



# BeagleBone Cookbook



# Table of contents

<b>1 Basics</b>	<b>3</b>
1.1 Picking Your Beagle	3
1.1.1 Problem	3
1.1.2 Solution	3
1.2 Getting Started, Out of the Box	3
1.2.1 Problem	3
1.2.2 Solution	3
1.2.3 Discussion	6
1.3 Verifying You Have the Latest Version of the OS on Your Bone	7
1.3.1 Problem	7
1.3.2 Solution	7
1.4 Running the Python Examples	7
1.4.1 Problem	7
1.4.2 Solution	7
1.5 Cloning the Cookbook Repository	7
1.5.1 Problem	7
1.5.2 Solution	7
1.6 Wiring a Breadboard	8
1.6.1 Problem	8
1.6.2 Solution	8
1.6.3 Breadboard wired to BeagleBone Black	8
1.7 Editing Code Using Visual Studio Code	9
1.7.1 Problem	9
1.7.2 Solution	9
1.8 Running Python and JavaScript Applications from Visual Studio Code	9
1.8.1 Problem	9
1.8.2 Solution	9
1.8.3 Finding the Latest Version of the OS for Your Bone	9
1.9 Running the Latest Version of the OS on Your Bone	13
1.9.1 Problem	13
1.9.2 Solution	13
1.10 Updating the OS on Your Bone	13
1.10.1 Problem	13
1.10.2 Solution	13
1.10.3 Discussion	14
1.11 Backing Up the Onboard Flash	14
1.11.1 Problem	14
1.11.2 Solution	14
1.12 Updating the Onboard Flash	14
1.12.1 Problem	14
1.12.2 Solution	15
<b>2 Sensors</b>	<b>17</b>
2.1 Choosing a Method to Connect Your Sensor	17
2.1.1 Problem	17
2.1.2 Solution	19
2.2 Input and Run a Python or JavaScript Application for Talking to Sensors	19

2.2.1	Problem	20
2.2.2	Solution	20
2.3	Reading the Status of a Pushbutton or Magnetic Switch (Passive On/Off Sensor)	23
2.3.1	Problem	23
2.3.2	Solution	23
2.4	Mapping Header Numbers to gpio Numbers	25
2.4.1	Problem	25
2.4.2	Solution	25
2.5	Reading a Position, Light, or Force Sensor (Variable Resistance Sensor)	26
2.5.1	Problem	26
2.5.2	Solution	26
2.6	Reading a Distance Sensor (Analog or Variable Voltage Sensor)	29
2.6.1	Problem	30
2.6.2	Solution	30
2.7	Reading a Distance Sensor (Variable Pulse Width Sensor)	32
2.7.1	Problem	32
2.7.2	Solution	32
2.8	Accurately Reading the Position of a Motor or Dial	34
2.8.1	Problem	34
2.8.2	Solution	34
2.8.3	See Also	37
2.9	Acquiring Data by Using a Smart Sensor over a Serial Connection	38
2.9.1	Problem	38
2.9.2	Solution	38
2.10	Measuring a Temperature	39
2.10.1	Problem	39
2.10.2	Solution	39
2.11	I <sup>2</sup> C tools	42
2.12	Reading the temperature via the kernel driver	42
2.13	Reading i2c device directly	45
2.14	Reading Temperature via a Dallas 1-Wire Device	45
2.14.1	Problem	45
2.14.2	Solution	45
2.15	Playing and Recording Audio	48
2.15.1	Problem	48
2.15.2	Solution	48
2.16	Listing the ALSA audio output and input devices on the Bone	49
2.16.1	Discussion	49
<b>3</b>	<b>Displays and Other Outputs</b>	<b>51</b>
3.1	Toggling an Onboard LED	52
3.1.1	Problem	52
3.1.2	Solution	52
3.2	Toggling an External LED	54
3.2.1	Problem	54
3.2.2	Solution	54
3.3	Toggling a High-Voltage External Device	56
3.3.1	Problem	56
3.3.2	Solution	56
3.4	Fading an External LED	56
3.4.1	Problem	56
3.4.2	Solution	57
3.5	Writing to an LED Matrix	60
3.5.1	Problem	60
3.5.2	Solution	60
3.6	Using I <sup>2</sup> C command-line tools to discover the address of the display	60
3.7	LED matrix display (matrixLEDi2c.py)	61
3.8	Driving a 5 V Device	62

3.8.1	Problem	62
3.8.2	Solution	63
3.9	Writing to a NeoPixel LED String Using the PRUs	63
3.9.1	Problem	63
3.9.2	Solution	63
3.10	Writing to a NeoPixel LED String Using LEDscape	64
3.11	Making Your Bone Speak	64
3.11.1	Problem	64
3.11.2	Solution	64
<b>4</b>	<b>Motors</b>	<b>67</b>
4.1	Controlling a Servo Motor	67
4.1.1	Problem	67
4.1.2	Solution	68
4.2	Controlling a Servo with an Rotary Encoder	72
4.2.1	Problem	72
4.2.2	Solution	72
4.3	Controlling the Speed of a DC Motor	74
4.3.1	Problem	74
4.3.2	Solution	74
4.4	See Also	77
4.5	Controlling the Speed and Direction of a DC Motor	77
4.5.1	Problem	77
4.5.2	Solution	77
4.6	Driving a Bipolar Stepper Motor	79
4.6.1	Problem	79
4.6.2	Solution	79
4.7	Driving a Unipolar Stepper Motor	81
4.7.1	Problem	82
4.7.2	Solution	82
<b>5</b>	<b>Beyond the Basics</b>	<b>85</b>
5.1	Running Your Bone Standalone	85
5.1.1	Problem	85
5.1.2	Solution	85
5.2	Selecting an OS for Your Development Host Computer	87
5.2.1	Problem	87
5.2.2	Solution	87
5.3	Getting to the Command Shell via SSH	87
5.3.1	Problem	87
5.3.2	Solution	88
5.3.3	Default password	88
5.4	Removing the <i>Message of the Day</i>	88
5.4.1	Problem	88
5.4.2	Solution	88
5.5	Getting to the Command Shell via the Virtual Serial Port	89
5.5.1	Problem	89
5.5.2	Solution	89
5.6	Viewing and Debugging the Kernel and u-boot Messages at Boot Time	89
5.6.1	Problem	89
5.6.2	Solution	90
5.7	Verifying You Have the Latest Version of the OS on Your Bone from the Shell	94
5.7.1	Problem	94
5.7.2	Solution	94
5.8	Controlling the Bone Remotely with a VNC	94
5.8.1	Problem	95
5.8.2	Solution	95
5.9	Learning Typical GNU/Linux Commands	96
5.9.1	Problem	96

5.9.2 Solution	96
5.10 Editing a Text File from the GNU/Linux Command Shell	97
5.10.1 Problem	97
5.10.2 Solution	97
5.11 Establishing an Ethernet-Based Internet Connection	97
5.11.1 Problem	97
5.11.2 Solution	97
5.12 Establishing a WiFi-Based Internet Connection	101
5.12.1 Problem	101
5.12.2 Solution	102
5.13 Sharing the Host's Internet Connection over USB	105
5.13.1 Problem	105
5.13.2 Solution	105
5.14 Setting Up a Firewall	108
5.14.1 Problem	108
5.14.2 Solution	108
5.15 Installing Additional Packages from the Debian Package Feed	109
5.15.1 Problem	109
5.15.2 Solution	110
5.16 Removing Packages Installed with apt	110
5.16.1 Problem	110
5.16.2 Solution	110
5.17 Copying Files Between the Onboard Flash and the MicroSD Card	111
5.17.1 Problem	111
5.17.2 Solution	111
5.18 Freeing Space on the Onboard Flash or MicroSD Card	112
5.18.1 Problem	112
5.18.2 Solution	112
5.19 Using C to Interact with the Physical World	114
5.19.1 Problem	114
5.19.2 Solution	114
<b>6 Internet of Things</b>	<b>117</b>
6.1 Accessing Your Host Computer's Files on the Bone	117
6.1.1 Problem	117
6.1.2 Solution	117
6.2 Serving Web Pages from the Bone	118
6.2.1 Problem	118
6.2.2 Solution	118
6.3 Interacting with the Bone via a Web Browser	118
6.3.1 Problem	118
6.3.2 Solution	120
6.4 First Flask - hello, world	120
6.5 Adding a template	120
6.6 Displaying GPIO Status in a Web Browser - reading a button	124
6.6.1 Problem	124
6.6.2 Solution	124
6.7 Controlling GPIOs	125
6.7.1 Problem	125
6.7.2 Solution	125
6.8 Plotting Data	128
6.8.1 Problem	128
6.8.2 Solution	128
6.9 Sending an Email	134
6.9.1 Problem	136
6.9.2 Solution	136
6.10 Sending an SMS Message	137
6.10.1 Problem	137

