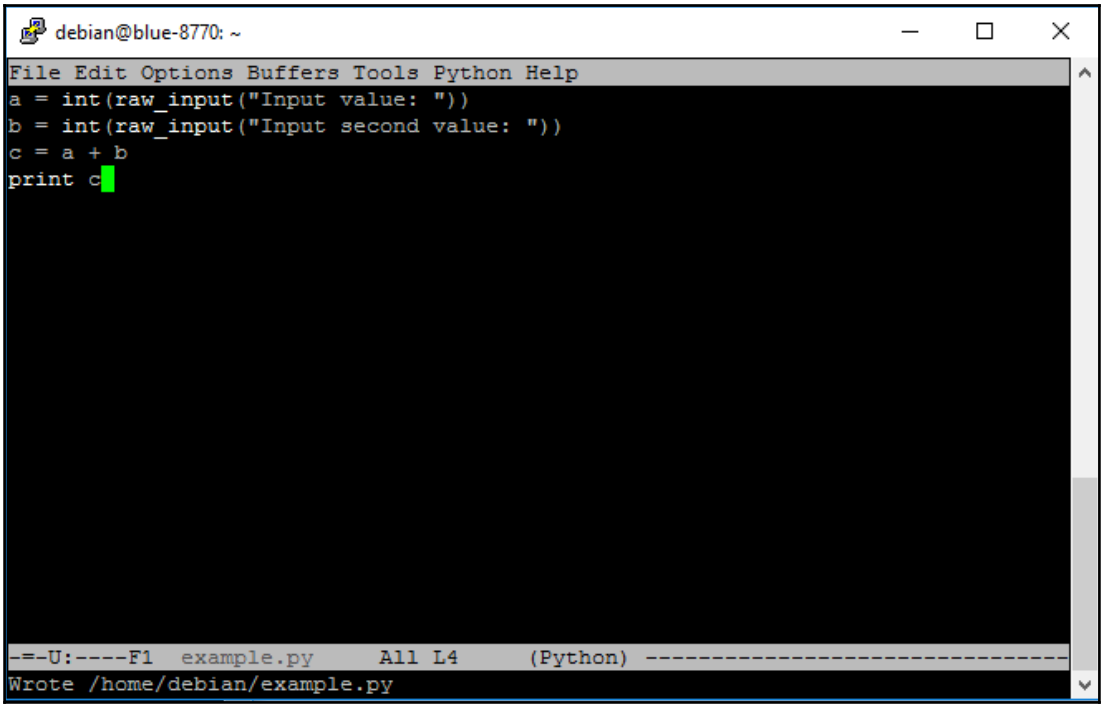
Now that you can get around and even edit programs, you can begin to use the BeagleBone Blue to create programs so you can control your robotic projects.

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. You'll use Python initially for two reasons. First, it is a simple language that makes intuitive sense and is very easy to use. Second, much of the open source functionality in the robotics world is available in Python. You'll also learn a bit of C in this chapter as well because sometimes, functionality is only available in C. But it makes the most sense to start in Python. To work the examples in this section, you'll need a version of Python installed to complete this section. Fortunately, the basic Debian system has a version already installed, so you are ready to begin. You can see what version is installed by typing Python --version.

You are going to learn just 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 tools, go to `www.codeacademy.com` or `http://www.learnpython.org/` and give it a try.

In this section, you'll learn how to create and run a Python file. It turns out that Python is an interactive language, so you could run Python and then type in commands one at a time at Command Prompt. But you want to use Python to create programs, so you are going to type your commands into a file using Emacs. Then, you'll run the file from the command line by invoking Python and specifying the command file. 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 these five lines:

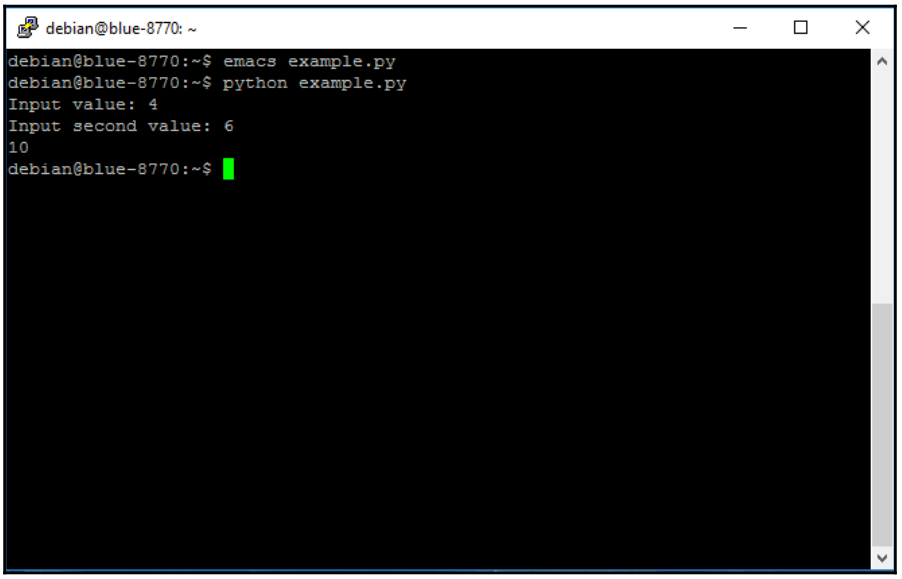


On your monitor, your code may be color-coded. I have removed the color coding here so it is easier to read.

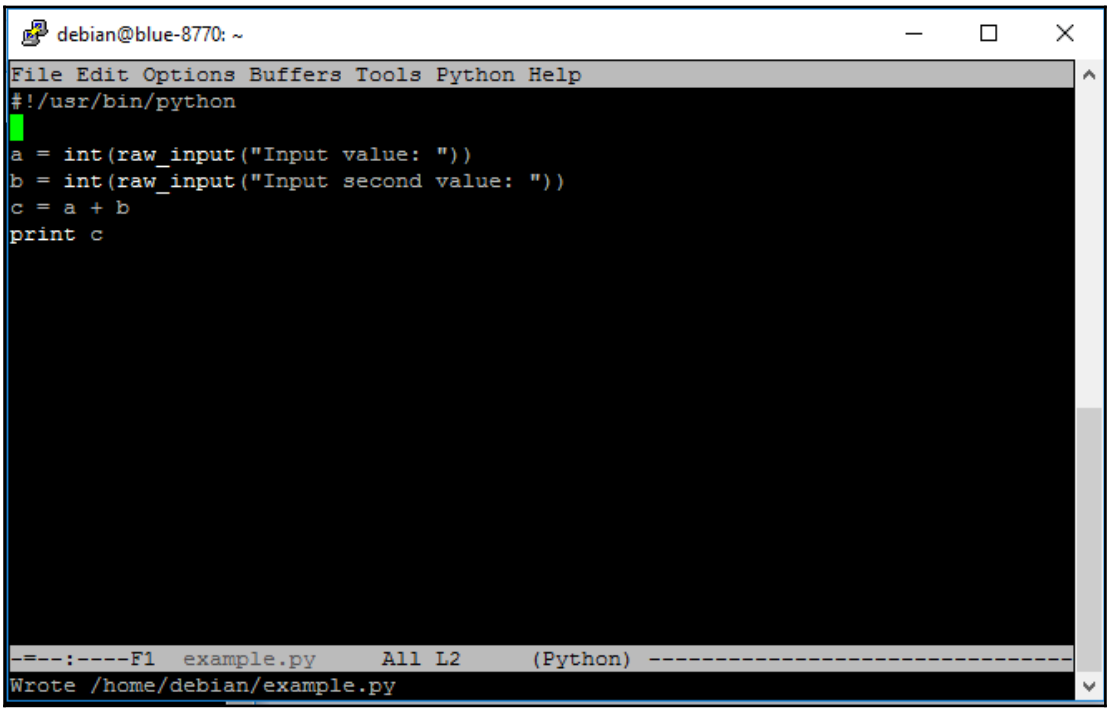
Here is an explanation of the code:

1. `a = int(raw_input("Input value: "))`: One of the basic needs of a program is to get input from the user. The `raw_input` command allows you to do that. The data will come to the program as a string; the `int()` function turns the string to an integer. In this case, the characters typed by the user will come in as a string, be converted into an integer, and then be stored in the storage location named variable a. The `"Input value: "` prompt will be shown to the user.
2. `b = int(raw_input("Input second value: "))`: This data will also be input by the user and stored in the storage location named variable `b`. The `Input second value:` prompt will be shown to the user.
3. `c = a + b`: This is an example of something you can do with the data; in this example, you can add the value in the storage location named variable a and the storage location named variable `b` and place it in the storage location labeled `c`.
4. `print c`: Another basic need of your program is to print out results. The print commands prints out the value stored in the storage location named variable `c`.

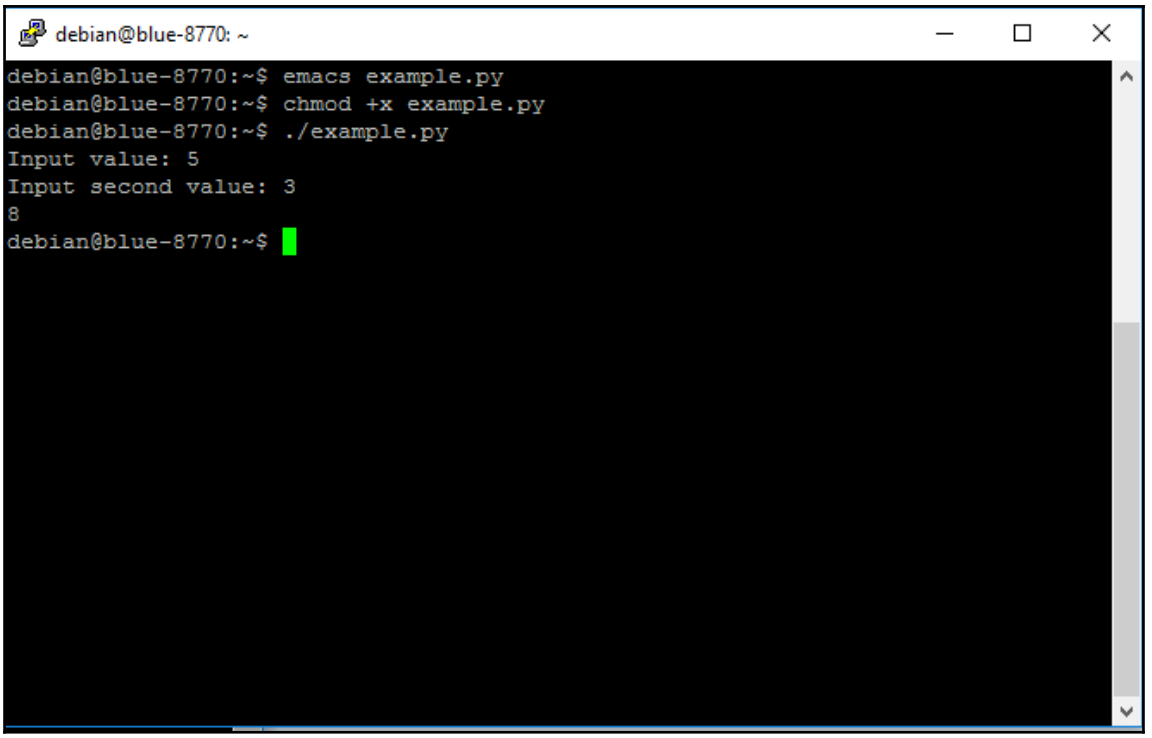
Once you have created your program, save it (using `cntrl-x cntrl-s`) and quit Emacs (using `cntrl-x cntrlc`). Now from the command line, run your program by typing `python example.py`. You should see something like this:



You can also run the program right from the command line without typing the `Python` filename by adding one line to the program. Now the program looks like this:



Adding `#!/usr/bin/python` as the first line simply makes this file available for you 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 believe it can execute it. You should be able to simply type `./example.py` and the program should run, as follows:



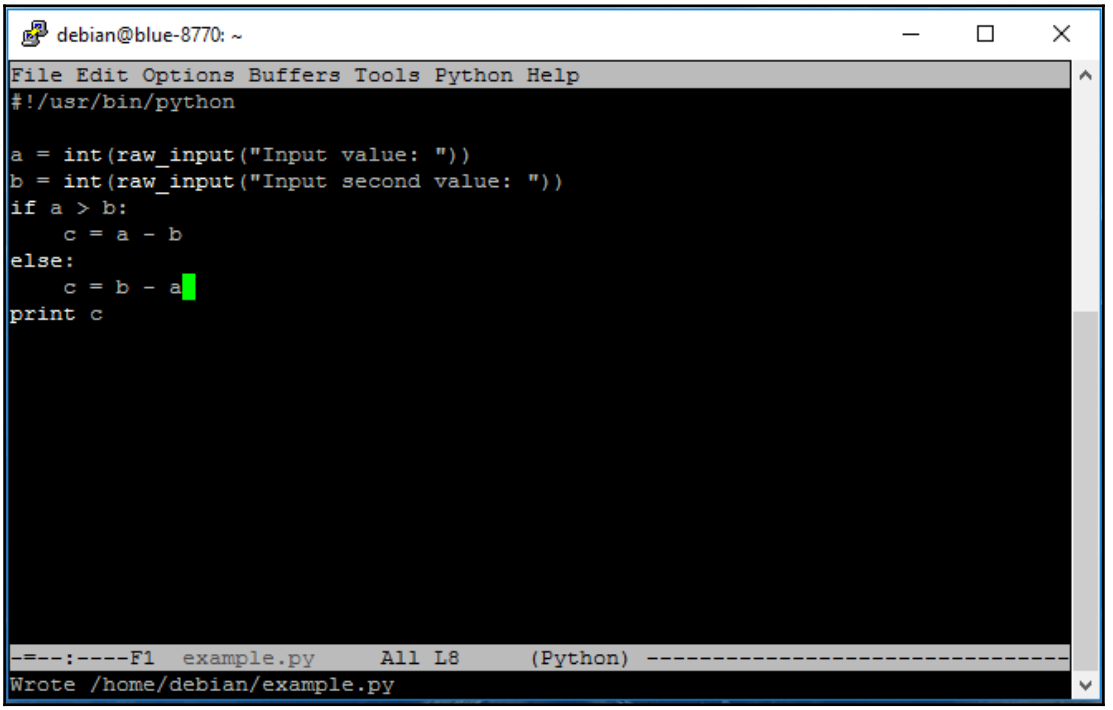
Note that if you simply type `example.py`, the system will not find the executable file. In this case, the file has not been registered with the system, so you have to give it a path to the file, in this case, `./example.py` is the current directory.

Now that you know how to create, enter, and run your simple Python programs, let's look at some programming constructs.

Some basic programming constructs on the BeagleBone Blue

Now that you know how to run a simple Python program on the BeagleBone Blue, let's look at some more complex programming constructs. Specifically, you'll learn what to do when you want to decide what instructions to execute and show how to loop your code to do the same thing more than once. You'll learn how to use libraries in Python code and how to organize statements into functions. Finally, you'll learn about some very basic object-oriented concepts.

As with the previous section, once you have the basic system and Emacs, you are ready to start coding. As you have seen, your programs normally start with the first line of code and then continue executing the next line until your program runs out of code. This is fine, but what if you want to decide between two different courses of action? You can do this in Python using an `if` statement. Here is some example code:

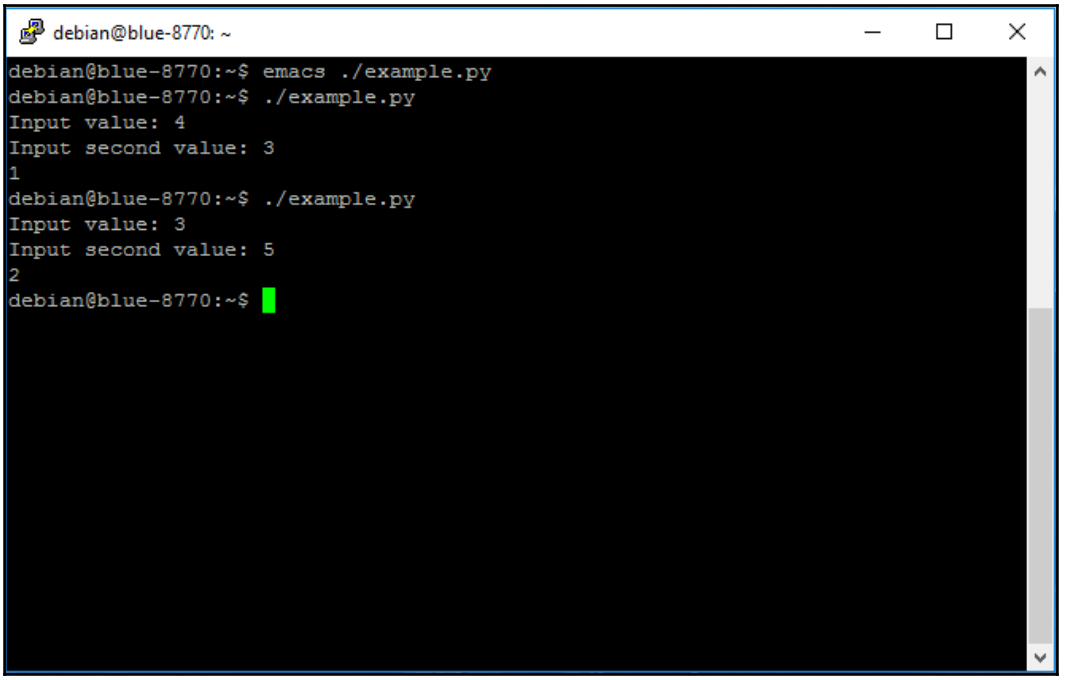


Here is the detail, line by line:

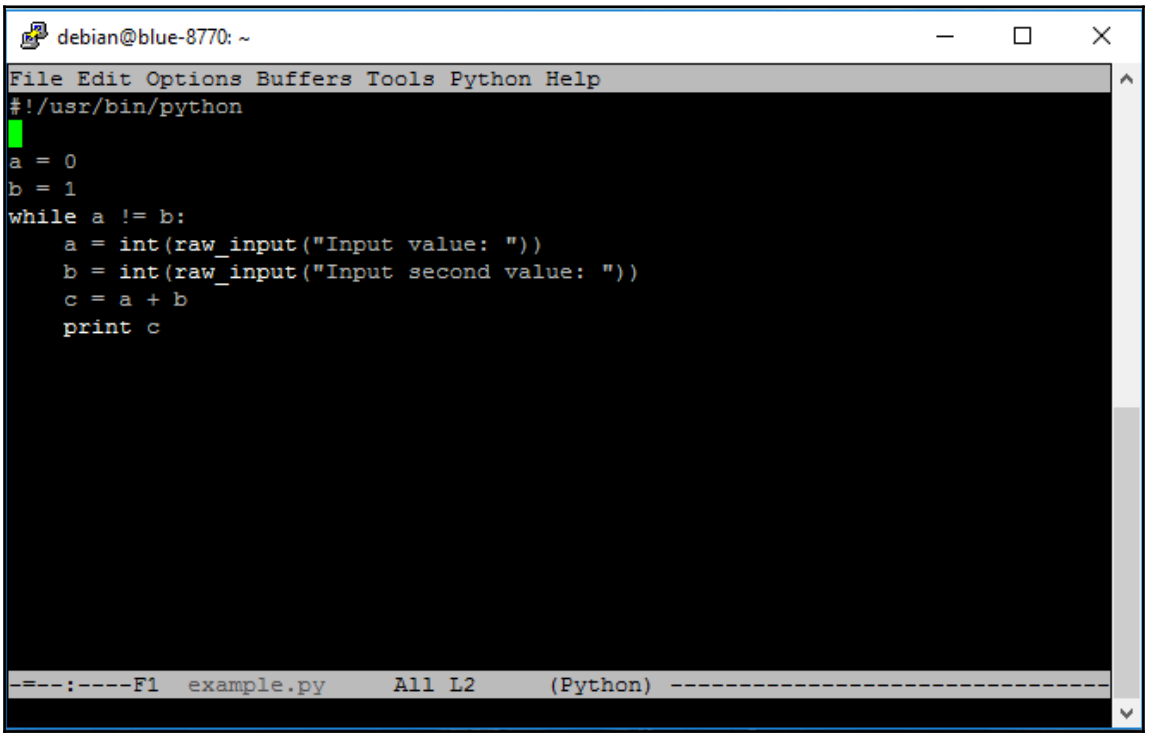
1. `#!/usr/bin/python`: This is included so you can make your program executable.
2. `a = int(raw_input("Input value: "))`: As discussed earlier, this allows the program to get input by the user and store it in variable a. The `"Input value: "` prompt will be shown to the user.
3. `b = int(raw_input("Input second value: "))`: This data will also be input by the user and stored in variable `b`. The `Input second value:` prompt will be shown to the user.

4. `if a > b:`: This is an if statement. The expression is evaluated, in this case, a check to see whether the data stored in variable a is greater than data stored in variable b. If it is true, the program will execute the next statement(s) that are indented. If not, it will skip those statement(s).
5. `else:`: This is an optional part of the command. If the expression in the `if` statement is evaluated as `false`, then the indented statement(s) will be executed.
6. `print c`: This prints out the data stored in variable `c`.

You can run this program a couple of times, checking both possibilities of the `if` expression:



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

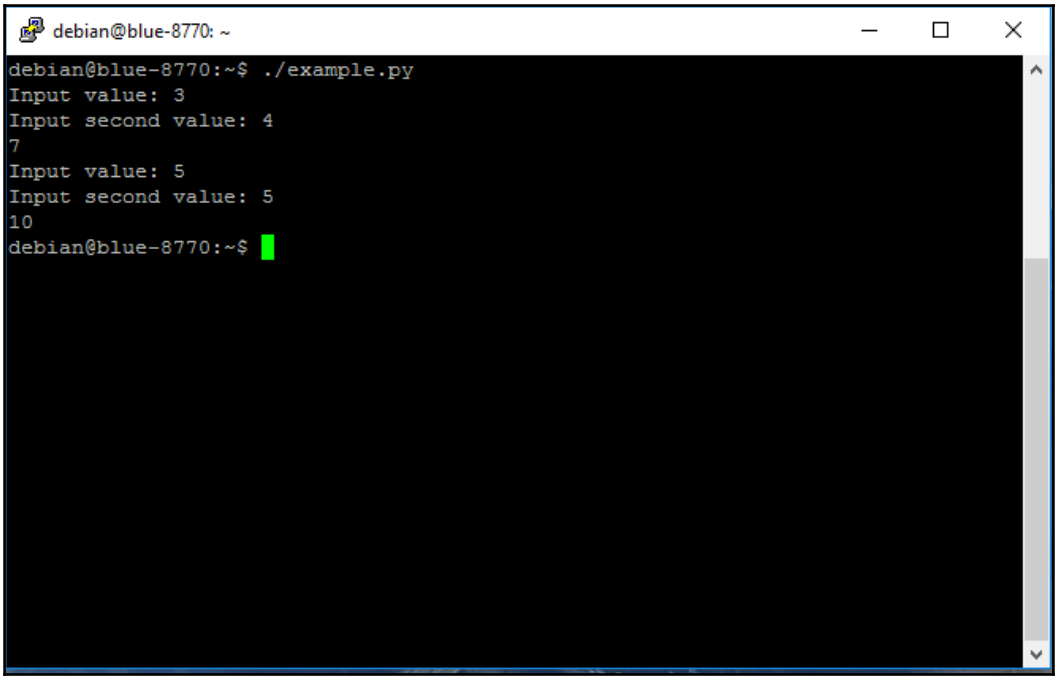


Here are the details of this code:

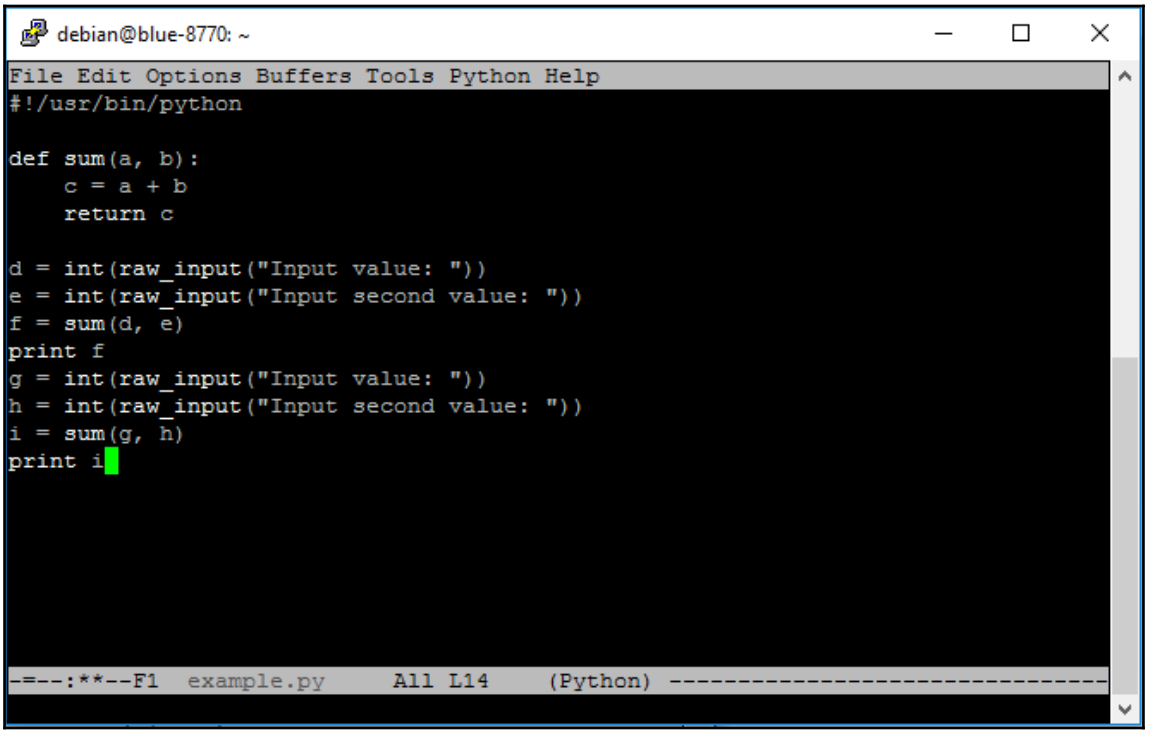
1. `#!/usr/bin/python`: This is included so you can make your program executable.
2. `a = 0`: Set the value of variable `a` to `0`. We'll need this only to make sure we execute the loop at least once.
3. `B = 1`: Set the value of variable `b` to `1`. We'll need this only to make sure we execute the loop at least once.
4. `While a != b:`: The expression `a != b` (in this case, `!=` means not equal to) is checked. If it is `true`, then the statement(s) that are indented are executed. When the statement evaluates as false, then the program jumps to the statements after the indented section.

5. `a = int(raw_input("Input value: "))`: As before, this allows the program to get input by the user and store it in variable a. The `"Input value: "` prompt will be shown to the user.
6. `b = int(raw_input("Input second value: "))`: This data will also be input by the user and stored in variable `b`. The `"Input second value: "` prompt will be shown to the user.
7. `c = a + b`: The variable `c` is loaded with the sum of the value of the data stored in variable `a` and the data stored in variable `b`.
8. `print c`: The `print` command prints out the value of data stored in variable `c`.

Now you can run the program, and note that when you enter the same value for `a` and `b`, the program stops:



The next concept we need to cover is how to put a set of statements into a function. A function is really just a set of instructions that you might want to run more than once. Instead of copying these lines over and over, if you put them in a function, you can simply call the function by entering its name and the program will go to the function and execute its lines of code. Here is the code:

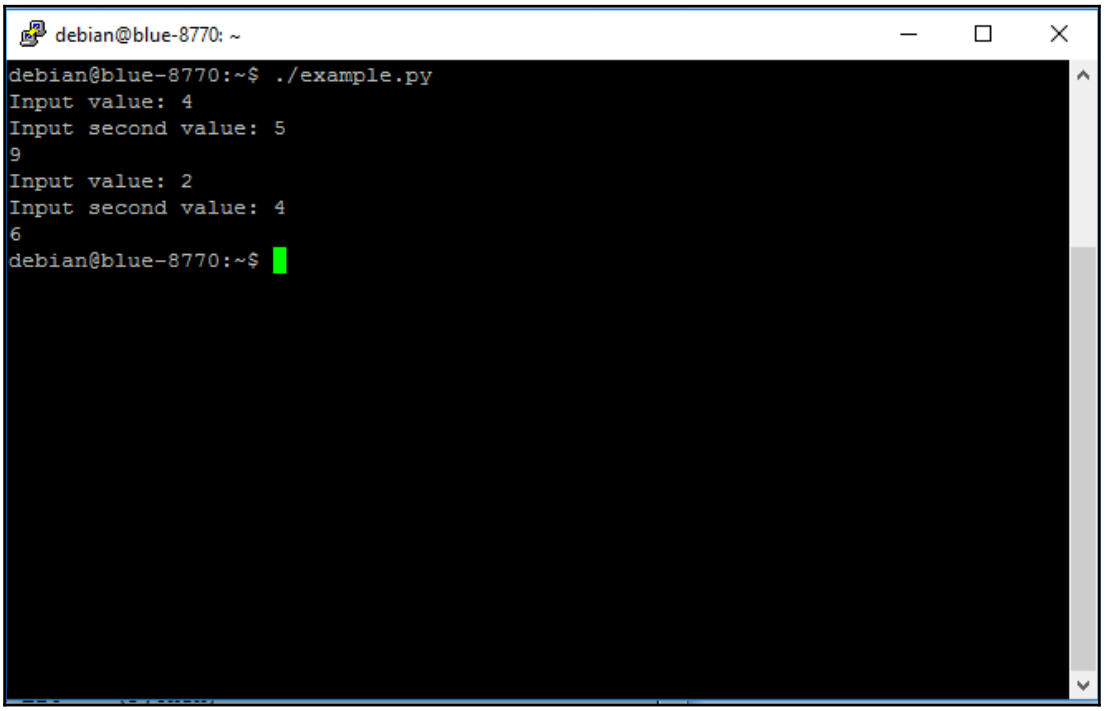


And here is the explanation of the code:

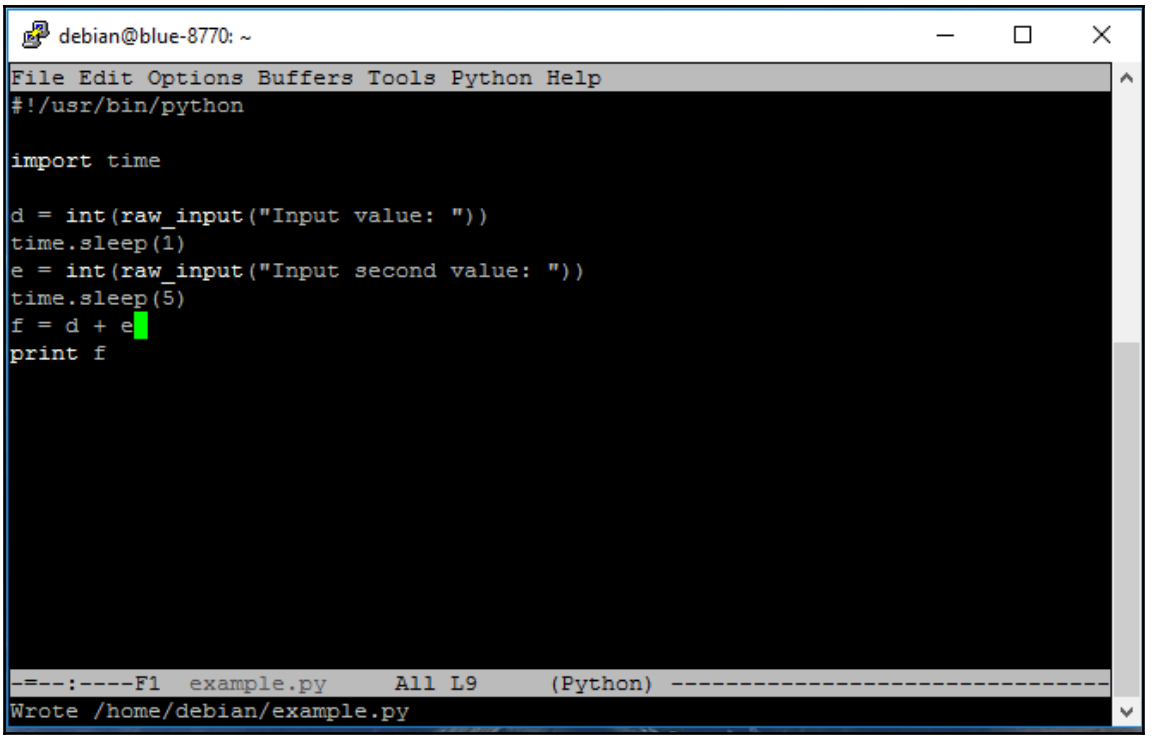
1. `#!/usr/bin/python`: This is included so you can make your program executable.
2. `def sum(a, b)`:: This defines a function whose name is sum. This function takes two arguments, the variables `a` and `b`.
3. `c = a + b`: Any time this function is called, it will add the value in the storage location variable a with the value in storage location variable `b` and place it in storage location variable `c`.

4. `return c`: When the function is finished, it will return the value stored in variable c to the calling expression.
5. `d = int(raw_input("Input value: "))`: As already discussed, this will take the data input by the user and store it in the variable d. The `"Input second value: "` prompt will be shown to the user.
6. `e = int(raw_input("Input second value: "))`: Again, this will take the data input by the user and store it in variable `b`. The `"Input second value: "` prompt will be shown to the user.
7. `f = sum(d, e)`: The `sum` function is called. The value in the storage location labeled variable `d` is copied into variable `a` in the `sum` function, and the value in the storage location variable `e` is copied to variable `b` in the `sum` function. The program then goes to the `sum` function and executes it. The return value is then stored in variable `f`.
8. `print f`: The `print` command prints out the value stored in variable `f`.
9. `g = int(raw_input("Input value: "))`: This will take the data input by the user and store it in the variable `g`. The `"Input second value: "` prompt will be shown to the user.
10. `h = int(raw_input("Input second value: "))`: Again, this will take the data input by the user and store it in variable `h`. The `"Input second value: "` prompt will be shown to the user.
11. `i = sum(g, h)`: The `sum` function is called. The value in the storage location labeled variable `g` is copied into the variable `a` in the `sum` function, and the value in the storage location variable `h` is copied to the variable `b` in the `sum` function. The program then goes to the sum function and executes it. The return value is then stored in the variable `i`.
12. `print i`: The `print` command prints out the value stored in variable `i`.

Thus, you can execute the sum twice without copying the statements twice. This decreases the code size and also the number of defects. And here is the result when you run the code:



Now that you know about function, another capability you may need is to add functions to your programs using libraries. Libraries include functions that you or someone else have created that you want to add to your code. You won't need to cut and paste the code into your file as long as the function exists and your system knows about it; then, you can include the library. So let's modify your code again:

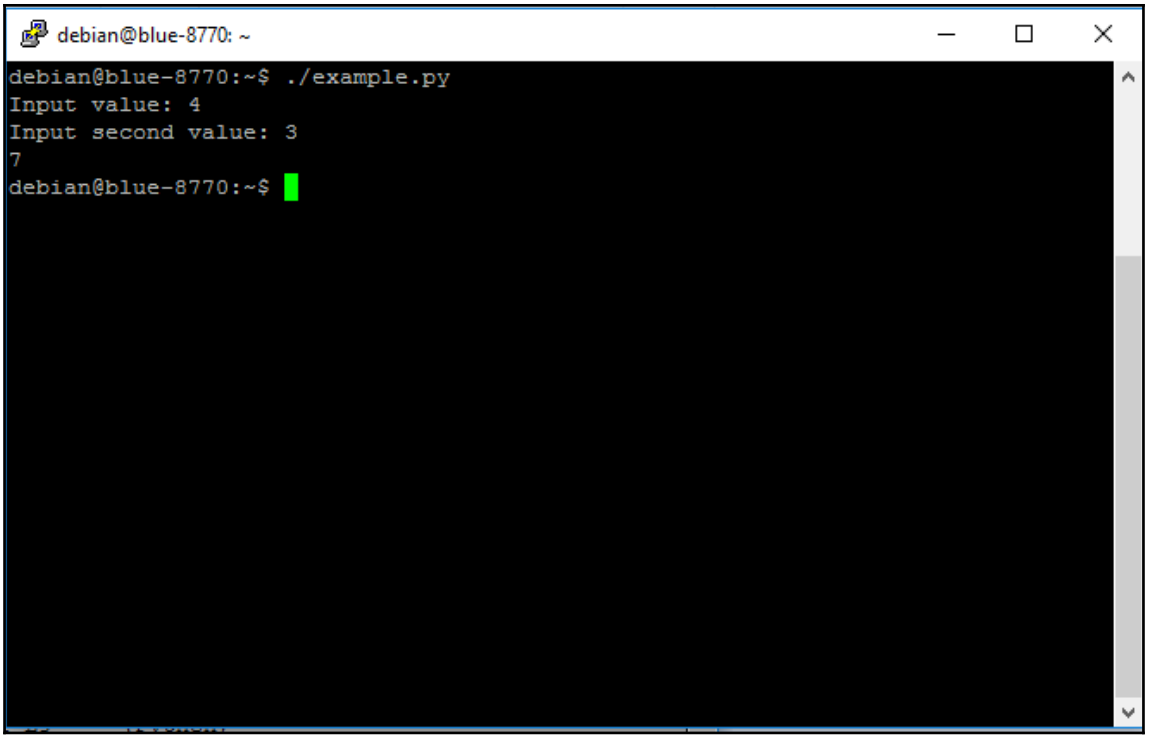


And here is the line-by-line description of the code:

1. `#!/usr/bin/python`: This is included so you can make your program executable.
2. `import time`: This includes the `time` library. The `time` library includes a function that allows you to pause for a certain number of seconds.
3. `d = int(raw_input("Input value: "))`: This data will input by the user and stored in variable `d`. The `"Input second value: "` prompt will be shown to the user.
4. `time.sleep(1)`: This line calls the `sleep` function in the `time` library for the `sleep` function, which will cause a `1` second delay.
5. `e = int(raw_input("Input second value: "))`: This data will also be input by the user and stored in variable `e`. The `"Input second value: "` prompt will be shown to the user.

6. `time.sleep(1)`: This line calls the sleep function in the `time` library for the `sleep` function, which will cause a `5` second delay.
7. `f = d + e`: The variable `d` is added to the variable `e` and the result is loaded into variable `f`.
8. `print f`: The `print` command prints out the value of the variable `f`.

And your result is as follows:



Of course, this looks very similar to other results. But you will notice a pause between when you enter the first value and the second value. There are thousands of these kinds of libraries, some that are included in the standard Python language and others that can be downloaded and installed. You'll see some examples of these in later chapters.

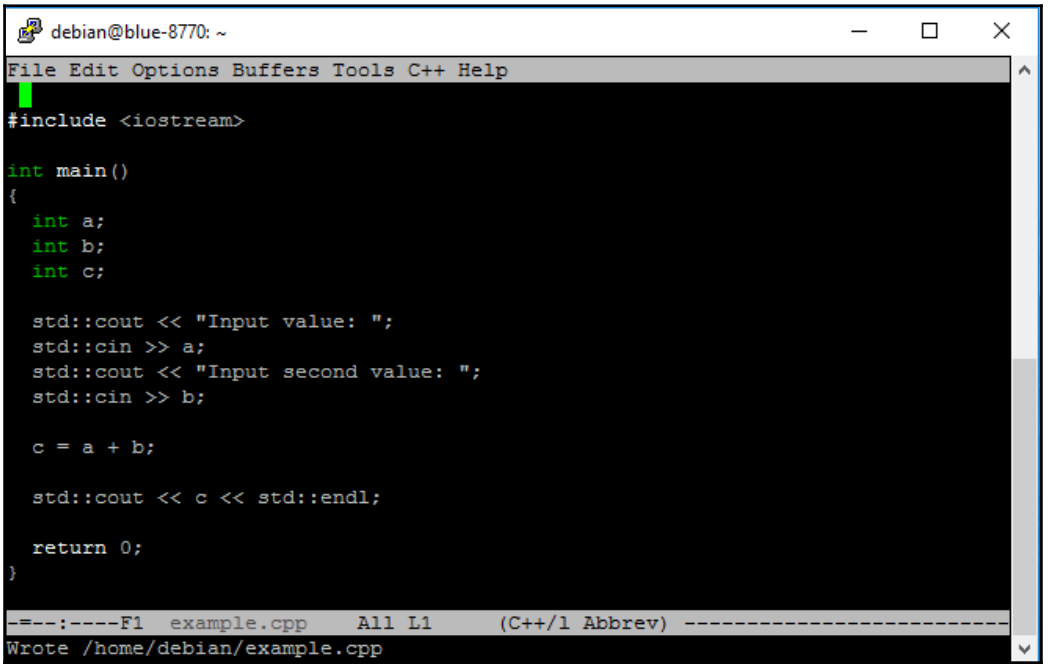
Now that you have an idea of the basics of Python coding, let's introduce you very briefly to the C coding language.