6.10.2 Solution . . . . .	137
6.11 Displaying the Current Weather Conditions . . . . .	138
6.11.1 Problem . . . . .	138
6.11.2 Solution . . . . .	138
6.12 Sending and Receiving Tweets . . . . .	140
6.12.1 Problem . . . . .	140
6.12.2 Solution . . . . .	140
6.13 Creating a Project and App . . . . .	140
6.14 Creating a tweet . . . . .	140
6.15 Deleting a tweet . . . . .	142
6.16 Wiring the IoT with Node-RED . . . . .	144
6.16.1 Problem . . . . .	144
6.16.2 Solution . . . . .	145
6.17 Starting Node-RED . . . . .	145
6.18 Building a Node-RED Flow . . . . .	145
6.19 Adding an LED Toggle . . . . .	146
6.20 Communicating over a Serial Connection to an Arduino or LaunchPad . . . . .	152
6.20.1 Problem . . . . .	152
6.20.2 Solution . . . . .	152
6.20.3 Discussion . . . . .	157
<b>7 The Kernel . . . . .</b>	<b>159</b>
7.1 Updating the Kernel . . . . .	159
7.1.1 Problem . . . . .	159
7.1.2 Solution . . . . .	159
7.2 Building and Installing Kernel Modules . . . . .	161
7.2.1 Problem . . . . .	161
7.2.2 Solution . . . . .	161
7.3 Compiling the Kernel . . . . .	163
7.3.1 Problem . . . . .	163
7.3.2 Solution . . . . .	163
7.4 Downloading and Compiling the Kernel . . . . .	163
7.5 Installing the Kernel on the Bone . . . . .	164
7.6 Installin a Cross Compiler . . . . .	166
7.6.1 Problem . . . . .	166
7.6.2 Solution . . . . .	166
7.7 Setting Up Variables . . . . .	167
7.8 Applying Patches . . . . .	167
7.8.1 Problem . . . . .	167
7.8.2 Solution . . . . .	167
7.9 Creating Your Own Patch File . . . . .	169
7.9.1 Problem . . . . .	169
7.9.2 Solution . . . . .	169
<b>8 Real-Time I/O . . . . .</b>	<b>171</b>
8.1 I/O with Python and JavaScript . . . . .	171
8.1.1 Problem . . . . .	171
8.1.2 Solution . . . . .	171
8.2 I/O with devmem2 . . . . .	175
8.2.1 Problem . . . . .	176
8.2.2 Solution . . . . .	176
8.3 I/O with C and mmap() . . . . .	177
8.3.1 Problem . . . . .	177
8.3.2 Solution . . . . .	177
8.4 Tighter Delay Bounds with the PREEMPT_RT Kernel . . . . .	180
8.4.1 Problem . . . . .	180
8.4.2 Solution . . . . .	180
8.5 Cyclictst . . . . .	181
8.6 I/O with simpPRU . . . . .	184

8.6.1	Problem	184
8.6.2	Solution	184
8.7	Background	184
<b>9</b>	<b>Capes</b>	<b>185</b>
9.1	Connecting Multiple Capes	185
9.1.1	Problem	185
9.1.2	Solution	185
9.2	LCD Backside	187
9.3	Audio cape pins	187
9.4	Moving from a Breadboard to a Protoboard	189
9.4.1	Problem	189
9.4.2	Solution	189
9.5	Creating a Prototype Schematic	190
9.5.1	Problem	190
9.5.2	Solution	190
9.6	Verifying Your Cape Design	190
9.6.1	Problem	190
9.6.2	Solution	193
9.7	Testing the quickBot motors interface (quickBot_motor_test.js)	193
9.8	Laying Out Your Cape PCB	198
9.8.1	Problem	198
9.8.2	Solution	198
9.9	Customizing the Board Outline	199
9.10	Fritzing tips	199
9.11	PCB Design Alternatives	204
9.11.1	EAGLE	204
9.11.2	DesignSpark PCB	207
9.11.3	Upverter	207
9.11.4	Kicad	208
9.12	Migrating a Fritzing Schematic to Another Tool	208
9.12.1	Problem	208
9.12.2	Solution	208
9.13	Producing a Prototype	210
9.13.1	Problem	210
9.13.2	Solution	210
9.14	Creating Contents for Your Cape Configuration EEPROM	214
9.14.1	Problem	214
9.14.2	Solution	214
9.15	Putting Your Cape Design into Production	215
9.15.1	Problem	215
9.15.2	Solution	215
<b>10</b>	<b>Parts and Suppliers</b>	<b>217</b>
10.1	Prototyping Equipment	217
10.2	Resistors	218
10.3	Transistors and Diodes	218
10.4	Integrated Circuits	218
10.5	Opto-Electronics	219
10.6	Capes	219
10.7	Miscellaneous	219
<b>11</b>	<b>Misc</b>	<b>221</b>
11.1	BeagleConnect Freedom	221
11.1.1	Useful Links	222
11.1.2	micropython Examples	222
11.2	Setting up shortcuts to make life easier	223
11.3	Setting up a root login	224
11.4	Wireshark	225

11.4.1	Running Wireshark on the Beagle	225
11.4.2	Running Wireshark on the host	226
11.4.3	Sharking the wpan radio	227
11.5	Find what UU is in i2cdetect	227
11.5.1	Problem	228
11.5.2	Solution	228
11.6	Converting a tmp117 to a tmp114	228
11.6.1	Problem	228
11.6.2	Solution	228
11.7	Documenting with Sphinx	237
11.7.1	Problem	237
11.7.2	Solution	237
11.8	Running Sparkfun’s qwiic Python Examples	238
11.8.1	Qwiic Alphanumeric display	239
11.9	Controlling LEDs by Using SYSFS Entries	239
11.9.1	Problem	239
11.9.2	Solution	239
11.10	Controlling GPIOs by Using SYSFS Entries	240
11.10.1	Problem	240
11.10.2	Solution	240
11.11	Reading a GPIO Pin via sysfs	240
11.12	Writing a GPIO Pin via sysfs	241
11.13	The Play’s Boot Sequence	242
11.13.1	Booting for the User	242
11.13.2	Booting for the Developer	242
11.13.3	Boot Flow	244
11.13.4	Source Code	245
11.14	Home Assistant	245
11.14.1	Mqtt	245





---

**Contributors**

- Author: [Mark A. Yoder](#)
  - Book revision: v2.1 beta
- 

A cookbook for programming Beagles



# Chapter 1

## Basics

When you buy BeagleBone Black, pretty much everything you need to get going comes with it. You can just plug it into the USB of a host computer, and it works. The goal of this chapter is to show what you can do with your Bone, right out of the box. It has enough information to carry through the next three chapters on sensors (*Sensors*), displays (*Displays and Other Outputs*), and motors (*Motors*).

### 1.1 Picking Your Beagle

#### 1.1.1 Problem

There are many different BeagleBoards. How do you pick which one to use?

#### 1.1.2 Solution

Check out the current list of boards: [boards](#)

### 1.2 Getting Started, Out of the Box

#### 1.2.1 Problem

You just got your Bone, and you want to know what to do with it.

#### 1.2.2 Solution

Many of the Beagles (beaglely-all-home, beagleplay-home, bba164-home, bba1-home, beaglev-ahead-home and beaglev-fire-home) have their own detailed **Quick start** guide. Here we present general instructions that work for all Beagles. Fortunately, you have all you need to get running: your Bone and a USB cable. Plug the USB cable into your host computer (Mac, Windows, or Linux) and plug the mini-USB connector side into the USB connector near the Ethernet connector on the Bone, as shown in [Plugging BeagleBone Black into a USB port](#).

The four blue **USER LEDs** will begin to blink, and in 10 or 15 seconds, you'll see a new USB drive appear on your host computer. [The Bone appears as a USB drive](#) shows how it will appear on a Windows host, and Linux and Mac hosts will look similar. The Bone acting like a USB drive and the files you see are located on the Bone.

Browse to <http://192.168.7.2:3000> from your host computer ([Visual Studio Code](#)). If the page is not found, run the following:



Fig. 1.1: Plugging BeagleBone Black into a USB port

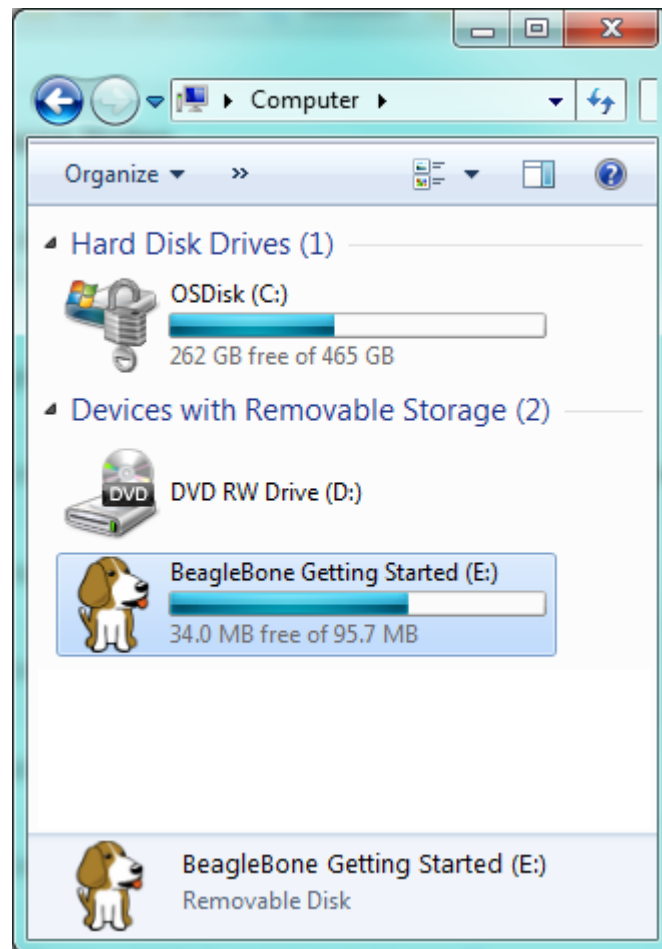


Fig. 1.2: The Bone appears as a USB drive

```
bone$ sudo systemctl start bb-code-server.service
```

Wait a minute and try the URL again.

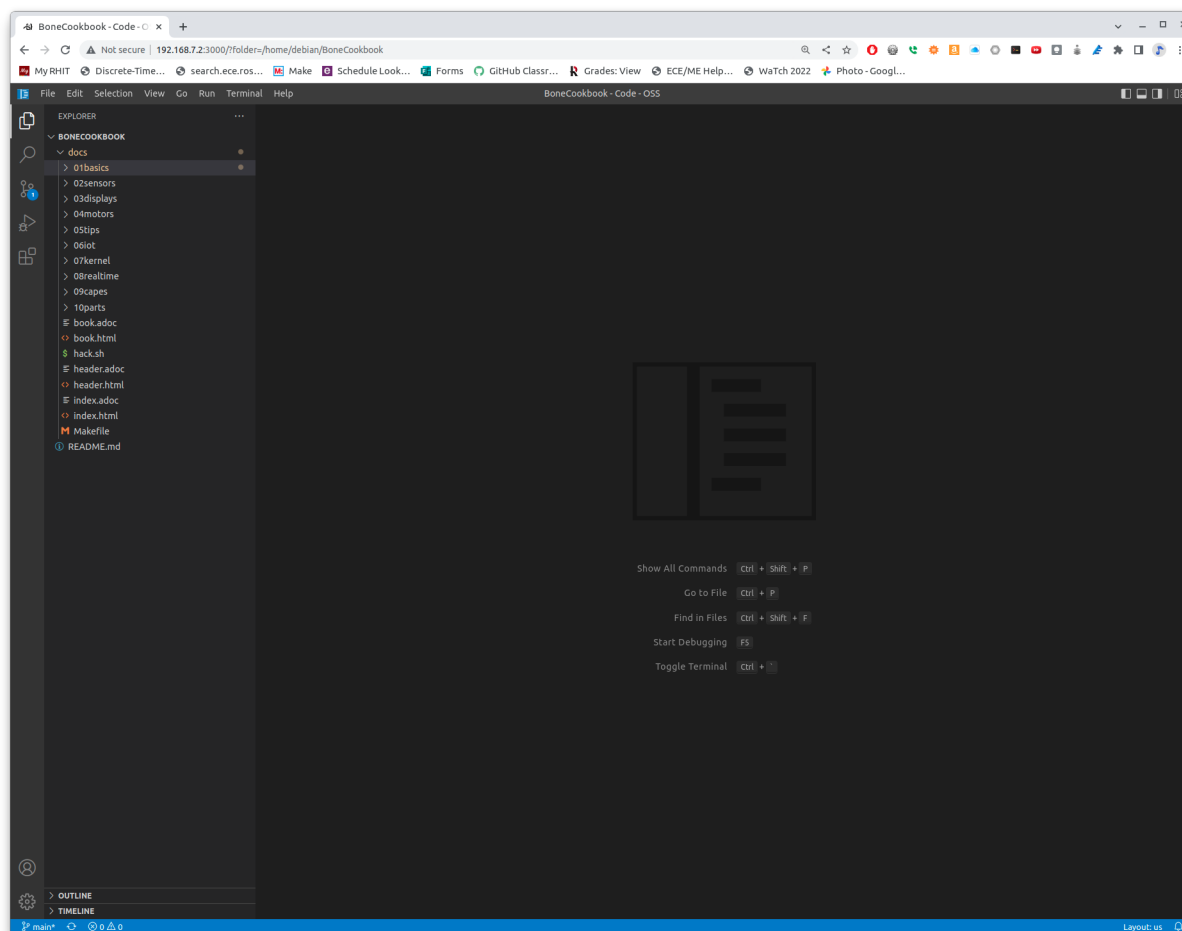


Fig. 1.3: Visual Studio Code

Here, you'll find *Visual Studio Code*, a web-based integrated development environment (IDE) that lets you edit and run code on your Bone! See [Editing Code Using Visual Studio Code](#) for more details.

### Warning:

Make sure you turn off your Bone properly. It's best to run the *halt* command:

```
bone$ sudo halt
```

```
The system is going down for system halt NOW! (pts/0)
```

This will ensure that the Bone shuts down correctly. If you just pull the power, it is possible that open files would not close properly and might become corrupt.

### 1.2.3 Discussion

The rest of this book goes into the details behind this quick out-of-the-box demo. Explore your Bone and then start exploring the book.

---

## 1.3 Verifying You Have the Latest Version of the OS on Your Bone

### 1.3.1 Problem

You just got BeagleBone Black, and you want to know which version of the operating system it's running.

### 1.3.2 Solution

This book uses [Debian](#), the Linux distribution that currently ships on the Bone. However this book is based on a newer version (BeagleBoard.org Debian Bullseye IoT Image 2023-06-03) than what is shipping at the time of this writing. You can see which version your Bone is running by following the instructions in [Getting Started, Out of the Box](#) to log into the Bone. Then run:

```
bone$ cat /etc/dogtag
BeagleBoard.org Debian Bookworm Minimal Image 2024-09-11
```

I'm running the **2024-09-11** version.

## 1.4 Running the Python Examples

### 1.4.1 Problem

You'd like to learn Python to interact with the Bone to perform physical computing tasks without first learning Linux.

---

**Note:** There are many JavaScript examples too, but they may not be as up to date as the Python examples.

---

### 1.4.2 Solution

Plug your board into the USB of your host computer and browse to <http://192.168.7.2:3000> using Google Chrome or Firefox (as shown in [Getting Started, Out of the Box](#)). In the left column, click on *examples*, then *BeagleBone* and then *Black*. Several sample scripts will appear. Go and explore them.

---

**Tip:** Explore the various demonstrations of Python and JavaScript. These are what come with the Bone. In [Cloning the Cookbook Repository](#) you see how to load the examples for the Cookbook.

---

## 1.5 Cloning the Cookbook Repository

### 1.5.1 Problem

You want to run the Cookbook examples.

### 1.5.2 Solution

Connect your Bone to the Internet and log into it. From the command line run:



```
bone$ git clone https://git.beagleboard.org/beagleboard/beaglebone-cookbook-  
→code  
bone$ cd beaglebone-cookbook-code  
bone$ ls
```

You can look around from the command line, or explore from Visual Studio Code. If you are using VSC, go to the *File* menu and select *Open Folder ...* and select *beaglebone-cookbook-code*. Then explore.

## 1.6 Wiring a Breadboard

### 1.6.1 Problem

You would like to use a breadboard to wire things to the Bone.

### 1.6.2 Solution

Many of the projects in this book involve interfacing things to the Bone. Some plug in directly, like the USB port. Others need to be wired. If it's simple, you might be able to plug the wires directly into the *P8* or *P9* headers. Nevertheless, many require a breadboard for the fastest and simplest wiring.

To make this recipe, you will need:

- Breadboard and jumper wires

The [Breadboard wired to BeagleBone Black](#) shows a breadboard wired to the Bone. All the diagrams in this book assume that the ground pin (*P9\_1* on the Bone) is wired to the negative rail and 3.3 V (*P9\_3*) is wired to the positive rail.

### 1.6.3 Breadboard wired to BeagleBone Black

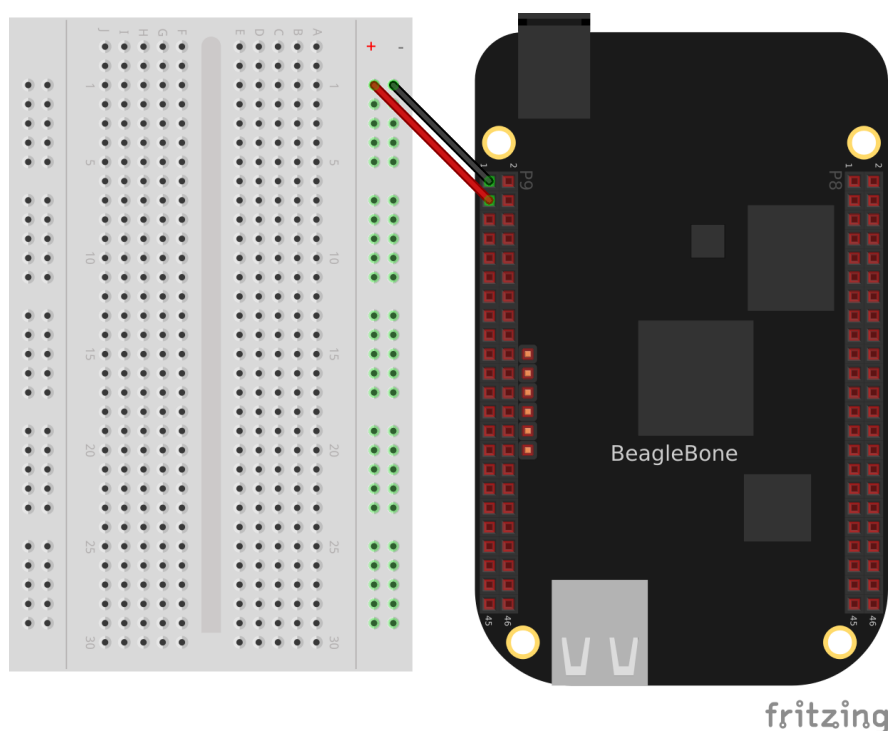


Fig. 1.4: Breadboard wired to BeagleBone Black

## 1.7 Editing Code Using Visual Studio Code

### 1.7.1 Problem

You want to edit and debug files on the Bone.