A brief introduction to the C programming language

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

As with Python, you will need to have access to the language capabilities. These come in the form of a compiler and build system, which turns your text files that contain programs to 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 are installed, let's walk through some simple examples. Here is the first C code example:



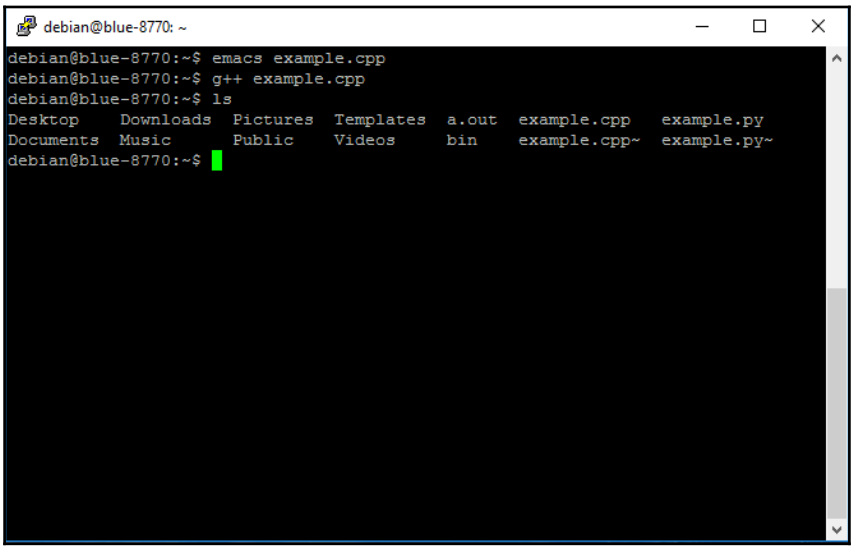
And here is an explanation of the code:

1. `#include <iostream>`: Much like Python, C uses libraries to bring in functions that others have written. This is a library that is included so your program can input data from the keyboard and output information to the screen.
2. `int main()`: Unlike Python, you need to define a starting point where you want your program to begin execution. C defines this as the main function.
3. `int a;`: This defines a variable named `a`, of type `int`. C is what we call a strongly typed language, which means you need to declare the type of the variable you are defining. The normal types are `int` , a number that has no decimal points; `float` , a number that requires decimal points; `char` , a character of text; and `bool` , a true or false value.
4. `int b;`: This defines a variable named `b`, of type `int`.
5. `int c;`: This defines a variable named `c`, of type `int`.
6. `std::cout << "Input value: ";`: This will display the string `"Input value: "` to the screen.
7. `std::cin >> a;`: This is the input that the user types will go into the variable `a`.
8. `std::cout << "Input second value: ";`: This will display the string `"Input second value: "` to the screen.
9. `std::cin >> b;`: This is the input that the user types will go into the variable `a`.
10. `c = a + b`: The statement is a simple addition of two values in variable `a` and variable `b`.
11. `std::cout << c << std::endl;`: The `cout` command prints out the value of `c`. The `endl` command at the end prints out a carriage return so that the next character appears on the next line.
12. `return 0;`: The main function ends and returns `0`.

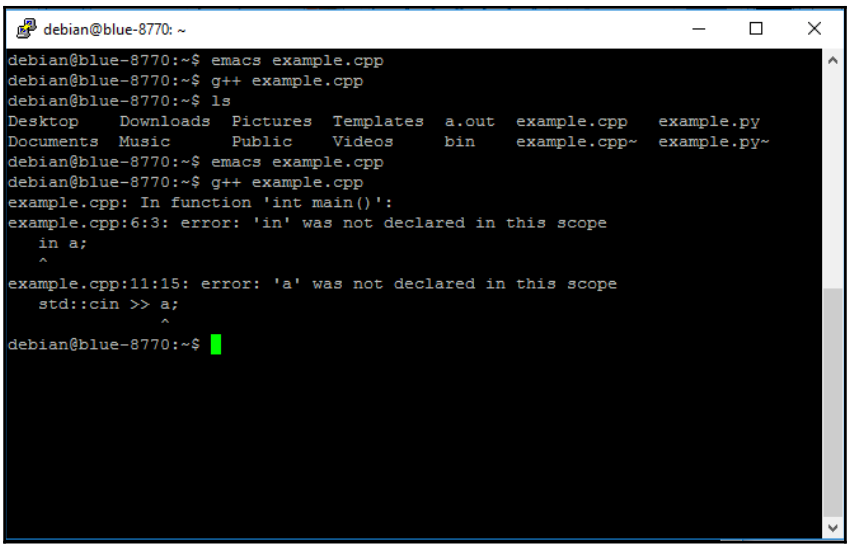
C is a compiled language. This means that you have to explicitly run a program that will translate these commands to the machine code the computer needs. To do this, after you have created the program, type `g++ example2.cpp`. This will then process your program, turning it into a file that the computer can execute. If you don't specify the name of the program that is created the system will use the default `a.out`.

You can specify the name of the executable using the `-o` option.

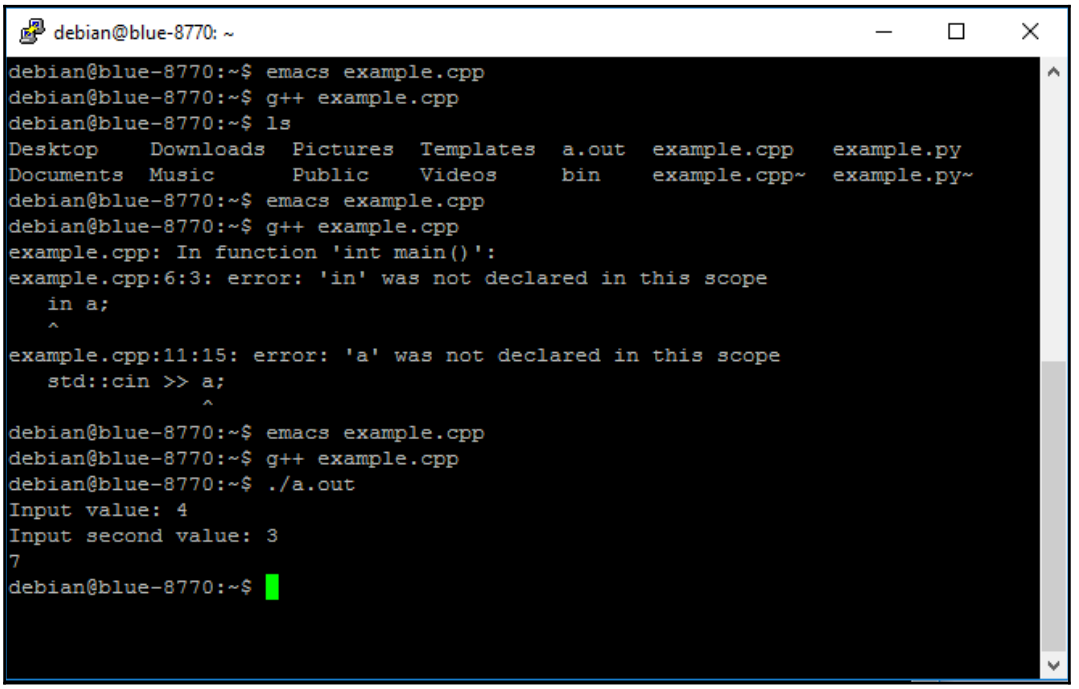
If you execute `ls` in your directory after you have compiled this, you should see the `a.out` file in your directory:



By the way, if you run into a problem, the compiler will try to help you figure out the problem. If, for example, you were to forget `t` before the `int a;` function, you would get the following error when you try to compile:



The error message indicates a problem in the int `main()` function and tells you that the variable `'a'` was not successfully declared. Once you have the file compiled, in order to run the executable, type `./a.out` and you should be able to create the following result:



There is not enough room to cover all the different aspects of the C language here; there are several good tutorials out 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 seemed fairly straightforward. However, if you have your functionality distributed between a lot of files, or need lots of libraries, the command-line approach to executing a compile can get unwieldy.

The C development environment provides a way to automate this process; it is called the make process. When using this, you create a text program named makefile that defines the files you want to include and compile, and 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. Here is a tutorial that talks more about this system: `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`, or `http://mrbook.org/blog/tutorials/make`.

Now you are equipped to edit and create your own programming files. The next chapters will provide you with lots of opportunity to practice your skills as you translate lines of code into cool robotic capabilities.

Summary

It is always a bit difficult to try new things. If this was your first attempt at programming, you may still feel a bit uncomfortable as I ask you to create or edit files. However, I will try to give you explicit instructions on what to type so that you can be successful. There is one major challenge when working with computers. They always do exactly what you tell them to do, not necessarily what you wanted them to do. So if you encounter problems, check several times to make sure that your code matches the example exactly. Now on to the next chapter and some actual coding! In the next chapter, you'll be building a simple wheeled platform that you can control with the BeagleBone Blue.

3

Making the Unit Mobile - Controlling Wheeled Movement

You can access your board now. You've learned some simple programming. Now you will add the capability to move the entire project. Perhaps the easiest way to make our projects mobile is to add a wheeled base. You'll do that in this chapter, where you will be introduced to some of the basics of controlling DC motors and using the BeagleBone Blue to control the speed and direction of your wheeled platform.

Even though you can access your board, you need to make it mobile to really call it a robot. In this chapter, you'll learn how to attach your board, both mechanically and electrically, to a board so that your projects can be mobile. Mobility; what could be more amazing than that?

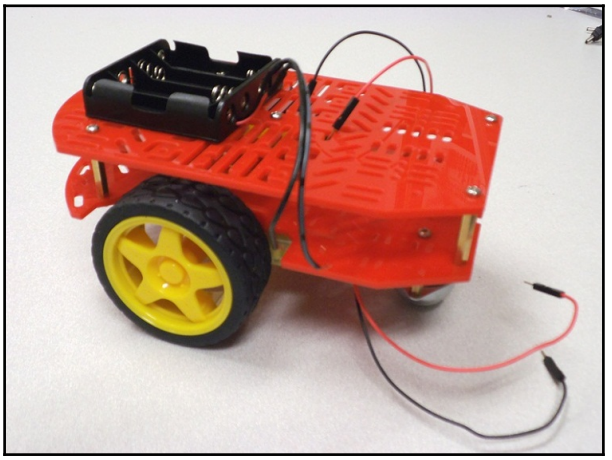
In this chapter, we will cover the following topics:

- Connecting the BeagleBone Blue directly to your wheeled platform
- Controlling your mobile platform programmatically using the BeagleBone Blue
- Adding a compass to enable simple path planning

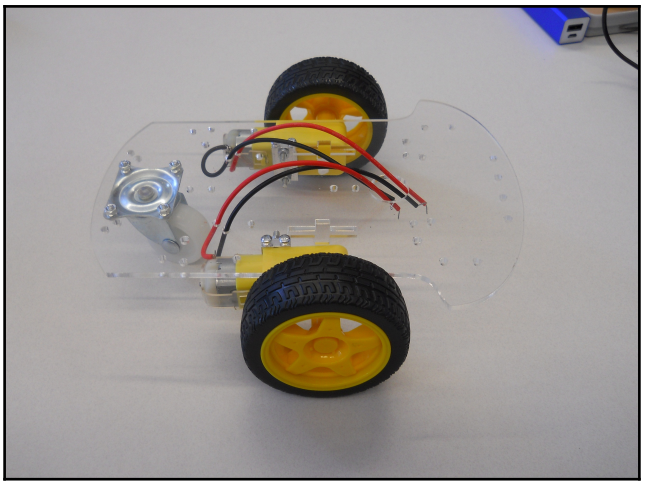
Getting started

You'll need to add some HW, specifically, a wheeled or tracked platform, to make your project mobile. There are a lot of choices. Some are completely assembled, others have some assembly required, and you may even choose to buy the components and construct your own custom mobile platform. Throughout this book, we're 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 can be assembled using simple tools (screwdriver and/or pliers).

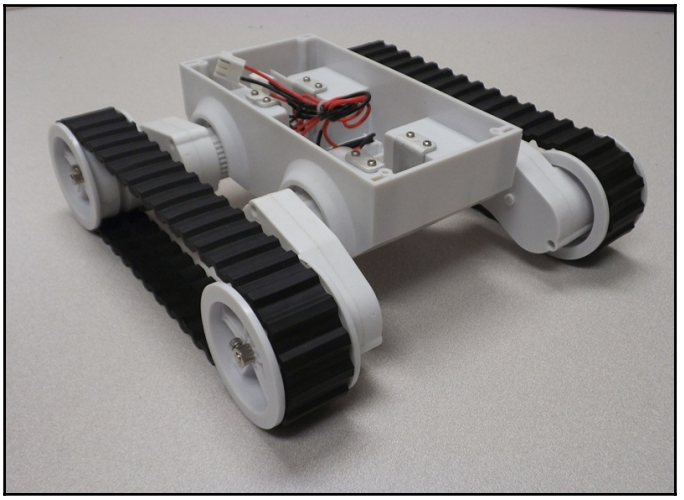
The easiest mobile platform is one that has two DC motors, each that controls a single wheel, with a small ball in the front or back. Here is a picture of one such platform, sold by *SparkFun*, at `www.sparkfun.com`, called the *Magician Chassis:*



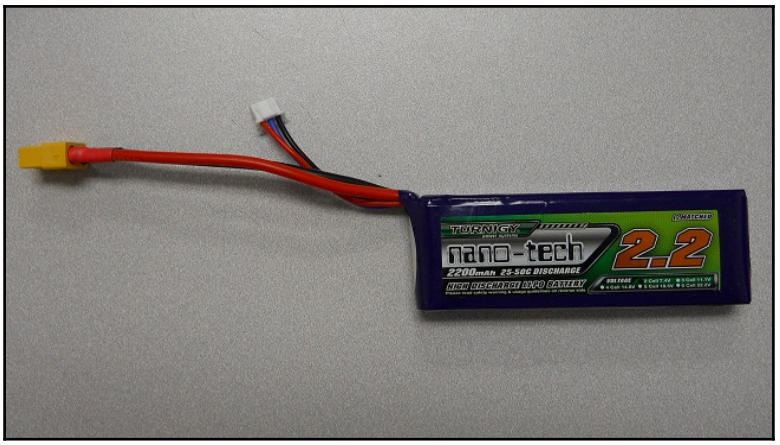
Fair notice: there is some assembly needed with this one, but it is fairly straightforward. There are some less expensive choices for two-wheeled, DC motor platforms, for example, this platform:



It is available on both `amazon.com` and `ebay.com`. You could also choose a tracked platform instead of a wheeled platform. Again, there are manufacturers that make already assembled units. Here is a picture of one such unit, made by *Dagu* called the *Dagu Rover 5 Tracked Chassis*:

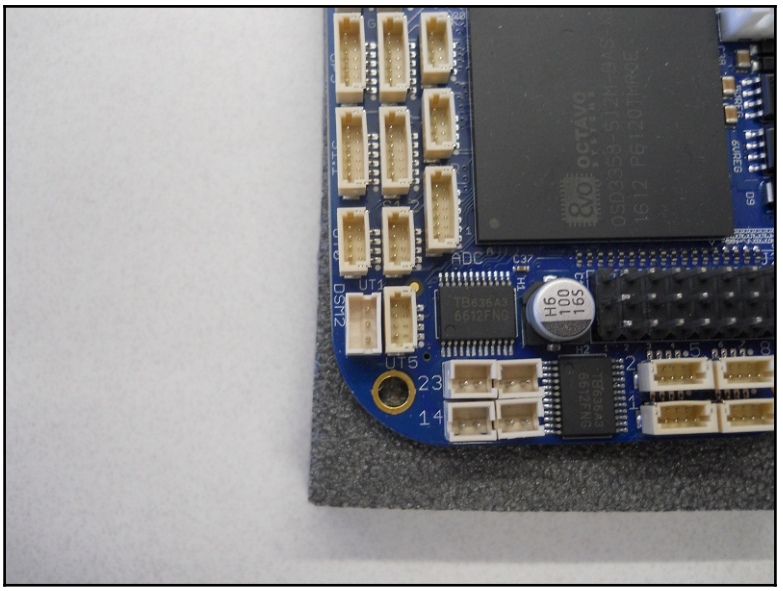


You simply have to choose a platform that can move using two DC motors and all of what you are shown here will work. Since you have a mobile platform, you'll need a mobile power supply for the BeagleBone Blue. The best choice is going to be a 2S LiPo battery, such as the ones used on many RC vehicles. Here is a picture of one such item:

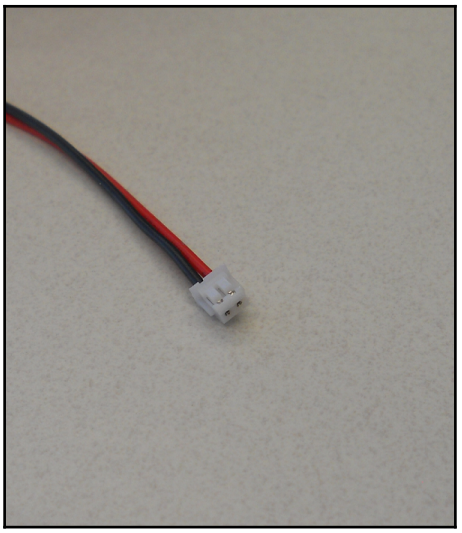


These are available at almost all online electronics stores or RC toy stores. You'll need to connect your battery to the BeagleBone Blue, but the standard connection on a 2S LiPo battery will connect directly to the BeagleBone Blue at the connection marked LiPo.

The final connection you'll need for your basic mobile platform is to connect your DC motors to the BeagleBone Blue so that you can control the speed of the motors. Fortunately, the BeagleBone Blue comes with connectors that are designed directly for that purpose. These are the 2314 2-pin connectors that are at the opposite end of the board as the host USB connector, shown here:



But you'll need some connectors to connect to the DC motors so you can plug them into the board. These are two-pin JST ZH connectors, and here is a picture:



You'll need two of these. Now that you have all the HW, let's walk through a quick tutorial of how the system works and then some step-by-step instructions on how to make your project mobile.

Controlling your mobile platform programmatically using the BeagleBone Blue

Now that you have your basic motor controller functionality up and running, you need to connect both motor controllers to the BeagleBone black. In this section, we will cover this and then show you how to control your entire platform programmatically.

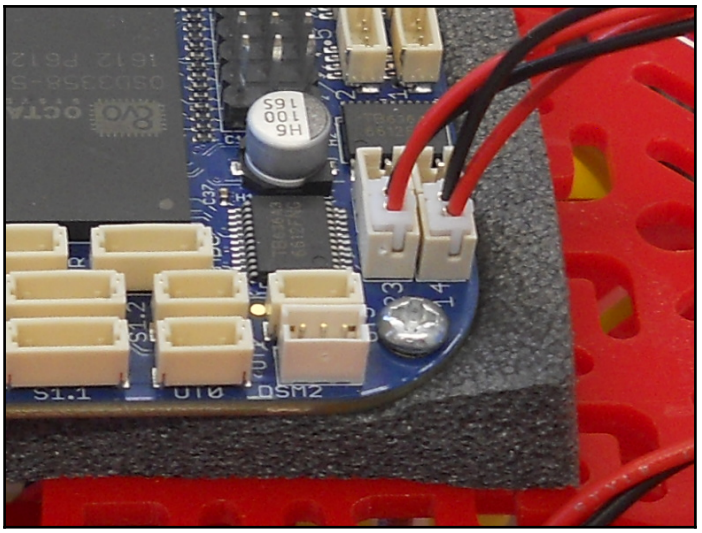
Before you get started connecting everything and making it all move, let's spend some time understanding some of the basics of motor control. Whether you chose the two-wheeled mobile platform or the tracked platform, the basic movement control is the same. The unit moves by engaging the motors. If the desired direction is straight, the motors are run at the same speed. If you want to turn the unit, the motors are run at different speeds. The unit can actually turn in a circle if you run one motor forward and one backward.

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

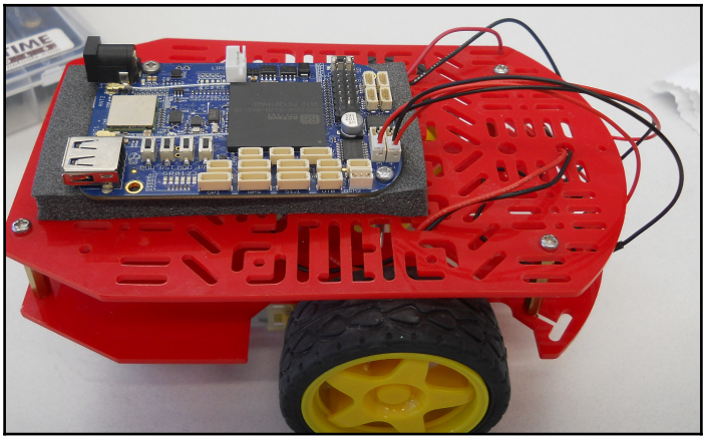
The magnitude and polarity of the voltage are not the only factors that are important when you think about controlling your motors, however. The power that your motor can apply to moving your platform is also determined by the voltage and the current supplied at its terminals.

Connecting the DC motors to the BeagleBone Blue

Fortunately, the BeagleBone Blue has built-in motor controllers. These motor controllers will get their power through the LiPo battery connector. All you'll need to do is connect the DC motors to the board via the motor connectors. So connect the two-pin JST ZH connectors to the red and black wires coming from the DC motors on your platform. Then, plug the connectors into the board, as follows:



Once you've made these connections, you can configure all of the HW on top of the mobile platform, perhaps like this:



Now you are ready to talk to your DC motors.

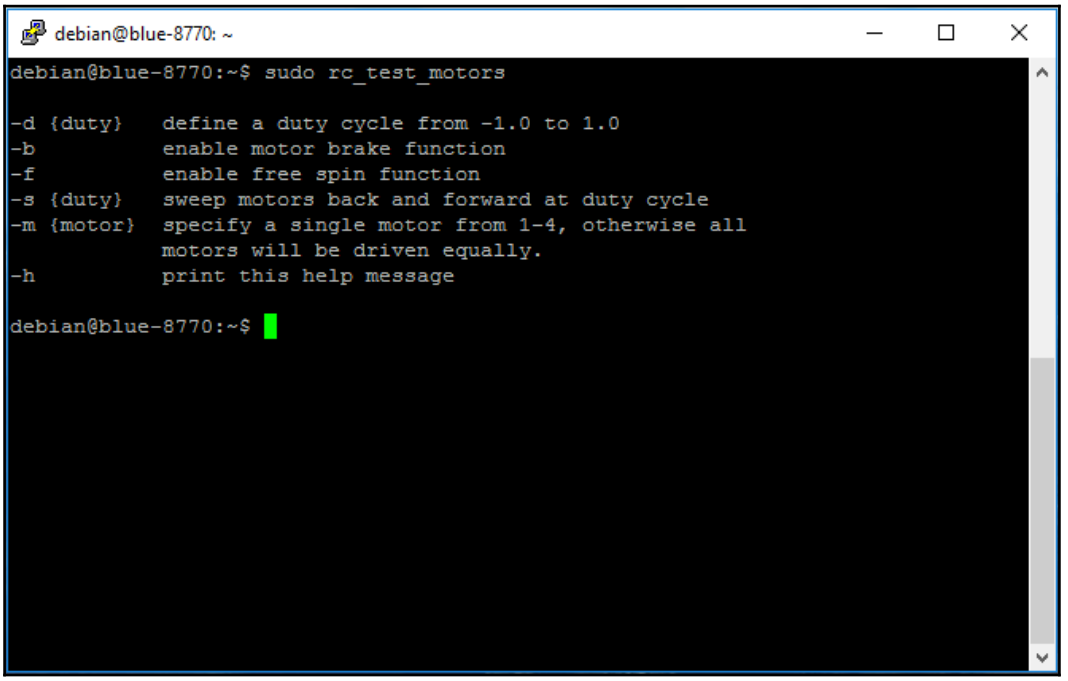
Controlling the DC motors programmatically

In order to talk to your DC motors, you'll need some software. Fortunately, this software is already installed on the BeagleBone Blue. Go to the `/root` directory and look for the `Robotics_Cape_Installer-master` directory. If you get a message of permission denied, then change to root privileges by typing `sudo -s`, and then you'll be able to access this directory.

If the software is not installed on your BeagleBone Blue, simply type `sudo apt-get install roboticscape` or go to `http://strawsondesign.com/#!manual-install` and look for the instructions on how to manually download the software.

You will use the software to engage with almost all of the hardware capabilities of the BeagleBone Blue, so let's start with how to program to control the DC motors.

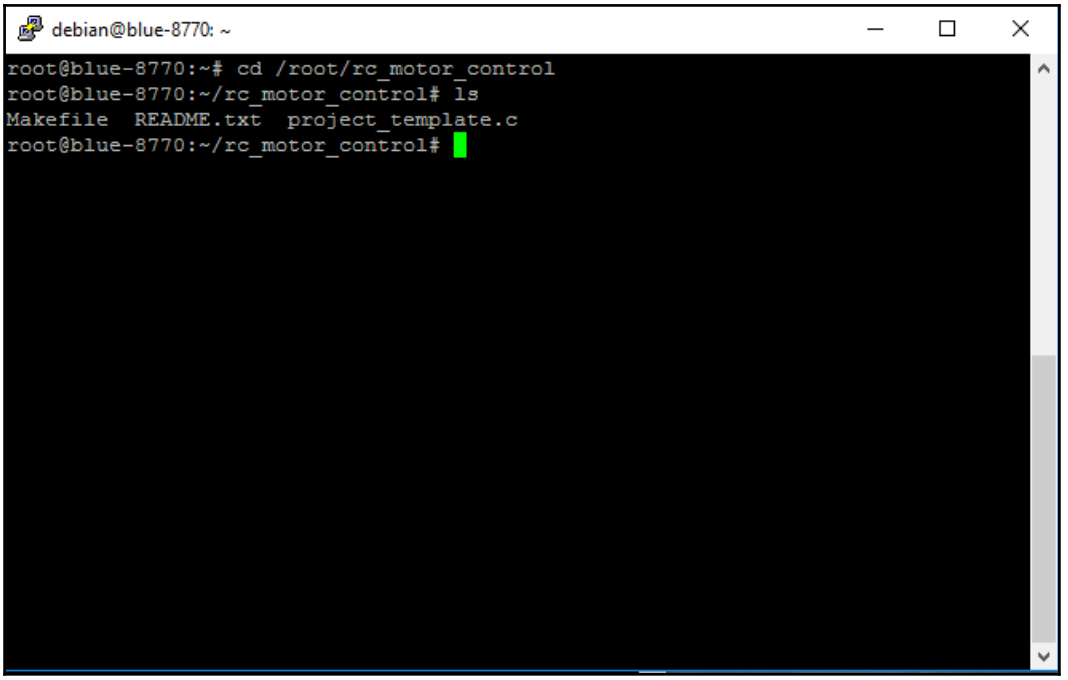
First, let's check to make sure that your motors are connected correctly by running a test program. Type `sudo rc_test_motors` and you should see the following:



Now you can test the motors by typing `sudo rc_test_motors -f -d 0.5` and this should result in both motors engaging forward at half speed. To stop the program, and thus stop the motors, press *Ctrl* + *C*.

Now that you know that your motors can be accessed by the BeagleBone Blue, you'll want to create your own program that can use the keyboard to control the speed and direction of your wheeled platform. To do this, you'll take some of the functionality of the `rc_test_motors` program and add the capability to input a key press and then control the motors.

To start the development process, you'll want to create your own project using some of the code from `rc_test_motors.c`. But first, you'll want to copy the project template into a new project directory by typing `cp -r /root/Robotics_Cape_Installer-master/project_template /root/rc_motor_control` and hitting Enter. Now you should have a new directory and a file to edit. So go to the `/root/rc_motor_control` directory. You should see this:



Now rename the source file by typing `mv project_template.c rc_motor_control.c` and hit Enter. Edit the `Makefile` file and change the `TARGET` variable to `TARGET = rc_motor_control` and then save the file.

Here is the source code for the file:

```
 /*********************************************************************
*********
    * rc_motor_control.c
    *
    * This is a simple dc motor control program. It takes in a
    character
    * and then controls the motors to move forward, reverse, left or
    right
*******************************************************************************
***/
    #include <rc_usefulincludes.h>
    #include <roboticscape.h>
 /*********************************************************************
 *********
    * int main()
    *
    * This main function contains these critical components
    * - call to initialize_cape
    * - main while loop that checks for EXITING condition
    * - switch statement to send proper controls to the motors
    * - cleanup_roboticscape() at the end
 ********************************************************************
 ********/
    int main(){
    char input;
    // always initialize cape library first
    rc_initialize();
    printf("nHello BeagleBonen");
    // done initializing so set state to RUNNING
    rc_set_state(RUNNING);
    // bring H-bridges of of standby
    rc_enable_motors();
    rc_set_led(GREEN,ON);
    rc_set_led(RED,ON);
    rc_set_motor_free_spin(1);
    rc_set_motor_free_spin(2);
    printf("Motors are now ready.n");
    // Turn on a raw terminal to get a single charactersystem("stty
    raw");
    do
    {
    printf("> ");
```

```
input = getchar();
switch(input){
case 'f':
rc_set_motor(1, 0.5);
rc_set_motor(2, 0.5);
break;
case 'r':
rc_set_motor(1, 0.5);
rc_set_motor(2, -0.5);
break;case 'l':
rc_set_motor(1, -0.5);
rc_set_motor(2, 0.5);
break;
case 'b':
rc_set_motor(1, -0.5);
rc_set_motor(2, -0.5);
break;
case 's':
rc_set_motor_brake_all();
break;
case 'q':
rc_disable_motors();
break;
default:
printf("Invalid Character.n");
}
}
while(input != 'q');
printf("Donen");
rc_cleanup();
system("stty cooked");
return 0;
}
```

This code is straightforward, but here are some of the specifics. In the first section, this is the code:

```
#include <rc_usefulincludes.h>
#include <roboticscape.h>
```

These two commands include the libraries you'll need to access the DC motor capability of the BeagleBone Blue.

In the top of `main()` function there is an initialization section. Here are some specifics on what these commands do:

```
rc_initialize();
```

This command initializes the hardware interfaces of the BeagleBone Blue:

```
rc_set_state(RUNNING);
```

This command sets the state of the hardware interfaces of the BeagleBone Blue to running:

```
rc_enable_motors();
```

This command enables the H-bridges for the DC motors so that the DC motor connectors are now active:

```
rc_set_led(GREEN,ON);
rc_set_led(RED,ON);
```

These commands turn on the green and red LEDs. This is not required, but it helps you know that the program is active and running and the robot is ready to move:

```
rc_set_motor_free_spin(1);
rc_set_motor_free_spin(2);
```

These commands set the control signals so that they are capable of driving the motors:

```
system("stty raw");
```

This sets the input mode so that you don't have to press the Enter key after each character input. Now that the program has initialized everything, it will enter a do-while loop and switch the statement so that each key press will come in, be interpreted, and then some resulting action will be sent to the DC motors:

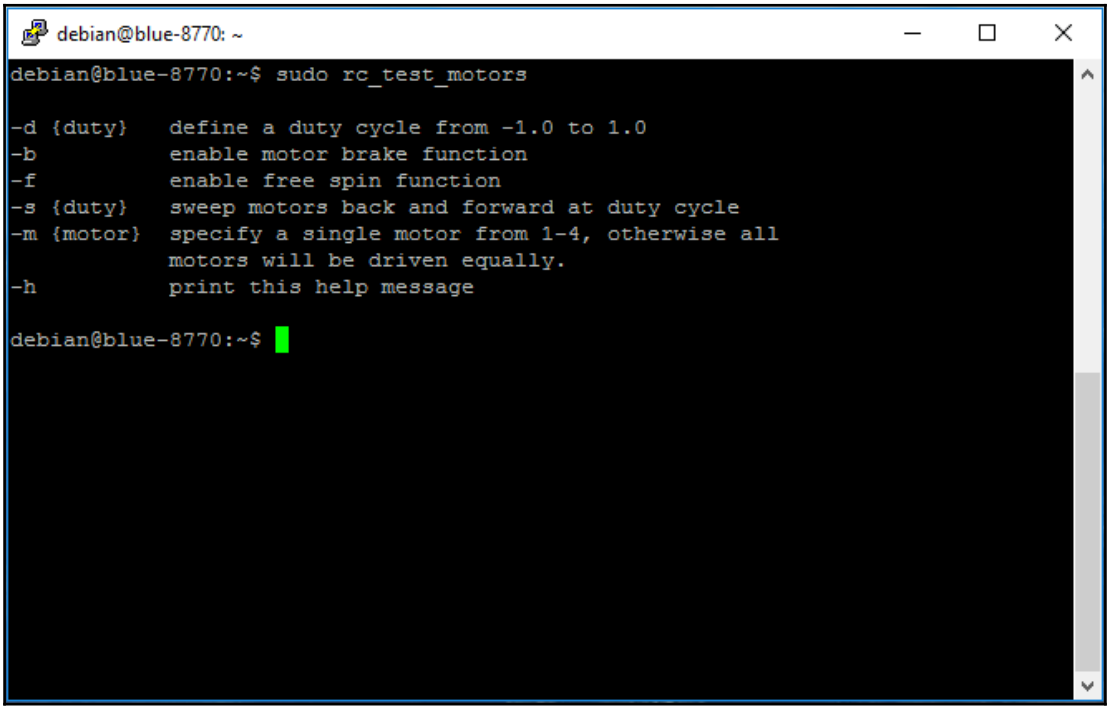
```
rc_set_motor(1, 0.5);
```

This command, used throughout the switch statement, moves an individual motor. The first argument of this statement determines which motor will move, and the second argument determines the speed at which it will move:

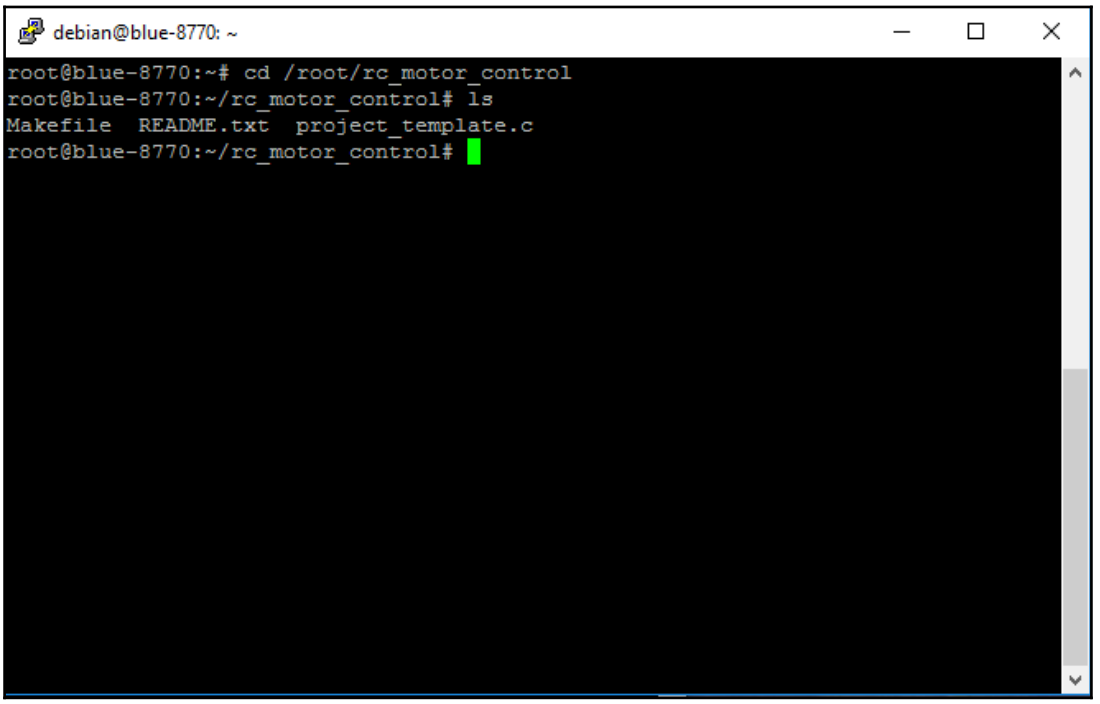
```
rc_cleanup();
system("stty cooked");
```

These last two commands reset the system; the first one resets the BeagleBone Blue's hardware interfaces and the second command resets the terminal so that it will require an Enter key when entering data.

After you have entered the program using your favorite editor, you'll need to compile it. First, type `make clean` as this will establish the need to compile your program. Then, type `make` and you should see something like this:



Once the program is compiled, you can run the program by typing `./rc_motor_control`; you will then see something like this:



Typing the character `f` should make your robot go forward, typing the character `b` should make it go in reverse, typing the character `r` should make it rotate to the right, typing the character `l` should make it rotate to the left. Typing the character `s` should make it stop, and typing the character `q` should stop the program.

Now your platform is mobile and you can control it from your remote computer.

Accessing the compass on the BeagleBone Blue

Your platform can move around, but if you are going to do some automated movement, you're going to want some of the additional information that the BeagleBone Blue can provide. This will come in the form of a compass direction that the built-in IMU can measure and return programmatically.

In order to access this information, you'll first need to run two different calibration programs. The first is `rc_calibrate_mag` and the second is `rc_calibrate_gyro`. When you run `rc_calibrate_mag`, it will direct you to move your platform around in as many ways as possible for 15 seconds. It will then use this date to calibrate the magnetic fields so that it can give a valid compass reading. The `rc_calibrate_gyro` program will also calibrate the gyros.

Then, you can try the test program for the compass by running the `rc_test_imu` program. This will show you readings from the gyros and the compass and verify that it is working correctly.

Once your compass is calibrated and tested, you'll want to write your very own program to control the wheeled platform. Create the program, as you did the `rc_motor_control.c` program, by creating a new directory `rc_wheeled_auto` and copying the project template files into the directory. You'll want to edit `Makefile` to set `TARGET = rc_wheeled_auto`. Then, you'll want to create the `rc_wheeled_auto.c` file with the following code:

```
/*******************************************************************
*********
   * rc_wheeled_auto.c
   *
   * This is a program that uses the compass to turn the wheeled
   * platform and then go a certain distance.
*******************************************************************
********/
   #include <rc_usefulincludes.h>
   #include <roboticscape.h>
   //struct to hold new data
   rc_imu_data_t data;
   void process_data();
   double angle;
   int distance;
   int turn;
/*******************************************************************
*********
   * int main()
   *
   * This main function contains these critical components
   * - call to initialize_cape
   * - set up the compass
   * - initiate the turn
   * - after it comes back - go a certain distance
   * - cleanup_roboticscape() at the end
*******************************************************************
*******/
   int main(int argc, char** argv){
```

```
// always initialize cape library first
rc_initialize();
printf("nHello BeagleBonen");
angle = atof(argv[1]);
if (angle > 0)
turn = 1;
else
turn = 0;
distance = atoi(argv[2]);
// done initializing so set state to RUNNING
rc_set_state(RUNNING);
// bring H-bridges of of standby
rc_enable_motors();
rc_set_led(GREEN,ON);
rc_set_led(RED,ON);
rc_set_motor_free_spin(1);
rc_set_motor_free_spin(2);
printf("Motors are now ready.n");
// start with default config and modify based on options
rc_imu_config_t conf = rc_default_imu_config();
conf.dmp_sample_rate = 20;
conf.enable_magnetometer = 1;
// now set up the imu for dmp interrupt operation
if(rc_initialize_imu_dmp(&data, conf)){
printf("rc_initialize_imu_failedn");
return -1;
}
rc_set_imu_interrupt_func(&process_data);
// set the unit turning
if (turn)
{
rc_set_motor(1, 0.2);
rc_set_motor(2, -0.2);
}
else
{
rc_set_motor(1, -0.2);
rc_set_motor(2, 0.2);
}
//now just wait, print_data() will be called by the interrupt
while (rc_get_state()!=EXITING) {
usleep(10000);
}
int movement = 0;
// Now move forward
while (movement < distance)
{
rc_set_motor(1, 0.2);
```

```
    rc_set_motor(2, 0.2);
    usleep(1000000);
    movement++;
    }
    rc_set_motor_brake_all();
    // shut things down
    rc_power_off_imu();
    rc_cleanup();
    return 0;
    }
/**************************************************************************
*********
    * int process_data()
    *
    * - Called each time the compass interrupts
    * - Compares angles to see if the platform has moved enough
    * - If it has, stop the platform
**************************************************************************
*******/
    void process_data() // imu interrupt function
    {
    printf("r");
    printf(" ");
    printf("Angle = %6.1fn",angle);
    printf("Distance = %2dn",distance);
    printf(" %6.1f |", data.compass_heading_raw*RAD_TO_DEG);
    printf(" %6.1f |", data.compass_heading*RAD_TO_DEG);
    if (turn)
    {
    if ((angle - data.compass_heading*RAD_TO_DEG) < 1.0)
    {
    rc_set_motor_brake_all();
    rc_set_state(EXITING);
    }
    }
    else
    if ((-angle + data.compass_heading*RAD_TO_DEG) < 1.0)
    {
    rc_set_motor_brake_all();
    rc_set_state(EXITING);
    }
    fflush(stdout);
    return;
    }
    Let's look at some of the key sections of this code:
    //struct to hold new data
    rc_imu_data_t data;
    void process_data();
```

```
double angle;
int distance;
int turn;
```

This section of the code creates the data structures required by your program. The `rc_imu_data_t data` is a data structure that holds the data that you'll be receiving back from the IMU. The `process_data()` function will be the function that is called each time the IMU interrupts with the data. The angle, distance, and turn variables hold the details from the user about the desired behavior.