### 1.7.2 Solution

Plug your Bone into a host computer via the USB cable. Open a browser (either Google Chrome or FireFox will work) on your host computer (as shown in [Getting Started, Out of the Box](#)). After the Bone has booted up, browse to <http://192.168.7.2:3000> on your host. You will see something like [Visual Studio Code](#).

Click the *examples* folder on the left and then click *BeagleBoard* and then *Black*, finally double-click `seqLEDs.py`. You can now edit the file.

---

**Note:** If you edit lines 33 and 37 of the `seqLEDs.py` file (`time.sleep(0.25)`), changing `0.25` to `0.1`, the LEDs next to the Ethernet port on your Bone will flash roughly twice as fast.

---

## 1.8 Running Python and JavaScript Applications from Visual Studio Code

### 1.8.1 Problem

You have a file edited in VS Code, and you want to run it.

### 1.8.2 Solution

VS Code has a *bash* command window built in at the bottom of the window. If it's not there, hit Ctrl-Shift-P and then type *terminal create new* then hit *Enter*. The terminal will appear at the bottom of the screen. You can run your code from this window. To do so, add `#!/usr/bin/env python` at the top of the file that you want to run and save.

---

**Tip:** If you are running JavaScript, replace the word **python** in the line with **node**.

---

At the bottom of the VS Code window are a series of tabs ([Visual Studio Code showing bash terminal](#)). Click the *TERMINAL* tab. Here, you have a command prompt.

Change to the directory that contains your file, make it executable, and then run it:

```
bone$ cd ~/examples/BeagleBone/Black/
bone$ ./seqLEDs.py
```

The `cd` is the change directory command. After you `cd`, you are in a new directory. Finally, `./seqLEDs.py` instructs the python script to run. You will need to press ^C (Ctrl-C) to stop your program.

### 1.8.3 Finding the Latest Version of the OS for Your Bone

#### Problem

You want to find out the latest version of Debian that is available for your Bone.

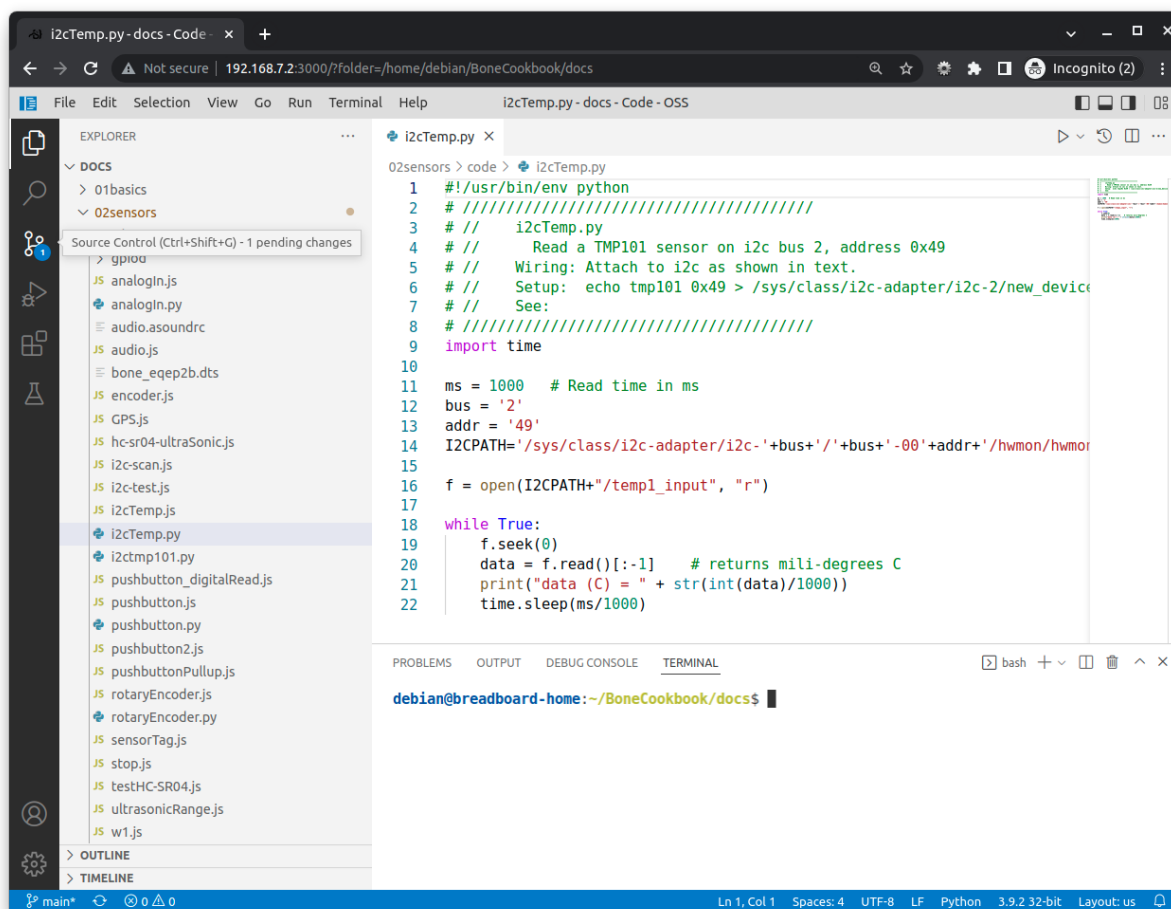


Fig. 1.5: Visual Studio Code showing bash terminal

## Solution

### bb-imager

The easiest way to see what the current images are and update your SD card is to use **bb-imager**. `beagle-ai-bb-imager` gives details on how to use it.

### forum

Another way to see the available images is to visit the beagleboard forum.

On your host computer, open a browser and go to <https://forum.beagleboard.org/tag/latest-images> This shows you a list of dates of the most recent Debian images (*Latest Debian images*).

---

**Todo:** Update for 2023-06-03

---

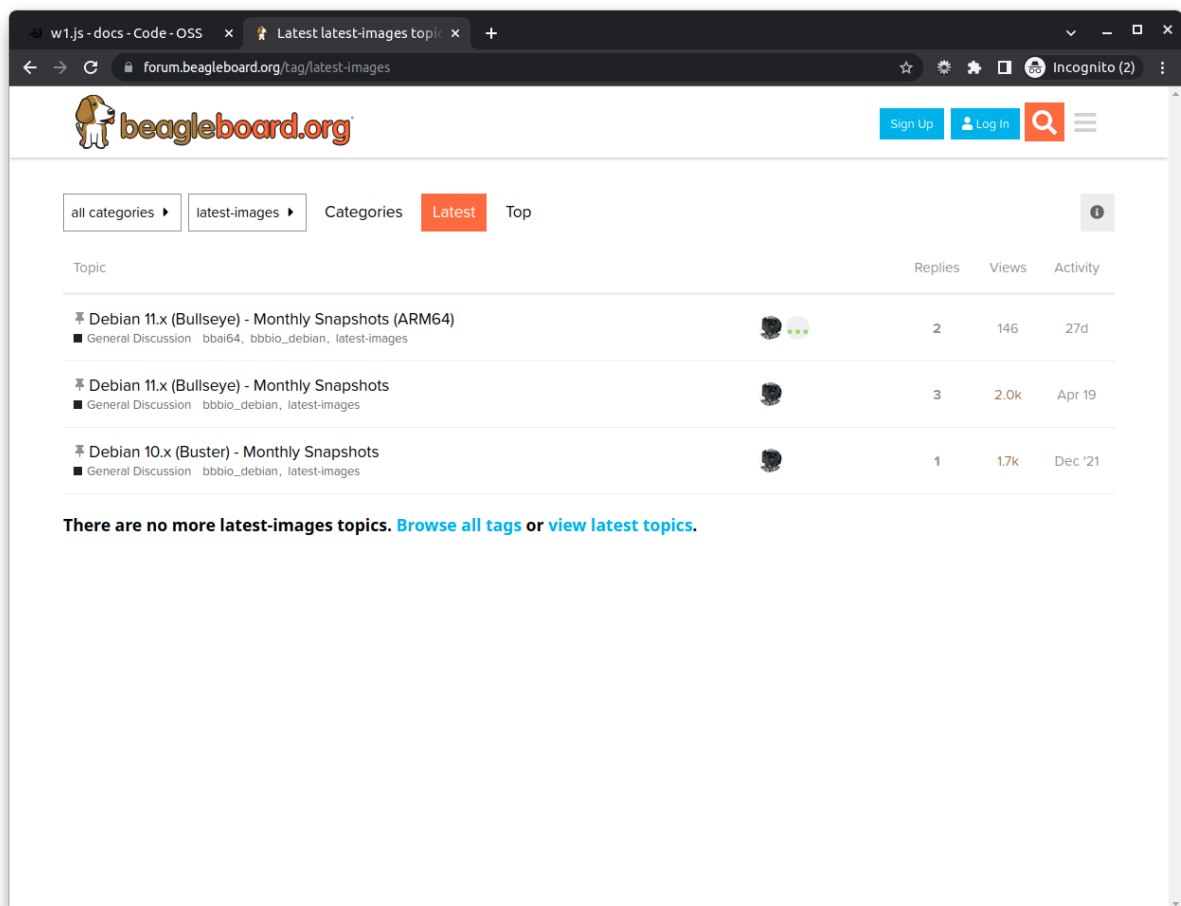


Fig. 1.6: Latest Debian images

At the time of writing, we are using the *Bullseye* image. Click on its link. Scrolling up you'll find *Latest Debian images*. There are three types of snapshots, Minimal, IoT and Xfce Desktop. IoT is the one we are running.