Take a look at these lines:

```
angle = atof(argv[1]);
if (angle > 0)
turn = 1;
else
turn = 0;
distance = atoi(argv[2]);
```

Take user input in the form of command-line arguments and set the angle, turn, and distance variables that you'll later use to control the robot. At the top of the main function are many of the same initializers as you used in the `rc_motor_control` program. Next come the lines:

```
// start with default config and modify based on options
rc_imu_config_t conf = rc_default_imu_config();
conf.dmp_sample_rate = 20;
conf.enable_magnetometer = 1;
// now set up the imu for dmp interrupt operation
if(rc_initialize_imu_dmp(&data, conf)){
printf("rc_initialize_imu_failedn");
return -1;
}
rc_set_imu_interrupt_func(&process_data);
```

These lines establish the configuration of the **IMU**. In this case, we set the sample rate to `20`, enable the magnetometer, and set up the system to call the `process_data()` function when the IMU interrupts the system with valid data. When everything is configured, the following lines start the unit turning:

```
// set the unit turning
if (turn)
{
rc_set_motor(1, 0.2);
rc_set_motor(2, -0.2);
}
else
```

```
{
rc_set_motor(1, -0.2);
rc_set_motor(2, 0.2);
}
```

After these lines are executed, the unit goes into a `while` loop:

```
while (rc_get_state()!=EXITING) {
usleep(10000);
}
```

These lines put the system in a while loop, mostly waiting in a paused state, ready to execute the `process_data()` function on each interruption from the enabled IMU. This loop is exited when `rc_get_state != EXITING` condition is true, when this condition is met you will then execute the `process_data()` function. This will occur when you reach the desired angle. Then, the following lines will be executed:

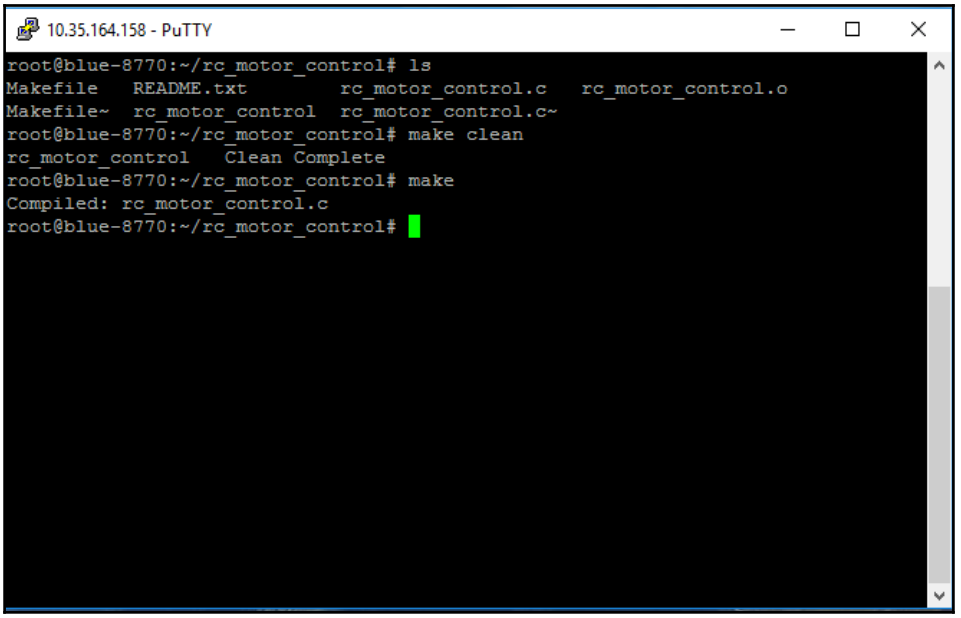
```
int movement = 0;
// Now move forward
while (movement < distance)
{
rc_set_motor(1, 0.2);
rc_set_motor(2, 0.2);
usleep(1000000);
movement++;
}
```

When the program executes these lines, the unit will move forward the distance variable in seconds. The final lines of the code stop the motor and prepare the system for the program to stop the execution. The `process_data()` function has the following key lines of code:

```
if (turn)
{
if ((angle - data.compass_heading*RAD_TO_DEG) < 1.0)
{
rc_set_motor_brake_all();
rc_set_state(EXITING);
}
}
else
if ((-angle + data.compass_heading*RAD_TO_DEG) < 1.0)
{
rc_set_motor_brake_all();
rc_set_state(EXITING);
}
fflush(stdout);
```

The code first uses turns to decide which direction the platform is moving in and then monitors the angle until it has reached the target. When it reaches the target, the code stops the motors and sets the `EXITING` state.

Compile the code by first typing `make clean`, the `make` command. This will create the `rc_wheeled_auto` executable. When you are ready to run the code, you'll type something like `./rc_wheeled_auto 30 2` and the platform should turn `30` degrees and then go for 2 seconds. Here is what you will see on the screen:



Now you can run the program and get the wheeled platform to go to a new position at the new angle and distance!

Summary

Now you have your mobile platform up and ready to move around. But, inevitably, when you move around you are going to encounter barriers. You'll want to sense those barriers. In the next chapter, you'll add some sensors so that you can avoid running into obstacles.

4
Avoiding Obstacles Using Sensors

In the previous chapters, you learned wheeled or tracked movement and then used the compass to move in a different specific direction. Now your robot can be manipulated through space. But what if you want your robot to sense the outside world, so you don't run into things? In this chapter, you'll discover how to add some sensors to help us avoid barriers.

In this chapter, we will cover the following the following topics:

- Connecting the BeagleBone Blue to a sonar sensor through an Arduino so that it can learn more about the world around it
- Showing how to use three sensors to give your mobile robot a robust view of the world around it
- Exploring some simple path planning algorithms that take into account obstacle avoidance

Different types of sensors used

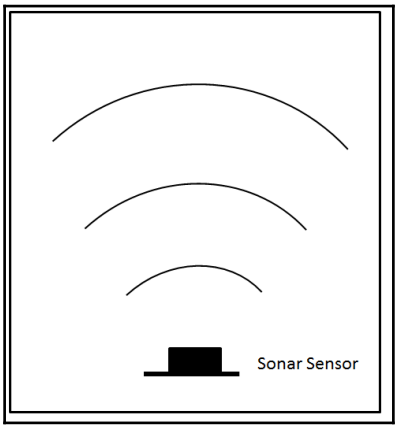
In this chapter, you'll need some sensors. There are three choices:

- Sonar
- LiDAR
- Infrared

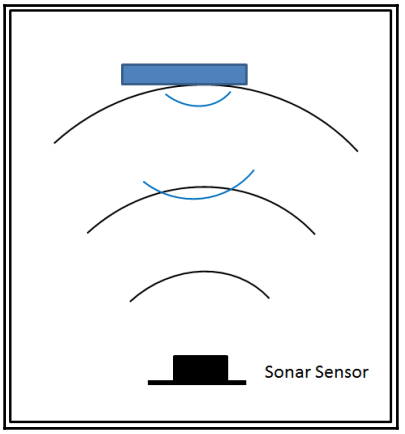
Let's take a brief look at each.

The sonar sensor

This type of sensor uses ultrasonic sound to calculate the distance to an object. The sensor consists of a transmitter and receiver. The transmitter creates a sound wave that travels out from the sensor, as illustrated in the following figure:



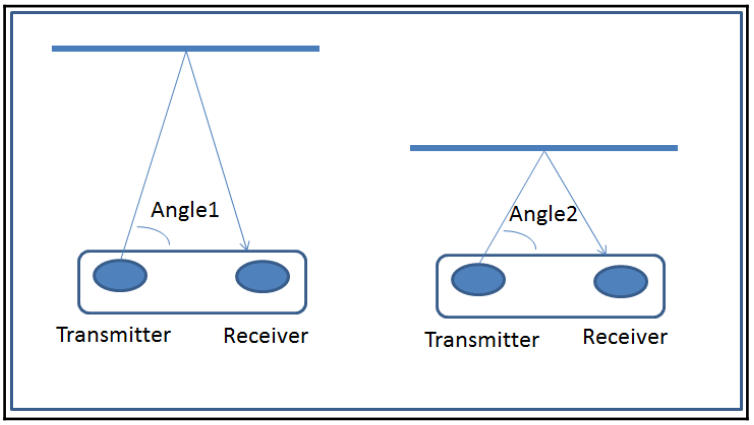
The device sends out a sound wave 10 times a second. If an object is in the path of these waves, then the waves reflect off the object. Then, this returns sound waves to the sensor, as shown in the following figure:



The sensor then measures any returning sound waves. It uses the time difference between when the sound wave was sent out and when it returns to measure the distance to the object. The advantages of the sonar sensor are that it is simple, inexpensive, and works well in all types of lighting. The disadvantages of this sensor are that it doesn't work well at longer distances and is not as accurate as other types of sensors.

The infrared sensor

Another type of distance sensor is a sensor that uses **Infrared** (**IR**) signals. The IR sensor also uses both a transmitter and a receiver. The transmitter transmits 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 figure:



The different angles give you an indication of the distance to 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. The advantages of the infrared sensors are that they are relatively inexpensive and easy to interface. However, they don't work well in situations with lots of light.

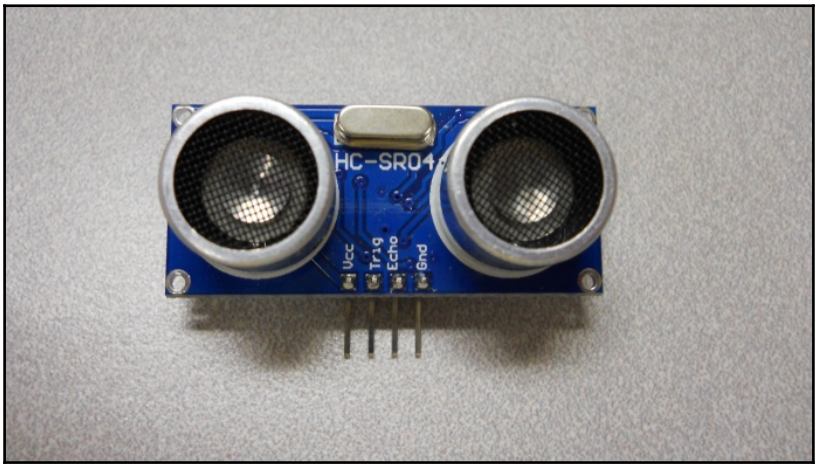
The LiDAR sensor

The LiDAR sensor is similar to the sonar sensor, but instead of sound waves, it sends out a pulse of very focused laser light. It uses a receiver to measure the return and also calculates the time of return to determine the distance traveled. Its advantages are that it is very accurate and also very good at long distances. Its disadvantage is that it is the most expensive of all the distance sensors.

Based on the advantages and disadvantages and also considering that you might want to use an array of distance sensors, for this project, you'll be constructing an array of distance sensors using a relatively inexpensive sonar sensor.

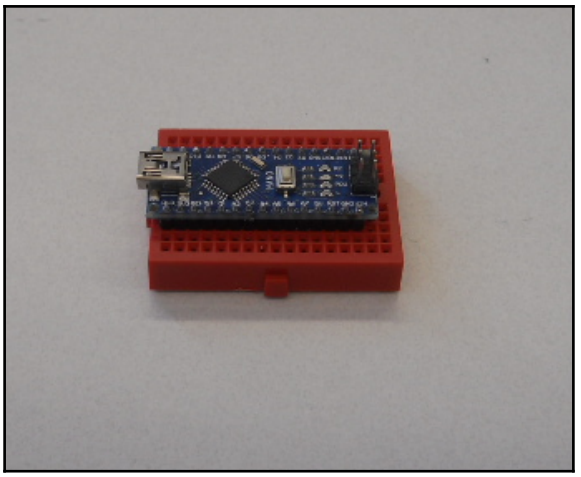
Connecting a sonar sensor to an Arduino

Here is a picture of a sonar sensor, the HC-SR04, an extremely inexpensive sensor that works well in these sorts of situations:



These sonar sensors are available at most online electronics stores, including `amazon.com`. There are ways to connect this sensor directly to the BeagleBone Blue; however, there are two disadvantages to this approach. First, you'd need to learn the **Programmable Real-Time Unit** (**PRU**) process for BeagleBone Blue development. These sensors need specifically timed pulses, and the PRU system would allow you to create them. Also, since you're going to use multiple sensors to look around your mobile platform, you would need a significant number of **General-Purpose Input/Output** (**GPIO**) connectors.

For these reasons, you're going to a use a simple and inexpensive processor, an Arduino, to connect to the sensors and then you'll communicate with that processor via a serial interface. There are several different choices of Arduinos, but one of the smallest and least expensive is the Arduino Nano. Here is a picture of this product:



You'll notice that it comes ready to install on a small breadboard that will make it easy to connect to the sonar sensors. So first, let's connect the Arduino and sensor, and then you'll write a program to communicate via the USB connection on the BeagleBone Blue.

In order to connect this sonar sensor to your Arduino, you'll need some female-to-male jumper cables. You'll notice that there are four pins to connect to the sonar sensor. Two of these supply the voltage and current to the servo. One pin, the trig pin, triggers the sensor to send out a sound wave. The echo pin then senses the return from this echo.

To access the sensor with the Arduino, make the following connections using the male to female jumper wires:

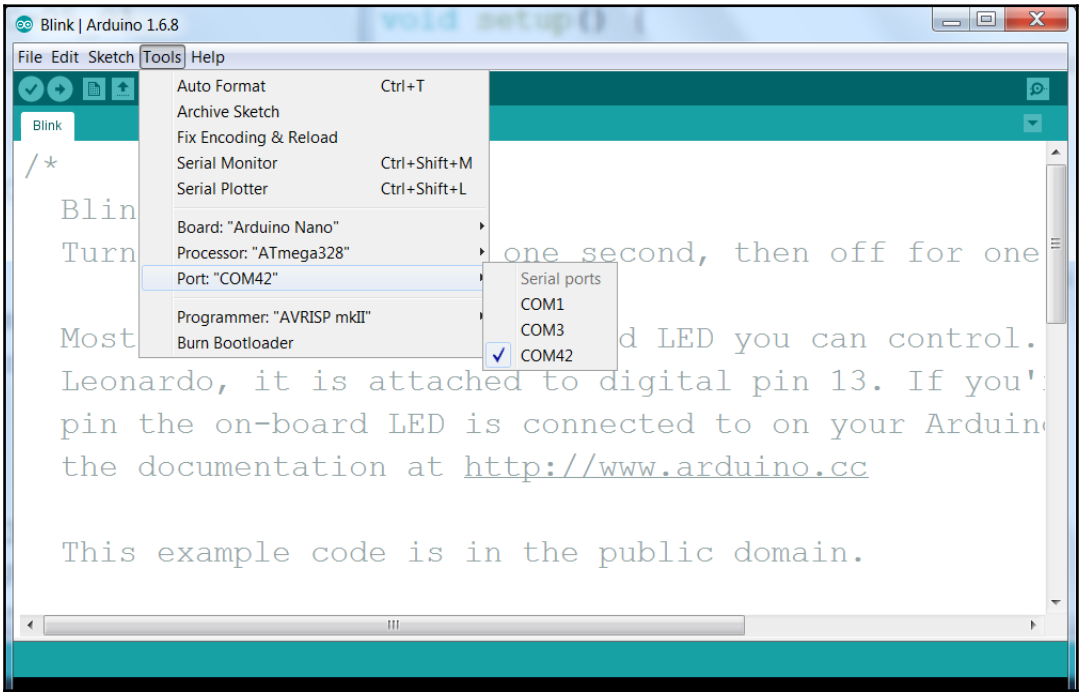
Arduino pin	**Sensor pin**
5V	Vcc
GND	Gnd
12	Trig
11	Echo

Accessing the sonar sensor from the Arduino IDE

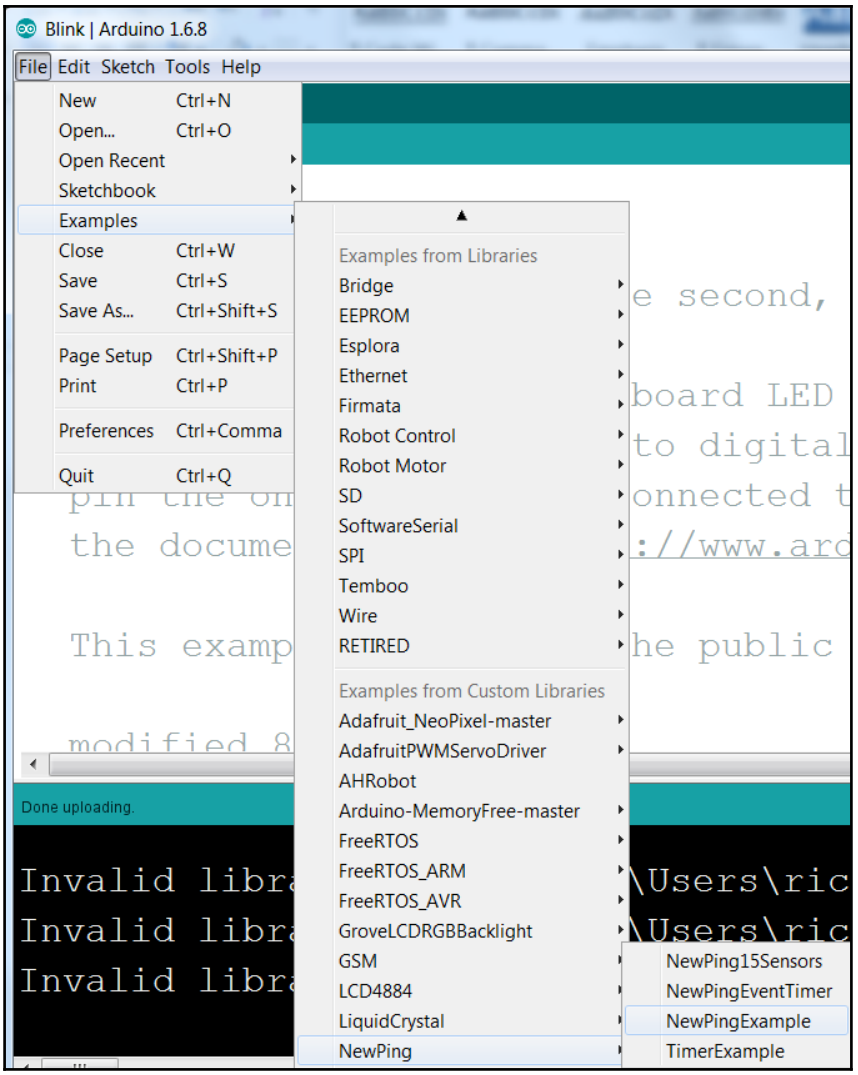
Now that the HW is connected, you'll need to download and install the Arduino IDE from `arduino.cc`. This simple IDE allows you to quickly program the Arduino.

If you are unfamiliar with the Arduino, you can learn a great deal using the `arduino.cc` website. There are also many YouTube videos that can give you information about the product.

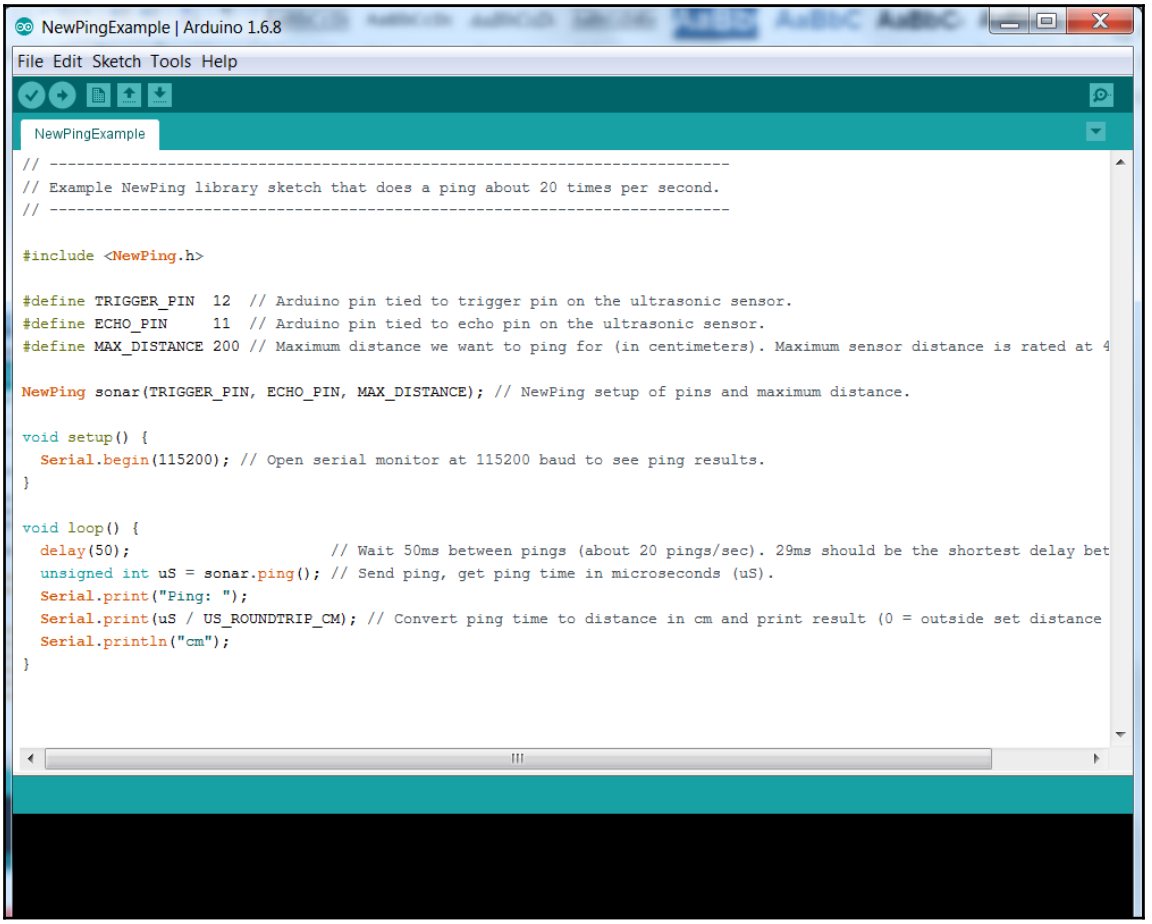
After you have downloaded and installed the Arduino IDE, you'll run the IDE and select the nano processor by navigating to **Tools** | **Boards** | **Arduino Nano**. You'll also need to set the processors by navigating to **Tools** | **Processor** | **ATmega328**. Finally, select your serial connection to the Arduino by navigating to **Tools** | **Ports** | **COM42** , as shown in the following screenshot:



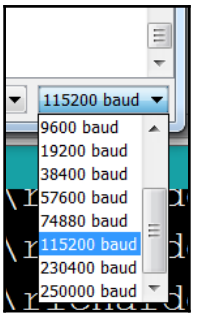
You'll also need to download a library that supports this sensor. One of the better libraries for this sensor is available at `https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home`. Download the `NewPing` library and then open the Arduino IDE. You can then include the library in the IDE by navigating to **Sketch** | **Import Library** | **Add Library** and then going to the download directory and selecting the `NewPing.zip` file. Once you have the library installed, you can access the example program by going to **File** | **Examples** | **NewPing** | **NewPingExample**, as shown in the following screenshot:


Blink | Arduino 1.6.8
File Edit Sketch Tools Help
New Ctrl+N
Open... Ctrl+O
Open Recent
Sketchbook
Examples
Close Ctrl+W
Save Ctrl+S
Save As... Ctrl+Shift+S
Page Setup Ctrl+Shift+P
Print Ctrl+P
Preferences Ctrl+Comma
Quit Ctrl+Q
Examples from Libraries
Bridge
EEPROM
Esplora
Ethernet
Firmata
Robot Control
Robot Motor
SD
SoftwareSerial
SPI
Temboo
Wire
RETIRED
Examples from Custom Libraries
Adafruit_NeoPixel-master
AdafruitPWMServoDriver
AHRobot
Arduino-MemoryFree-master
FreeRTOS
FreeRTOS_ARM
FreeRTOS_AVR
GroveLCDRGBBacklight
GSM
LCD4884
LiquidCrystal
NewPing
NewPing15Sensors
NewPingEventTimer
NewPingExample
TimerExample
Done uploading.
Invalid libra
Invalid libra
Invalid libra

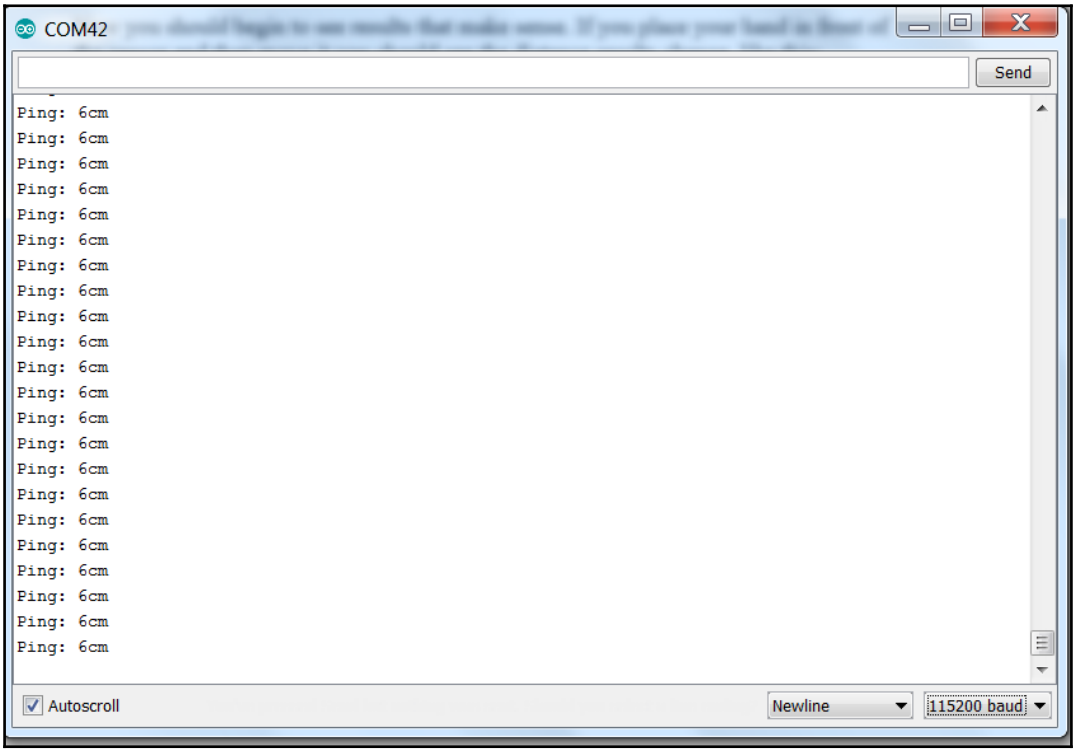
You'll then see the following code in the IDE:



You should then upload the code to the Arduino and open a serial terminal by navigating to **Tools** | **Serial Monitor** in the IDE. Initially, you'll see characters that make no sense, you need to change the serial port baud rate by selecting this field in the lower-right corner of the **Serial Monitor**, as shown in the following screenshot:



Now you should begin to see results that make sense. If you place your hand in front of the sensor and then move it, you should see the distance results change, as shown in the following screenshot:



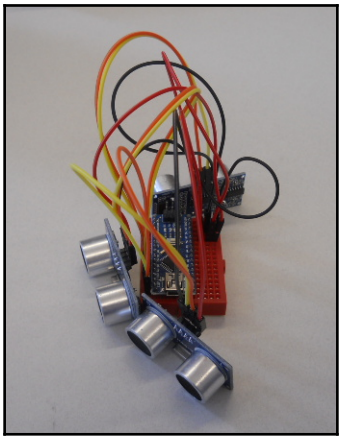
You can now measure the distance to an object using your sonar sensor using the Arduino. The next step is to connect three sensors to the Arduino.

Creating an array of sensors

To do this, connect the additional pins, as follows:

Arduino pin	**Sensor pin**
5V	Sensor 1 Vcc
GND	Sensor 1 GND
12	Sensor 1 Trig
11	Sensor 1 Echo
5V	Sensor 2 Vcc
GND	Sensor 2 GND
10	Sensor 2 Trig
9	Sensor 2 Echo
5V	Sensor 3 Vcc
GND	Sensor 3 GND
8	Sensor 3 Trig
7	Sensor 3 Echo

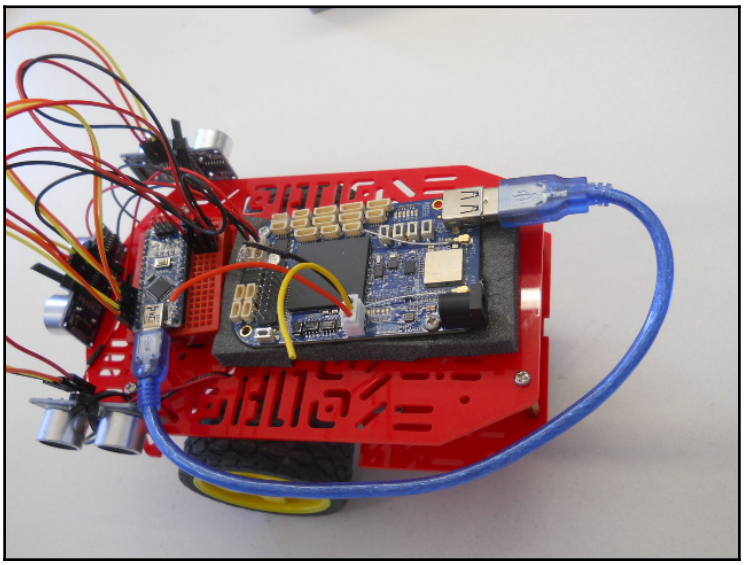
Here is a picture of the three sensors assembled:



Now that the sensors are connected, you'll need to add some functionality to the Arduino code to communicate with all three sensors. Here is that Arduino code:

```
#include <NewPing.h>
#define TRIGGER_PIN1 12
#define ECHO_PIN1 11
#define TRIGGER_PIN2 10#define ECHO_PIN2 9
#define TRIGGER_PIN3 8
#define ECHO_PIN3 7
#define MAX_DISTANCE 200
NewPing sonar1(TRIGGER_PIN1, ECHO_PIN1, MAX_DISTANCE);
NewPing sonar2(TRIGGER_PIN2, ECHO_PIN2, MAX_DISTANCE);
NewPing sonar3(TRIGGER_PIN3, ECHO_PIN3, MAX_DISTANCE);
void setup() {
  Serial.begin(115200);
}
void loop() {
  delay(500);
  unsigned int uS1 = sonar1.ping();
  unsigned int uS2 = sonar2.ping();
  unsigned int uS3 = sonar3.ping();
  Serial.print(uS1 / US_ROUNDTRIP_CM);
  Serial.print(",");
  Serial.print(uS2 / US_ROUNDTRIP_CM);
  Serial.print(",");
  Serial.println(uS3 / US_ROUNDTRIP_CM);
}
```

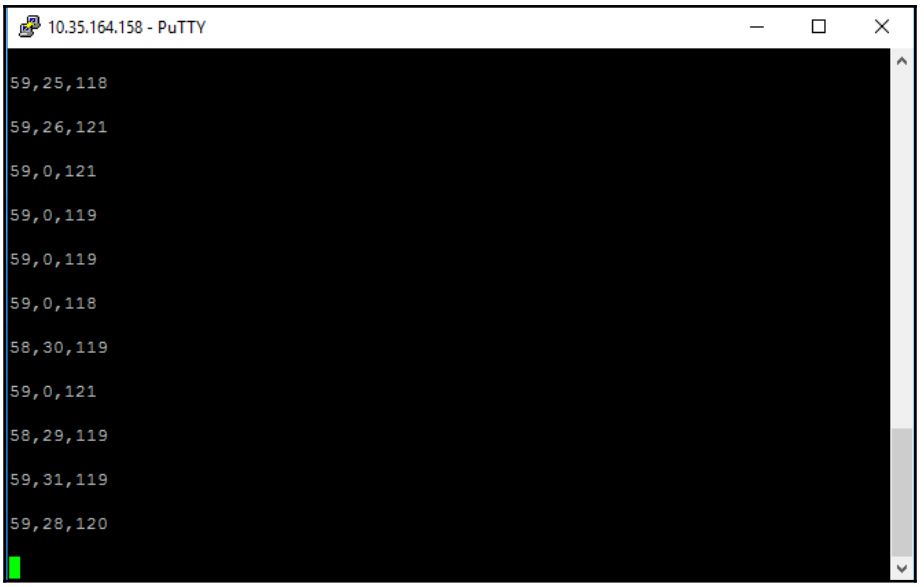
Once you've downloaded the code, you can disconnect the sensors from the host computer where you are developing the code and connect it to the BeagleBone Blue through the slave USB port, as shown in the following image:



With sonar sensors connected to the USB port of the BeagleBone Blue it is easy to create a simple Python program to run on the BeagleBone Blue that executes a while loop, reading the value from the Arduino each time through the loop . Here is that code:

```
import serial
ser = serial.Serial('/dev/ttyUSB0', 115200)
while True:
x = ser.readline()
print x
```

The code is quite simple. First, you'll import the `serial` library. Then, you open the `/dev/ttyUSB0` port. The loop reads the data coming from the Arduino. When you run this code, you should see the following output:



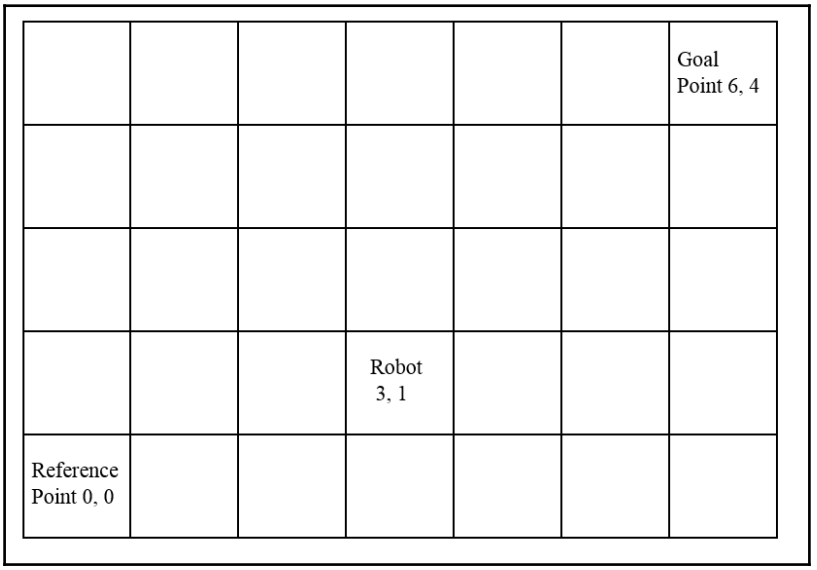
Each of the three measurements is now flowing to the BeagleBone Blue. You can now use this, along with the code you created earlier, to do some simple path planning.

Dynamic path planning with your mobile platform

With the three sensors providing coverage to check for barriers from the front as well as from either side you can add to your path planning capabilities. You can use the program that you wrote in `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*. Before you start, here are some basics of dynamic path planning. Let's first address the problem where you know where you want to go and need to execute a path without barriers and then add in barriers.

Basic path planning

In order to talk about dynamic path planning, that is, planning a path where you don't know what barriers you might encounter, you'll need a framework to understand where your robot is as well as to determine the location of the goal. One common framework is an *x-y* grid. Here is a drawing of such a grid:



There are three key points:

- The lower-left point is a fixed reference position. The directions *x* and *y* are also fixed; all other positions will be measured with respect to this position and these directions.
- Another important point is the starting location of your robot. Your robot will then keep track of its location using its *x* coordinate or position with respect to some fixed reference position in the *x* direction and its *y* coordinate and then its position with respect to some fixed reference position in the *y* direction to the goal. It will use the compass to keep track of these directions.
- The third important point is the position of the goal, also given in *x* and *y* coordinates with respect to the fixed reference position. If you know the starting location and the starting angle of your robot, you can plan an optimum (shortest distance) path to this goal. To do this, you can use the goal location and the robot location and some fairly simple math to calculate the distance and angle from the robot to the goal.

To calculate the distance, use the following equation:

$$d = \sqrt{((Xgoal - Xgoal)^2 + (Ygoal - Yrobot)^2)}$$

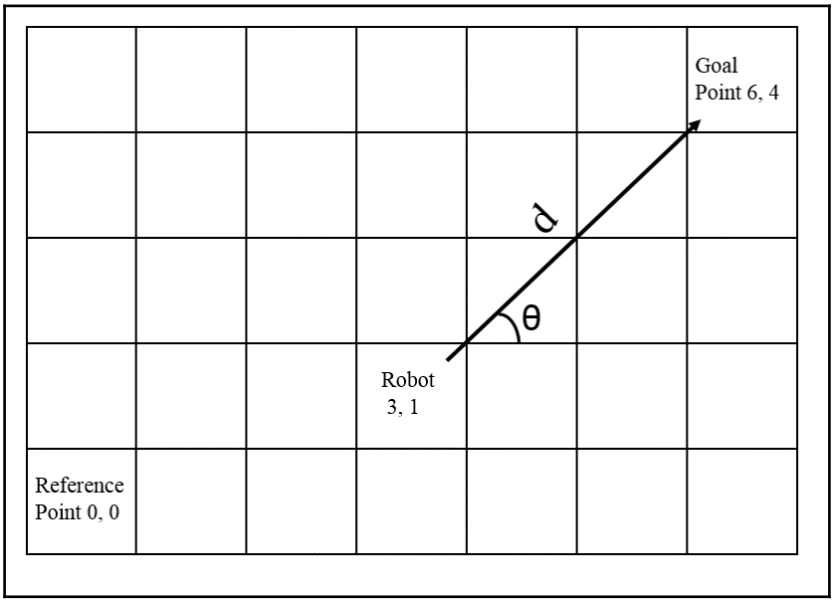
$$d = \sqrt{((Xgoal - Xgoal)^2 + (Ygoal - Yrobot)^2)}$$

You'll use this equation to tell your robot how far to travel to the goal. A second equation will tell your robot the angle it needs to travel:

$$\theta = \arctan(\frac{Ygoal - Yrobot}{Xgoal - Xrobot})$$

$$\theta = \arctan(\frac{Ygoal - Yrobot}{Xgoal - Xrobot})$$

Here is a graphical representation of these two pieces of information:



Now that you have a goal angle and distance, you can program your robot to move. To do this, you'll write a program to do path planning and call the movement functions you created in `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*. You'll need, however, to know how the distance that your robot travels in a second so you can tell your robot in seconds, not distance units, how far to travel. If you then know the angle and the distance, you can move your robot to the goal.

Here are the steps you'll program:

1. Calculate the distance in units that your robot will need to travel to reach the goal. Convert this into the number of seconds in order to achieve this distance.
2. Calculate the angle that your robot will need to travel to reach the goal. You'll use the compass and the functions you wrote to turn your robot in the program you wrote in `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement* , to turn to achieve this angle.
3. Now call the program from `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*, with the proper arguments to have your program move the correct distance.

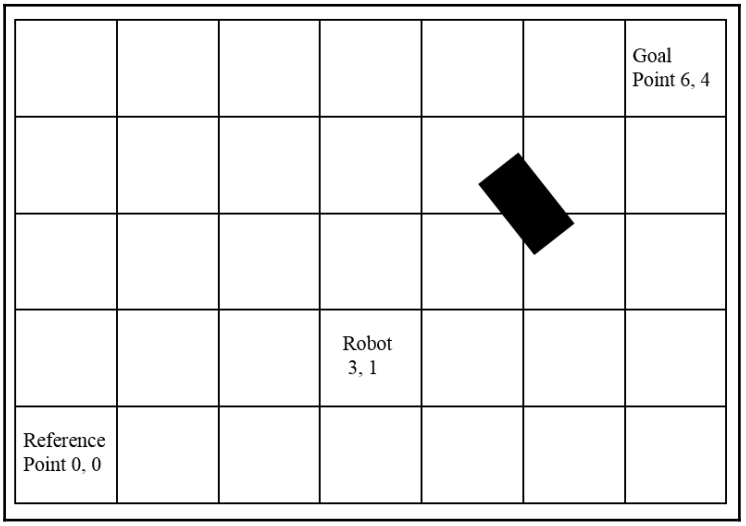
That's it. Now here's some very simple Python code that executes on the BeagleBone Blue to turn the robot and move it forward. Here is a listing of the program:

```
import serial
import time
import math
import os
xpos_robot = int(raw_input("Robot X Position: "))
ypos_robot = int(raw_input("Robot Y Position: "))
xpos_goal = int(raw_input("Goal X Position: "))
ypos_goal = int(raw_input("Goal Y Position:
"))distance =
math.sqrt((xpos_goal - xpos_robot)**2 + (ypos_goal -
pos_robot)**2)
angle = round(math.degrees(math.atan2((ypos_goal -
ypos_robot
(xpos_goal - xpos_robot))))
print distance, angle
os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(angle) +
' ' + str(distance))
```

In this program, the user enters the current robot and goal location. Then, some simple math computes the angle and distance to the new location. To keep it simple, a single unit distance is the amount the robot travels in 1 second. An `os.system()` call then invokes the program written in `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*, and moves the robot to the desired location.

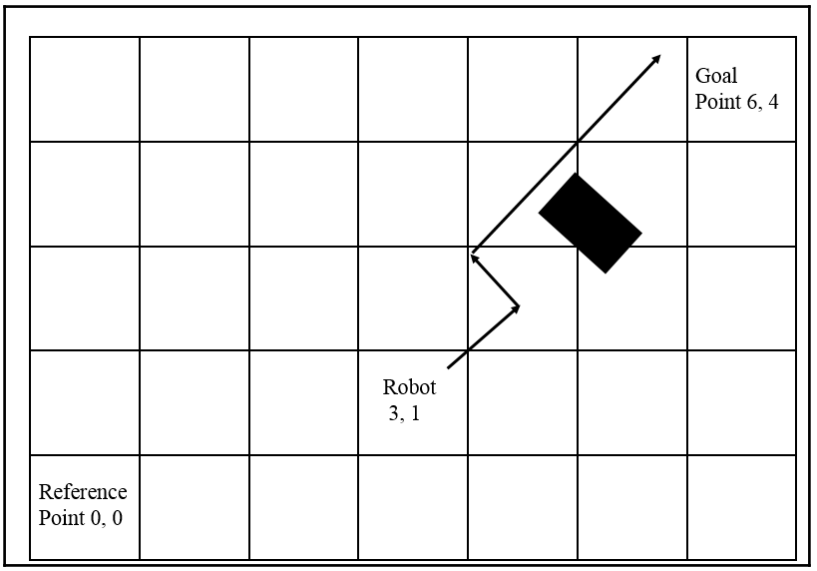
Avoiding obstacles

Planning paths without obstacles is, as has been shown, quite easy. However, it becomes a bit more challenging when your robot needs to walk around obstacles. Let's look at a case where there is an obstacle in the path you calculated previously. It might look like this:



You can still use the same path planning algorithm to find the starting angle; however, you'll now need to use your sonar sensor to detect the obstacle. To keep this all very simple, you'll do this in steps. You'll turn and move toward the obstacle for a one-time distance. Then, you'll check whether there is a barrier. When your sonar sensor detects the obstacle, you'll need to recalculate a path to avoid the barrier. To keep this all very simple, we'll assume that the barrier is quite small and that you'll always turn to the left. You'll turn to the left 90 degrees and then move down the barrier at a one-time distance. If the path is no longer blocked, based on the side sonar reading, you'll move on more time distance and then turn back to the path toward the desired path to the goal.

In this case, using these rules, the robot will travel the following path:



And here is the code to detect the barrier, turn to the left, then down the sensor until it is gone, and then turn back toward the goal:

```
import serial
import time
import math
import os
import serial
def getFrontSensor(ser, sensorMat):
ser.reset_input_buffer()
sensor = ser.readline()
sensorMat = sensor.split(",")
print sensorMat[1]
sensor_distance = int(sensorMat[1])
return sensor_distance
def getLeftSensor(ser, sensorMat):
ser.reset_input_buffer()
sensor = ser.readline()
sensorMat = sensor.split(",")
print sensorMat[0]
sensor_distance = int(sensorMat[0])
return sensor_distance
def getRightSensor(ser, sensorMat):
ser.reset_input_buffer()
sensor = ser.readline()
```

```
 sensorMat = sensor.split(",")
 print sensorMat[2]
 sensor_distance = int(sensorMat[2])
 return sensor_distance
 xpos_robot = int(raw_input("Robot X Position: "))
 ypos_robot = int(raw_input("Robot Y Position: "))
 xpos_goal = int(raw_input("Goal X Position: "))
 ypos_goal = int(raw_input("Goal Y Position: "))
 distance = math.sqrt((xpos_goal - xpos_robot)**2 +
(ypos_goal -
 ypos_robot)**2)
 angle = round(math.degrees(math.atan2((ypos_goal -
ypos_robot),
 (xpos_goal - xpos_robot))))
 print distance, angle
 ser = serial.Serial('/dev/ttyUSB0', 115200)
 not_at_goal = True
 distance_traveled = 0
 sensorMat = [0, 0, 0]
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(angle) +
 ' ' +     str(0))
 while not_at_goal:
 barrier = getFrontSensor(ser, sensorMat)
 if (barrier < 10):
 print "Barrier"
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(90) +
 ' ' + str(0))
 while(barrier > 0 and barrier < 15):
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(0) + ' ' +
 str(1))
 barrier = getLeftSensor(ser,sensorMat)
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
 str(0) + ' ' +
 str(1))
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(-90) + ' '
 + str(1))
 else:
 os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' +
str(0) + ' ' +
 str(1))
 distance_traveled += 1
 if (distance - distance_traveled) < 0:
 not_at_goal = False;
 ser.close()
```

This algorithm is quite simple; it won't get your robot perfectly to the goal, but fairly close. There are other algorithms, much too large to include here, that have much more complex responses to barriers. You could also provide more complex decision processes about which way to turn in order to avoid an object. Again, there are many different path-finding algorithms. Refer to `http://www.academia.edu/837604/A_Simple_Local_Path_Planning_Algorithm_for_Autonomous_Mobile_Robots` for an example. These more complex algorithms can be explored using the basic functionality that you have built in this chapter.

Summary

Congratulations! You can now detect and avoid walls and other objects that might be a barrier to your robot. You can also use these sensors to detect objects that you might want to find. In the next chapter, you'll learn how to add vision to your project. You'll learn how to see the world around you as well as be able to find colors and motion.

5
Allowing Our BeagleBone Blue to See

Our projects can communicate via voice, and now we are going to provide the details of adding a webcam to our system so that we can add vision to our projects. We'll use this functionality in lots of different applications; it will make our projects seem very advanced. Fortunately, adding hardware and software to provide vision to our projects is both easy and inexpensive.

Vision will open a set of possibilities for your project. These can range from simple motion detection to advance capabilities such as facial recognition, object identification, and even object following. You can also use vision to detect the world around you and avoid obstacles. Vision is central to our existence as human beings and can provide a whole new level of coolness to your projects.

In this chapter, you will learn:

- How to connect your USB camera to our BeagleBone Blue and view the images
- How to download and install OpenCV, a fully featured vision library
- How to use the vision library to detect colored objects

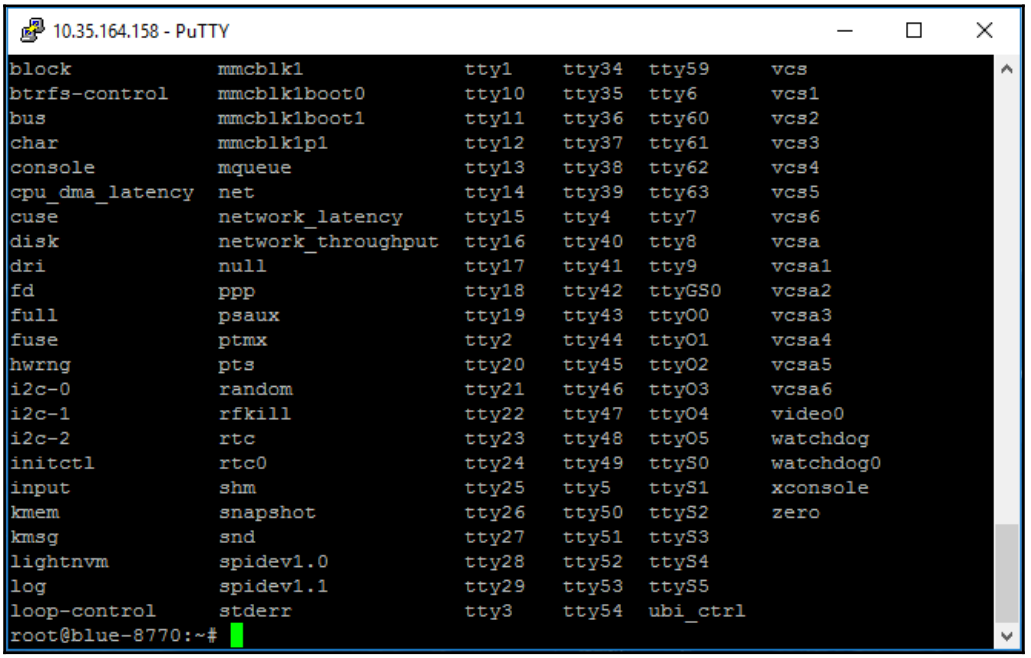
To complete this mission, I'll assume you already have the BeagleBone Blue with a LAN connection to the outside world and a 5V power supply. You'll need to add a USB webcam to this. Try to find one of fairly recent manufacture. You may try to use that webcam that you bought many years ago and that has now been relegated to the project shelf to work, but the odds are low that the webcam will work and the money you save will not be worth the frustration.

In most cases, we won't need to connect this single device through a powered USB hub; however, if you encounter problems, realize that lack of USB power could possibly be the problem.

Connecting your USB camera to your BeagleBone Blue and viewing the images

Our first step in enabling computer vision is to connect your USB camera to your USB port on the BeagleBone Blue. For this example, you'll use an iHome webcam, a standard USB webcam available at many electronics online outlets. To make sure you can access your USB webcam, I like to use a program called `guvcview`. Install this by typing `sudo apt-get install guvcview`.

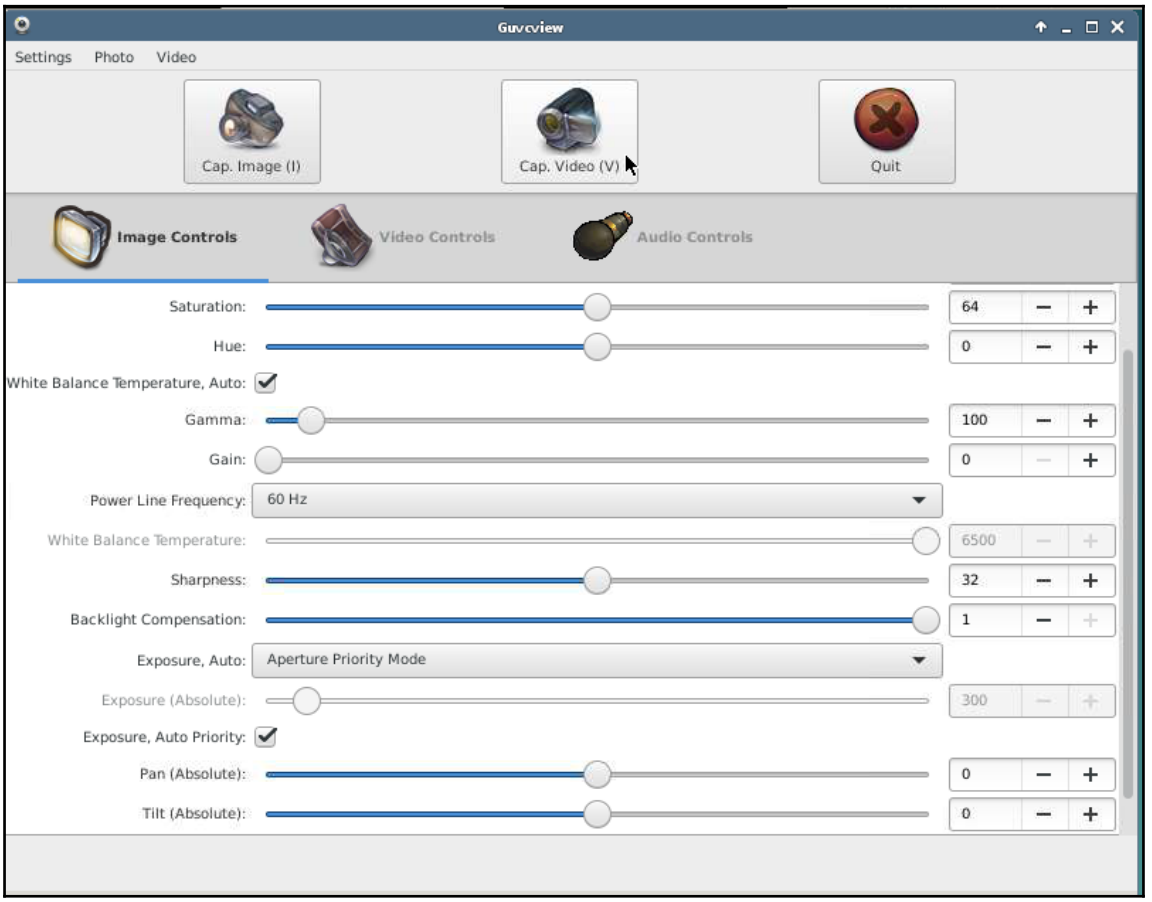
Connect your USB camera. Then, apply power to the BeagleBone Blue. After the system is booted, you can check whether the BeagleBone Blue has found your USB camera. Go to the `/dev` directory and type `ls`. You should see something like this:



What you are looking for is the video device; this is our webcam. So if you see it, then you can be sure that the system at least knows that your camera is there.

Now let's use guvcview to view the output of the camera. Since this is going to need to output some graphics, you'll need to use VNC server. To use a VNC server, make sure you start the server on the BeagleBone Blue by typing VNC server via SSH. Then, start up your VNC viewer on your host computer. Once you have a VNC viewer window, open a Terminal window on the BeagleBone Blue and type `sudo guvcview`. You should see something like this:

The video window displays what the webcam sees, and the **GUVCViewer Controls** window lets you control the different aspects of the camera. For this particular camera, the defaults work fine. This will not always be the case. If you get a black screen for the camera, you may need to adjust the settings. Select the **Controls** window and the **Video & Files** tab. You will see a window where you can adjust the settings for your camera.



Perhaps the most important setting is the **Resolution** setting. If you see a black screen for your camera, adjust the resolution down; often, this will resolve the issue. This will also tell you what resolutions are supported for your camera. Once you have the camera up and running and a desired resolution is documented, you can move on to downloading and installing OpenCV.

Your system can now see the outside world. Guvcview can actually capture images or video and store them as files, so you can use it to capture images or pictures. However, OpenCV is a much more attractive option as it provides a fully featured set of image processing capability as well.

Downloading and installing OpenCV, a fully featured vision library

You'll need to install OpenCV, a complete vision library that provides an amazing array of tools for you to capture, process, and save your images. First, you'll need to download a set of libraries and OpenCV itself. There are several possible steps; I'm going to suggest one 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 this order:

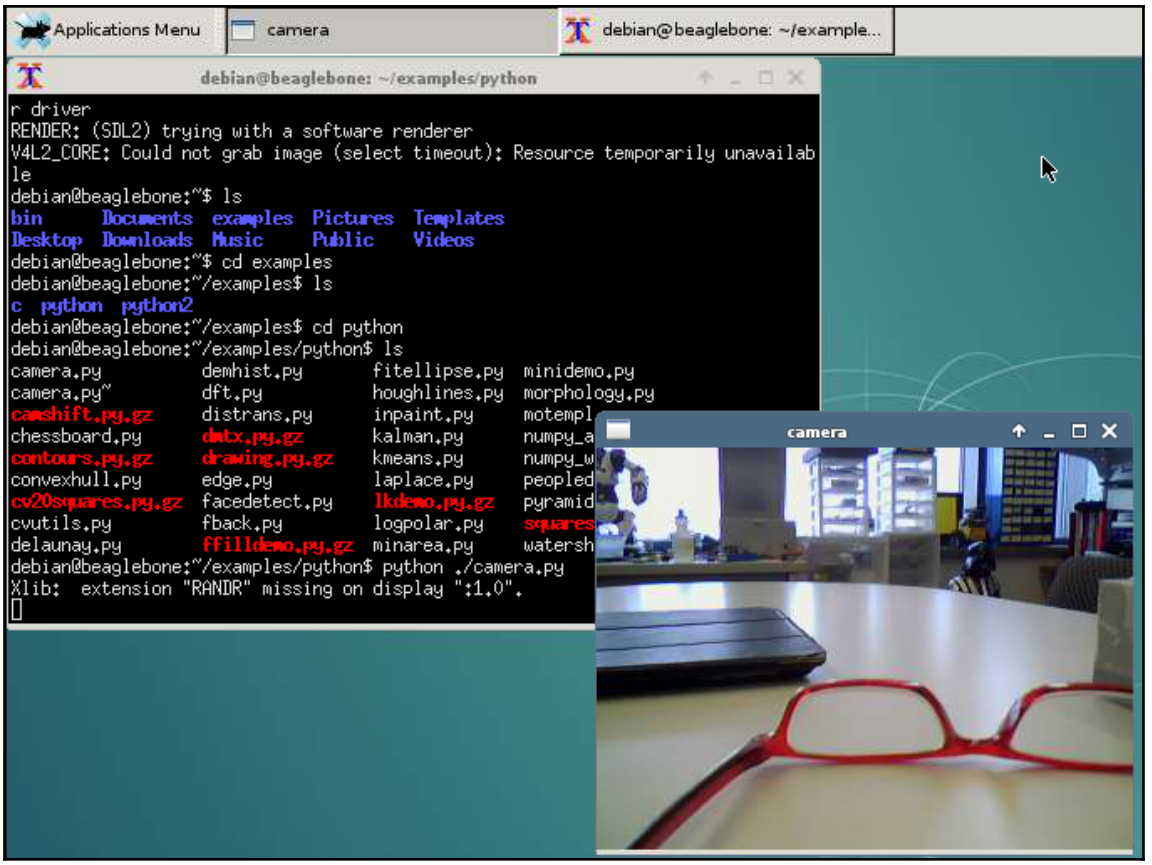
1. `sudo apt-get update`: If you haven't done this in a while, it is a good idea to do this now before you start. You're going to be downloading a number of new software packages, so it is good to make sure everything is up to date.
2. `sudo apt-get install libavformat-dev`: This library provides a way to code and decode audio and video streams.
3. `sudo apt-get install python-opencv`: This is the Python development kit for OpenCV, which is required as you are going to use some Python to play with OpenCV; it is just too easy not to use it.
4. `sudo apt-get install libcv2.4 libcvaux2.4 libhighgui2.4`: This is basic OpenCV library. Note the number. This will almost certainly change as newer versions of OpenCV become available. If 2.4 does not work, either try 2.5 or Google the latest version of OpenCV.
5. `sudo apt-get install opencv-doc`: This is the documentation for OpenCV, including a set of example programs.
6. `sudo apt-get install libgl1-mesa-swx11`: This installs a substitute of OpenGL that you'll need when you want to access your OpenCV capabilities remotely through the VNC server.

Make sure you are in your home directory and then type `cp -r /usr/share/doc/opencv-doc/examples`. This will copy all the examples to your home directory. Now you are ready to try out the OpenCV library. It is easier to use Python when programming simple tasks, so let's start with the Python examples. If you prefer the C examples, feel free to explore.

Try one of the Python examples. Change directory to the Python examples by typing `cd ./examples/python`. In this directory, you will find a number of useful examples; let's start with the most basic ones. It is called `camera.py`. You can try running this example; however, to do this, you'll need to connect to the BeagleBone Blue via the VNC server. Once you are connected, you'll first want to change the `camera.py` code to set the initial resolution of the window. Edit `camera.py` to look like this:

```
#!/usr/bin/python
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:
breakcv.DestroyAllWindows()
```

Now bring up a Terminal window and type in `python camera.py` You should see something like this:



Your project can now see! You will use this capability to do a number of impressive tasks that will use this video. You may want to play with the resolution to find the optimum for your application. Bigger images are great; they give you more of a view on the world, but they also take up significantly more processing power. You'll play with this more as we actually ask your system to do some real image processing.

Using the vision library to detect colored objects

A useful task that OpenCV and your webcam can do is to track objects. This might be useful if you are building a system that needs to track and follow a colored object. OpenCV makes this amazingly simple by providing some high-level libraries that can help with this task. I'm going to do this in Python, as I find it much easier to work with than C. If you're a coder who feels more comfortable in C, these instructions should be fairly easy to translate to the C environment.

If you'd like, create a directory to hold your image-based work. From your home directory, you can create an `imageplay` directory name by typing `mkdir imageplay` while in your home directory. Then, change directory to the directory you just created by typing `cd imageplay`. Once there, let's bring over your `camera.py` file as a starting point by typing `cp /home/debian/examples/python/camera.py camera.py`. Now you are going to edit the file until it looks something like this:

```
#!/usr/bin/python
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, (10, 120, 100), (70, 255, 255),
threshold_img)
cv.ShowImage("Color Tracking", img)
cv.ShowImage("Threshold", threshold_img)
if cv.WaitKey(10) == 27:
break
cv.DestroyAllWindows()
```

Let's look specifically at the changes you need to make to `camera.py`. The first three lines you add are these:

```
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)
```

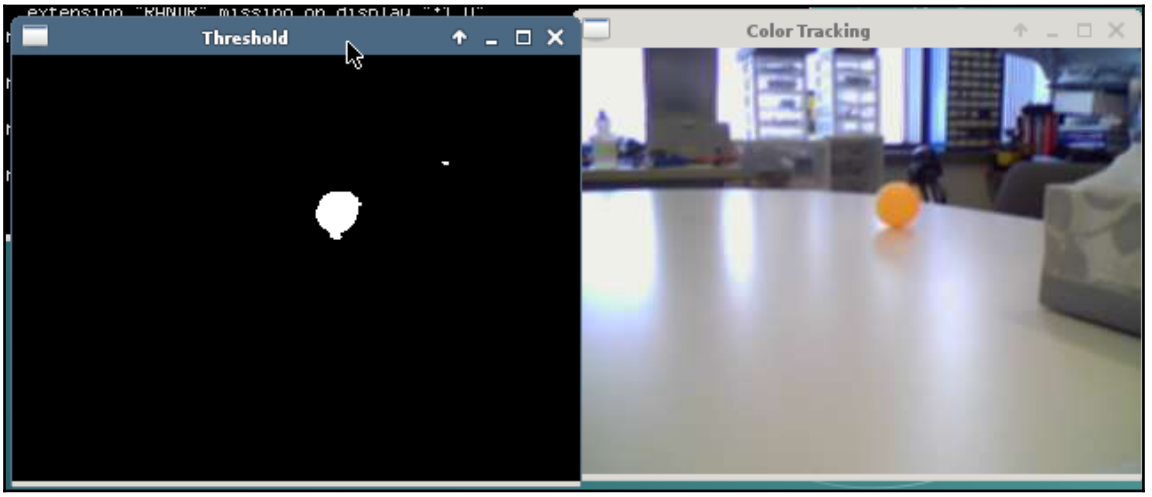
You are going to use the OpenCV library to first smooth the image and take out any large deviations. The next two lines create a new image that stores the image in values of hue (color), saturation, and value instead of the RED, GREEN, and BLUE pixel values of the original image.

Then, add these two lines:

```
threshold_img = cv.CreateImage(cv.GetSize(hue_img), 8, 1)
cv.InRangeS(hue_img, (10, 120, 100), (70, 255, 255), threshold_img)
```

You are going to create yet one more image, this time, a black and white image that is black for any pixel that is not between two certain color values. The `(10, 120, 100), (70, 255, 255)` determine the color range. In this case, I have a standard orange ping pong ball and I want to detect the color orange.

Now run the program by typing `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; in this case, I use my orange ping pong ball. Then, move it into the frame. You should see something like this:



Notice the white pixels in your threshold image showing where the ball is located. You can add a bit more OpenCV code that will provide you with the actual location of the ball. You can actually draw an indicator in your original image file, indicating the location of the ball. Edit the `camera.py` file to look like this:

```
#!/usr/bin/python
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, (10, 120, 100), (70, 255, 255), threshold_img)
storage = cv.CreateMemStorage(0)
contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP,
     cv.CV_CHA\
IN_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, (0, 255, 0))
cv.ShowImage("Color Tracking", img)
if cv.WaitKey(10) == 27:
break
cv.DestroyAllWindows()
 The first change you are going to make here is to add these lines:
 storage = cv.CreateMemStorage(0)
 contour = cv.FindContours(threshold_img, storage,
 cv.CV_RETR_CCOMP,
 cv.CV_CHAIN_APPROX_SIMPLE)
```

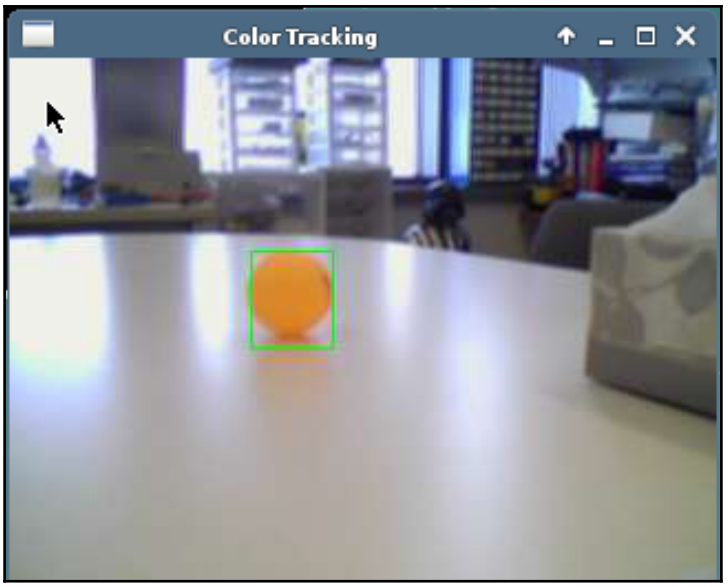
These lines find all the areas on your image that are within your threshold. There may be more than one, so you want to capture them all. Now you will add a while loop that will let you step through all the possible contours:

```
points = []

while contour:
```

The next few lines will then get the information for each of your contours. You want to identify the corners. Then, you can check whether the area is big enough to be of concern. If it is, you will add a rectangle to your original image identifying where we think it is:

```
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, (0, 255, 0))
Now that the code is ready, you can run it. You should see
something like this:
```



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 now you can use your mobile platform to move to the right and the left when the colored object moves to the right and the left.

Here is the code:

```
#!/usr/bin/python
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, (10, 120, 100), (70, 255, 255),
threshold_img)
storage = cv.CreateMemStorage(0)
contour = cv.FindContours(threshold_img, storage, cv.CV_RETR_CCOMP,
cv.CV_CHA\
IN_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, (0, 255, 0))
x, y = pt1
if x < 10:
os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' + str(20) + ' '
+ str(0))
if x > 320:
os.system('/root/rc_wheeled_auto/rc_wheeled_auto ' + str(-20) + ' '
+ str(0))
cv.ShowImage("Color Tracking", img)
if cv.WaitKey(10) == 27:
break
cv.DestroyAllWindows()
```

The difference between this code and the previous set is fairly straightforward; it uses the techniques you used in `Chapter 4`, *Avoiding Obstacles Using Sensors*. Here, you use the `x` position of the ball; if it is too close the right-hand side of the viewport, you'll move the robot to the right, and if it is too close to the left-hand side of the viewport, you'll move the robot to the left.

You can do all sorts of incredible things with just a few lines of code. Another common feature we may want to add to our projects is motion detection. If you'd like to try, there are several good tutorials; try looking at `http://derek.simkowiak.net/motion-tracking-with-python/`, `http://stackoverflow.com/questions/3374828/how-do-i-track-motion-using-opencv-in-python`, or `https://github.com/RobinDavid/Motion-detection-OpenCV`.

Summary

Your projects can see now! Your projects can respond to changes in the physical environment that can sense using your webcam. In the next chapter, the capability you will add is to make your projects able to speak and listen.

6
Providing Speech Input and Output

In this chapter, you'll be adding a microphone and speaker, but more than that, you'll add functionality so that your project can both recognize voice commands and also respond via the speaker. This will free you up from typing in commands and let you interact with your projects in an impressive way.

Besides, what self-respecting robot wants to carry around a keyboard? No, you want to interact in natural ways with your projects, and what you learn in this chapter will enable that.

In this chapter, you'll learn about the following topics:

- Hooking up speakers and a microphone 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 have your robot initiate action

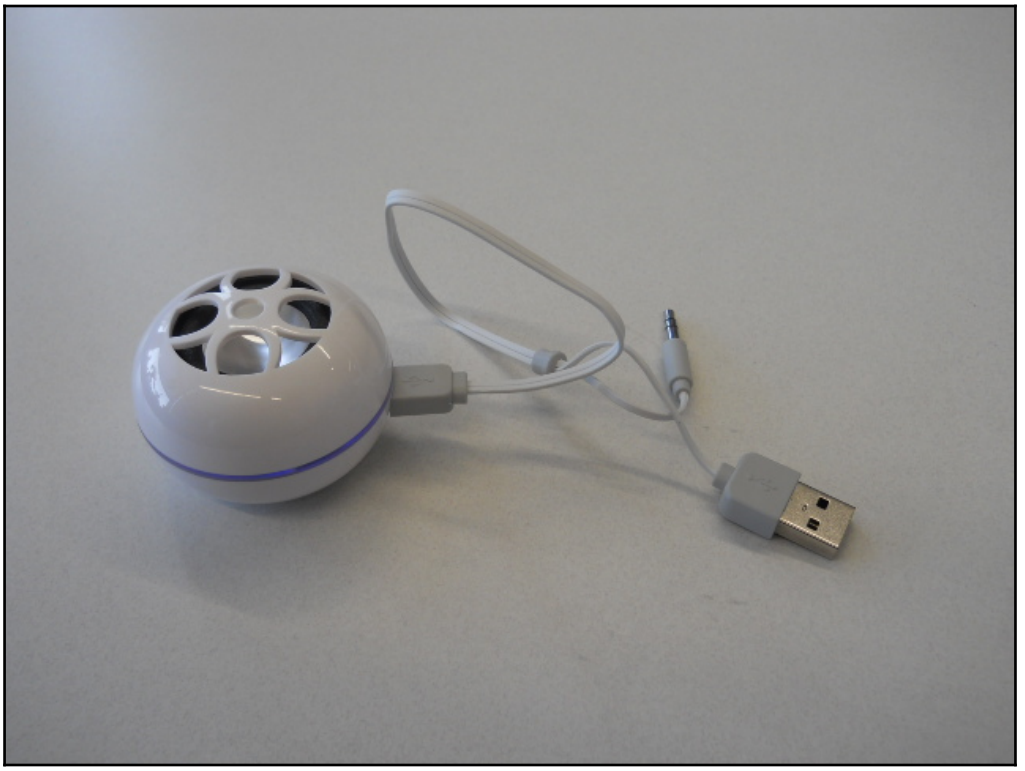
Hardware prerequisites

For this chapter, you're going to need several pieces of hardware:

1. First, you'll need a USB device that supports microphone in and speaker out. These are available at most online electronics outlets. One such device is shown next:

2. Then, you'll need a microphone that can plug into the USB device. Again, these are available at most online electronics outlets. Here is an example:

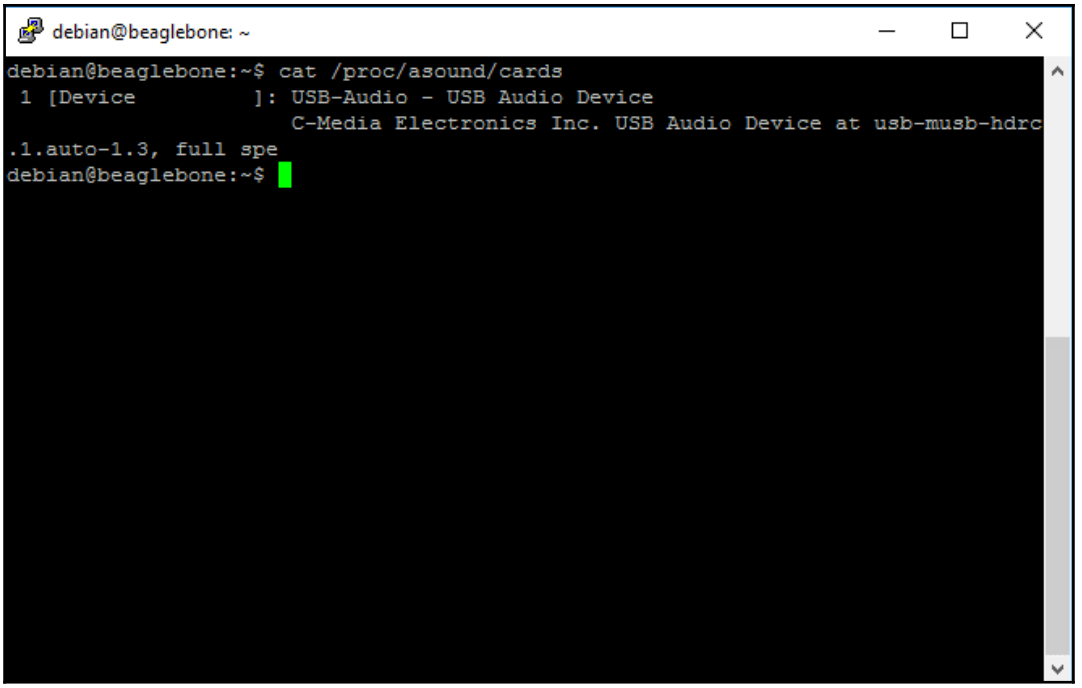
3. Finally, you'll need a powered speaker that can plug into the USB device. When you get the speaker, make sure it is powered. Your board will generally not be able to drive a passive speaker with enough power for these applications. This can either be internal battery power, or it can be powered via a USB. Here is a picture of such a device:



Fortunately, these devices are very inexpensive and are widely available. You'll need to get these before you can proceed.

Connecting the hardware and making an input sound

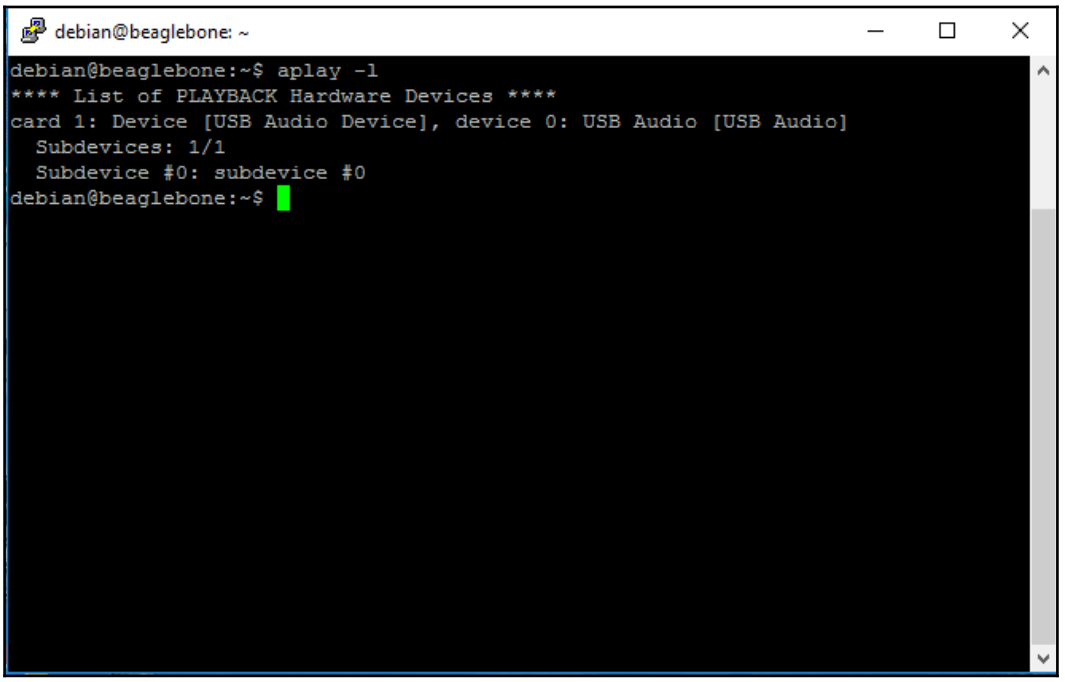
To start, you are going to hook up the hardware and make and input sound. Plug in the power and SSH into the BeagleBone Blue. Now type in `cat/proc/asound/cards`. You should see the following response:



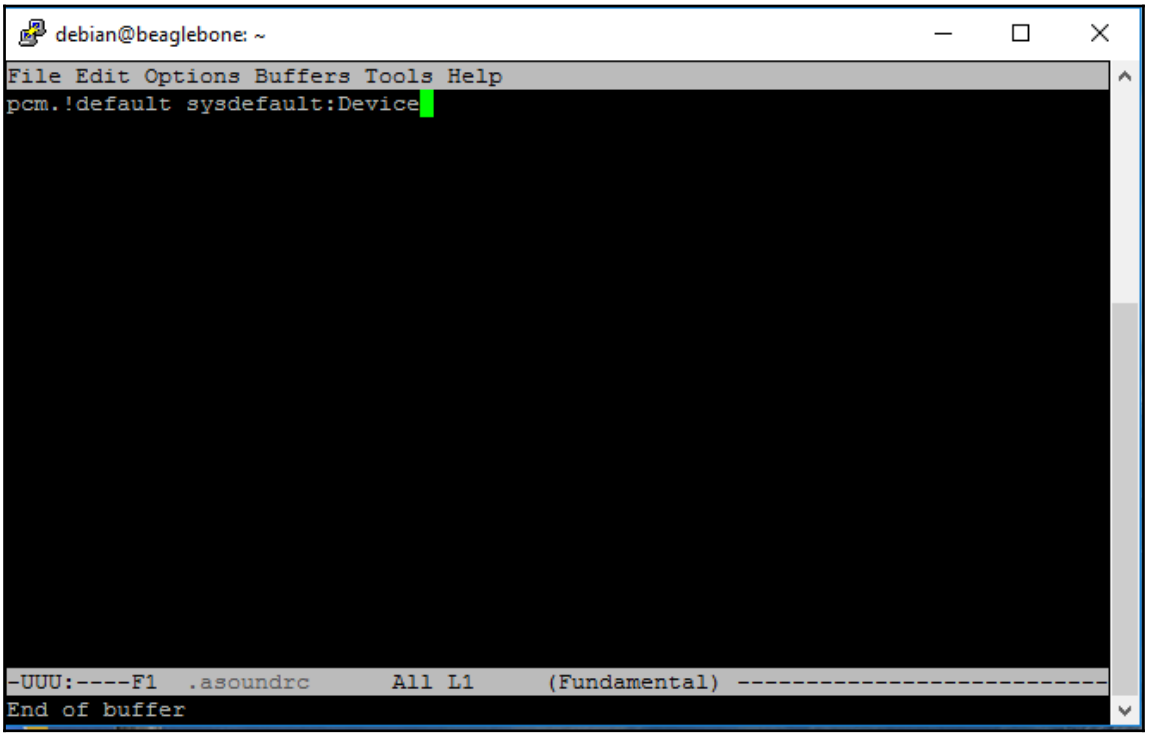
Note that the system indicates that you are connected to your USB audio plugin. Now you can use the USB card to both create and record sound.

First, let's get some music going. This will let you know that your USB sound device is working. You'll need to first configure your system to look for your USB card and play and record sound from there as the default. To do this, you'll need to add a library to your system. Install the library include files by typing `sudo apt-get install libasound2-dev`. This will install the basic capability that you need.

Let's make sure your system knows about your USB sound device. At the prompt, type in `aplay -l`. You should see the following:



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. Using your favorite editor, create the `.asoundrc` file and put the following in it:

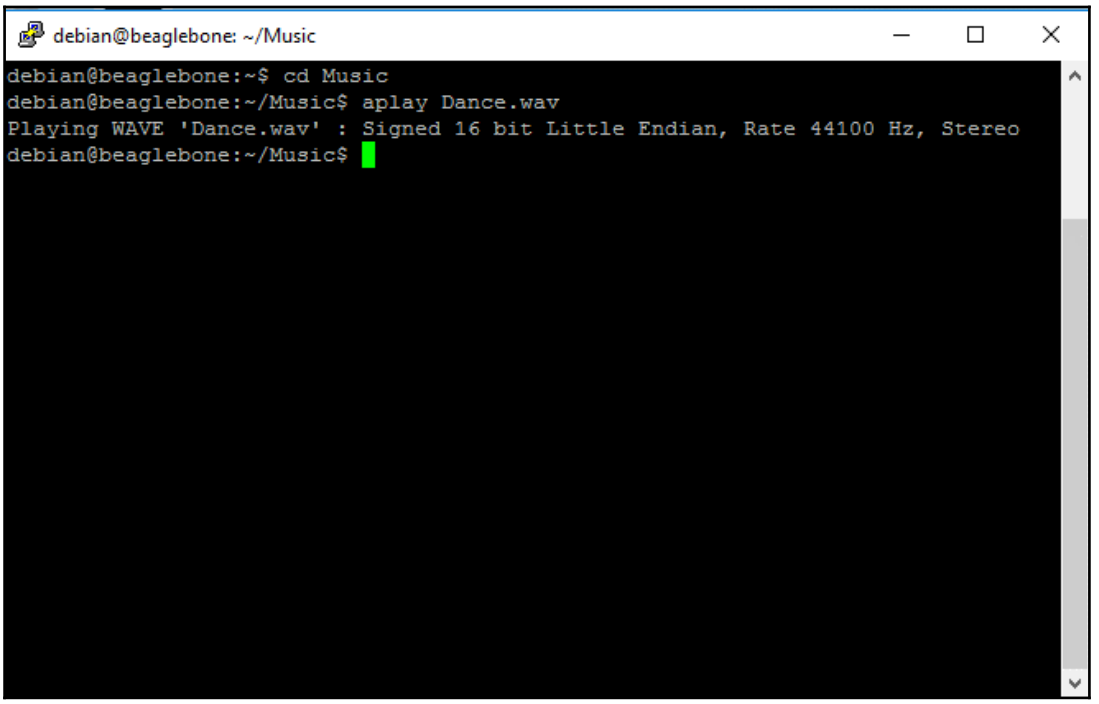


This will tell the system to use your USB device as a default. Once you have completed this, reboot your system.

Later, you will be using root to send voice commands to your robot. To do this, you'll need to create the `.asoundrc` file in the `/root` directory as well.

You will now try to actually play some music. To do this, you need a sound file and a way to play it. You can use `WinScp` from a Windows machine to transfer a simple `WAV` file to the `Music` subdirectory of your BeagleBone Blue. You are going to use an application called `aplay` to play your sound. In my case, I downloaded some music under the name `Dance.wav`.

Now type `aplay Dance.wav` to play your music. You will see, and hopefully hear, this result:



If you aren't hearing any music, check to make sure power is applied to your speaker. It could also be that you need to set the volume. To do this, you'll want to open up a VNC server-VNC viewer pair to your BeagleBone Blue and then navigate to **Applications Menu | Multimedia | Audio Mixer**. Once there, you'll want to select the **Select Controls...** button at the bottom and then enable the volume controls for your speaker and microphone.