These are the images you want to use if you are flashing a Rev C BeagleBone Black onboard flash, or flashing a 4 GB or bigger microSD card. The image beginning with *am335x-debian-11.3-iot-* is used for the non-AI boards. The one beginning with *am57xx-debian-* is for programming the Beagle AI's.

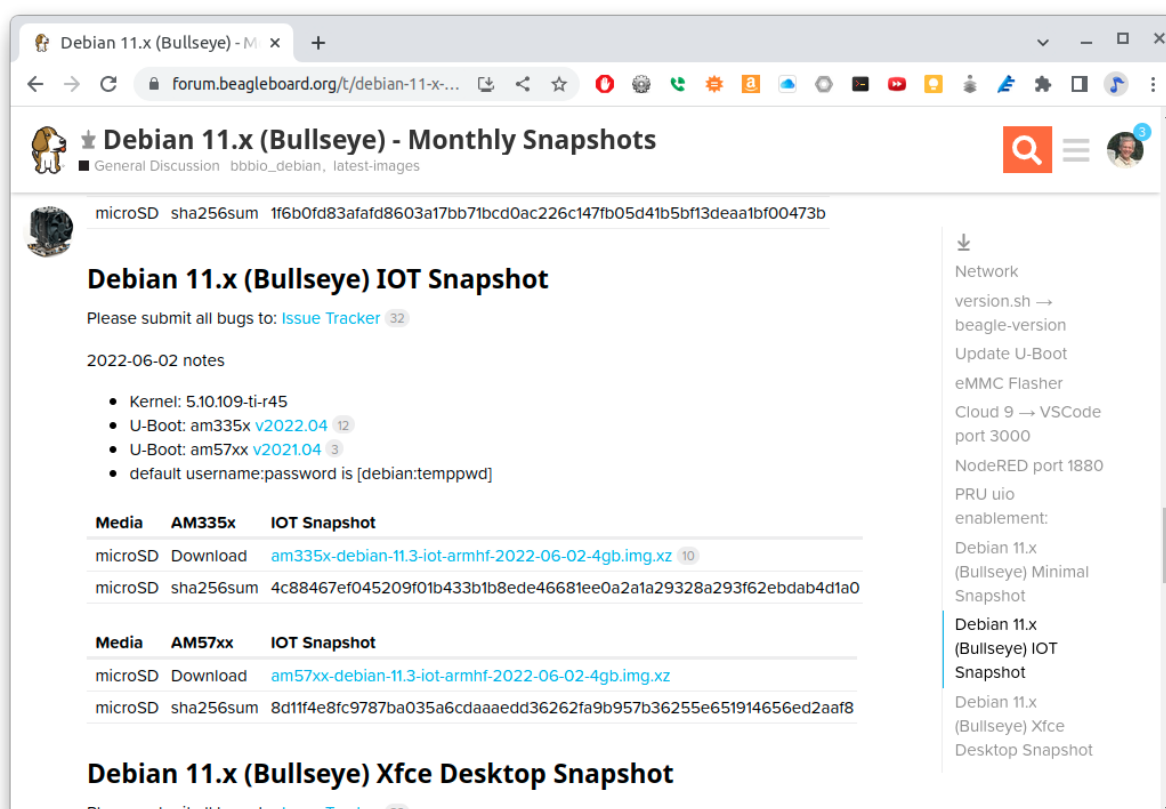


Fig. 1.7: Latest Debian images

---

**Note:** The onboard flash is often called the *eMMC* memory. We just call it *onboard flash*, but you'll often see *eMMC* appearing in filenames of images used to update the onboard flash.

---

Click the image you want to use and it will download. The images are some 500M, so it might take a while.

## 1.9 Running the Latest Version of the OS on Your Bone

### 1.9.1 Problem

You want to run the latest version of the operating system on your Bone without changing the onboard flash.

### 1.9.2 Solution

This solution is to flash an external microSD card and run the Bone from it. If you boot the Bone with a microSD card inserted with a valid boot image, it will boot from the microSD card. If you boot without the microSD card installed, it will boot from the onboard flash.

---

**Tip:** If you want to reflash the onboard flash memory, see [Updating the Onboard Flash](#).

---

---

**Note:** I instruct my students to use the microSD for booting. I suggest they keep an extra microSD flashed with the current OS. If they mess up the one on the Bone, it takes only a moment to swap in the extra microSD, boot up, and continue running. If they are running off the onboard flash, it will take much longer to reflash and boot from it.

---

Download the image you found in [Finding the Latest Version of the OS for Your Bone](#). It's more than 500 MB, so be sure to have a fast Internet connection. Then go to <http://beagleboard.org/getting-started#update> and follow the instructions there to install the image you downloaded.

## 1.10 Updating the OS on Your Bone

### 1.10.1 Problem

You've installed the latest version of Debian on your Bone ([Running the Latest Version of the OS on Your Bone](#)), and you want to be sure it's up-to-date.

### 1.10.2 Solution

Ensure that your Bone is on the network and then run the following command on the Bone:

```
bone$ sudo apt update
bone$ sudo apt upgrade
```

If there are any new updates, they will be installed.

---

**Note:** If you get the error *The following signatures were invalid: KEYEXPIRED 1418840246*, see [eLinux support page](#) for advice on how to fix it.

---

### 1.10.3 Discussion

After you have a current image running on the Bone, it's not at all difficult to keep it upgraded.

## 1.11 Backing Up the Onboard Flash

### 1.11.1 Problem

You've modified the state of your Bone in a way that you'd like to preserve or share. Note, this doesn't apply to boards that don't have onboard flash (PocketBeagle and BeagleY-AI).

### 1.11.2 Solution

The [eLinux wiki](#) page on [BeagleBone Black Extracting eMMC contents](#) provides some simple steps for copying the contents of the onboard flash to a file on a microSD card:

- Get a 4 GB or larger microSD card that is FAT formatted.
- If you create a FAT-formatted microSD card, you must edit the partition and ensure that it is a bootable partition.
- Download [beagleboneblack-save-emmc.zip](#) and uncompress and copy the contents onto your microSD card.
- Eject the microSD card from your computer, insert it into the powered-off BeagleBone Black, and apply power to your board.
- You'll notice *USER0* (the LED closest to the S1 button in the corner) will (after about 20 seconds) begin to blink steadily, rather than the double-pulse "heartbeat" pattern that is typical when your BeagleBone Black is running the standard Linux kernel configuration.
- It will run for a bit under 10 minutes and then *USER0* will stay on steady. That's your cue to remove power, remove the microSD card, and put it back into your computer.
- You will see a file called *BeagleBoneBlack-eMMC-image-XXXXX.img*, where *XXXXX* is a set of random numbers. Save this file to use for restoring your image later.

---

**Note:** Because the date won't be set on your board, you might want to adjust the date on the file to remember when you made it. For storage on your computer, these images will typically compress very well, so use your favorite compression tool.

---

---

**Tip:** The [eLinux wiki](#) is the definitive place for the BeagleBoard.org community to share information about the Beagles. Spend some time looking around for other helpful information.

---

## 1.12 Updating the Onboard Flash

### 1.12.1 Problem

You want to copy the microSD card to the onboard flash. Note, this doesn't apply to boards that don't have onboard flash (PocketBeagle and BeagleY-AI).

### 1.12.2 Solution

If you want to update the onboard flash with the contents of the microSD card,

- Repeat the steps in [Running the Latest Version of the OS on Your Bone](#) to update the OS.
- Attach to an external 5 V source. *you must be powered from an external 5 V source*. The flashing process requires more current than what typically can be pulled from USB.
- Boot from the microSD card.
- Log on to the bone and edit `/boot/uEnv.txt`.
- Uncomment out the last line `cmdline=init=/usr/sbin/init-beagle-flasher`.
- Save the file and reboot.
- The USR LEDs will flash back and forth for a few minutes.
- When they stop flashing, remove the SD card and reboot.
- You are now running from the newly flashed onboard flash.

**Warning:** If you write the onboard flash, **be sure to power the Bone from an external 5 V source**. The USB might not supply enough current.

When you boot from the microSD card, it will copy the image to the onboard flash. When all four *USER* LEDs turn off (in some versions, they all turn on), you can power down the Bone and remove the microSD card. The next time you power up, the Bone will boot from the onboard flash.





## Chapter 2

# Sensors

---

**Note:** Although the examples given here are originally for the BeagleBone Black, many will also work on the other Beagles too. See the *tabs* for details on running the examples on other boards.

---

In this chapter, you will learn how to sense the physical world with BeagleBone Black. Various types of electronic sensors, such as cameras and microphones, can be connected to the Bone using one or more interfaces provided by the standard USB 2.0 host port, as shown in [The USB 2.0 host port](#).

---

**Note:** All the examples in the book assume you have cloned the Cookbook repository on [git.beagleboard.org](http://git.beagleboard.org). Go here [Cloning the Cookbook Repository](#) for instructions.

---

### BeagleBone Black

The two 46-pin cape headers (called *P8* and *P9*) along the long edges of the board ([Cape Headers P8 and P9](#)) provide connections for cape add-on boards, digital and analog sensors, and more.

### BeagleY-AI

The 40-pin hat header along the long edge of the board provides connections for hat add-on boards, digital and analog sensors, and more.

The simplest kind of sensor provides a single digital status, such as off or on, and can be handled by an *input mode* of one of the Bone's 65 general-purpose input/output (GPIO) pins. More complex sensors can be connected by using one of the Bone's seven analog-to-digital converter (ADC) inputs or several I<sup>2</sup>C buses.

[Displays and Other Outputs](#) discusses some of the *output mode* usages of the GPIO pins.

All these examples assume that you know how to edit a file ([Editing Code Using Visual Studio Code](#)) and run it, either within the Visual Studio Code (VSC) integrated development environment (IDE) or from the command line ([Getting to the Command Shell via SSH](#)).

## 2.1 Choosing a Method to Connect Your Sensor

### 2.1.1 Problem

You want to acquire and attach a sensor and need to understand your basic options.

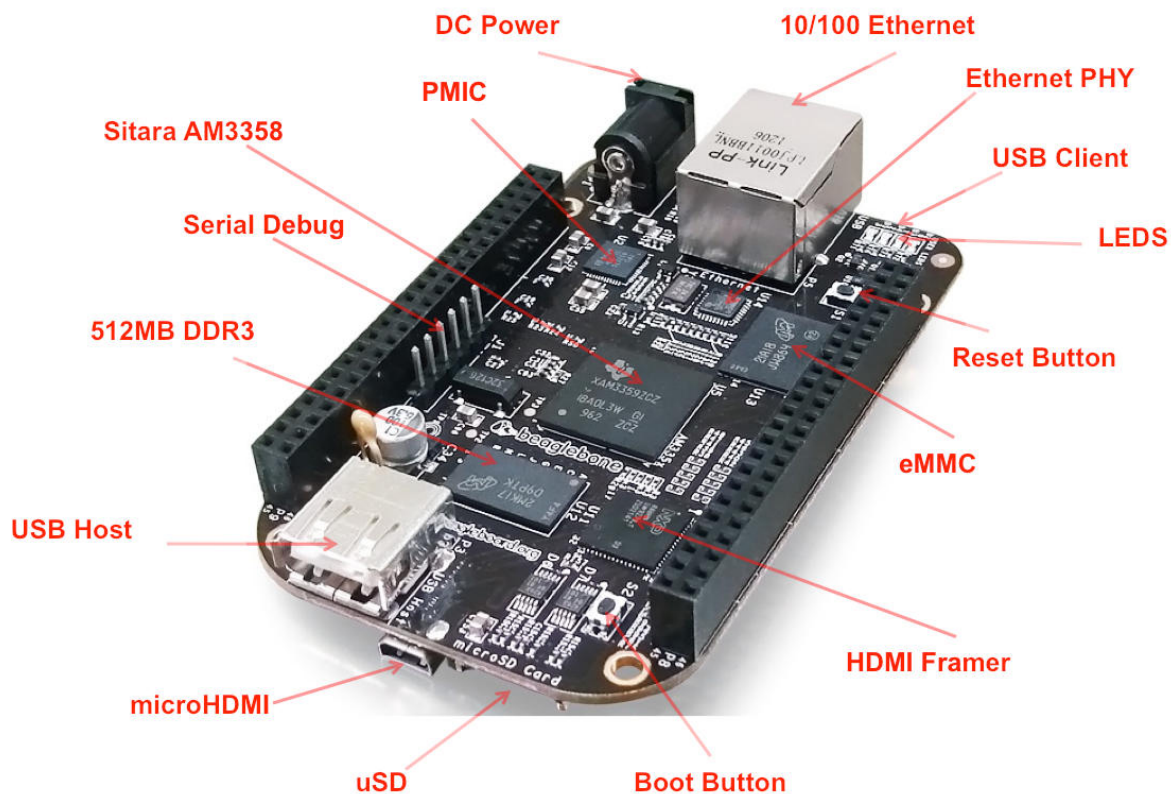


Fig. 2.1: The USB 2.0 host port

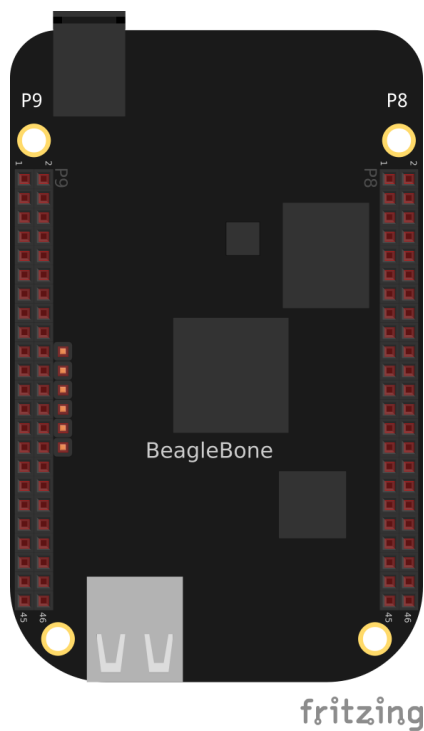


Fig. 2.2: Cape Headers P8 and P9

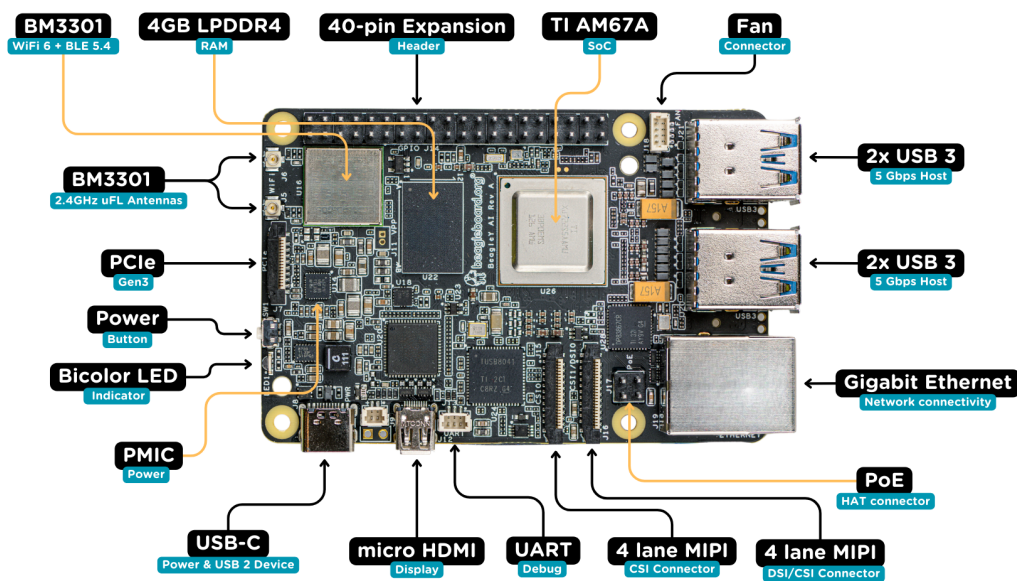


Fig. 2.3: BeagleY-AI Front View

## 2.1.2 Solution

### BeagleBones

*Some of the many sensor connection options on the Bone.* shows many of the possibilities for connecting a sensor.

### BeagleY-AI

*BeagleY-AI pinout* shows many of the possibilities for connecting a sensor.

You will see pins referenced in several ways. While this is confusing at first, in reality, we can pick our favorite way and stick to it.

The two main ways of referring to GPIOs is **by their number**, so GPIO2, GPIO3, GPIO4 etc. as seen in the diagram below. This corresponds to the SoC naming convention. For broad compatibility, BeagleY-AI re-uses the Broadcom GPIO numbering scheme used by RaspberryPi.

The second (and arguably easier) way we will use for this tutorial is to use the **actual pin header number** (shown in dark grey). So, for the rest of the tutorial, if we refer to **hat-08-gpio** we mean the **8th pin of the GPIO header**. Which, if you referenced the image below, can see refers to **GPIO14 (UART TXD)**

Go to <https://pinout.beagleboard.io/> to see an interactive version of the figure.

Choosing the simplest solution available enables you to move on quickly to addressing other system aspects. By exploring each connection type, you can make more informed decisions as you seek to optimize and troubleshoot your design.