You should then see this:

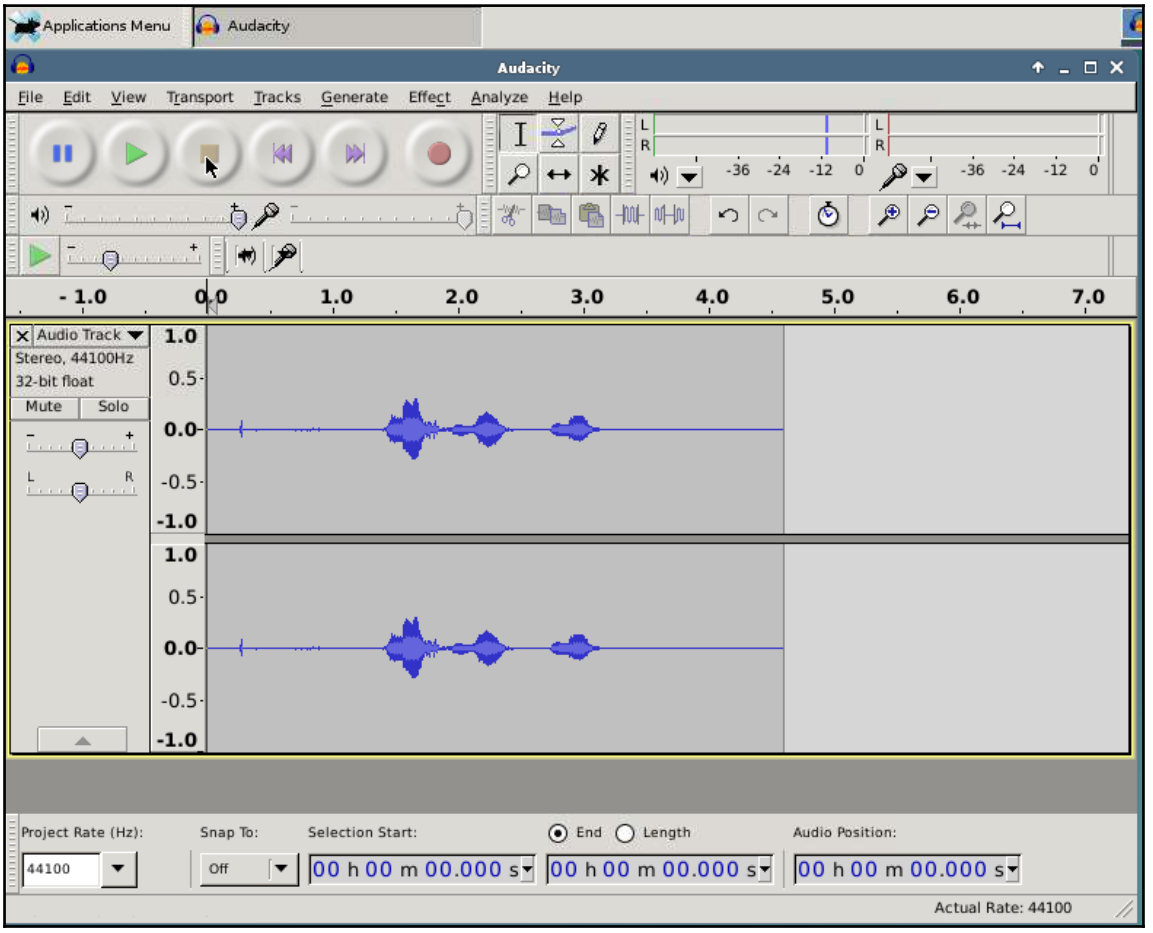
Now make sure everything is unmuted and set to the desired volume.

You can also set your speaker and microphone settings using text commands. Refer to `https://wiki.archlinux.org/index.php/Advanced_Linux_Sound_Architecture#Keyboard_volume_control`for more information.

By the way, `aplay` can be a bit finicky with respect to the types of file it might accept, so you may have to try different **.wav** files until you get one it likes.

Now you can play music or other sound items using your BeagleBone Blue system. You can change the volume of your speaker. You're ready for the next step now.

To make sure you can record your audio, you can use an application called **Audacity**. It is free and is very useful for you to be able to see your speech. Download the application by typing `sudo apt-get install audacity`. You can then run the application using the VNC server by navigating to **Applications Menu** | **Multimedia** | **Audacity**. You can then simply press the red button and record your speech. Playing it back is as easy as hitting the play button. Here is a picture of the interface:



If you are not able to record, you might want to check your **Audio Mixer** settings. Now that you can play music and record sound, you can make your application talk.

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

Now that we can both get sound out of your BeagleBone Blue, let's start doing something useful with this capability. You're going to start by enabling eSpeak, an open source application that provides us with a computer voice with a bit of personality.

The great news is that here, you get a huge check of functionality for free. The program you are going to use is eSpeak, an open source voice generation application. To get this functionality, download the eSpeak library by typing `sudo apt-get install espeak`. Now, let's see whether your BeagleBoard Blue has a voice. Type the command `espeak "hello"`. The speaker should emit a computer voiced hello. If it does not, make sure that the speaker is on and that its volume is high enough to hear it.

Now that you 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 at the command line. For example, type in `espeak -v+f3 "hello"` and you should hear a female voice. Add a Scottish accent by typing `espeak -ven-sc+f3 "hello"`. My personal favorite is the West Midlands accent using a female voice, `espeak -ven-sc+f3 "hello"`.

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 using your editor to create a text file called speak and then type in this command: `espeak -f speak.txt`.

There are lots of choices with respect to eSpeak. Feel free to play around with those, choosing your favorite. Don't expect, however, that you'll get the kind of voices that you hear from computers in the movies. Those are actors, not computers, although one day, we will get to that point where computers will sound a lot like people.

Using Pocketsphinx to interpret your voice commands

Now that our projects can speak, you will want them to listen as well. This isn't nearly as simple as the speaking part, but thankfully, you have some significant help. You are going to download a set of capabilities called Pocketsphinx, and using these, you will provide your project with the ability to listen to your commands.

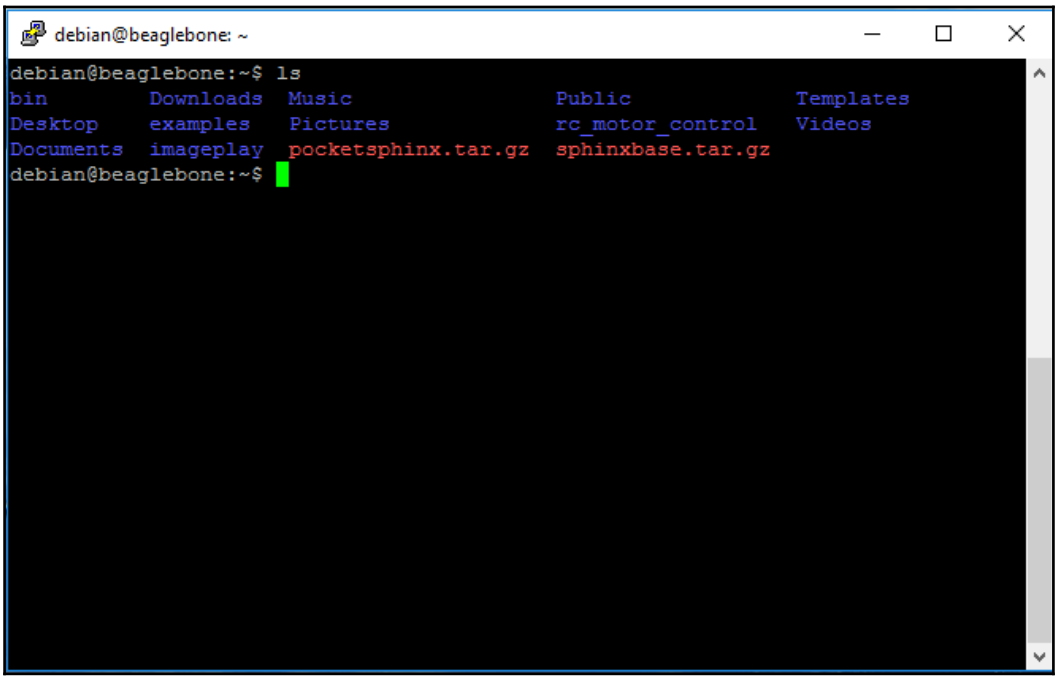
The first step is to download the Pocketsphinx capability. Unfortunately, this is not quite as user-friendly at the eSpeak process, so follow along carefully. First, go to the Sphinx website, hosted by Carnegie Mellon University at `http://cmusphinx.sourceforge.net/`. This is an open source project that provides you with the speech recognition SW you will need. With your smaller, embedded system, you will be using the Pocketsphinx version of this code.

You will need to download two pieces of software, sphinxbase and Pocketsphinx. Download these by issuing the following commands first, and these should now be in the `/usr/debian` directory of your BeagleBone Blue. However, before you build these, you need three libraries.

The first of these is `libasound2-dev`. Now if you have completed the first and second objectives in this chapter, you will have downloaded this package. If you skipped that part because you didn't want your project to speak, you'll need to download it now using `sudo apt-get install libasound2-dev`. If you are unsure, just do it again; it will tell you whether it's already there.

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 in `sudo apt-get install bison`. The final package you'll need is `swig`. This package type is `sudo apt-get install swig`.

If everything is installed and downloaded, you can build Pocketsphinx. First, your home directory should look like this, with the `tar.gz` files of both Pocketsphinx and Sphinxbase:

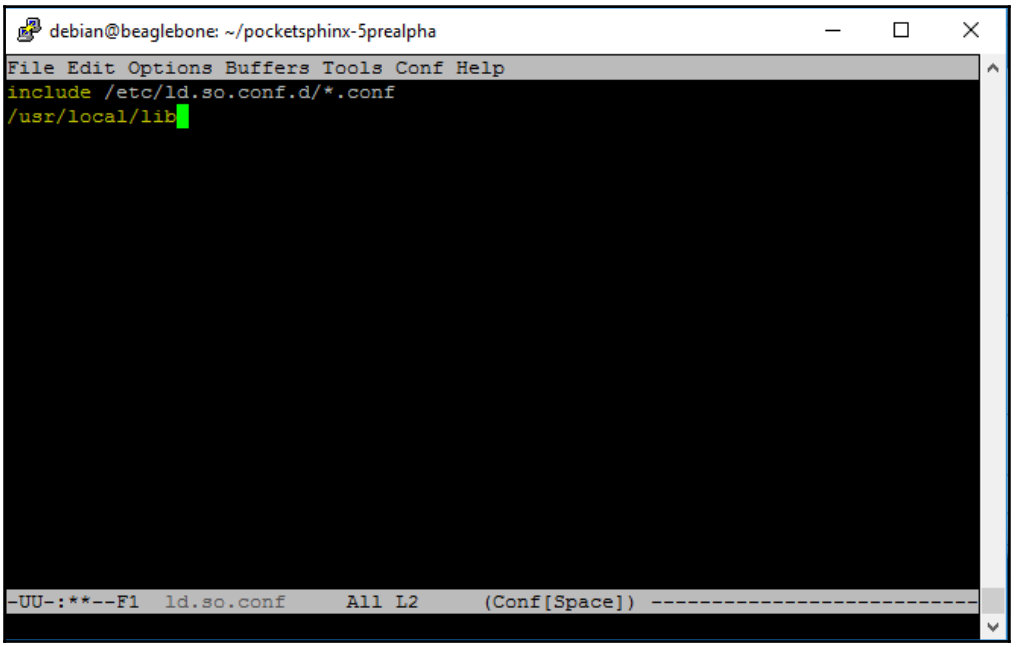


You will start by unpacking and building the sphinx base. Type in `tar -xzvf sphinxbase-5prealpha.tar.gz`. This should unpack all the files from your archive into a directory called `sphinxbase-5prealpha`. Now `cd` to that directory.

Now you will build the application. Start by issuing the `./configure --enable-fixed` command. This will first check to make sure everything is OK with the system and then configure a build. Now type in `make`. This builds all the files you'll need as a part of the Pocketsphinx base system. Finally, type in `sudo make install`, which will install all of the Pocketsphinx base capability.

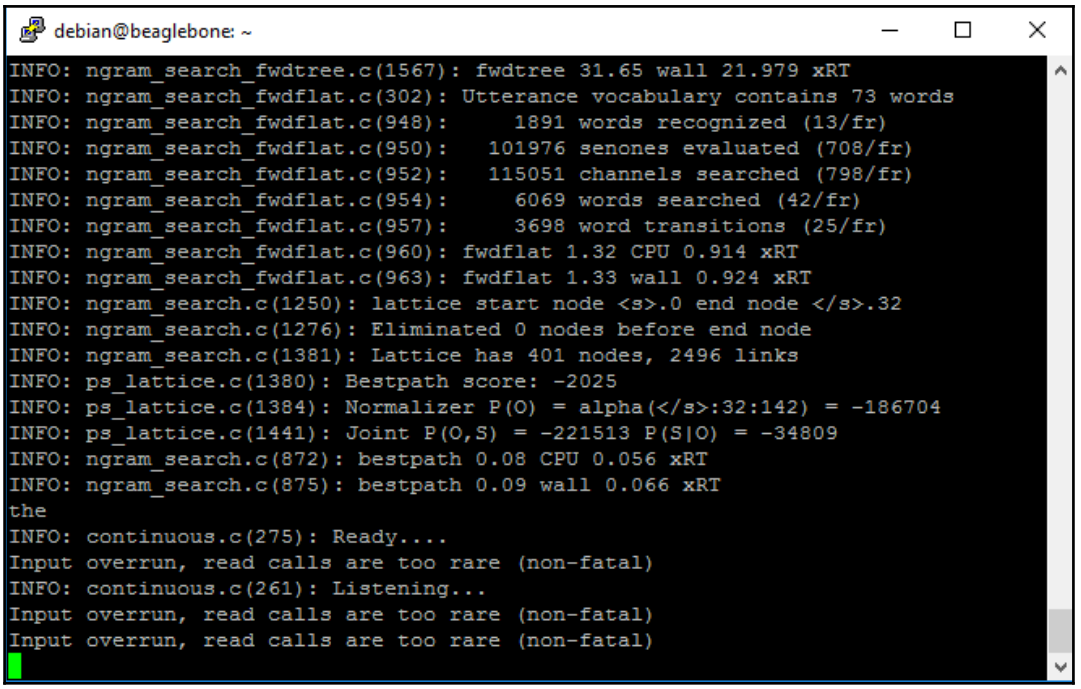
Now you need to make the second part of the system: the PocketSphinx code itself. Go to the home directory and un-archive the code by typing `tar -xzvf pocketsphinx-5prealpha.tar.gz`. The files should now be un-archived, and you can now build the code. We'll follow similar steps for these files, first `cd` to the Pocketsphinx directory, then type `./configure` to see whether we are ready to build the files. Then type `make`, wait for a good while for everything to build, and then type `sudo make install`.

Once you have the completed the installation, you'll need to let the system know where your files are. To do this, you will need to edit the `/etc/ld.so.conf` as root. You will add the last line to file so it now should look like this:



Now type `sudo /sbin/ldconfig` and the system will be aware of your Pocketsphinx libraries. Everything is installed; you can try your speech recognition. Just type `pocketsphinx_continuous -samprate 16000 -inmic yes`. This program takes in input from the mic, sampled at 16000 times a second, and turns it into speech. After running the command, you'll get all kinds of information that won't have much meaning for us and then get to the point where the listening prompt on the screen signals the system should be ready to receive a command.

Once you get to the listening prompt on the screen, the system should be ready to receive a command. Say hello into the mic. When it senses that you have stopped speaking, it will process your speech, again giving us all kinds of interesting information that has no meaning for us, but it should eventually show this screen:



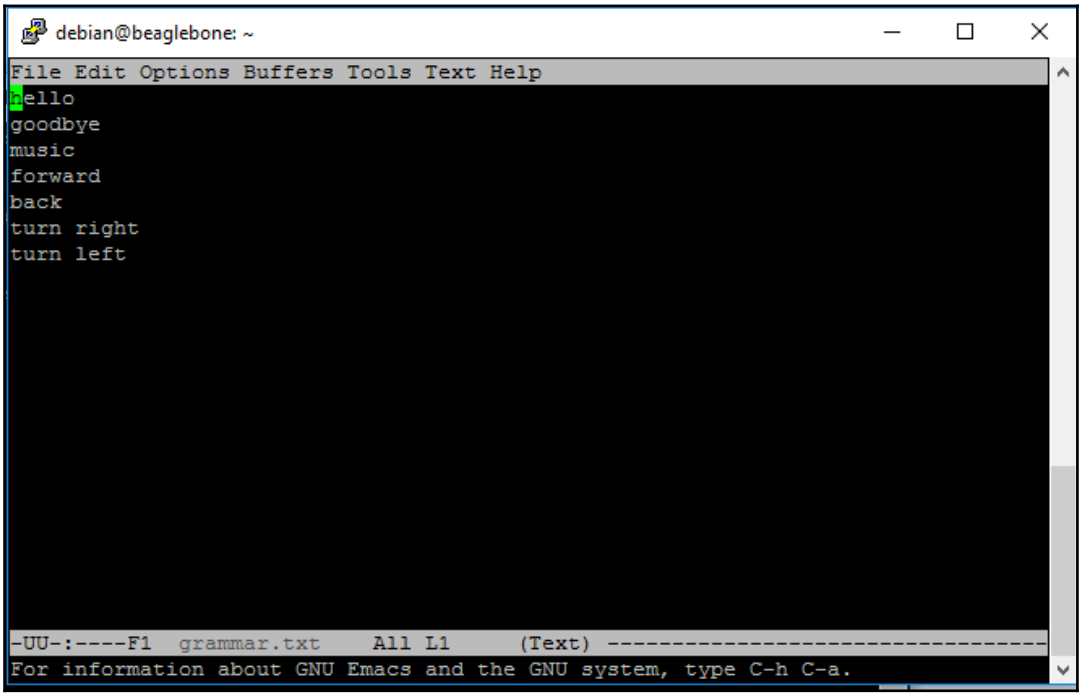
Note that the system indicates that you said the. It recognized your speech but not the exact word. You can try other words and phrases. The system is very sensitive, so it may also pick up background noise. You are also going to find that it is not very accurate.

There are two ways to make it more accurate. One is to train the system to more accurately understand your voice. I am not going to detail that process here; it is a bit complex, and if you want to know more, feel free to go to the CMU Pocketsphinx website.

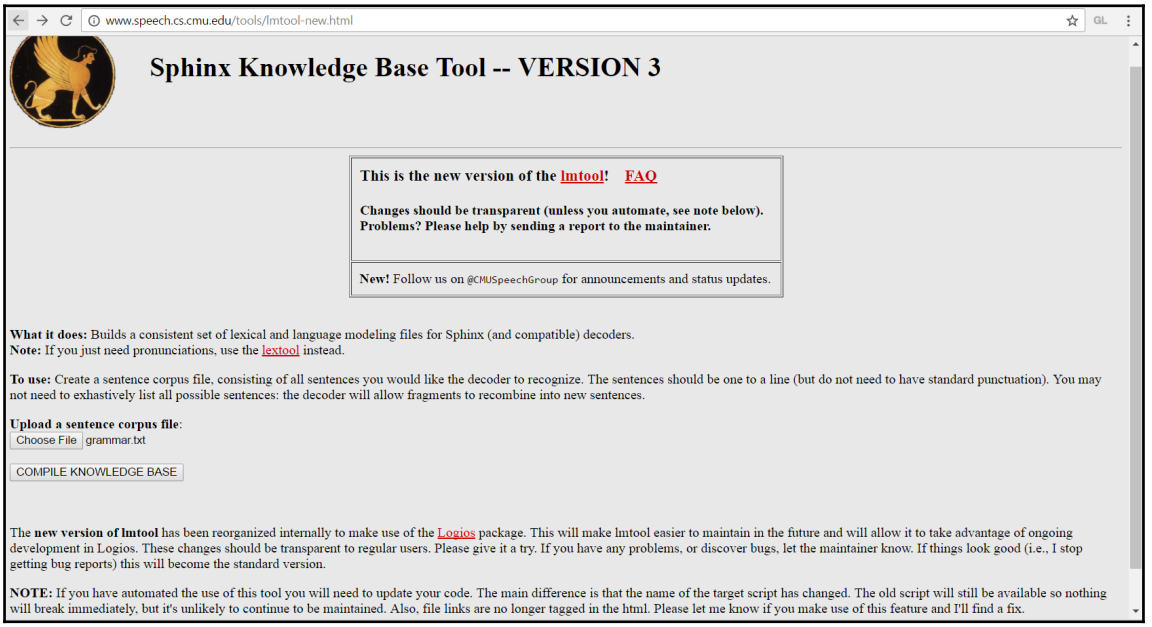
The second way to improve accuracy is to limit the number of words that your system can use to determine what you are saying. The default has literally thousands of words that are possibilities, so if two words are close, it may choose the wrong word as opposed to the word you spoke. In order to make the system more accurate, you are going to restrict the words it has to choose from. You do this by making your own grammar.

You can either do this on the BeagleBone Blue, or you can do it on your host computer and then download the appropriate files to the BeagleBone Blue.

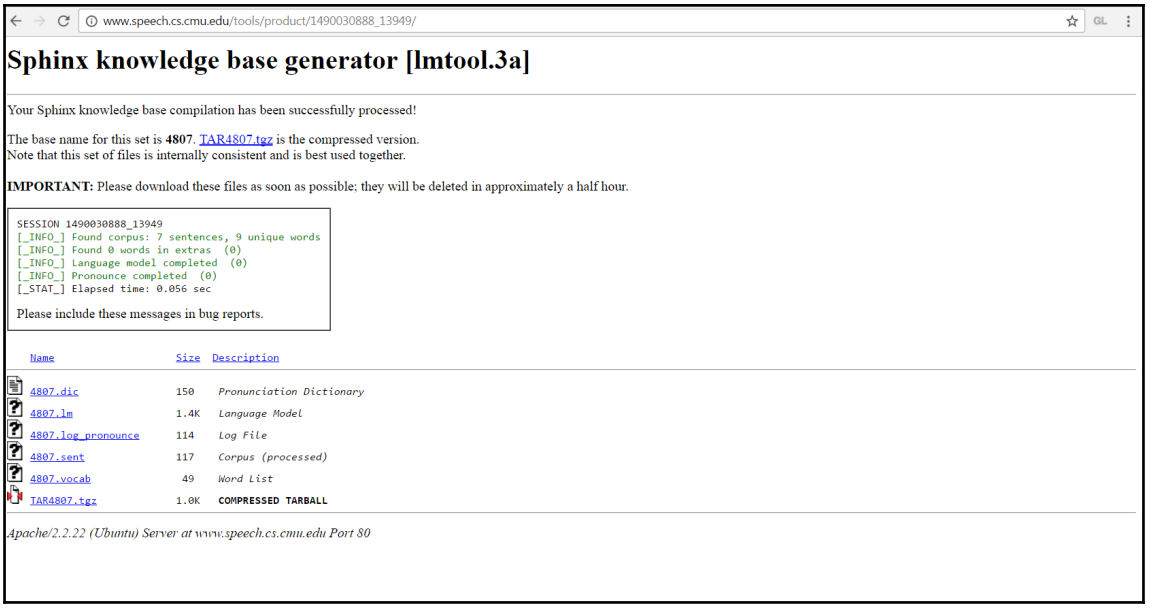
The first step is to create a file with the words or phrases you want the system to recognize. Then, you use a web tool to create two files that the system will use to define your grammar. The first step is to create a file called `grammar.txt` and insert the following text:



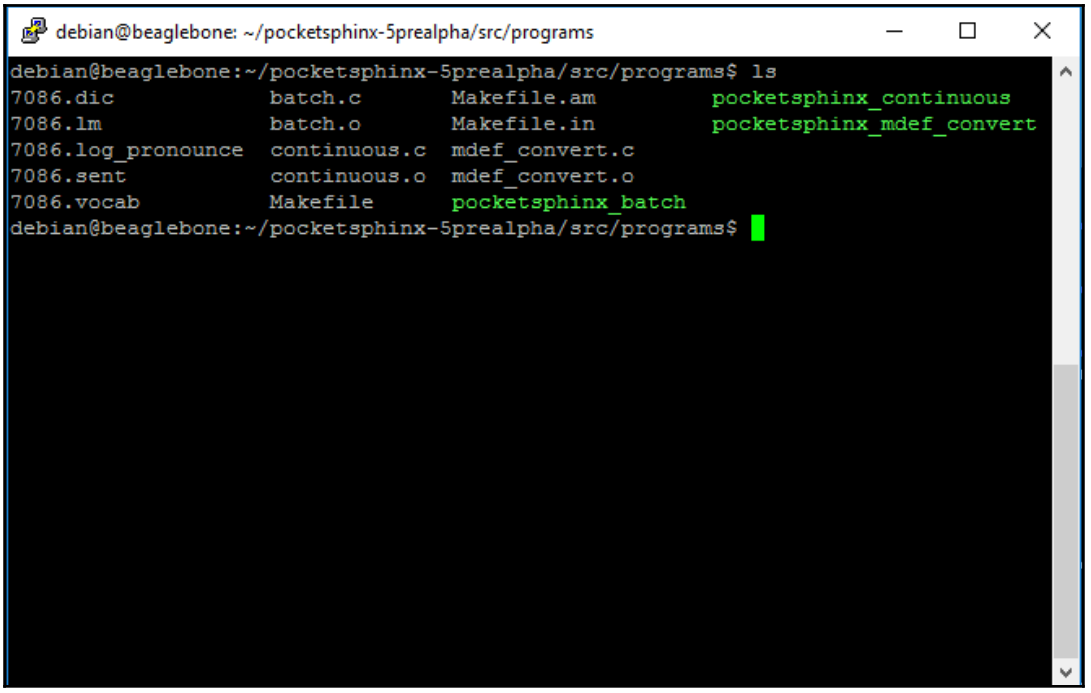
Now you must use the CMU web browser tool to turn this file into two files that the system can use to define its dictionary. Go to a web browser window and type in this URL: `http://www.speech.cs.cmu.edu/tools/lmtool-new.html`. You should see this screen:



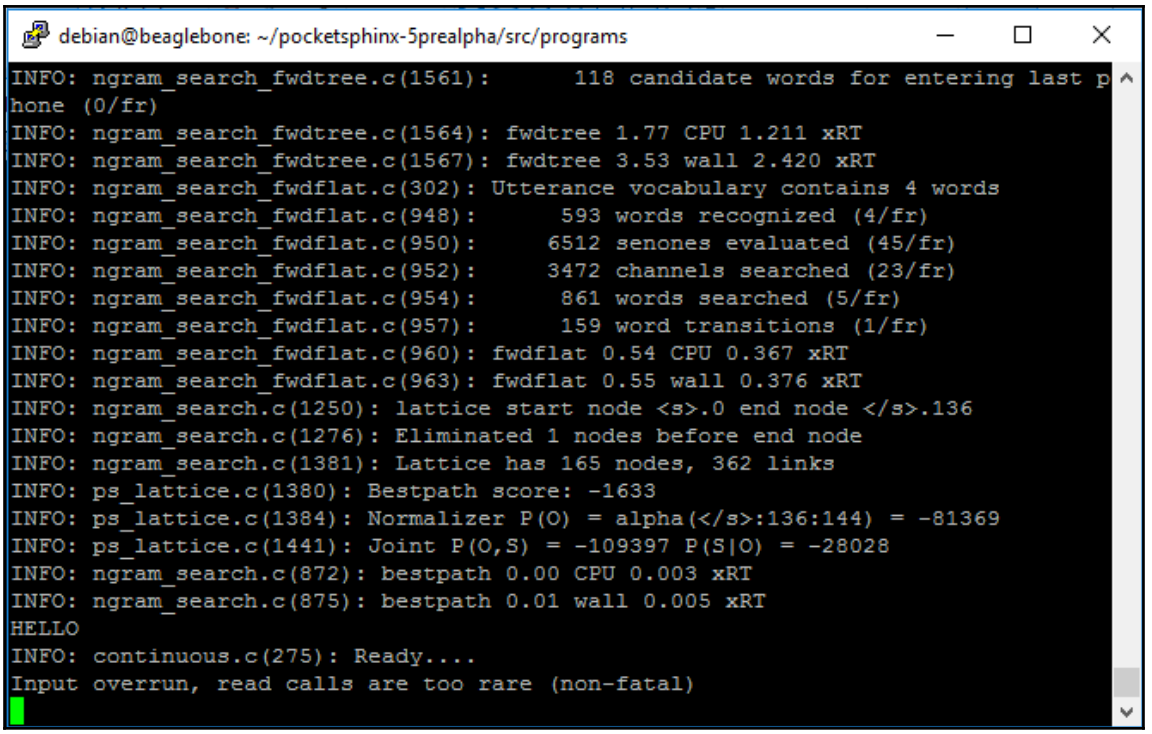
If you hit the **Choose File** button, you can then find and select your file. Now select **COMPILE KNOWLEDGE BASE**, and the following window should pop up:



You need to download the `.tgz` file the tool created, in this case, the `TAR7086.tgz` file, to your BeagleBone Blue. Move it to the `/home/debian/pocketsphinx-5prealpha/src/programs` directory and un-archive it using `tar -xzvf` and the filename. You should end up with the following programs in the directory:



Now you can invoke `pocketsphinx_continuous` to use this dictionary by typing `pocketsphinx_continuous -lm 1565.lm -dict 1565.dic -samprate 16000 -inmic yes`, and it will now look in that dictionary as it tries to find matches to your commands. Saying hello should now result in something like this:

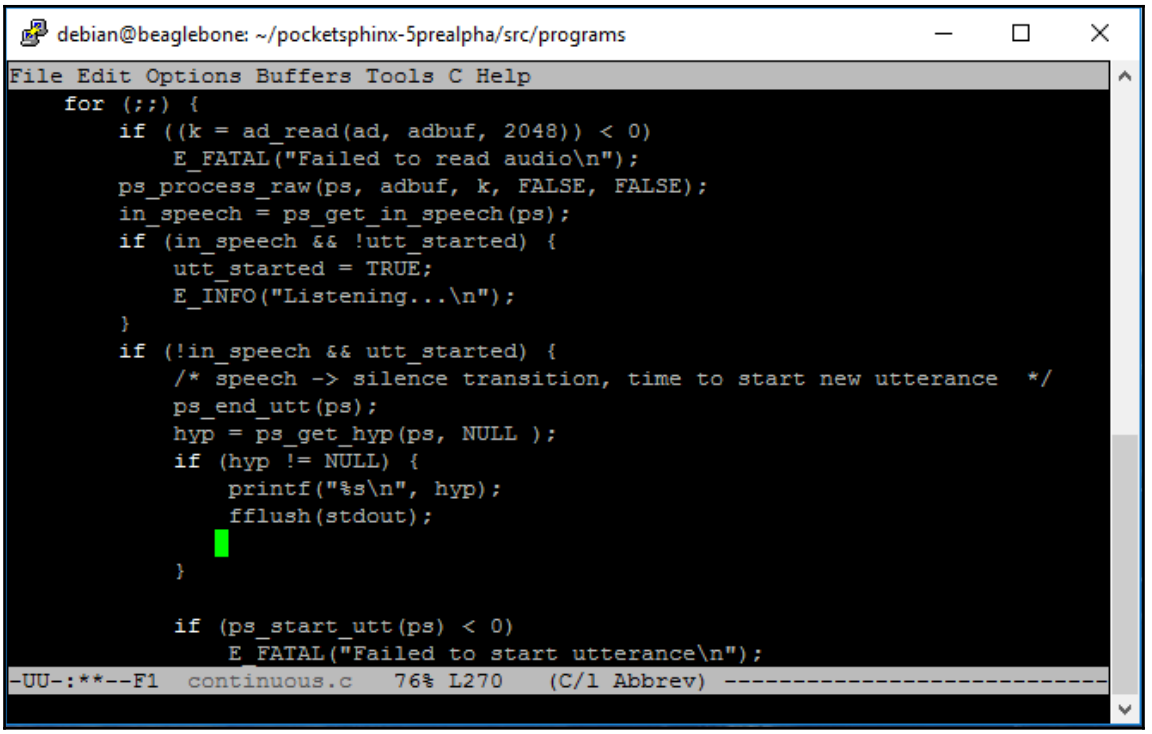


Now that your BeagleBone can actually understand your commands, you can use these to send voice commands to your robot.

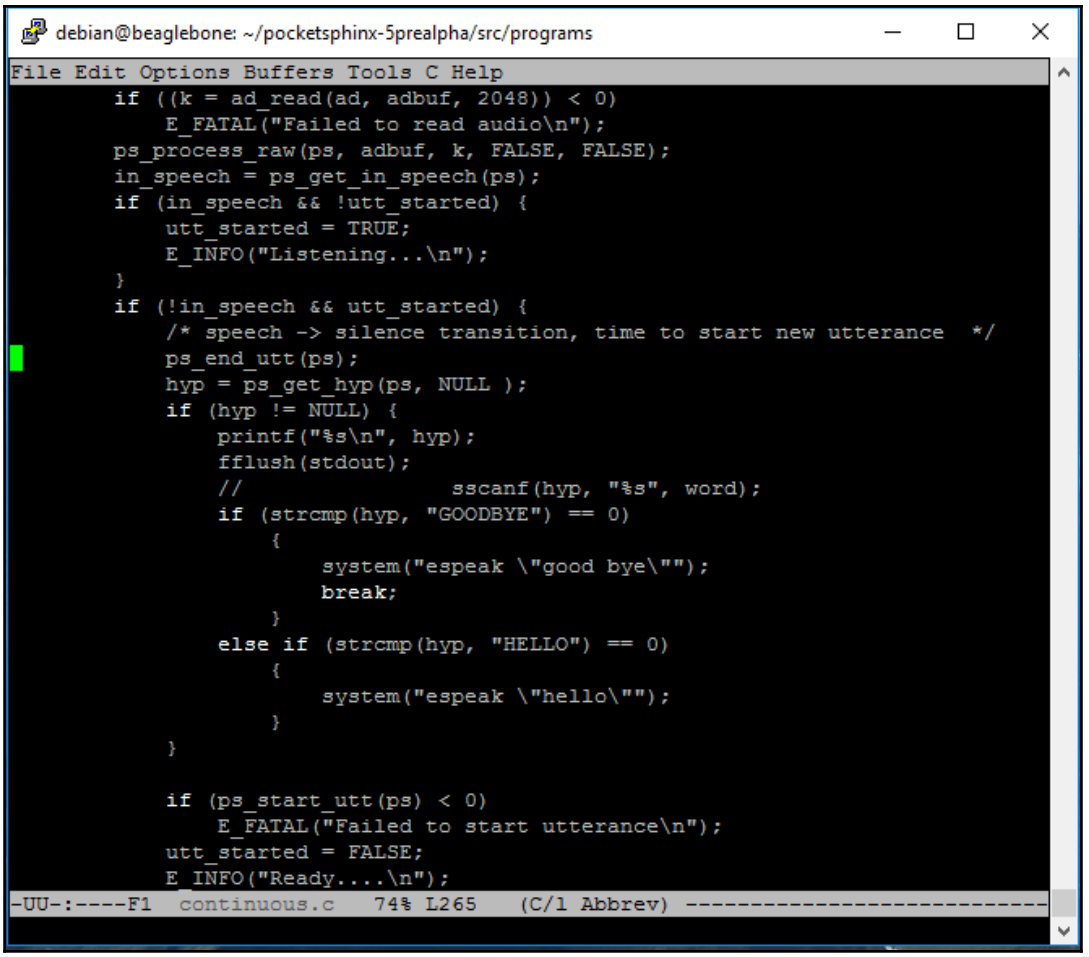
Providing the capability to interpret your commands and have your robot initiate action

Now that your system can both hear and speak, you want to provide the capability to respond to your speech and perhaps even execute some commands based on the speech input. I will assume that you've been successful at each step in this chapter and your BeagleBone black can both speak and listen. You're going to configure the system to respond to your simple commands now.

In order to respond, you're going to edit the `continuous.c` code in the `/home/debian/src/programs` directory. You could create our own `.c` file, but this file is already set up in the `makefile` system and will serve as an excellent starting spot. I like to first make a copy of the current file into `continuous.c.old`. This way, I can always get back to the starting program if it is 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 following section in the code:

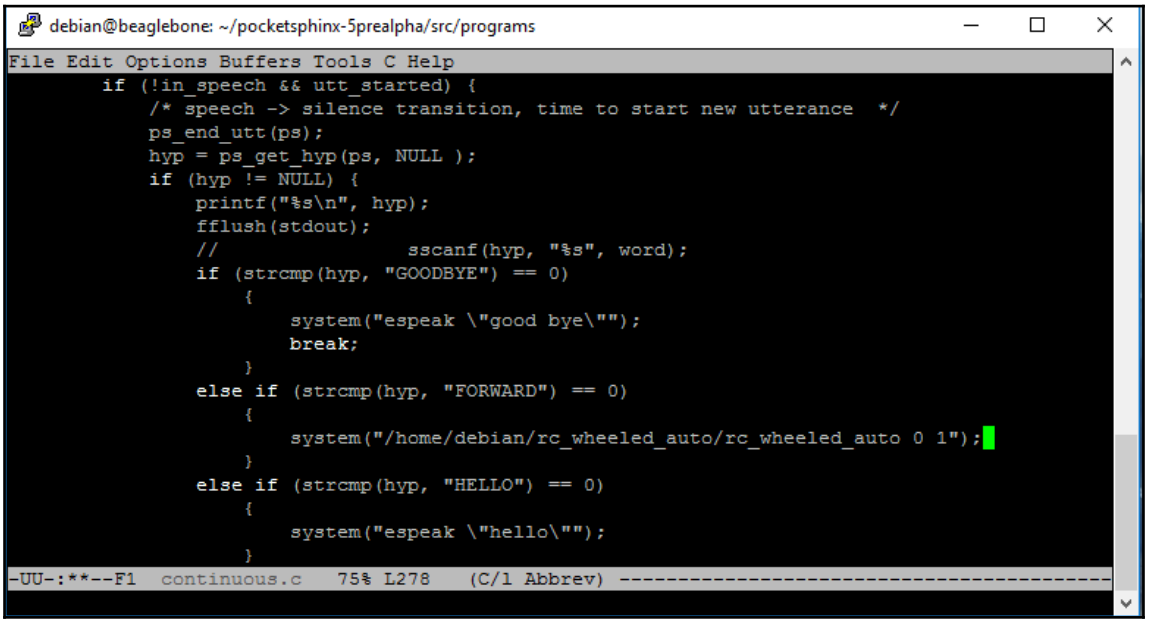


In this section of the code, the word has already been decoded and is held in the `hyp` variable. You can add code here to make your system do things based on the value associated with the word the software has decoded. First, let's try to add the capability to respond to hello and also respond to goodbye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



Now you need to rebuild your code. Since the make system already knows how to build the `pocketsphinx_continuous` program, any time you make a change to the `continuous.c` file, it will rebuild the application. Simply type in `make`. The file will compile and create a new version of `pocketsphinx_continuous`. To run our new version, type in `pocketsphinx_continuous -lm 1565.lm -dict 1565.dic -samprate 16000 -inmic yes`. Make sure you type `./` at the start; if not, the system has another version of `pocketsphinx_continuous` in the library that it will run.

If everything is set correctly, saying hello should result in a response of hello from your BeagleBone Blue. Saying goodbye should elicit a response of goodbye as well as shutting down the program. Note that the system command can be used to actually run any program we might run with a command line. You can now use this to have your program start and run other programs based on the commands. Here is the code to add a forward motion calling the `/root/rc_wheeled_auto/rc_wheeled_auto 0 1` program. You'll want to make sure you are running your `./pocketsphinx_continuous` as the super user by typing `sudo -s` before you run the command. If you get an error message that the system does not recognize your input device, you'll need to create the `.asoundrc` file in the `/root` directory.



Now your BeagleBone Blue will both listen and respond as well as actually execute a command.

Summary

Now your project can both hear and speak. You can use this later when you want to interface with your project without typing commands or using display. Hopefully, you also feel more comfortable installing new hardware and software onto your system. You'll be using this skill throughout the book as you look at even more complex projects.

In the next chapter, we will learn how our project can add the BeagleBone Blue to a quadruped for a legged motion.

7

Making the Unit Very Mobile - Controlling Legged Movement

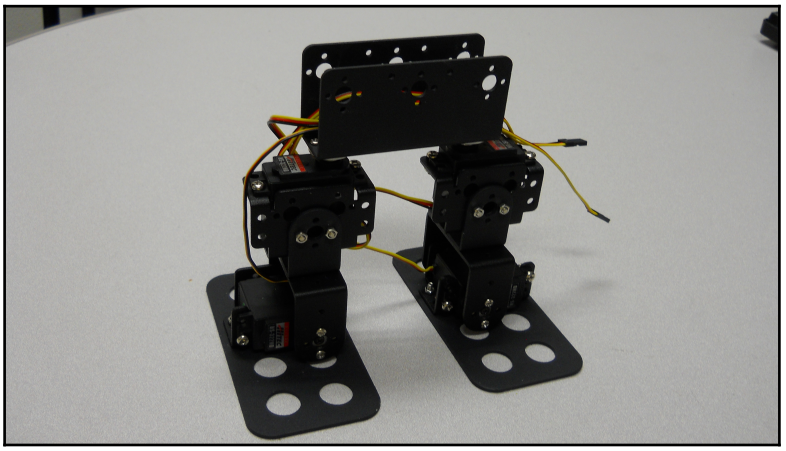
Even though you've learned to make your robot mobile by adding wheels or tracks, this mobile platform will only work well on smooth, flat surfaces. Often, you'll want your robot to work in environments where it is not smooth or flat; perhaps you'll even want your robot to go up stairs or over curbs. 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 the BeagleBone Blue to a two-legged mobile platform using four servos
- 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

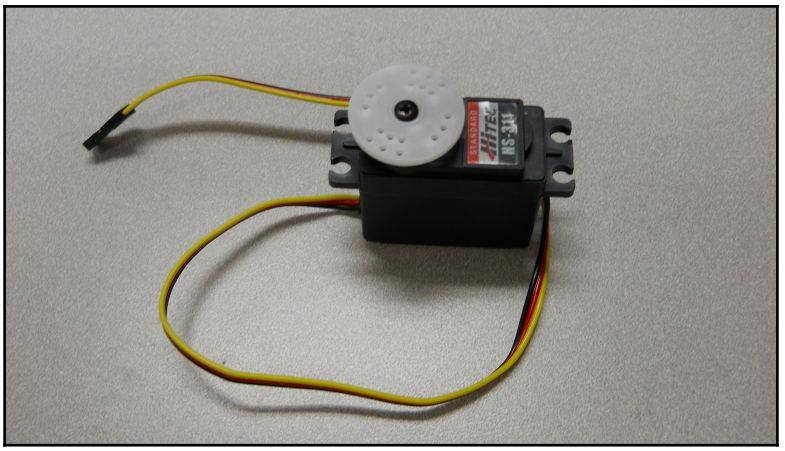
In this chapter, you'll need to add a legged platform to make your project mobile. There are a lot of choices. As before, some are completely assembled, others are "some assembly required," and you may even choose to buy the components and construct your own custom mobile platform. Also, as before, we're going to assume that you don't want to do any soldering or mechanical machining yourself, so let's look at the several choices that are available completely assembled or can be assembled with simple tools (screwdriver and/or pliers).