## 2.2 Input and Run a Python or JavaScript Application for Talking to Sensors

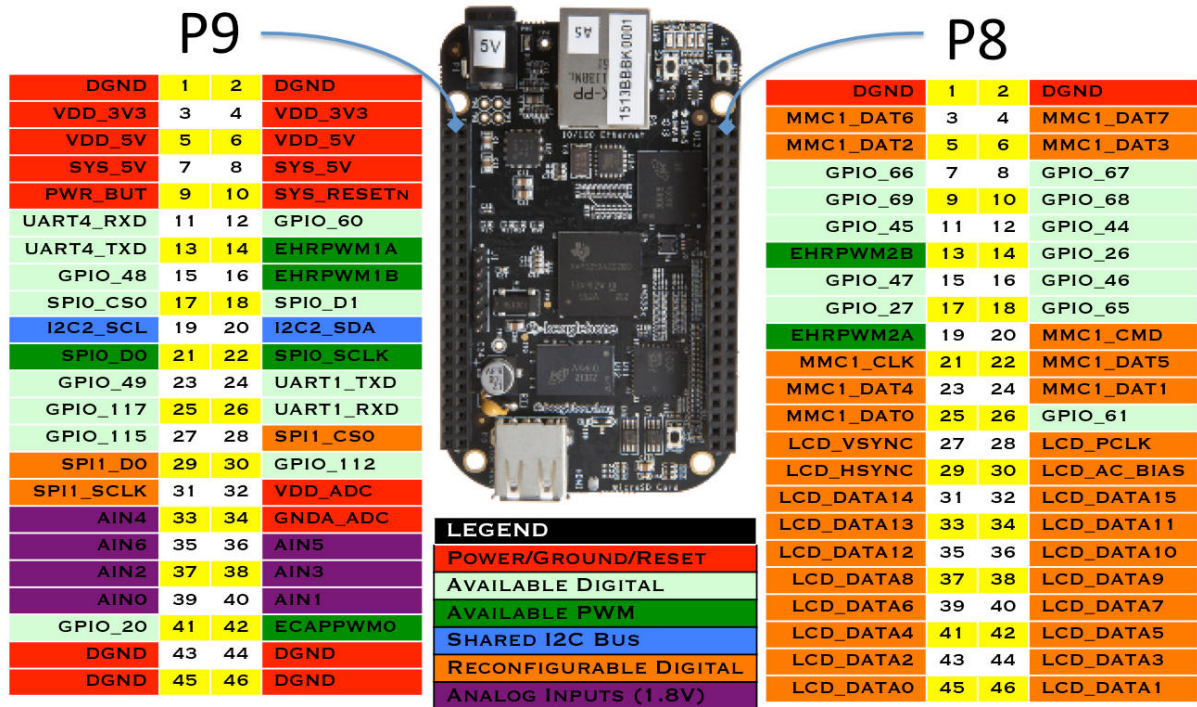


Fig. 2.4: Some of the many sensor connection options on the Bone.

### 2.2.1 Problem

You have your sensors all wired up and your Bone booted up, and you need to know how to enter and run your code.

### 2.2.2 Solution

You are just a few simple steps from running any of the recipes in this book.

- Plug your Bone into a host computer via the USB cable (*Getting Started, Out of the Box*).
- Start Visual Studio Code (*Editing Code Using Visual Studio Code*).
- In the *bash* tab (as shown in *Entering commands in the VSC bash tab*), run the following commands:

```
bone$ cd
bone$ cd beaglebone-cookbook-code/02sensors
```

Here, we issued the *change directory (cd)* command without specifying a target directory. By default, it takes you to your home directory. Notice that the prompt has changed to reflect the change.

---

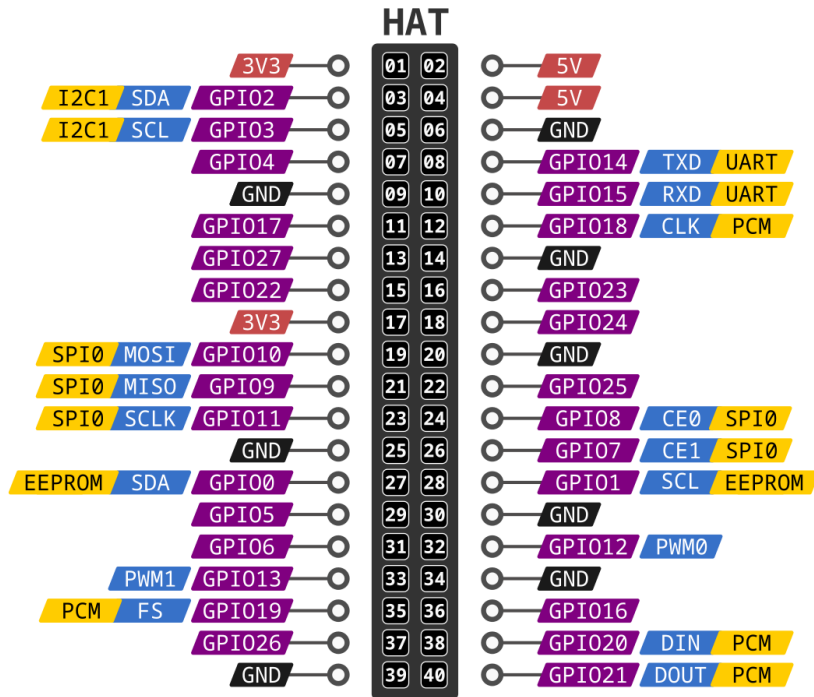
**Note:** If you log in as *debian*, your home is */home/debian*. If you were to create a new user called *newuser*, that user’s home would be */home/newuser*. By default, all non-root (non-superuser) users have their home directories in */home*.

---

**Note:** All the examples in the book assume you have cloned the Cookbook repository on [git.beagleboard.org](http://git.beagleboard.org). Go here [Cloning the Cookbook Repository](#) for instructions.

---

- Double-click the *pushbutton.py* file to open it.



- GND
- POWER
- GPIO NUMBER
- PIN FUNCTION

# BeagleY-AI

40-pin HAT header pinout

Fig. 2.5: BeagleY-AI pinout

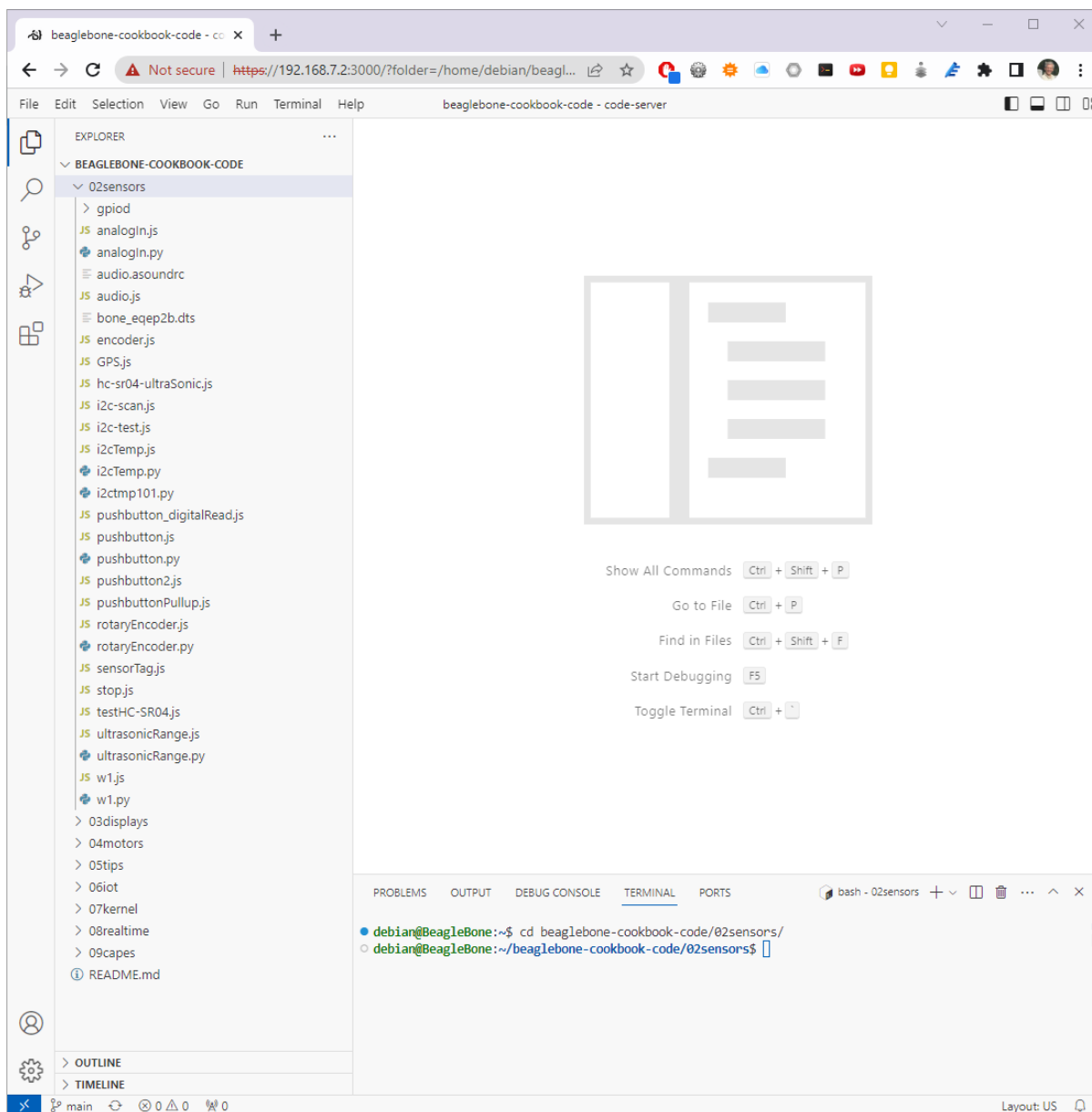


Fig. 2.6: Entering commands in the VSC bash tab

- Press `^S` (Ctrl-S) to save the file. (You can also go to the File menu in VSC and select Save to save the file, but Ctrl-S is easier.) Even easier, VSC can be configured to autosave every so many seconds.
- In the *bash* tab, enter the following commands:

```

debian@beaglebone:beaglebone-cookbook/code/02sensors$ ./pushbutton.py
data = 0
data = 0
data = 1
data = 1
^C

```

This process will work for any script in this book. (See the following sections for instructions on how to wire the pushbutton.)