One of the easiest legged mobile platforms is one that has two legs and four servo motors. Here is a picture of this type of platform:

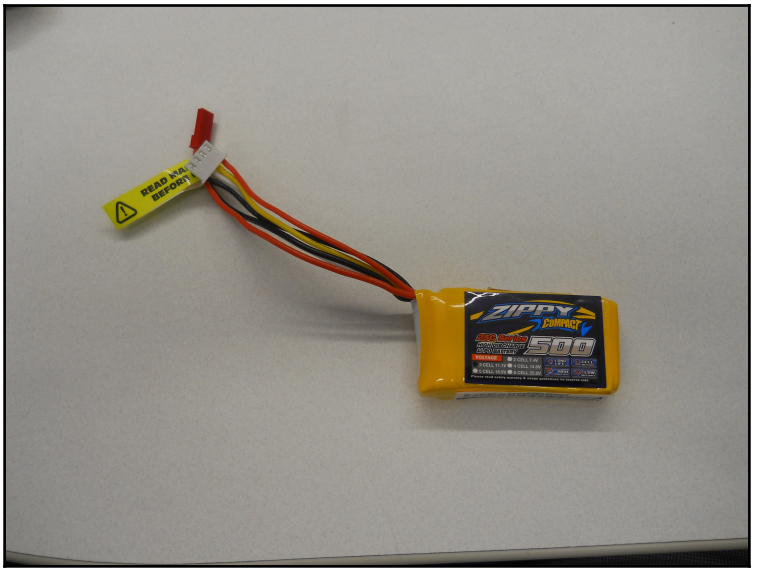


You'll use this platform in this chapter because it is the simplest to program and is the least expensive, requiring only four servos. To construct this platform, you must purchase the parts and then assemble it yourself. The instructions and parts list are at `http://www.lynxmotion.com/images/html/build112.htm`. One of the easiest ways to obtain all the required mechanical parts (except servos) is to purchase a biped robot kit with six **degrees of freedom** (**DOF**); this will contain the parts needed to construct our four-servo biped. These six-DOF bipeds can be purchased from eBay or `http://www.robotshop.com/2-wheeled-development-platforms-1.html`. You won't be able to build and control your six-DOF biped, as this would require the BeagleBone Blue to connect to 12 servos. But at least all the parts are there.

You'll also need to purchase the servos. For this type of robot, you can use standard-size servos. I like the Hitec HS-311 or HS-322 for this robot: they are inexpensive but powerful enough. You can get them on Amazon or eBay. Here is a picture of an HS-322:



As in the last chapter, you'll need a mobile power supply for the BeagleBone Blue. Again, the most useful battery will be a 2S LiPo battery, such as this one:

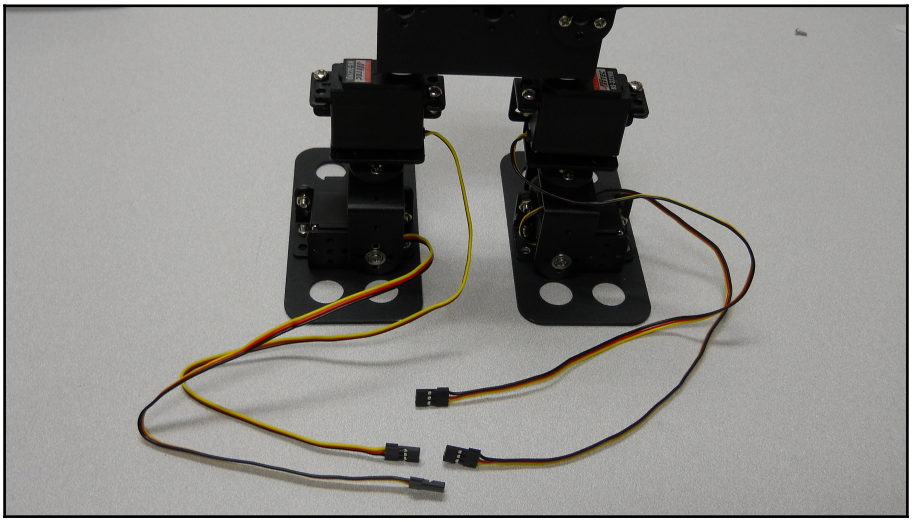


Now that you have all the hardware, let's walk through a quick tutorial of how a two-legged system with servos works and then some step-by-step instructions to make your project walk.

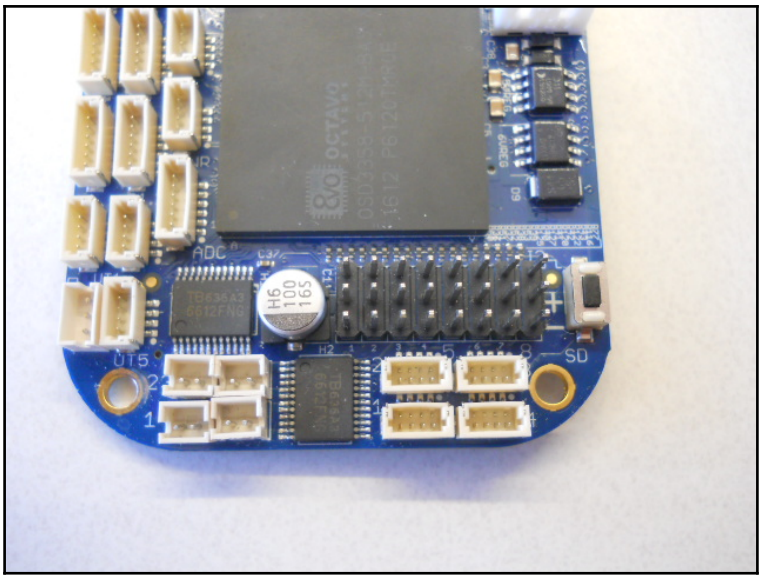
Connecting the BeagleBone Blue to your mobile platform

Now that you have a legged platform and a servo motor controller, you are ready to make your projects 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 and at a given speed, servos are generally designed to move in a limited set of angles. In other words, in the DC motor world, you generally want your motors to spin with a continuous rotation speed that you control, while in the servo world, you want your motor to move to a specific position that you control.

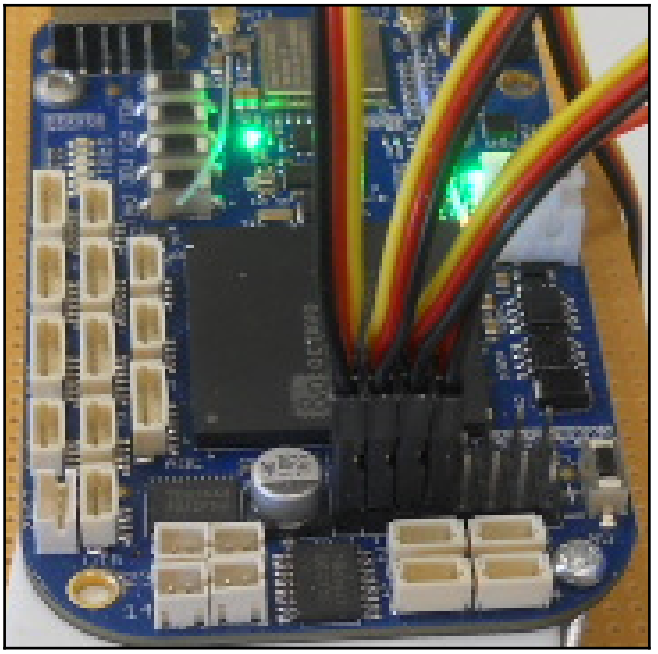
To make your project walk, you first need to connect the BeagleBone Blue to the four servos. Here is a picture of our two-legged robot and the four different servo connections:



In order to be consistent, let's connect your four servos to the connections marked 0 through 3 on the controller using this configuration; 0: left foot, 1: left hip, 2: right foot, and 3: right hip. Here is a picture of the BeagleBone Blue; this will tell you where to connect your servos:



Notice the three pin servo connectors. You'll connect to connections 1, 2, 3, and 4, as shown here:



Now you are ready to control your servos. Start with a simple test command. If you run the `rc_test_servos` program, you'll see the specifics for the test program designed to exercise the servos. Type `rc_test_servos -v -c 1 -p 0.0` and see whether the first servo responds. If it does not, try to change the command to `rc_test_servos -v -c 1 -p 1.0`. You should now be able to control your servos. Let's control them via a Linux program.

Creating a Linux program to control your mobile platform

So you know that you can set your servos. In this section, you'll create a program that will let you talk to your servos a bit more intuitively. You'll issue commands that tell the set of servos to go to specific angles, and they will go to those angles. You can then add a set of such commands to allow your legged mobile robot to lean left, lean right, or even take a step forward.

Let's start with a simple program that will make your legged mobile robot's servos go to 90 degrees (which should be somewhere close to the middle of the 180 degrees we can set. You'll start with the `rc_test_servos` code and modify it. To do this, copy the code to its own directory, along with the `Makefile` for these types of programs. Here is the code:

```
#include "/usr/include/rc_usefulincludes.h"
#include "/usr/include/roboticscape.h"
int main(int argc, char *argv[]){
double servo_pos;
int frequency_hz = 50; // default 50hz frequency to
send pulses
int servo;
int toggle;
servo = atoi(argv[1]);
servo_pos = atof(argv[2]);
if(rc_initialize()){
fprintf(stderr,"ERROR: failed to run rc_initialize(),
are you root?n");
return -1;
}
rc_enable_servo_power_rail();
while(rc_get_state()!=EXITING){
rc_send_servo_pulse_normalized(servo, servo_pos);
rc_set_led(GREEN, toggle);
toggle = !toggle;
rc_usleep(1000000/frequency_hz);
```

```
}
rc_cleanup();
return 0;
}
```

Here is an explanation of the code:

- `#include "/usr/include/rc_usefulincludes.h"` and `#include "/usr/include/roboticscape.h"`: These two lines include the functionality associated with controlling the BeagleBone Blue's servo driver capability.
- `int main(int argc, char *argv[]){`: This line of code is where the program starts execution. To run the code, you'll type in two arguments: the servo and then the desired position for the servo.
- `double servo_pos;`: This variable will hold the value of the servo position.
- `int frequency_hz = 50; // default 50hz frequency to send pulses`: This sets the default frequency of the pulses to 50 Hz. If your particular servo requires a different frequency, change this number.
- `int servo;`: This variable holds the value of the servo you want to move.
- `int toggle;`: These are two variables you'll use in the program.

- `servo = atoi(argv[1])`: This sets the servo to be moved to the first argument on the command line.
- `servo_pos = atof(argv[2]);`: This sets the servo to move the position specified in the command line.
- `if(rc_initialize()){fprintf(stderr,"ERROR: failed to run rc_initialize(), are you root?n");return -1;}`: This initializes the drivers for the BeagleBone Blue, with an error check.
- `rc_enable_servo_power_rail();`: This sets the power rail to 6V for the servos.
- `while(rc_get_state()!=EXITING){`: We keep running this loop until the user asks to exit.
- `rc_send_servo_pulse_normalized(servo, servo_pos);`: This sends the commands to set the servos.

- `rc_set_led(GREEN, toggle);` and `toggle = !toggle;`: This toggles the green LED.
- `rc_usleep(1000000/frequency_hz);`: This delays the commands so that they're sent at the correct frequency.
- `rc_cleanup();`: This cleans up the drivers for the BeagleBone Blue.

Once you have the program entered, you can run it by typing `sudo ./rc_move_servos 1 0.0`. This will set servo `1` to the middle of its range.

Now you can create a Python program that will access your `rc_move_servos` program to allow you to access the different servos. Here is the code for that:

```
#!/usr/bin/python
import os
def set_angle(servo, angle):
angle = angle/60.0
args = (str(servo), str(angle))
os.system("./rc_move_servos " + str(servo) + " " +
str(angle))
pid = os.fork()
servo = 1
while servo:
servo = int(raw_input("Servo: "))
if servo:
angle = float(raw_input("Angle: "))
set_angle(servo, angle)
```

Here is an explanation of these lines of code:

- `#!/usr/bin/python`: This line configures the file so it can be run without adding the Python call on the command line.
- `import os`: This imports the `os` library, the library that allows you to access the `rc_move_servo` command.
- `def set_angle(servo, angle)`: This is function to which you'll pass the servo number you want to control, together with the desired angle.
- `angle = angle/135.0`: This scales the angle to the `+/01.5` degrees that the C program wants.
- `args = (str(servo), str(angle))`: These set the arguments for the program you are going to call with the servo number and angle number.
- `os.system("./rc_move_servos " + str(servo) + " " + str(angle))`: This calls your executable program that moves the servo to the desired angle.
- `pid = os.fork()`: This sets up the system to call the executable in a forked process so that the program can continue without getting hung up waiting for the `rc_move_servos` program to come back.
- `servo = 1`: This sets the default servo.
- `while servo:`: This loops until a `0` is entered for `servo`.

- `servo = int(raw_input("Servo: "))`: Get the servo that is to be moved from the user.
- `if servo:`: If `servo` is not `0`, then go ahead and ask for the angle and then move the servo.
- `angle = float(raw_input("Angle: "))` : Get the desired angle to move to.
- `set_angle(servo, angle)`: Call the `set_angle` function to move the servo.

You can also create a Python program that can actually make the robot move in a coordinated fashion. Here is simple program that will cause the robot to lean to the right:

```
#!/usr/bin/python
import time
import os
def set_angle(servo, angle):
angle = angle/60.0;
args = (str(servo), str(angle))
os.system("rc_test_servos -v -c " + str(servo) + " -p"
+ str(angle) + "&")
set_angle(3, 90.0)
time.sleep(1)
set_angle(5, 30)
time.sleep(2)
os.system("rc_test_servos -v -p 0.0")
```

In this case, you will be using your `set_angle` command to set your servos to manipulate your robot. This set of commands uses the feet to lean to the right.

You can also use a set of commands to lean to the left, or take a step with the right foot, or with the left foot. Here is a table with those commands:

Movement	**Servo movements**

Lean right	`set_angle(3, -70)`
	`tile.sleep(.5)`
	`set_angle(1, -30)`
	`time.sleep(5)`

Lean left	`set_angle(5, 0)`
	`tile.sleep(.5)`
	`set_angle(0, 20)`
	`time.sleep(5)`

Step forward left	`set_angle(3, -70)`
	`time.sleep(.5)`
	`set_angle(1, -30)`
	`time.sleep(.5)`
	`set_angle(2, 30)`
	`time.sleep(.5)`
	`set_angle(3, 0)`
	`time.sleep(.5)`
	`set_angle(1,0)`
	`time.sleep(5)`

Step forward right	`set_angle(1, 70)`
	`time.sleep(.5)`
	`set_angle(3, 30)`
	`time.sleep(.5)`
	`set_angle(4, -100)`
	`time.sleep(.5)`
	`set_angle(1, 0)`
	`time.sleep(.5)`
	`set_angle(3, 0)`
	`time.sleep(5)`

You can create a program for each of these movements; then, in the next section, you can call them from your `continuous.c` program. Here is the code for your robot to take two steps:

```
#!/usr/bin/python
import time
import os
def set_angle(servo, angle):
angle = angle/60.0;
args = (str(servo), str(angle))
 os.system("./rc_move_servos " + str(servo) + " " +
str(angle) + "&")
# Step right
set_angle(5, -70)
time.sleep(.5)
set_angle(3, -30)
time.sleep(.5)
set_angle(4, 30)
time.sleep(.5)
set_angle(5, 0)
time.sleep(.5)
set_angle(3, 0)
time.sleep(2)
# step left
```

```
set_angle(3, 70)
time.sleep(.5)
set_angle(5, 20)
time.sleep(.5)
set_angle(6, -20)
time.sleep(.5)
set_angle(3, 0)
time.sleep(.5)
set_angle(5, 0)
time.sleep(2)
os.system("rc_test_servos -v -p 0.0")
```

Following these principles, you can make your robot do many amazing things: walk forward, walk backward, dance, turn around...any number of movements are possible. The best way to learn is to try new and different positions with the servos.

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. And once you have begun the program, you can't alter its behavior. In this section, you will use the principles from `Chapter 2`, *Programming the BeagleBone Blue*, to issue voice commands to 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 rusty on how this works, review `Chapter 6`, *Providing Speech Input and Output*. We are going to make a simple modification to the `continuous.c` program in `/home/debian/pocketsphinx-5prealpha/src/programs`. To do this, type `cd /homedebian/pocketsphinx-5prealpha/src/programs` and then type `emacs continuous.c`. The changes will be made in the same section as your other voice commands.

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

1. `else if (strcmp(hyp, "FORWARD") == 0)`: This checks the word as recognized by our voice command program. If it corresponds with the word `FORWARD`, you will execute everything inside the `if` statement. We use `{ }` to tell the system which commands go with this `else if` clause.

2. `system("espeak "moving robot"")`: This executes eSpeak, which should tell you that we are about to move our biped robot with the program.
3. `system("/home/debian/biped/step_right.py")`: This is the program you will execute. In this case, your mobile platform will do whatever the program `step_right.py` tells it to do. In this program should be your `set_angle` values to allow your robot to step forward.

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`. The mobile platform will now translate 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 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 moves your robot forward, a different backwards, while another to turn right or left.

Summary

Congratulations! Your robot should now be able to move around in any way you'd like to program. You can even have the robot dance. In the next chapter, you'll learn how to add GPS capability to your robot so it can find its way around.

8

Using a GPS Receiver to Locate Your Robot

Based on the previous projects, you now have mobile robots that can move around, accept commands, see, and even avoid obstacles. This project will help you locate your robot while it moves, which can be useful for a robot that is fully autonomous. The robot is mobile, but let's not let it get lost. You're going to add a GPS receiver so that you can always know your location.

In this chapter, you will learn:

- How to connect the BeagleBone Blue to a GPS device
- How to access GPS programmatically and determine how to move to a location

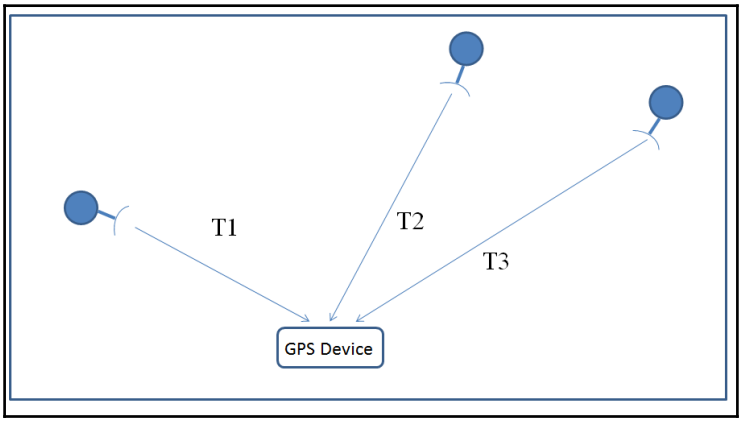
In this project, you'll need a GPS device. There are a lot of options and they come with many different interfaces, but because we want to avoid using a soldering iron or other complex connection processes, we're going to choose one with a USB interface. Here is an image of a device I have used for some of my projects:

The model number is **ND-100S** from **GlobalSat**. It is small, inexpensive, and supports Windows, Apple, and Linux, so your 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 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.

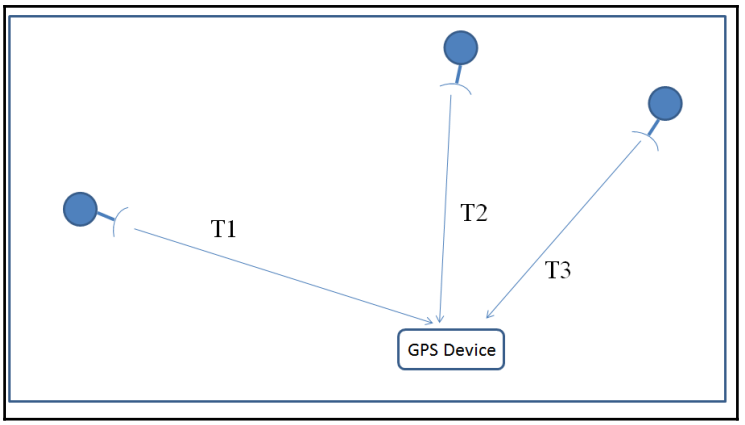
Connecting the BeagleBone Blue to a GPS device

Unpack your GPS device; it is time to get started. 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. 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.

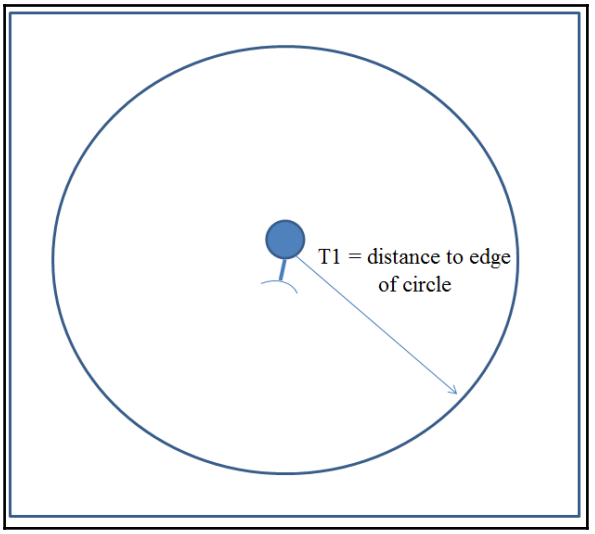
Each of these satellites transmits a very accurate time signal that your device can receive and interpret. It receives the time signal from each of these satellites, and then based on the delay, that is the time it takes the signal to reach the device, it calculates the receiver's position based on a procedure called triangulaion. The following two diagrams illustrate how the device uses the delay differences from three satellites to calculate its position:



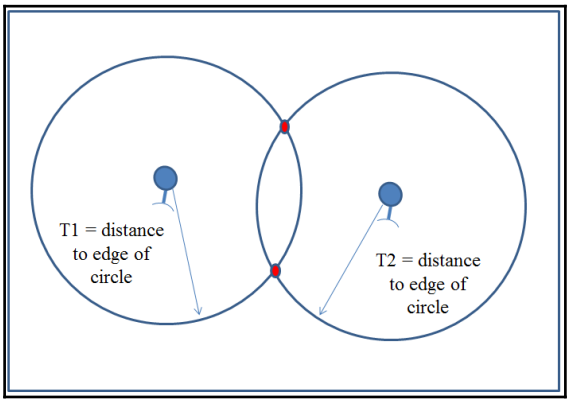
The GPS device is able to detect the three signals and the time delays associated with receiving these signals. In the following diagram, the device is at a different location and the time delays associated with the three signals have changed:



The time delays of the signals **T1**, **T2**, and **T3** can provide the GPS with an absolute position, using a mathematical process called **triangulation**. Since the position of the satellite is 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, let's show an example in two dimensions. If the GPS device knows one distance to a satellite, based on the amount of time delay, you can draw a circle around the satellite at that distance and know that your GPS device is on that sphere, as shown in the following diagram:



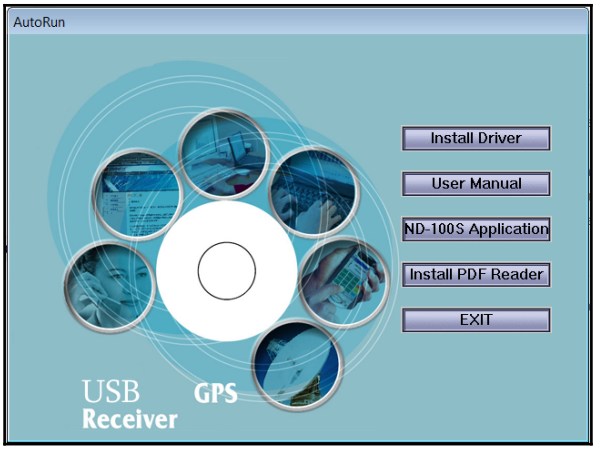
If you have two satellite signals and know the distance between the two, you can draw two circles, as shown in this next diagram:



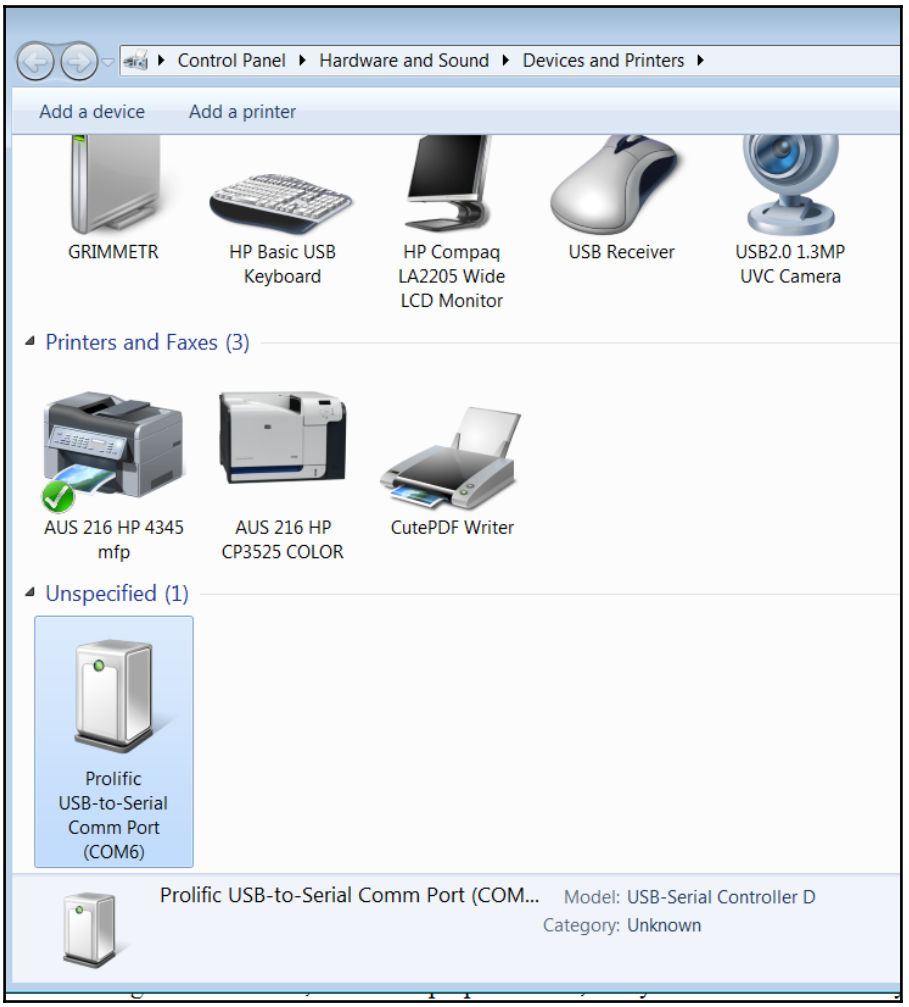
However, since you know that you can only be at points on the circle, you must be at one of the two points that are on both circles. Adding an additional satellite would eliminate one of these two points, providing you with an exact location. You need more satellites if you are going to do this in all three dimensions.

Now, it is time to connect the device. As a first step, I suggest you connect the dongle to your PC. This will let you know that the unit works and will help you understand the device a little better. Later you'll connect it to the BeagleBone Blue.

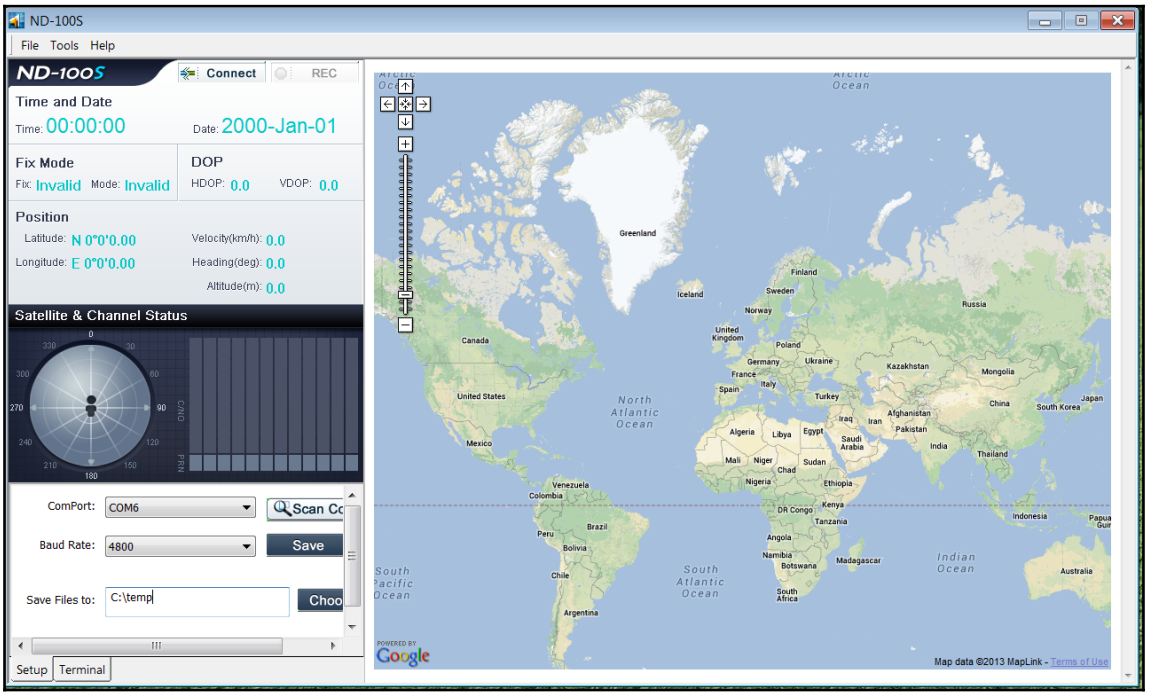
In order to install the system on your PC, insert the CD and run the setup program. You should see something like this:



Click on both the **Install Driver** and **ND-100S Application** buttons and follow the default instruction procedures. Once 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 would indicate that the device has been plugged in. The system will recognize the device and install the proper drivers, and you will have access to your device (this may take a few minutes). To ensure that the device has been installed, check your **Devices and Printers** start menu selection (if running Windows 7). You should see the following:



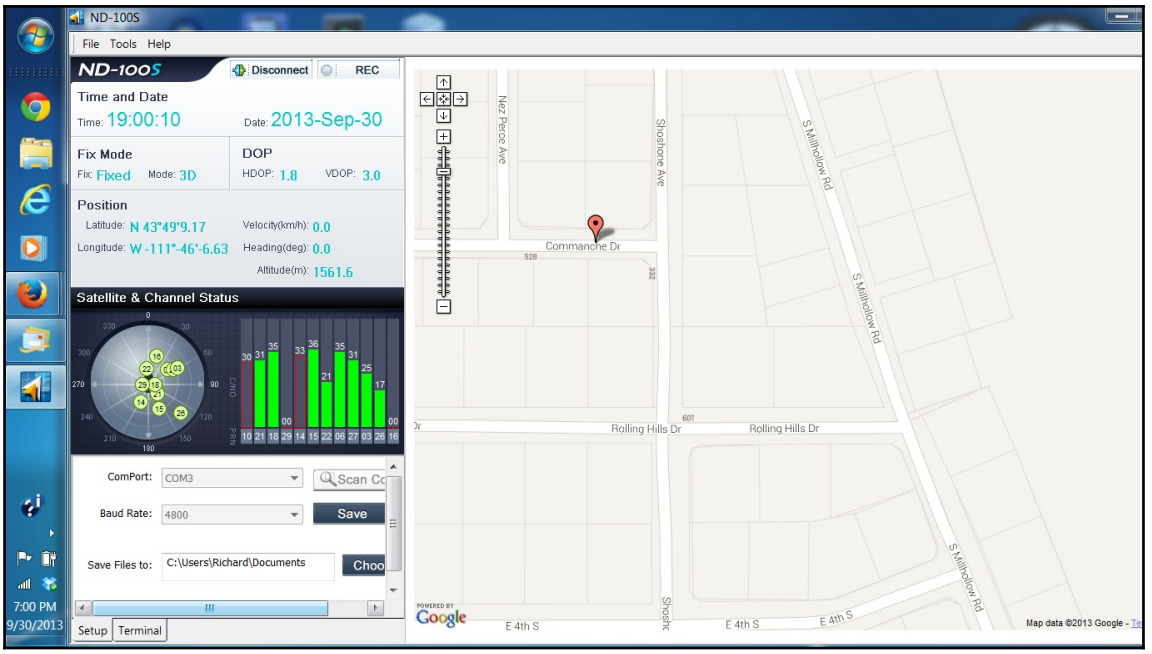
Once the device is installed, you can also run the application that comes on the CD-ROM. On startup, it should look something like following:



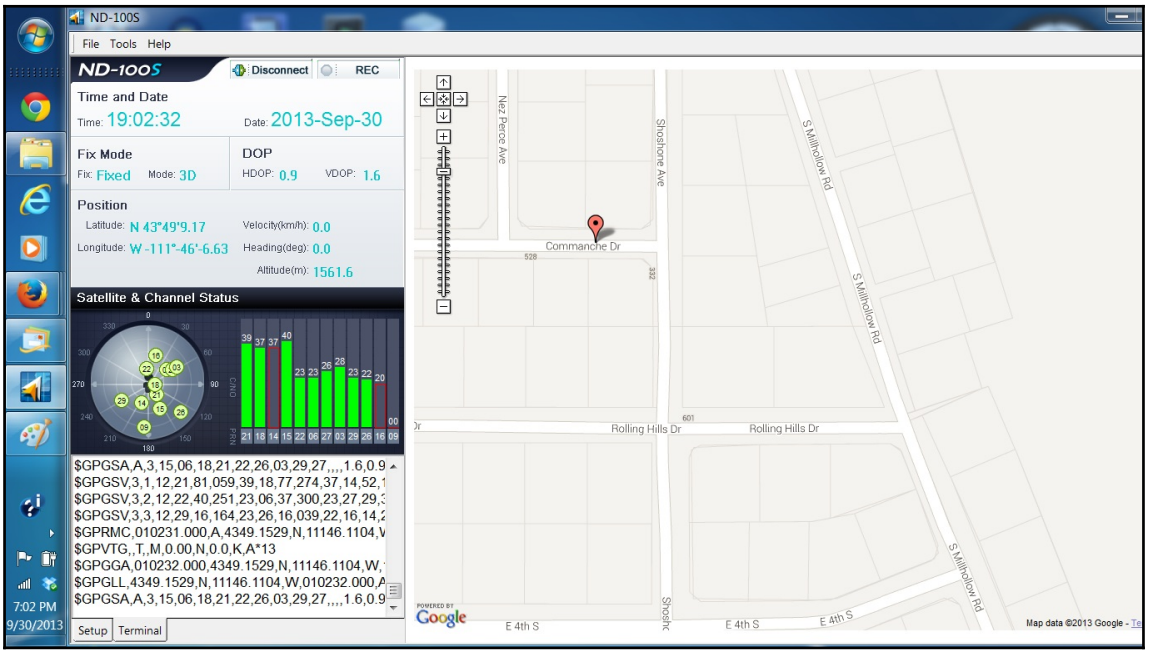
Now press the **Connect** selection button on the top-left of the screen. You should now see the following:

Unfortunately, if you are in a building or in a place where receiving information from the GPS satellites is difficult, the device may struggle to find its position. If you want to know that the system is working, even though it may struggle to find signals, select the **Terminal** tab selection in the lower-left corner. You should see something like the following:

Note that the lower-left window indicates the device trying to find its location. Initially, the unit in my office was unable to locate the satellites; not surprising for a building designed to restrict the transmission of signals into and out of the building. Following the same procedure on my laptop shows the following:

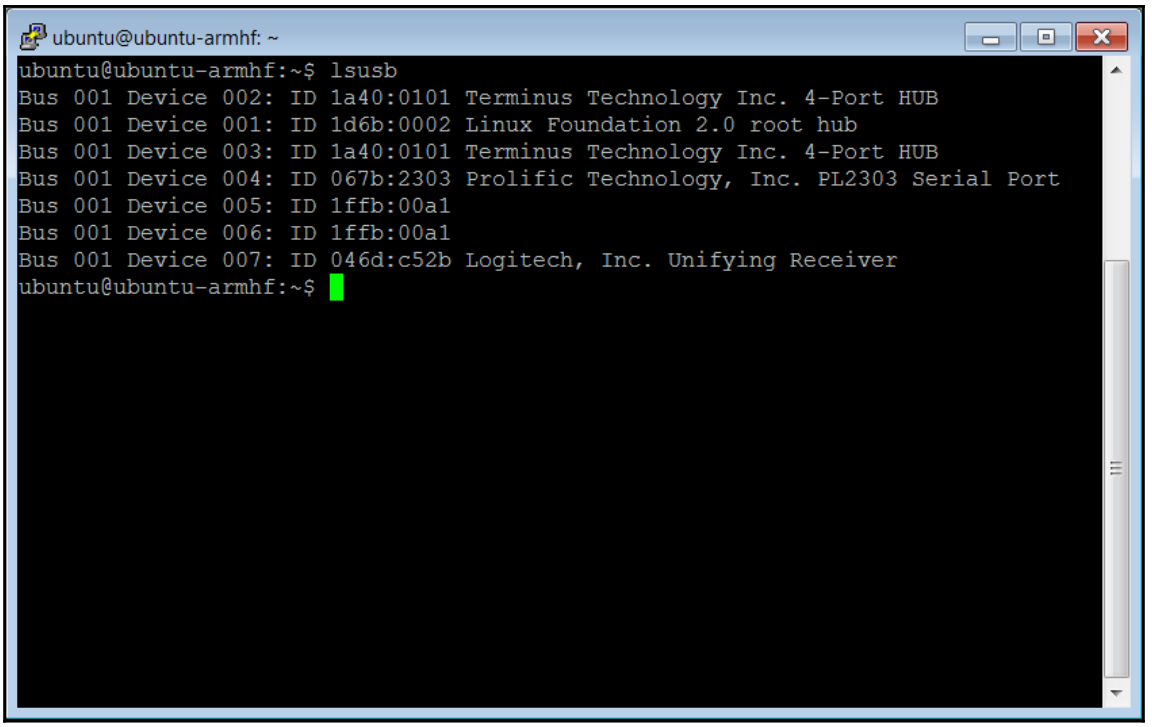


You'll note the blue LED at the end of the GPS flashing. Now you have a position. If you select the **Terminal** tab, it would show the raw data coming back from the GPS:



You'll use this raw data in your next section to plan your path to other positions. So, in an environment where GPS data is available, the unit is able to sync up and show your position. The next step is to hook it to your BeagleBone Blue 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 following:



The device is shown as `Prolific Technology, Inc. PL2303 Serial Port`. Your device is now connected to your BeagleBone Blue.

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 following code:

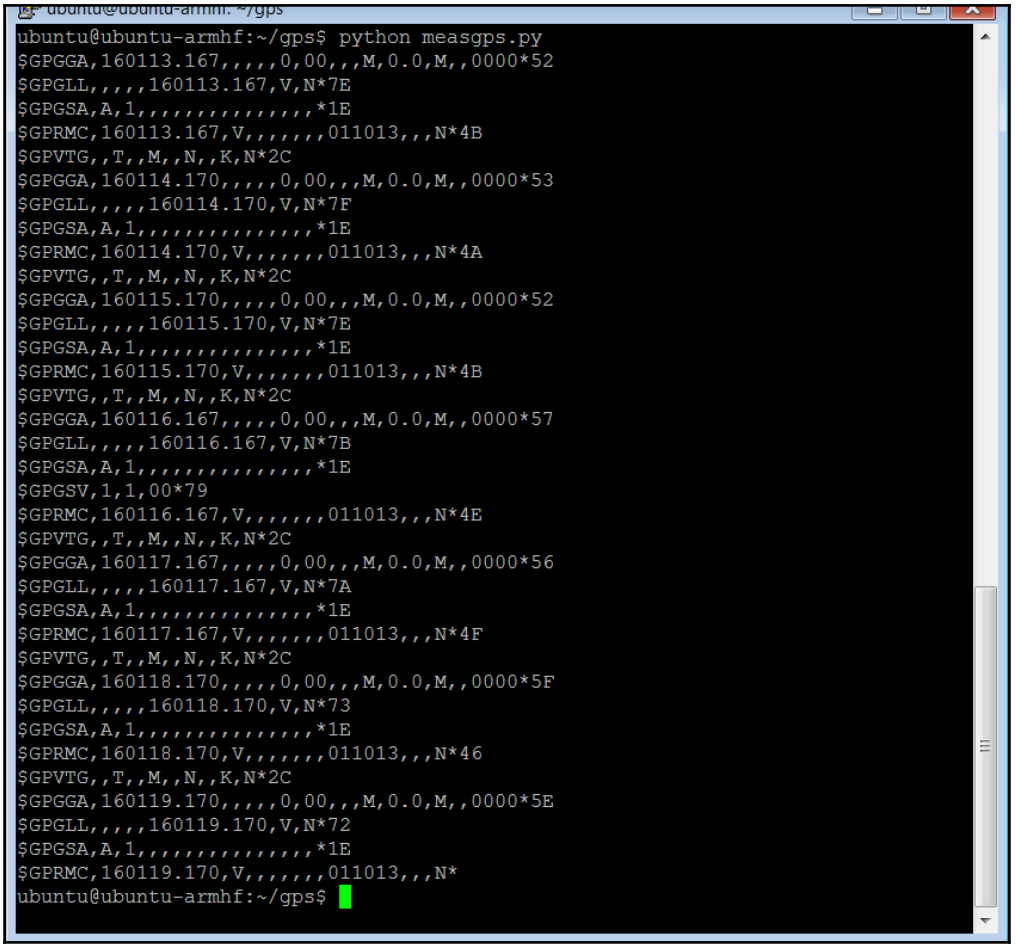
```
#!/usr/bin/python
   import serial
   ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)
    x = ser.read(1200)
    print(x)
```

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

- `#!/usr/bin/python`: This line simply makes this file available for us to execute from the command-line.
- `import serial`: We import the `serial` library. This allows us to interface the USB GPS sensor.

- `ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)`: This command sets up the serial port to use the `/dev/ttyUSB0` device, which is our GPS sensor using a baud rate of `4800` and a timeout of `1`.
- `x = ser.read(1200)`: This command then reads in a set of values from the USB port. In this case, we read `1200` bytes, which includes a fairly full set of our GPS data.
- `print x`: This final command then prints out the value.

Once you have this file created, 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 something like the following:



The device is providing raw readings back to you, which is a good sign. Unfortunately, there isn't much good data here, as the unit again is inside. How do we know this? Look at one of the lines that starts with `$GPRMC`. This line should tell you your current latitude and longitude values. The GPRS is reporting:

```
$GPRMC,160119.170,V,,,,,,,011013,,,N*
```

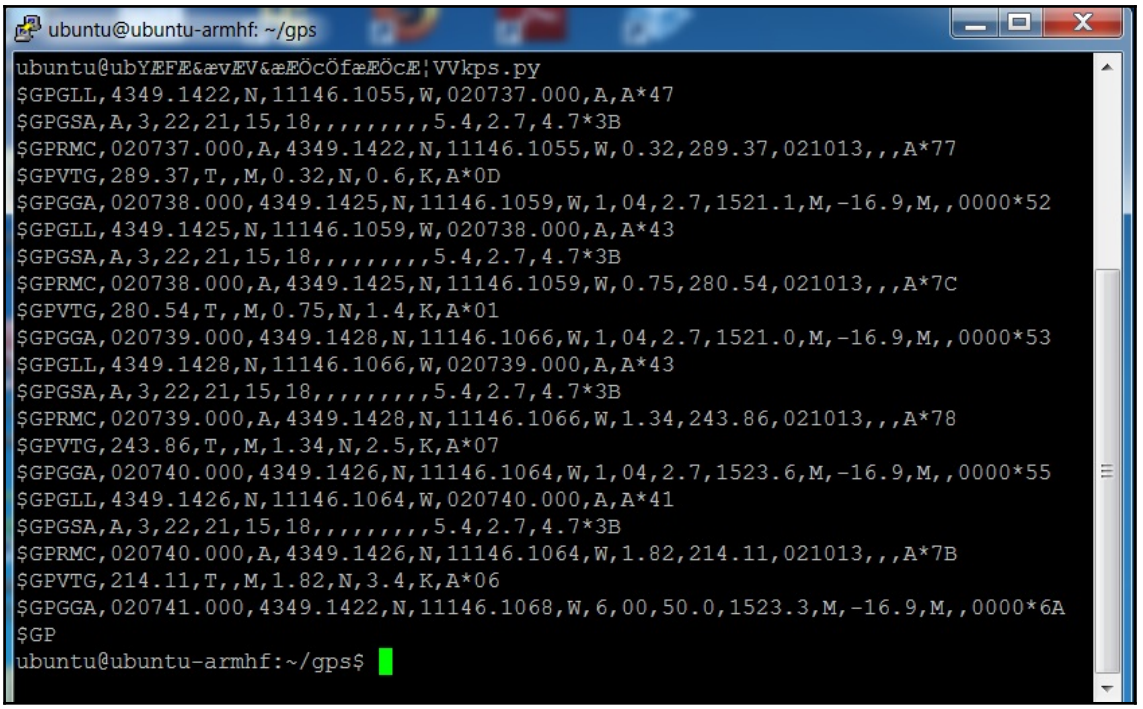
This line of data should take the following form, with each field separated by a comma:

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

Following is a table providing explanation of each of these fields:

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 with which the device is moving
8	`3`	Course: The angle direction in which the device is moving
9	`130694`	Datestamp
10	`0004.2`	Magnetic variation: Variation from magnetic and true north
11	`W`	East or west
12	`*70`	Checksum

In our case, field 2 is reporting `A`, or that the unit cannot find enough satellites to get a position. Taking the unit outside, we can get something like the following on our `measgps.py`:



Note that the `$GPRMC` line now reads as follows:

```
$GPRMC,020740.000,A,4349.1426,N,11146.1064,W,1.82,214.11,021013,,,A*7B
```

Our values are now:

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 with which the device is moving
8	`214.11`	Course: The angle direction in which the device is moving
9	`021013`	Datestamp
10		Magnetic variation: Variation from magnetic and true north
11		East or west
12	`*7B`	Checksum

Now you have some indication of where you are. However, it is in a raw form that may not mean much. In the next section, you will figure out how to do something with these readings.

Accessing the GPS programmatically and determining how to move to a location

Now that you can access your GPS device, let's work on accessing the data programmatically. Your project should now have the GPS connected and have access to querying 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 completed the last section, you should be able to receive raw data from the GPS unit. Now you want to be able to take this data and do something with it; for example, find your current location and altitude, and then decide if your target location is to the west, east, north, or south.

First, get the information out of the raw data. As noted earlier, your position and speed are in the `$GPMRC` output of your GPS. Write a program to simply parse out several important pieces of information from that data. So open a new file (you can name it `location.py`) and edit it as follows:

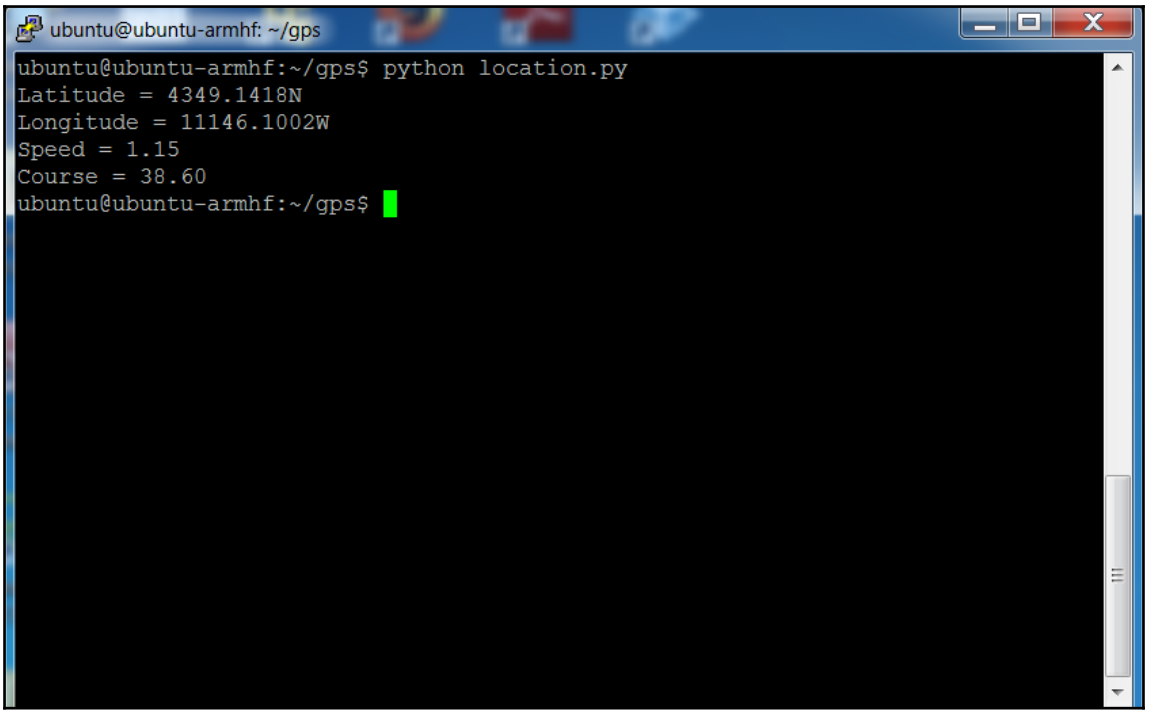
```
#!/usr/bin/python
    import serial
    ser-ser.Serial('/dev/ttyUSB0', 4800, timeout = 1)
    x = ser.read(500)
    pos1 = x.find("$GPRMC")
    pos2 = x.find("n", pos1)
    loc=x[pos1:pos2]
    data = loc.split(',')
    if data[2] == 'V':
```

```
   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]
```

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

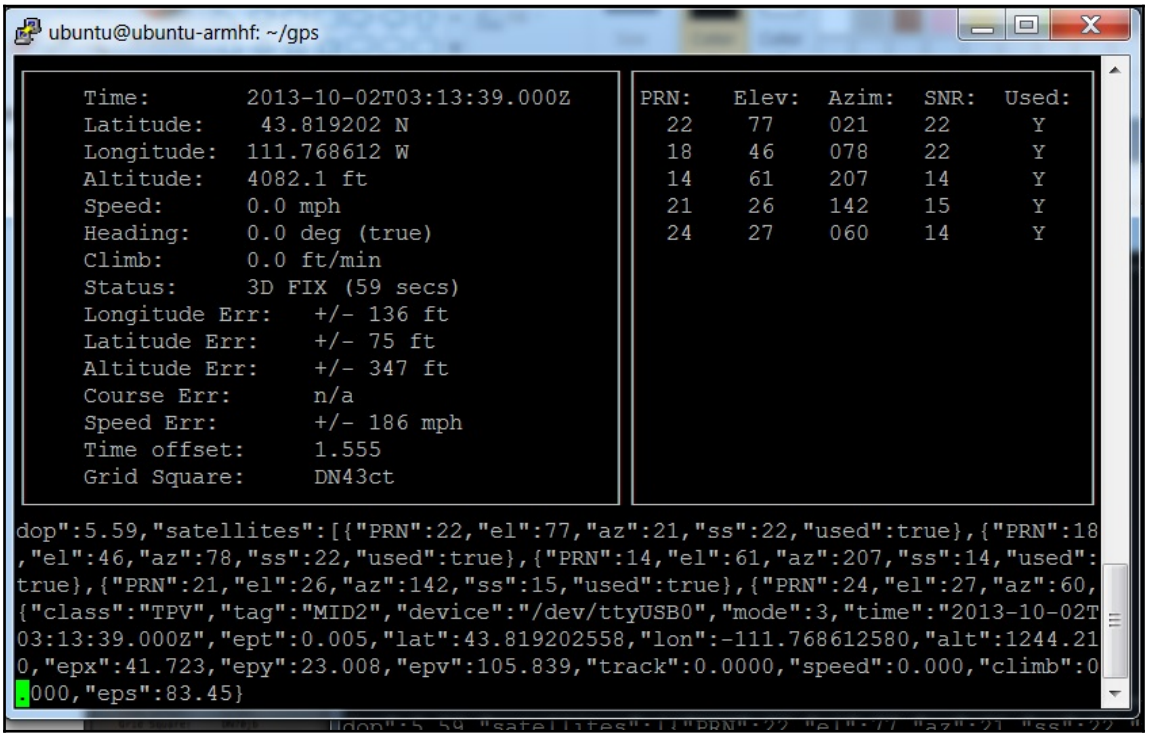
- `The#!/usr/bin/Python`: As always, this line simply makes this file available for you to execute from the command line.
- `import serial`: You again import the `serial` library. This allows you to interface the USB GPS sensor.
- `ser = serial.Serial('/dev/ttyUSB0', 4800, timeout = 1)`: The first command sets up the serial port to use the `/dev/ttyUSB0` device, which is your GPS sensor using a baud rate of `4800` and a timeout of `1`.
- `x = ser.read(500)`: This command then reads in a set of values from the USB port. In this case, you read `500` values, which includes a fairly full set of your GPS data.
- `pos1 = x.find("$GPRMC")`: This finds the first occurrence of `$GPRMC` and sets the value of `pos1` to that position. In this case, you want to isolate the `$GPRMC` response line.
- `pos2 = x.find("n", pos1)`: This finds the end of this line of data.
- `loc = x[pos1:pos2]`: The variable `loc` will now holds the line with all the information you are interested in.
- `data = loc.split(',')`:=This will take your comma-separated line and break it into an array of values.
- `if data[2] == 'V'`: You now check to see if 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.

Here is a screenshot showing the results when my device was able to find its location:



Once you have the data, you can do some interesting things with it. For example, you might want to figure out the distance 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 distance and bearing to other waypoints based on your current location. You can easily add this code to your `location.py` file to update your robot on the distance and bearing to other waypoints.

Now your robot knows where it is and the direction it needs to go to, 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 a set of functionality held in the `gpsd` library. To install this capability, type `sudo apt-get install gpsd gpsd-clients`; this will install the `gpsd` SW. This SW works by starting a background program (called `daemon`) that communicates with your GPS device. You can then just query the program to get the data. To make sure this works, type `cgps` and a program that was installed with the `gpsd` library would open, and you should see the following:



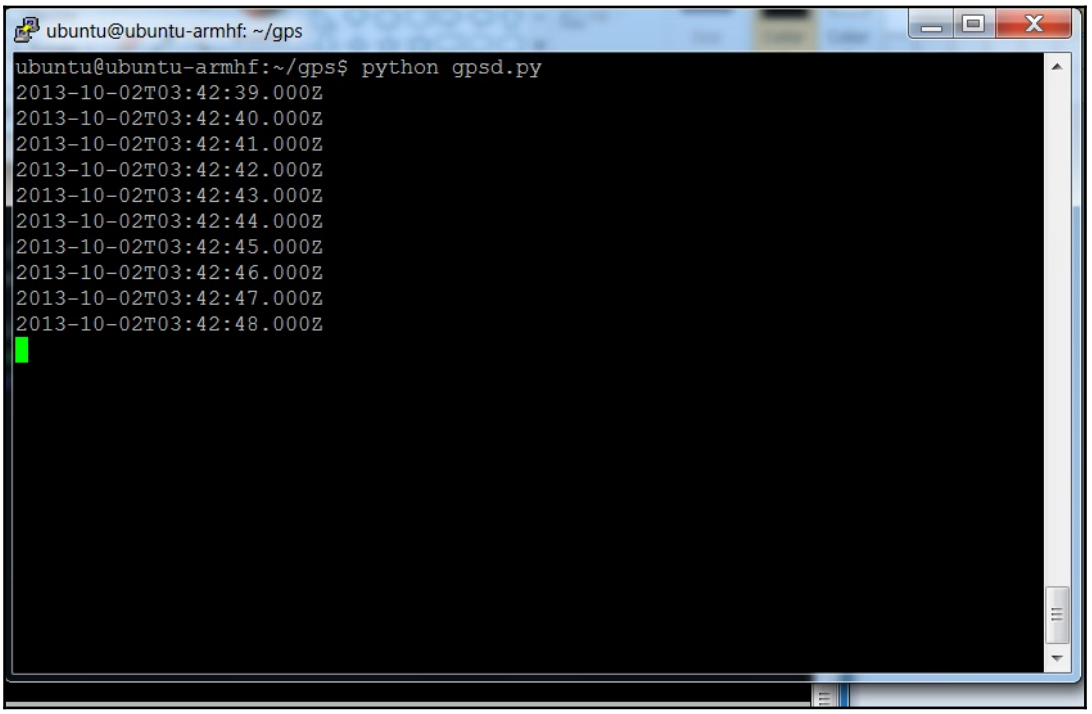
This displays both the formatted data and some of the raw data that is coming from the GPS sensor. We can also access this information from a program. To do this, edit a new file called `gpsd.py` with the following code:

```
#!/usr/bin/python
    import serial
    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
```

Here are the details of your code:

- `#!/usr/bin/Python`: As always, the first 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 allows you to access the `gpsd` functionality.
- `session = gps.gps("localhost", "2947")`: This opens a communication path between the `gpsd` functionality and our program. This opens port `2947`, assigned to the `gpsd` functionality on the local host.
- `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 (by typing `Ctrl C`).
- `report = session.next()`: When a report is ready, it goes into the variable `report`.
- `if report['class'] == 'TPV':`: This checks to see if the report will give you the type of report that you need.
- `if hasattr(report, 'time'):`: This makes sure that `report` holds time data.
- `print report.time`: This prints out the time data. I use this in my example because it is always returned, even if the GPS is not able to see enough satellites to return position data. To see other possible attributes, see `http://www.catb.org/gpsd/gpsd_json.html` for details.

Once you have created the program, you can run it by typing `python gpsd.py`. The following is a possible output of running the program:



Congratulations! Your robot can now get around without getting lost. You can use the information to plan routes to different waypoints and track where your robot has been. One of the ways to display positional information is to use 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. An excellent tutorial on this is available at `https://www.sparkfun.com/tutorials/403`. You won't need to execute the HW configuration part of the tutorial, but will be able to start with the section *Read a GPS and plot position with Python.*

Summary

Now that you have a GPS attached, you can identify where your robot is and plan a path to a new location. In the next chapter, you'll learn how to build robots that can fly, sail, and go under the water.

9
By Land, By Sea, By Air

You've spent lot of time building robots that can navigate on land; now let's look at the possibilities for robots that can navigate in the air or on the water. By now I hope you are comfortable accessing the USB control channels, and talking with other hardware such as servo controllers and other devices that can communicate over USB. Instead of leading out through each step, in this chapter I'm going to point you in the right direction and 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. I hope you are now ready to explore a bit on your own, for these projects can be quite complex, and I'm not going to lead you through each step. Rather, I'm going to try and point you in the right direction and then include lots of links to others who are working in these areas.

In this chapter, you will learn:

- How the BeagleBone Blue can be used in robots that can sail
- How the BeagleBone Blue can be used in robots that can fly
- How the BeagleBone Blue can be used in robots that can go under the water

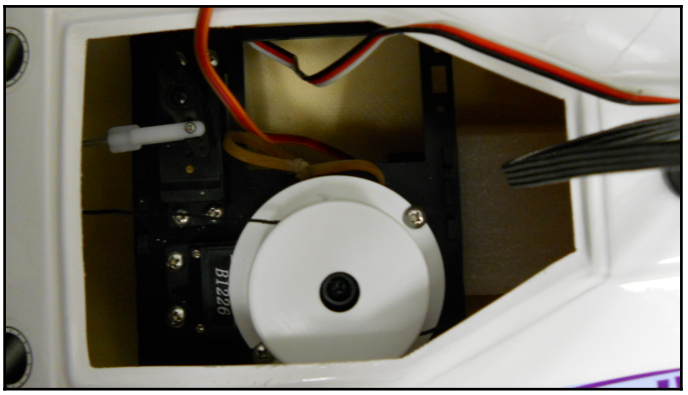
You're going to need to add HW to your robotics in order to complete these projects. Since the hardware is different for each of these projects, I'll introduce the hardware in each individual section.

The BeagleBone Blue and robots that can sail

Now that you've discovered that the BeagleBone Blue can guide platforms that can move on land, let's turn to a completely different type of mobile platform: - one that can sail. In this section, you'll discover how to add the BeagleBone Blue to a sailing platform and utilize it to control your sail boat.

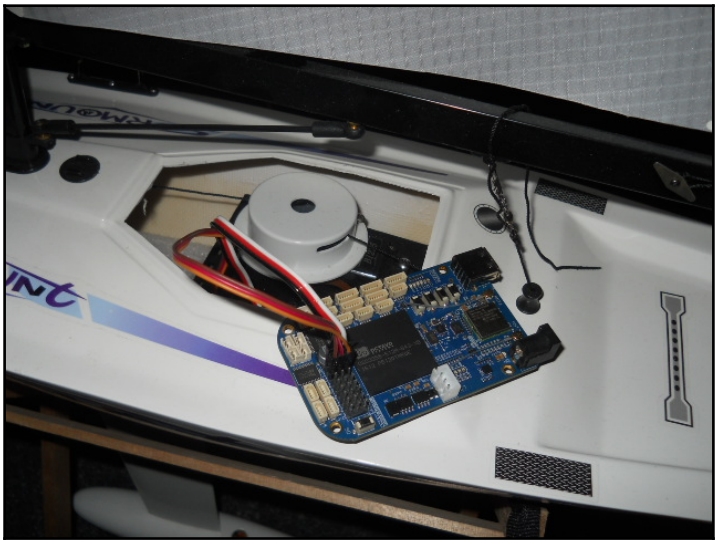
Fortunately, sailing on the water is about as simple as walking on land. First, however, you need a sailing platform. Here is a picture of an RC sailing platform that can be modified to accept control from the BeagleBone Blue:

In fact, many RC controller boats can be modified to add the BeagleBone Blue. All you need is space to put the processor, the battery, and any additional control circuitry that you might need. In this case, the sailing platform has basically two controls: a rudder that is controlled by a servo and a second servo that controls the position of the sail. These are shown next:



To automate the control of the sail boat, you need your BeagleBone Blue and a LiPo battery. As you learned in `Chapter 7`, *Making the Unit Very Mobile - Controlling Legged Movemen*t, the BeagleBone Blue can control servos directly.

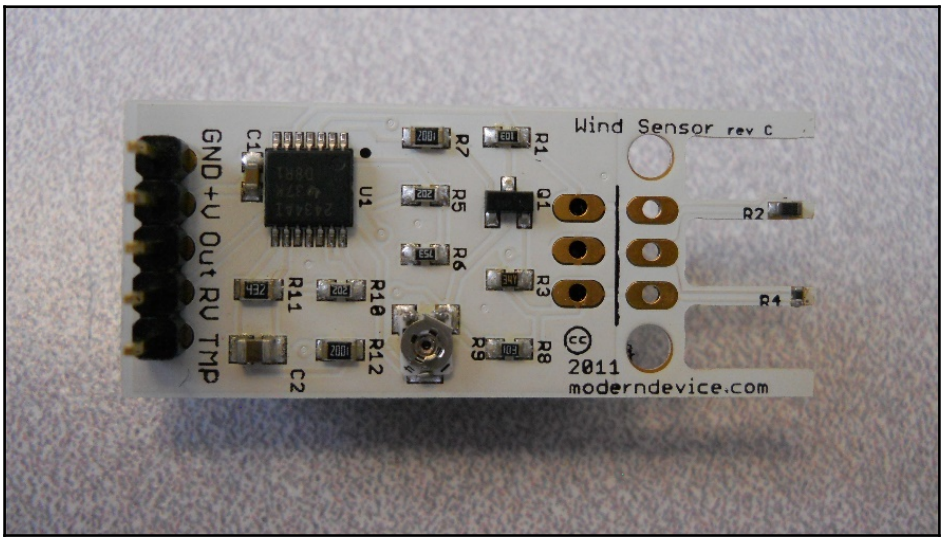
Once you have assembled your sail boat, you need to first hook up the BeagleBone Blue controller to the servos on the boat. You should try these outside the boat first; something like this:



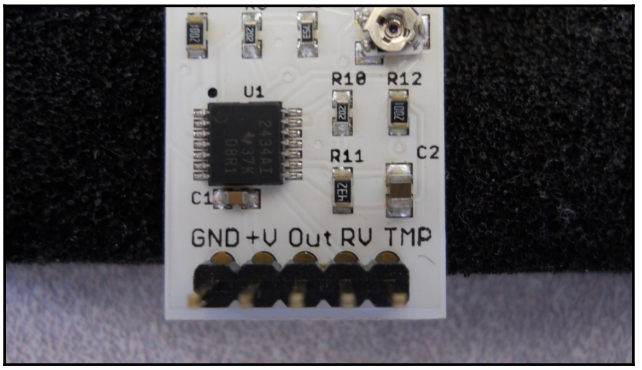
Just as in `Chapter 7`, *Making the Unit Very Mobile - Controlling Legged Movemen*t, you can use code to control the servo. You may also want to be able to sense when the wind is blowing. There is a device that you can add to your sailboat that will give you a wind reading.

Connecting an analog airspeed sensor

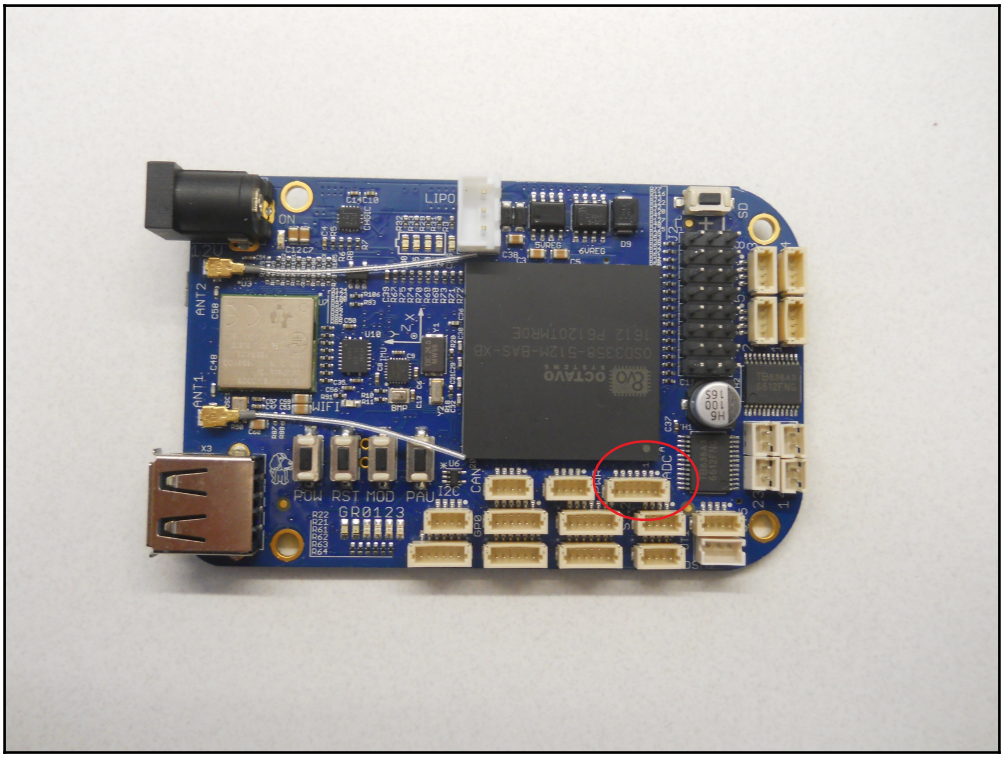
Since you are going to use wind as your power source, you'll want to know both the direction and strength of the wind. You can do this with an analog wind sensor. Here is a picture of a wind sensor from `https://moderndevice.com/product/wind-sensor/`, that is fairly inexpensive:



You can mount it to the mast if you like; perhaps use a small piece of heavy duty tape and mount it to the top of the mast. In order for the BeagleBone Blue to talk with this, you need to connect it to the GPIO connector pins. The following is a close-up of the connections that the wind sensor requires:



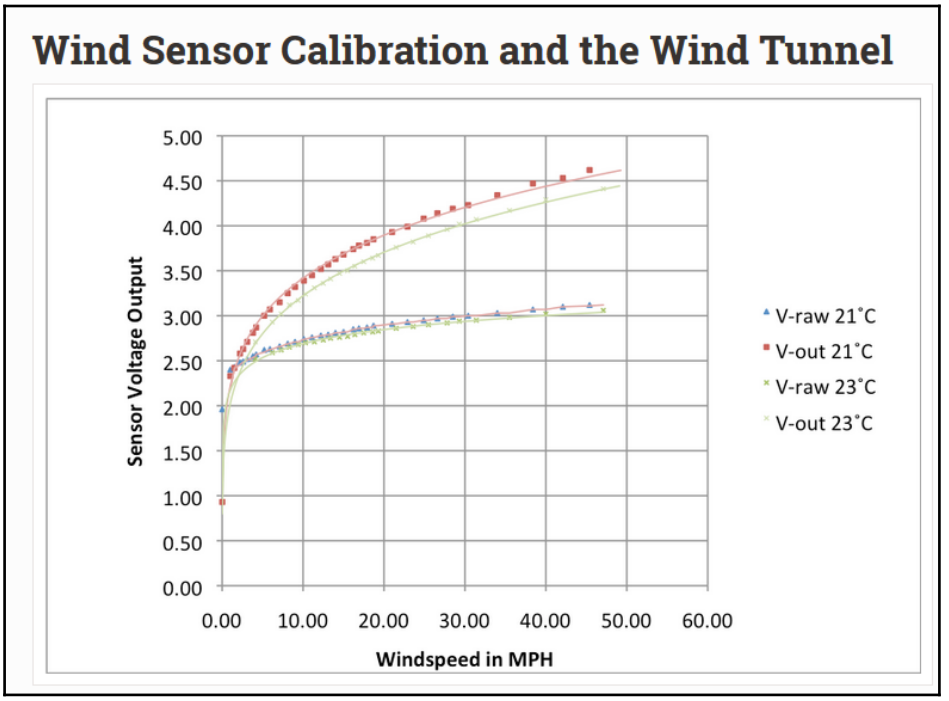
You'll need a **GND** and **+V** connection; the **+V** will be to the 6V of the BeagleBone Blue, available on one of the center pins of the servo connectors. The **GND** will be connected to the ground connection of the BeagleBone Blue ADC connector. Here is the location of that connector, circled in red:



In this connector, the first pin, that is the pin on the farthest right in the preceding picture, is GND. The second pin on this connector is 1.8 volts; this is not enough voltage to drive the wind sensor, so you won't use that connection as you will be supplying power from the servo connector. The third pin is the input to channel 1 on the ADC of the BeagleBone Blue.

The TMP pin on this sensor is a pin that also outputs temperature measurement that is available; you won't use it on this project.

There are two connections on the wind sensor that you can use to sense wind and connect to channel 1 on the ADC of the BeagleBone Blue. The out pin is an output that you can actually calibrate using the small potentiometer on the device. The RV is the raw voltage output of the sensor. Here is a graph, from the manufacturer's website, of the two different outputs at different temperatures and wind values:



You'll use the connection labeled Out and you'll connect it to the A1 pin of the ADC connector on the BeagleBone Blue. However, you can't connect this input directly to this pin, as the upper limit of the ADC is 1.8V and this input can come in at up to 5V. You need to build a voltage divider circuit to translate the 0 to 5V signal to one that is 0 to 1.8V. A voltage divider is a simple circuit that uses two resistors of correct value to scale the voltage. See `https://learn.sparkfun.com/tutorials/voltage-dividers` for details on the principles behind the voltage divider, and a calculator to allow you to decide on resistor values. In this case, you will want to reduce the voltage by 1.8/6 or 0.3. If you chose a 1 kOhm resistor as R2, then you'll want a 2 kOhm resistor for R1.

Choosing these resistors values also makes sure that the resistors you choose are not so large that the device cannot drive them with enough current for the BeagleBone Blue to sense the voltage. They must also not be so small that the device drives the BeagleBone Blue with too much current. The choices of these two resistor values meets these requirements for this device.

Now that the hardware is connected, you'll want to read the values from the wind sensor programmatically.

Getting sensor data from the airspeed sensor

You'll use the same library of code that you've used in the previous chapters to access the ADC connector and read the ADC value. The command you would use to read the value is:

```
adc_value = rc_adc_raw(1)
```

In this code, a value from `0` to `4096` is read into the variable `adc_value`, based on the size of the signal from the wind sensor connected to the ADC channel 1 connector. If you use this command, the value `0` corresponds to no signal and `4096` corresponds to the max signal. You can also use `rc_adc_volt(1)`; then the value `0` corresponds to zero voltage and the value `1.8` corresponds to a 1.8V signal.

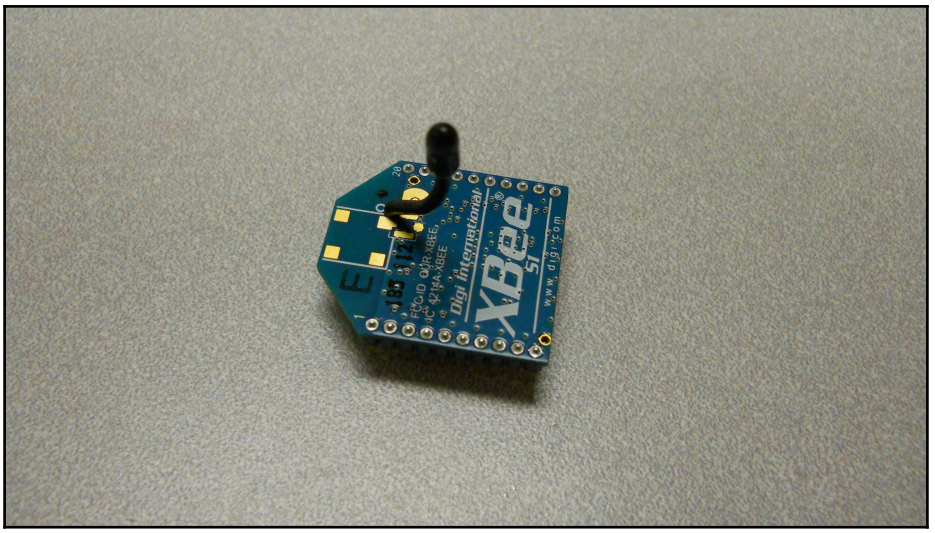
Note that this will give you an indication of the strength of the wind, but the reading is not calibrated. You can run your own calibration process to turn this value into a wind strength indicator. You will also not be able to tell the absolute direction of the wind, as the wind sensor won't know if it is coming or going. You'll need to use the results of the force of the wind on your sail boat to determine whether you are sailing with or against the wind.

Now you have a way to tell the wind direction.

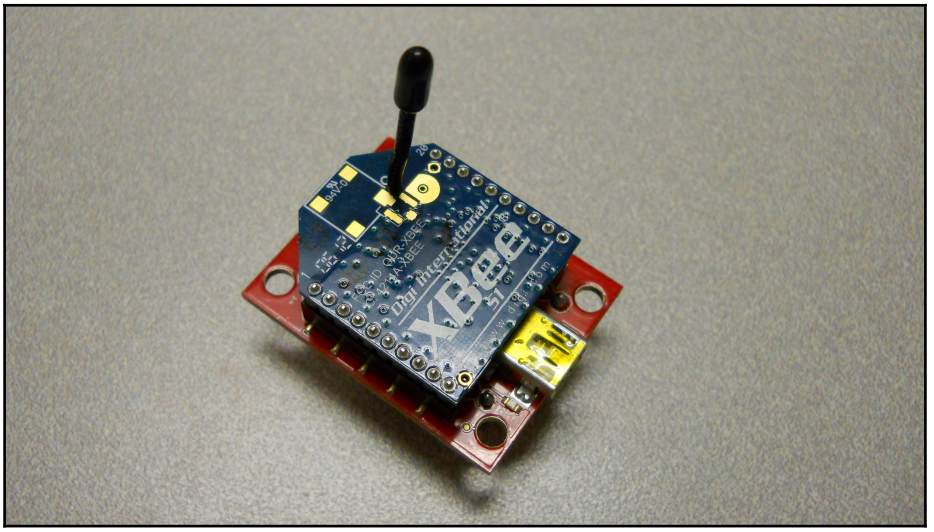
Long range control of the BeagleBone Blue

You can control the BeagleBone Blue remotely using a Wi-Fi connection and an SSH Terminal. However, this link is limited to about 30 meters, which will be fine for a small pond but you might lose your boat if you try sailing in a larger body of water.

A possible solution is to use **ZigBee** wireless devices to connect your sail boat to a computer. Here is a picture of a ZigBee device, called the **XBee**:



You'll need two of those and a USB shield for each. You can get these at a number of places, including `www.adafruit.com`. Here is a picture of the device on the shield:



Now you can connect your computer and your BeagleBone Blue via this wireless network. The advantage is that it is dedicated and can have a range of up to a mile using the right devices. A web site that provides an excellent example of how to configure and have two computers talk over this type of dedicated wireless link is `http://examples.digi.com/get-started/basic-xbee-802-15-4-chat/`.

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 can add your GPS and then have your sailboat sail all by itself.

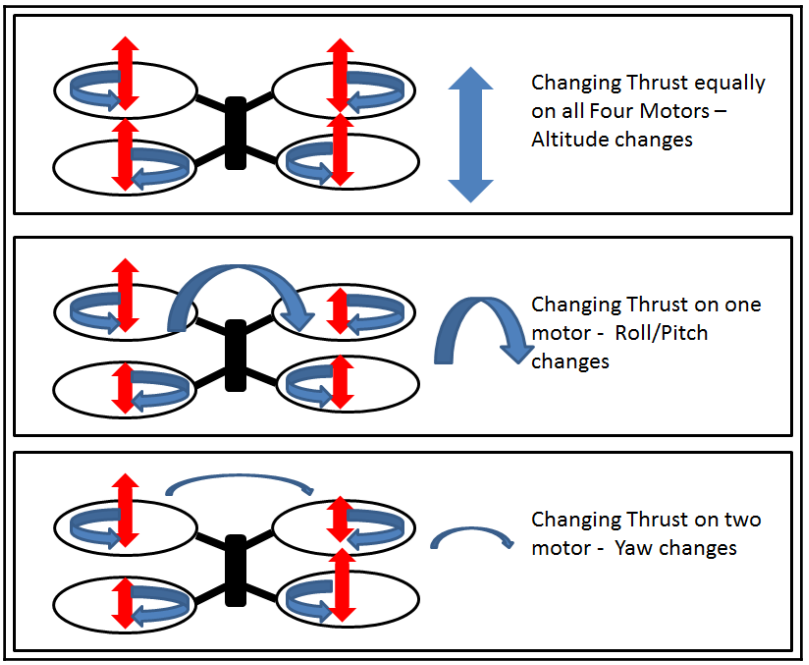
BeagleBone Blue and robots that can fly

You've now built robots that can move around on a wheeled structure, robots that can have legs, and robots that can sail. You can also build robots that can fly, relying on the BeagleBone blue to control their flight. There are several possible ways to incorporate the BeagleBone into a flying robotic project, but perhaps the most straightforward is to add it to a quadcopter project.

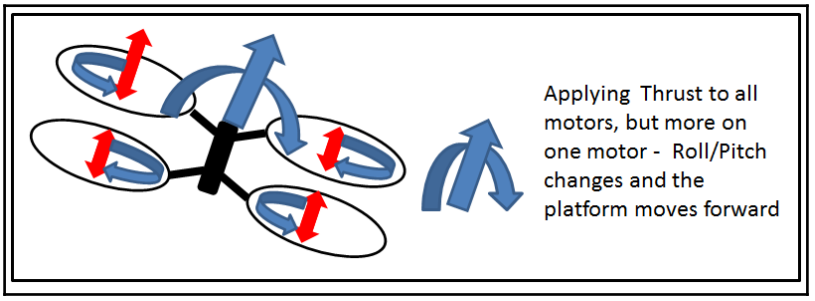
Quadcopters are a unique subset of flying platforms that have become very popular in the last few years. They utilize the same vertical lift concept as helicopters; however, they employ not one but four motor or propeller combinations to provide an enhanced level of stability. Here is a picture of just such a platform:

The quadcopter has two sets of counter-rotating propellers, which simply means that two of the propellers rotate one way and the other two rotate the other to provide thrust in the same direction. This provides a platform that is inherently stable. Controlling the thrust on all the four motors allows you to change pitch, roll, and yaw of the device.

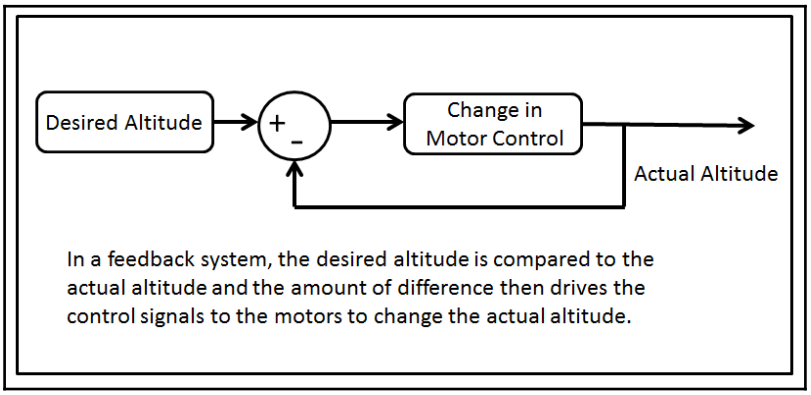
Here is a diagram that may be helpful:



As you can see, controlling the relative speeds of the four motors allows you to control the various ways the device can change position. To move forward, or really in any direction, we combine a change in roll or pitch with a change in thrust, so that instead of going up, the device moves forward, as shown in this diagram:



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. But there are simply too many aspects of your device that can vary to know this well enough to rely on a fixed set of signals. Instead, this platform uses a series of measurements of 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. Here is a diagram of 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 be increased, 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 be decreased, 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 wind comes along and blows the quadcopter up or down.

One application of the BeagleBone Blue 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 but it is a very complex task and the detail of its implementation is beyond the scope of this book. There are some individuals in the open-source software and hardware space working on this problem. See `https://diydrones.com/profiles/blogs/beaglebone-blue-released-linux-enabled-autopilot-for-80?xg_source=activity`for details.

The BeagleBone Blue can still be utilized in this type of robotic projects, without focusing on the low-level control, by introducing another embedded processor to do the low-level control, and using the BeagleBone blue to manage the high level tasks, such as using the vision system of the BeagleBone Blue to identify a colored ball and then guiding the platform toward it. Or, as in the sail boat example, using the BeagleBone to coordinate GPS tracking and long-range communications via ZigBee. This is the type of example that I'll cover in this section.

The first thing you'll need is a quadcopter. There are three approaches to this: one, purchase an already assembled quadcopter; two, purchase a kit and construct it yourself; or three, buy the parts and construct the quadcopter. In either case, to complete this section, you'll need to choose one that uses the ArduPilot as its flight control system. This flight system uses a flight version of the Arduino to do the low level feedback control we talked about earlier. The advantage to 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 `ardupilot.com`. This will give you some information on the flight controller, and the store has several already assembled quadcopters. If you think assembling a kit is the right approach, try `www.unmannedtechshop.co.uk/multi-rotor.html` or `www.buildyourowndrone.co.uk/ArduCopter-Kits-s/33.htm`, as each of these not only sells assembled quadcopters but 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 `blog.tkjelectronics.dk/2012/03/quadcopters-how-to-get-started`, `blog.oscarliang.net/build-a-quadcopter-beginners-tutorial-1/`, or `http://www.arducopter.co.uk/what-do-i-need.html`for excellent instructions.

You might be tempted to purchase one of the very inexpensive quadcopters that are being offered on the market. For this project, you will need two key characteristics of the quadcopter. First, the quadcopter flight control will need a USB port so you can connect the BeagleBone Blue to it. Second, it will need to be large enough with enough thrust to carry the extra weight of the BeagleBone Blue, a battery, and perhaps a web cam or other sensing device.

No matter which path you choose, another excellent source for information is `code.google.com/p/arducopter`. This gives you some information on how the ArduPilot works, and also talks about Mission Planner, the open-source control SW that will be used to control the ArduPilot on your quadcopter. This SW runs on the PC and communicates to the quadcopter in one of two ways: either directly through a USB connection or through a radio connection. It is the USB connection that you will communicate between the BeagleBone Blue and the ArduPilot.

The first step, when working in this space, is to build your quadcopter and get it working with an RC radio. When you allow the BeagleBone Blue 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, then you should begin to use the ardupilot in autopilot mode. To do this, download the SW from `ardupilot.com/downloads`. You can then run the SW and you should see something like this:

You can then connect your ArduPilot to the SW and press the connect button in the upper right corner. I will not walk you through how to use the SW to plan an automated flight plan as there is plenty of documentation for that on the `ardupilot.com` website. What you want to do is to hook up your BeagleBone Blue to the ArduPilot on your quadcopter so that it can control the flight of your quadcopter much as the Mission Planner does, but at a much lower and more specific level. You will use the USB interface, just as the Mission Planner does.

To connect the two devices, you'll need to modify the Arduino code and create some BeagleBone Blue code, then simply connect the USB interface of the BeagleBone Blue to the Ardupilot and you can issue 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, albeit using the Raspberry Pi as the controller, is available at `http://oweng.myweb.port.ac.uk/build-your-own-quadcopter-autopilot/`.

Now that you can fly your quadcopter using the BeagleBone Blue, you can use the same GPS and ZigBee capabilities, mentioned in other areas of the book, to make your quadcopter semi-autonomous.

Your quadcopter can also act completely autonomously as well. Adding a 3G modem to the project allows you to track your quadcopter, no matter where it goes, as long as it can receive a cell signal. Here is a picture of such a modem:

This can be purchased on `amazon.com`, but also at your cellular service provider. Once you have purchased your modem, simply google instructions on how to configure it in Linux. An example project is given at `http://www.adafruit.com/blog/2013/08/23/sky-drone-fpv-uses-3g4g-cell-network-to-provide-long-range-rc/`.

The BeagleBone Blue in robots that can go under the water

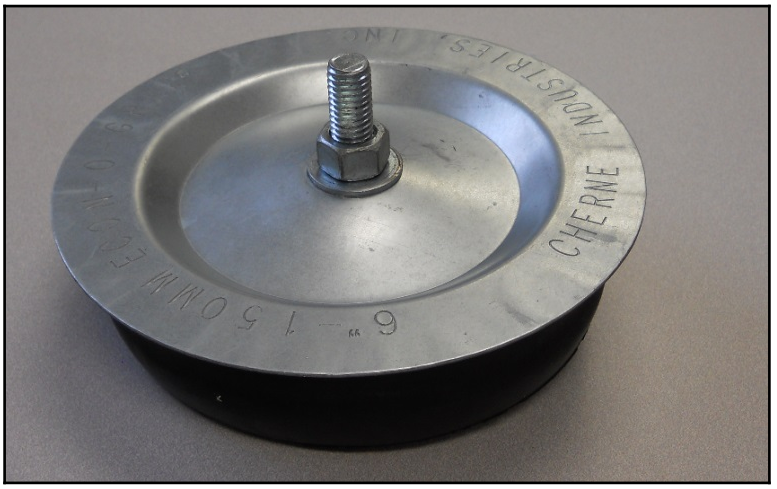
You've explored the possibilities of walking robots, flying robots, and sailing robots. The final frontier is robots that can actually manoeuvre under the water. It only makes sense that you use the same techniques that you've mastered to explore the undersea world. In this section, I'll detail how to 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 to either buy an assembled robot or assemble one yourself. If you'd like to buy an assembled ROV, try `http://openrov.com`. This project, funded through **Kickstarter,** provides a complete package, including the electronics based on the BeagleBone Blue. If you are looking to build your own, there are several websites that document possible instructions for you to follow. One of them is `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 your BeagleBone Blue. Here is a picture of a fairly simple underwater platform built.

My physical design is based more on the latter design, using mostly plastic PVC piping. Here is a picture:

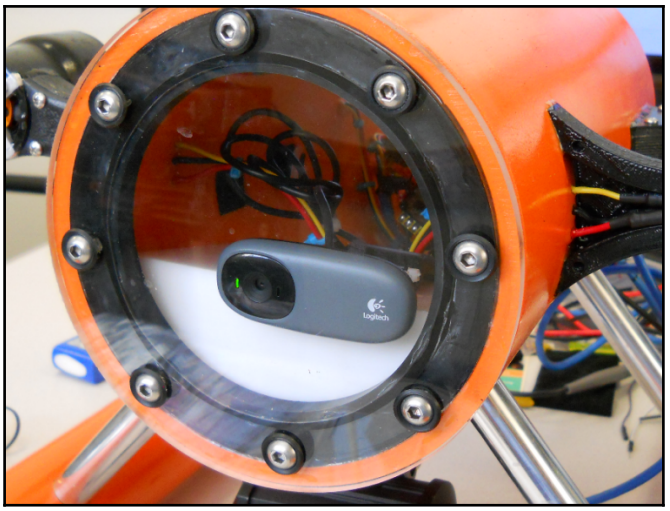
This hardware design was done using standard PVC for the main chamber and for the two chambers that hold the ballast. The main chamber is 12 inch long and 6 inch in diameter. The two smaller chambers are 20 inches by 2 inches PVC pipes. These two small chambers are capped at both ends and hold the weight required to make your ROV neutrally buoyant. Be prepared to add significant weight to your ROV. I had to add 25 pounds of lead to mine to get it to do anything but bob on top of the surface!

On the rear of the ROV, you'll want some way to be able to gain access to the main chamber while also making it water tight when you want to try your ROV. For this purpose, I used a CHERNE Model # 271578 6 in. Econo-Grip plug. These are available at most plumbing stores and also online. Here is a picture:

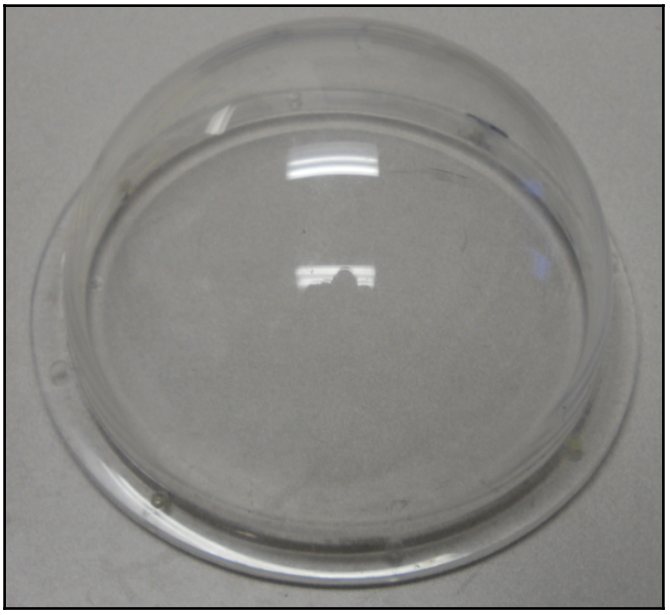


The concept of the plug is quite simple. As you tighten the nut, the rubber seal spreads and seals off the PVC pipe.

One of the most important components of your ROV is the clear plastic front casing, as you'll need to be able to use a camera to see the world under water. Here you have two choices. First is you can go with a flat piece of clear acrylic. Here is a picture of the ROV with a flat piece of acrylic:



Alternately, you can also choose a rounded dome of clear acrylic. Here is a picture of one:



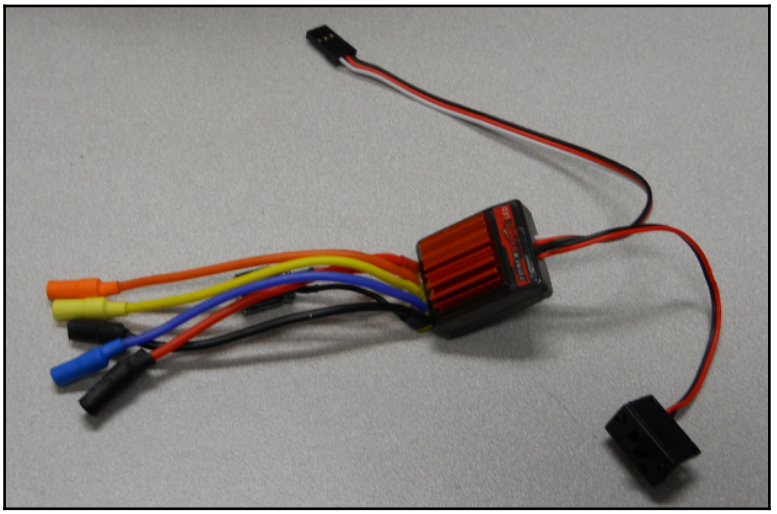
This came from a company called **EZ Tops**, and can be ordered online at `www.eztopsworldwide.com/smalldomes.htm`. You will put this on the end of your ROV, assembled with a gasket and some small bolts. This clear plastic will give you a good view of the underwater world and reduce the drag when moving in the forward direction.

The final piece of hardware that is important is the light. I chose a Cree LED work light, as it was waterproof and provided a significant amount of light while only using 18 watts. They operate from 9 to 12 volts, so I added a 3S RC LiPo battery to power the light.

You'll need two batteries for your project. One 3S RC LiPo battery to power the DC brushless motors and BeagleBone Blue, and another 3S RC LiPo battery to power the LED light.

Whatever physical design you choose, you'll need to control the motors, and the BeagleBone Blue can do this task. In this case, I chose fairly standard brushless DC motors, the **Turnigy D2836/9 950KV Brushless Outrunner Motor** available from `hobbyking.com`, and then fitted them with RC boat propellers, the **Traxxas Propeller Left 4.0mm Villain EX TRA1584** available at `rcplanet.com`. These motors work just fine underwater and are easy to control with radio control **Electronic Speed Controllers** (**ESC**).

For this project, you'll need four brushless DC motors and four ESC controllers. You'll need to make sure the ESCs will be able to control the motors to go both forward and backwards. Here is a picture of one such unit:



This particular unit is a **Turnigy Trackstart 25A ESC**, made normally for an RC car, and is available at many RC outlets (both retail and online). The connections on this unit are straightforward. The red and black wires with plugs go to an RC battery; in this case, a 3S 11.1 volt LiPo RC battery. The other three connections go to the motor. This particular ESC comes with a switch but you won't use it in this particular project. The last connection is a three wire connector similar to a servo connection. You'll connect this to the BeagleBone Blue to the servo connectors.

For details about the ESC to brushless DC motor connection, check your ESC documentation at `http://www.hobbyking.com/hobbyking/store/uploads/981860436X852229X19.pdf`. To control the ESCs, you'll use code very similar to the code you used in `Chapter 7`, *Making the Unit Very Mobile - Controlling Legged Movement*, as they work on the same PWM principle as the servos. Instead of driving the four servos to make the unit move, you'll drive the four ESCs with the servo controls to drive the motors to a certain speed.

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, however, for some challenges, and always plan on a bit of rework for, as they say, what goes up must come down. And what sails out doesn't always return.

Summary

Now you know how to build a wide range of projects using the BeagleBone Blue. In the final chapter, you'll learn a bit more about complex software systems for the BeagleBone Blue and more ways to control your projects remotely using Bluetooth and your cell phone.

10
System Dynamics

You've spent lots of time on individual functionality, and your robotic projects now have lots of functionality that can be implemented in any individual robot. This chapter will bring all these parts together into something of a framework that can help allow the different parts to work together.

You don't want your robot to just walk, or just talk, or just see. You want it to do all of these in a coordinated package. In this chapter, you'll learn how to programmatically bring together all of these individual capabilities that will make your projects seem, well, intelligent.

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

- Connect a game pad controller to control your robot from your host computer
- Control your robot via a web interface

Finally, you're done purchasing hardware. In this chapter, you'll be adding functionality via software.

Controlling your robot via a game pad controller

You've assembled your robot and can now control it via the BeagleBone Blue. You can run the control program using a VNC server configuration. Doing this will allow you basic control of the robot wirelessly. However, it would be easier to control your quadcopter using a game controller.

What you want is a controller that has more immediate control of the different aspects of your robot. Perhaps the most practical is a game controller that has two joysticks and several additional buttons. Using these as input mechanisms will make controlling your robot with the BeagleBone Blue much easier.

To add the game controller, you'll need to first find a game controller that can connect to your computer. If you are using Microsoft Windows as the OS on the host computer, then pretty much any USB controller that can connect to a PC will work. The same type of controller also works if you are using Linux for the remote computer. In fact, you could use another BeagleBone Blue as the remote computer.

Since the joystick will be connected to the remote computer, you'll need to run two programs, one on the remote computer and one on the BeagleBone Blue on your robot. You'll also need a way to communicate between them. In this example, you'll use the wireless LAN interface and a client-server model of communication. For an excellent tutorial on this type of model and how it is used in a gaming application, see `http://www.raywenderlich.com/38732/multiplayer-game-programming-for-teens-with-python`. You'll run the server program on the remote computer and the client program on the BeagleBone Blue on your robot.

Once you have the controller connected, you'll need to create a Python program on the BeagleBone Blue, that will take the signals sent from the client and send the correct signals to the various hardware you want to control. For this example, you'll use the simple two wheeled vehicle documented in `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*.

The program running on the wheeled vehicle is the client program. But before you create this client program, you'll need to install to libraries that will allow this client to work. The first is a library called `pygame`. Install this by typing `sudo apt-get install python-pygame`. Then you'll need a LAN communication layer library called `PodSixNet`. This will allow the two applications to communicate. To install this, follow the instructions at `http://mccormick.cx/projects/PodSixNet/`. Now you are ready to create the client program on the BeagleBone Blue on the robot. Here is the listing of the program:

```
 import pygame
 import math
 from PodSixNet.Connection import ConnectionListener,
 connection
 from time import sleep
 import serial
 import os
 class QuadGame(ConnectionListener):
def Network_close(self, data):
  exit()
```

```
def Network_gamepad(self, data):
if data["type"] == 10:
  #print "Pressed button "
  #print data["info"]["button"]
  if data["info"]["button"] == 0:
   os.system('/root/rc_wheeled_auto/rc_wheeled_auto '
+ str(0) + ' ' + str(1))
  if data["info"]["button"] == 5:
   os.system('/root/rc_wheeled_auto/rc_wheeled_auto '
+ str(10) + ' ' + str(0))
  if data["info"]["button"] == 4:
   os.system('/root/rc_wheeled_auto/rc_wheeled_auto '
+ str(-10) + ' ' + str(0))
def __init__(self):
 address=raw_input("Address of Server: ")
 try:
  if not address:
   host, port="localhost", 8000
  else:
   host,port=address.split(":")
  self.Connect((host, int(port)))
 except:
  print "Error Connecting to Server"
  print "Usage:", "host:port"
  print "e.g.", "localhost:31425"
 exit()
 print "Quad client started"
 self.running=False
 while not self.running:
  self.Pump()
  connection.Pump()
  sleep(0.01)
bg=QuadGame()
while 1:
if bg.update()==1:
 break
bg.finished()
```

You'll call the control program from `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement;* this code should look familiar.

In the second part of the code is the class called `QuadGame`. This class will take the inputs from the game controller connected to the server and turn them into commands that will be sent to the robot.

Here is a table of those controls:

Joystick control	Robot control
Joystick forward	Forward
Joystick right	Turn right
Joystick left	Turn left

The next set of the code includes the initialization of the interface layer, including how the user will specify the server machine. The final piece of code initializes the game loop, which loops while taking the inputs and sends them to the servo controller and on to the robot.

You'll also need a server program running on the remote computer to take the signals from the game controller and send them to the client. You'll be writing this code in Python using Python version 2.7, which can be installed from `python.org`. Additionally, you'll need to install the `pygame` library. If you are using Linux on the remote computer, then type `sudo apt-get install python-pygame`. If you are using Microsoft Windows on the remote machine, then follow the instructions at `http://www.pygame.org/download.shtml`. You'll also need the LAN communication layer described previously. You can find a version that will run on Microsoft Windows or Linux at `http://mccormick.cx/projects/PodSixNet/`. Here is a listing of the server code:

```
import pygame
import PodSixNet.Channel
import PodSixNet.Server
from pygame import *
from time import sleep
init()
from time import sleep
class ClientChannel(PodSixNet.Channel.Channel):
def Network(self, data):
 print data
def Close(self):
 self._server.close(self.gameid)
class BoxesServer(PodSixNet.Server.Server):
channelClass = ClientChannel
def __init__(self, *args, **kwargs):
 PodSixNet.Server.Server.__init__(self, *args,
**kwargs)
 self.games = []
 self.queue = None
 self.currentIndex=0
def Connected(self, channel, addr):
 print 'new connection:', channel
 if self.queue==None:
```

```
  self.currentIndex+=1
  channel.gameid=self.currentIndex
  self.queue=Game(channel, self.currentIndex)
def close(self, gameid):
  try:
  game = [a for a in self.games if a.gameid==gameid]
[0]
  game.player0.Send({"action":"close"})
 except:
  pass
def tick(self):
 if self.queue != None:
  sleep(.05)
  for e in event.get():
   self.queue.player0.Send({"action":"gamepad",
"type":e.type, "info":e.dict})
 self.Pump()
class Game:
def __init__(self, player0, currentIndex):
 #initialize the players including the one who started
the game
 self.player0=player0
#Setup and init joystick
j=joystick.Joystick(0)
j.init()
#Check init status
if j.get_init() == 1: print "Joystick is initialized"
#Get and print joystick ID
print "Joystick ID: ", j.get_id()
#Get and print joystick name
print "Joystick Name: ", j.get_name()
#Get and print number of axes
print "No. of axes: ", j.get_numaxes()
#Get and print number of trackballs
print "No. of trackballs: ", j.get_numballs()
#Get and print number of buttons
print "No. of buttons: ", j.get_numbuttons()
#Get and print number of hat controls
print "No. of hat controls: ", j.get_numhats()
print "STARTING SERVER ON LOCALHOST"
# try:
address=raw_input("Host:Port (localhost:8000): ")
if not address:
host, port="localhost", 8000
else:
host,port=address.split(":")
boxesServe = BoxesServer(localaddr=(host, int(port)))
while True:
```

```
        boxesServe.tick()
        sleep(0.01)
```

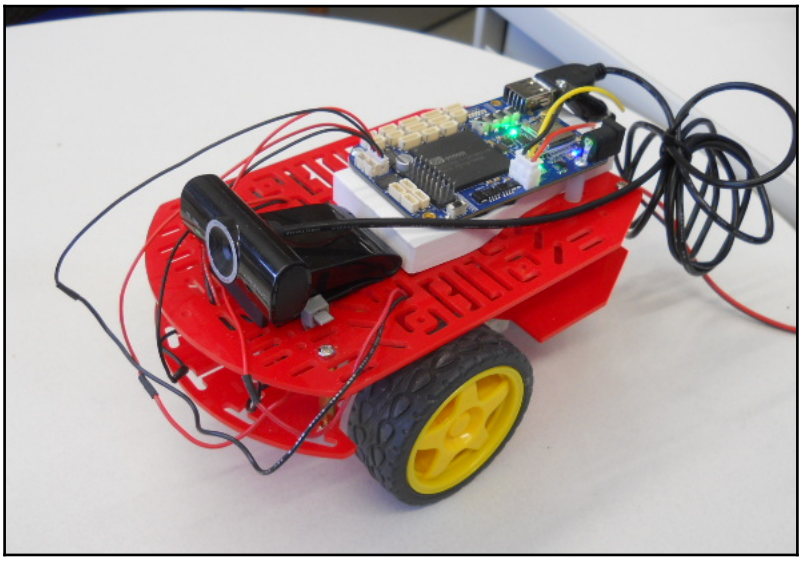
This first part creates three classes. The first, `ClientChannel`, establishes a communication channel for your project. The second, `BoxServer`, sets up a server so that you can communicate the joystick action to the BeagleBone Blue on the robot. Finally, the `Game` class just initializes a game that contains everything you'll need.

Now you can control your robot via a game controller!

Controlling your robot via a web interface

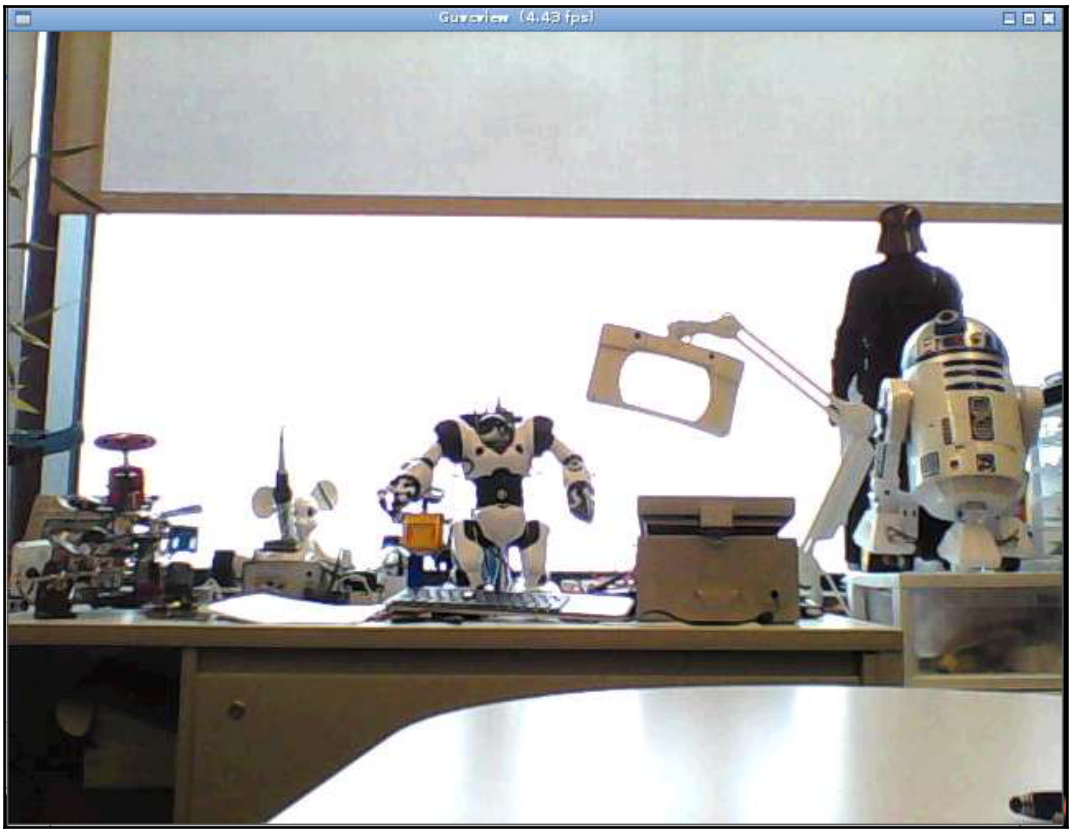
You might also want to control your BeagleBone Blue project from a remote device, perhaps a mobile phone or a tablet. In this section, you'll learn how to create a web server control mechanism for your robot so that you can control it from any web browser via WLAN.

For this part of the project, I'll assume you want to access the device, control two DC motors, and use a connected webcam to give you remote interface access to the video information you'll need to control the direction of the robot. Here is a picture of this sort of robot:



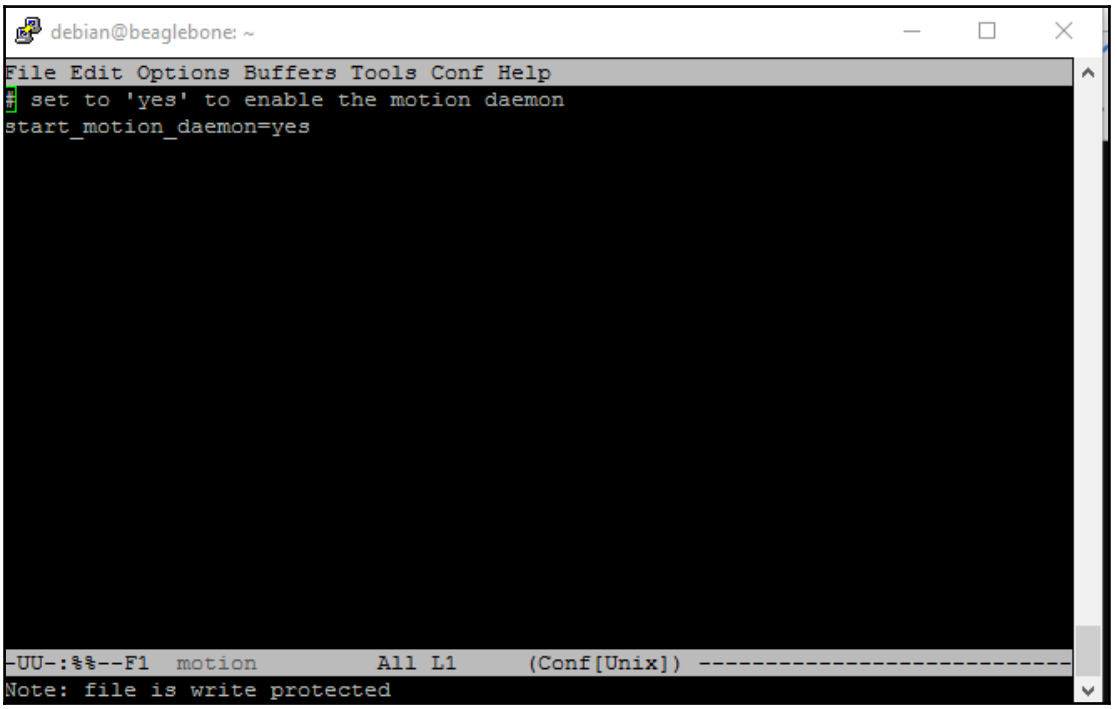
I'll also assume that you have read `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*, so that you know the basics of making the wheels move, and `Chapter 5`, *Allowing Our BeagleBone Blue to See*, so that you know how to connect and use the webcam.

Now that you have all of this hooked up, check access to the web camera by using `guvcview`. Here is what you should see:



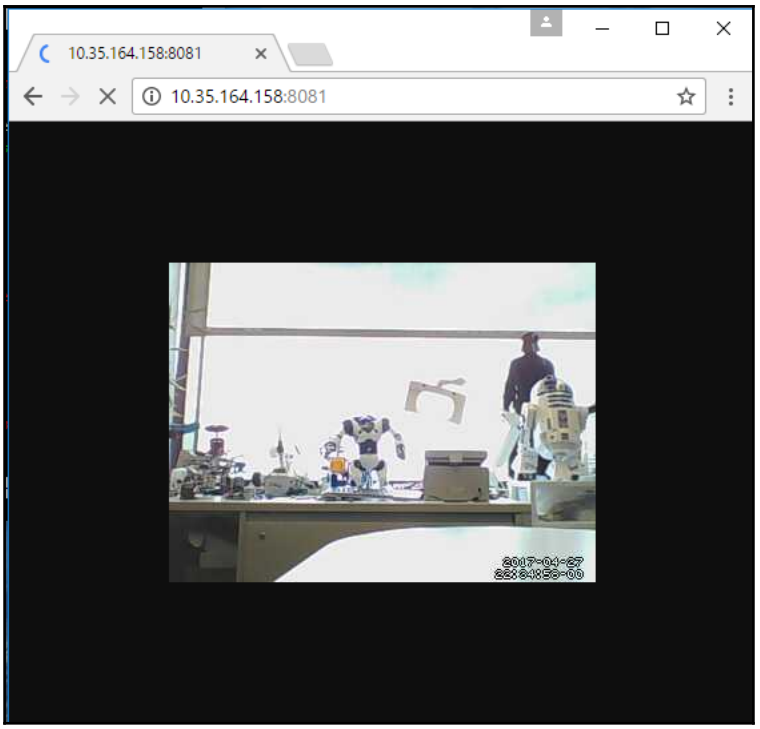
Now that you can access the camera, you need to add some software that can capture a stream of video and present it via a web interface. To do this, first install a library called motion by typing `sudo apt-get install motion`.

Now that the library is installed, you'll want to turn on the motion daemon, or the code that will run the streaming service all the time. Edit the `/etc/default/motion` file to look like this:



You'll need to set the correct permissions for the application by typing `sudo chown motion:motion /var/lib/motion/`. The last step is to edit the file `/etc/motion/motion.conf`, the configuration file for the application. You're going to want to set the `stream_localhost` variable to off; this will allow streaming from other locations other than the local host, in this case the BeagleBone Blue.

Now that you have everything set up, you can restart the application by typing `sudo /etc/init.d/motion restart`. Now you can open a web page to your BeagleBone Blue and type the IP address and port `8081`, as follows, and you should see your webcam output:



Now that you can access your webcam from any web page, including ones opened on a tablet or a mobile device, you'll want to add some control capability. Returning to `Chapter 3`, *Making the Unit Mobile - Controlling Wheeled Movement*, you should first recreate the Python code to make the motors move and provide control via the command line. Here is that code:

```
/*******************************
*****************************************
* rc_wheeled_auto.c*
* This is a program that uses the compass to turn the
wheeled
* platform and then go a certain distance.
*********************************
*****************************************/
#include <rc_usefulincludes.h>
#include <roboticscape.h>
//struct to hold new data
rc_imu_data_t data;
void process_data();
double angle;
int distance;
```

```
int turn;

/************************************
*************************************
* int main()*
* This main function contains these critical
components
* - call to initialize_cape
* - set up the compass
* - initiate the turn
* - after it comes back - go a certain distance
* - cleanup_roboticscape() at the end
**********************************
*************************************/

int main(int argc, char** argv){
// always initialize cape library first
rc_initialize();
printf("\nHello BeagleBone\n");
angle = atof(argv[1]);
if (angle > 0)
turn = 1;
else
turn = 0;
distance = atoi(argv[2]);
// done initializing so set state to RUNNING
rc_set_state(RUNNING);
// bring H-bridges of of standby
rc_enable_motors();
 rc_set_led(GREEN,ON);
rc_set_led(RED,ON);
rc_set_motor_free_spin(1);
rc_set_motor_free_spin(2);
printf("Motors are now ready.\n");
// start with default config and modify based on
option
rc_imu_config_t conf = rc_default_imu_config();
conf.dmp_sample_rate = 20;
conf.enable_magnetometer = 1;
// now set up the imu for dmp interrupt operation
if(rc_initialize_imu_dmp(&data, conf)){
printf("rc_initialize_imu_failed\n");
return -1;
}
rc_set_imu_interrupt_func(&process_data);
// set the unit turning
if (turn)
{
```

```
 rc_set_motor(1, 0.2);
 rc_set_motor(2, -0.2);
}
else
{
 rc_set_motor(1, -0.2);
 rc_set_motor(2, 0.2);
}
//now just wait, print_data() will be called by the
interrupt
while (rc_get_state()!=EXITING) {
usleep(10000);
}
int movement = 0;
// Now move forward
while (movement < distance)
{
 rc_set_motor(1, 0.2);
 rc_set_motor(2, 0.2);
 usleep(1000000);
 movement++;
}
rc_set_motor_brake_all();
// shut things down
rc_power_off_imu();
rc_cleanup();
return 0;
}
/*******************************
*********************************************
* int process_data()
*
* - Called each time the compass interrupts
* - Compares angles to see if the platform has moved
enough
* - If it has, stop the platform
 *******************************
*********************************************/
void process_data() // imu interrupt function
{
printf("\r");
printf(" ");
printf("Angle = %6.1f\n",angle);
printf("Distance = %2d\n",distance);
printf(" %6.1f |",
data.compass_heading_raw*RAD_TO_DEG);
printf(" %6.1f |", data.compass_heading*RAD_TO_DEG)
if (turn)
```

```
{
if ((angle - data.compass_heading*RAD_TO_DEG) < 1.0)
{
 rc_set_motor_brake_all();
 rc_set_state(EXITING);
 }
}
else
if ((-angle + data.compass_heading*RAD_TO_DEG) < 1.0)
 {
 rc_set_motor_brake_all();
 rc_set_state(EXITING);
 }
 fflush(stdout)
 return;
}
```

Remember that when you are ready to run the code, you type something like `./rc_wheeled_auto 30 2` and the platform should turn 30 degrees and then go for two seconds.

Now that you have this capability, you'll want to add the ability to call this from your web page. To do this, you're going to use a capability called Flask. It is a tool that lets you talk from a web page to a Python program. Install this by typing `pip install Flask`.

If you'd like to learn more about Flask, see `http://flask.pocoo.org/docs/0.11/quickstart/`.

Now that you have Flask installed, you create an HTML program that can access the streaming video and also send commands to a Python program. So create a `/templates` directory on the BeagleBone Blue and then put this code in a file in the directory:

```
<html>
<head>
<script   src="https://ajax.googleapis.com/
ajax/libs/jquery/3.1.1/jquery.min.js"></script>
</head>
<body>
<img src="http://0.0.0.0:8081" />
<!--Enter the IP Address of your BeagleBone Blue-->
<div style="float:right">
</div>
<div style=" height:400px; width:300px; float:right;">
<center>
<h2>Control Robot</h2><br><br>
```

```
 <a href="#" id="up" style="font-size:30px;
text- decoration:none;"> &#55358;&#55358;
    <br>Forward</a><br><br></center>
<a href="#" id="left" style="font-size:30px;
text- decoration:none;">
&#55358;&#55358;Left
</a>        
<a href="#" id="right" style="font-size:30px;
text-decoration:none;"> Right &#55358;&#55358;</a>
<br>  <br>
 <center><a href="#" id="down"
style="font-size:30px;text-decoration:none;">
Backward<br> &#55358;&#55358;</a></center>
</div>
<script>
$( document ).ready(function(){
$("#down").on("mousedown", function() {
$.get('/down_side');
 }).on('mouseup', function() {
 $.get('/stop');
});
$("#up").on("mousedown", function() {
 $.get('/up_side');
}).on('mouseup', function() {
$.get('/stop');
});
$("#left").on("mousedown", function() {
 $.get('/left_side');
 }).on('mouseup', function() {
 $.get('/stop');
});
$("#right").on("mousedown", function() {
$.get('/right_side');
}).on('mouseup', function() {
 $.get('/stop');
});
});
</script>
</body>
</html>
```

It is important to note that you'll need to put the IP address of your BeagleBone Blue and the video stream in the code.

You'll also need to create a Python program that can be accessed from the web page. Here is the code:

```
from flask import Flask
from flask import render_template, request
import time
import os
app = Flask(__name__)
@app.route("/")
def index():
return render_template('robot.html')
@app.route('/left_side')
def left_side():
data1="LEFT"
 os.system("/usr/debian/rc_motor/rc_wheeled_auto "
+     str(90) + " " + str(0))
return 'true'
@app.route('/right_side')
def right_side():
data1="RIGHT"
os.system("/usr/debian/rc_motor/rc_wheeled_auto "
+  str(-90) + " " + str(0))
return 'true'
@app.route('/up_side')
def up_side():
data1="FORWARD"
os.system("/usr/debian/rc_motor/rc_wheeled_auto "
+ str(0) + " " + str(1))
return 'true
@app.route('/down_side')
def down_side():
data1="BACK"
os.system("/usr/debian/rc_motor/rc_wheeled_auto "
+  str(180) + " " + str(1))
return 'true'
@app.route('/stop')
def stop():
data1="STOP"
os.system("/usr/debian/rc_motor/rc_wheeled_auto "
+ str(0) + " " + str(0))
return 'true'
if __name__ == "__main__":
print "Start"
app.run(host='0.0.0.0',port=5010)
```

In this case, the program simply takes the commands from the web page and sends them, through an `os` call, to the `rc_wheeled_auto` executable to move the robot.

That's it. Once you are connected to the web page, you can press one of the buttons and the robot should then move. You can access the web page from a remote computer, or from a remote tablet or other mobile device.

Summary

Now you have a way to start coordinating complex functionalities for your robot. Your robot can walk, talk, see, hear, and even sense its environment, all at the same time. You can add any of these different capabilities to build a robotic project that can do almost anything!

Index

A

adafruit
  reference 179
analog airspeed sensor
  connecting 175, 177
  reference 175
  sensor data, obtaining 178
Arduino IDE
  reference 88
Arduino
  sonar sensor, connecting 87
array of sensors
  creating 93, 94, 95
Audacity 125

B

BeagleBone Blue
  accessing, remotely via WLAN 20, 35
  adding, to sailing platform 173, 174
  compass, accessing 77, 78, 81
  connecting 9, 11, 16, 18
  connecting, to GPS device 153, 156, 159, 161, 163, 165
  connecting, to mobile platform 143, 145
  controlling, from long range 178
  DC motors, connecting 69
  files, creating 43
  files, editing 43
  files, saving 44
  filesystem navigation, Linux commands 38
  powering 10, 11, 15, 18
  programming constructs 49, 51, 53, 55
  python programs, creating 45, 47, 48, 49
  python programs, running 46, 47, 49
  reference 43
  used, for controlling mobile platform programmatically 69

C

C programming language 59, 60, 61, 62
colored objects
  detecting, vision library used 111, 116
commands
  interpreting capability, providing 135, 138
compass, BeagleBone Blue
  accessing 77, 81
Connmanctl application 22

D

DC motors
  connecting, to Beagle Bone 69
  controlling programmatically 71, 72, 74, 75, 76, 77
dedicated wireless
  reference link 180
degrees of freedom (DOF) 141

E

Electronic Speed Controllers (ESC)
  about 189
  reference 190
Emacs 44
eSpeak
  used, for communication in robot voice 126
EZ Tops
  reference 189

F

Flask
  reference 202

G

game pad controller
  used, for controlling robot 192, 196
gedit 44
General-Purpose Input/Output (GPIO) 87
Global Positioning System (GPS)
  about 153
  accessing programmatically 166, 171
GlobalSat 153
GPS device
  BeagleBone Blue, connecting to 153
GPS distance
  reference 168
GPS tracking
  reference 171
guvcview 105

H

hardware
  connecting 120, 122
  prerequisites 118

I

IMU 81
Infrared (IR) signals 86
infrared sensor 86
input sound
  creating 120, 122

K

Kickstarter 186

L

legged mobile platform
  about 141
  reference 141
LiDAR sensor 86
Linux commands
  used, for filesystem navigation 38, 39, 40, 41, 42, 43
Linux program
  creating, for controlling mobile platform 145, 148, 150
local path planning algorithm
  reference 103
location
  migrating to 166, 171

M

mobile platform
  basic path, planning 97, 99
  BeagleBone Blue, connecting to 143, 145
  controlling, with Linux program 145, 149
  creating 68
  dynamic path, planning 96
  obstacles, avoiding 100
  prerequisites 65
  voice commands, issuing 150

N

nano 44
ND-100S 153

O

OpenCV
  downloading 108, 110
  installing 108, 110
operating system
  accessing 18

P

Pocketsphinx
  reference 127
  used, for voice command interpretation 127, 130
PodSixNet
  reference 192
Programmable Real-Time Unit (PRU) 87
programming constructs 49, 55, 57
project
  making mobile 140
Python
  programming concepts, URL 46

Q

quadcopters
  about 180
  building 180, 184
  reference 183

R

Remote Operated Vehicle (ROV) 186
robot
  action, initiating 136, 138
  controlling, via game pad controller 192, 196
  controlling, via web interface 196, 202, 205

S

sensors
  infrared sensor 86
  LiDAR sensor 86
  options 84
  sonar sensor 85
sonar sensor
  about 85
  accessing, from Ardino IDE 88
  connecting, to Arduino 87
Sphinx
  reference 132

T

text commands
  reference 124
tightvncserver
  about 30
  reference 31
triangulation 154
Turnigy D2836/9 950KV Brushless Outrunner Motor
  reference 189
Turnigy Trackstart 25A ESC 190

U

underwater robots
  exploring 186, 189
unit mobile
  creating 65, 67
USB camera
  connecting, to BeagleBone Blue 105
  images, viewing 105, 108

V

vi 44
Vim 44
vision library
  used, for colored object detection 111, 116
voice commands
  interpreting, Pocketsphinx used 127, 130, 134

W

web interface
  robot, controlling 196, 199, 205
WLAN
  BeagleBone Blue, remote access via 20, 22, 24, 29, 35

X

XBee 179
Xfce 30

Z

ZigBee 179