## 2.3 Reading the Status of a Pushbutton or Magnetic Switch (Passive On/Off Sensor)

### 2.3.1 Problem

You want to read a pushbutton, a magnetic switch, or other sensor that is electrically open or closed.

### 2.3.2 Solution

Connect the switch to a GPIO pin and read from the proper place in `/sys/class/gpio`.

To make this recipe, you will need:

- Breadboard and jumper wires.
- Pushbutton switch.
- Magnetic reed switch. (optional)

You can wire up either a pushbutton, a magnetic reed switch, or both on the Bone, as shown in [Diagram for wiring a pushbutton and magnetic reed switch input](#).

The code below reads GPIO port `P9_42`, which is attached to the pushbutton.

---

**Note:** If you are using a BeagleY-AI, wire the button to **GPIO23** which is **hat-16**. This also appears at **gpiochip0** and line **7**.

---

### Python

Listing 2.1: Monitoring a pushbutton (pushbutton.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      pushbutton.py
4  # //      Reads P9_42 and prints its value.
5  # //      Wiring:      Connect a switch between P9_42 and 3.3V
6  # //      Setup:
7  # //      See:
8  # //////////////////////////////////////
9  import time
10 import gpiod
11 import os

```

(continues on next page)



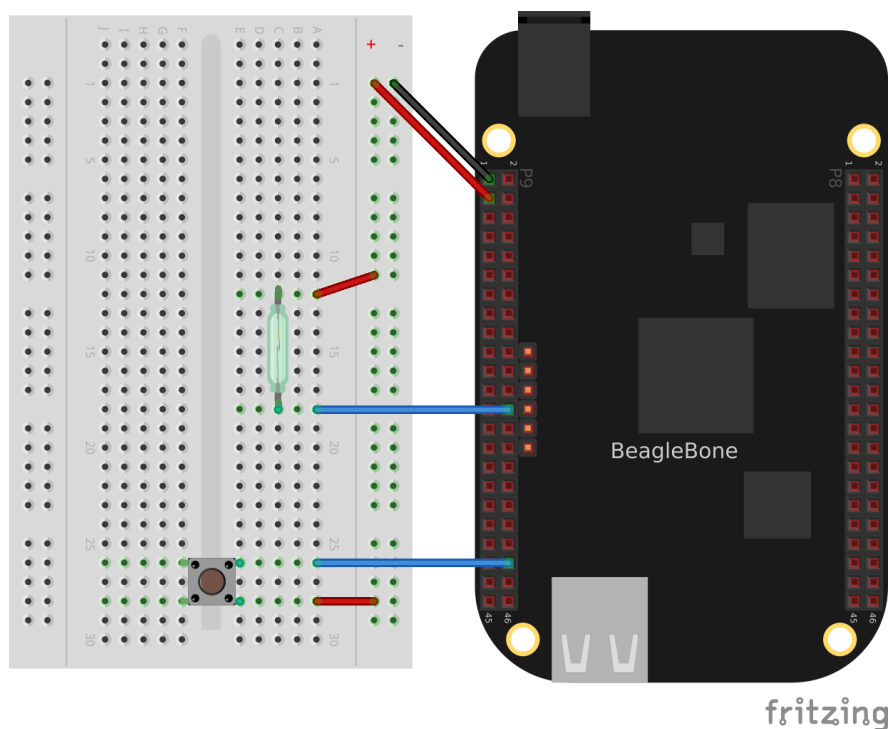


Fig. 2.7: Diagram for wiring a pushbutton and magnetic reed switch input

(continued from previous page)

```

12
13 ms = 100    # Read time in ms
14 CHIP = 'gpiochip0'
15 LINE_OFFSET = [7] # P9_42 is gpio 7
16 chip = gpiod.Chip(CHIP)
17 lines = chip.get_lines(LINE_OFFSET)
18 lines.request(consumer='pushbutton.py', type=gpiod.LINE_REQ_DIR_IN)
19
20 while True:
21     data = lines.get_values()
22     print('data = ' + str(data[0]))
23     time.sleep(ms/1000)

```

pushbutton.py

**c**

Listing 2.2: Monitoring a pushbutton (pushbutton.c)

```

1 ///////////////////////////////////////////////////////////////////
2 //      pushbutton.c
3 //      Reads P9_42 and prints its value.
4 //      Wiring:      Connect a switch between P9_42 and 3.3V
5 //      Setup:
6 //      See:
7 ///////////////////////////////////////////////////////////////////
8 #include <gpiod.h>
9 #include <stdio.h>
10 #include <unistd.h>
11
12 #define CONSUMER      "pushbutton.c"

```

(continues on next page)

(continued from previous page)

```

13
14 int main(int argc, char **argv)
15 {
16     int chipnumber = 0;
17     unsigned int line_num = 7;
18     struct gpiod_line *line;
19     struct gpiod_chip *chip;
20     int i, ret;
21
22     chip = gpiod_chip_open_by_number(chipnumber);
23     line = gpiod_chip_get_line(chip, line_num);
24     ret = gpiod_line_request_input(line, CONSUMER);
25
26     /* Get */
27     while(1) {
28         printf("%d\r", gpiod_line_get_value(line));
29         usleep(100);
30     }
31 }

```

pushbutton.c

Put this code in a file called *pushbutton.py* following the steps in [Input and Run a Python or JavaScript Application for Talking to Sensors](#). In the VSC *bash* tab, run it by using the following commands:

```

bone$ ./pushbutton.py
data = 0
data = 0
data = 1
data = 1
^C

```

The command runs it. Try pushing the button. The code reads the pin and prints its current value.

You will have to press ^C (Ctrl-C) to stop the code.

If you want to run the C version do:

```

bone$ gcc -o pushbutton pushbutton.c -lgpiod
bone$ ./pushbutton
data = 0
data = 0
data = 1
data = 1
^C

```

If you want to use the magnetic reed switch wired as shown in [Diagram for wiring a pushbutton and magnetic reed switch input](#), change *P9\_42* to *P9\_26* which is *gpio 14*.

## 2.4 Mapping Header Numbers to gpio Numbers

### 2.4.1 Problem

You have a sensor attached to the P8 or P9 header and need to know which gpio pin it is using.

### 2.4.2 Solution

The *gpioinfo* command displays information about all the P8 and P9 header pins. (Or the HAT header pins if you are on the BeagleY-AI.) To see the info for just one pin, use *grep*.

```
bone$ gpioinfo | grep -e chip -e P9.42
gpiochip0 - 32 lines:
    line 7: "P8_42A [ecappwm0]" "P9_42" input active-high [used]
gpiochip1 - 32 lines:
gpiochip2 - 32 lines:
gpiochip3 - 32 lines:
```

Or, if on the BeagleY-AI.

```
bone$ gpioinfo | grep -e chip -e GPIO23
gpiochip0 - 24 lines:
    line 7: "GPIO23" unused input active-high
gpiochip1 - 87 lines:
gpiochip2 - 73 lines:
```

This shows P9\_42 (GPIO32) is on chip 0 and pin 7. To find the gpio number multiply the chip number by 32 and add it to the pin number. This gives  $0*32+7=7$ .

For P9\_26 you get:

```
bone$ gpioinfo | grep -e chip -e P9.26
gpiochip0 - 32 lines:
    line 14: "P9_26 [uart1_rxd]" "P9_26" input active-high [used]
gpiochip1 - 32 lines:
gpiochip2 - 32 lines:
gpiochip3 - 32 lines:
```

$0*32+14=14$ , so the P9\_26 pin is gpio 14.

## 2.5 Reading a Position, Light, or Force Sensor (Variable Resistance Sensor)

### 2.5.1 Problem

You have a variable resistor, force-sensitive resistor, flex sensor, or any of a number of other sensors that output their value as a variable resistance, and you want to read their value with the Bone.

### 2.5.2 Solution

---

**Note:** The BeagleY-AI doesn't have ADC's, so you can skip this section.

---

Use the Bone's analog-to-digital converters (ADCs) and a resistor divider circuit to detect the resistance in the sensor.

The Bone has seven built-in analog inputs that can easily read a resistive value. [Seven analog inputs on P9 header](#) shows them on the lower part of the P9 header.

To make this recipe, you will need:

- Breadboard and jumper wires.
- 10k trimpot or
- Flex resistor (optional)
- 22 kΩ resistor

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	GPIO_22	19	20	GPIO_63
GPIO_3	21	22	GPIO_2	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	GPIO_15	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_14	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GNDA_ADC	GPIO_9	33	34	GPIO_81
AIN6	35	36	AIN5	GPIO_8	35	36	GPIO_80
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 2.8: Seven analog inputs on P9 header

### A variable resistor with three terminals

[Wiring a 10 kΩ variable resistor \(trimpot\) to an ADC port](#) shows a simple variable resistor (trimpot) wired to the Bone. One end terminal is wired to the ADC 1.8 V power supply on pin *P9\_32*, and the other end terminal is attached to the ADC ground (*P9\_34*). The middle terminal is wired to one of the seven analog-in ports (*P9\_36*).

The section below shows the code used to read the variable resistor. Add the code to a file called *analogIn.py* and run it; then change the resistor and run it again. The voltage read will change.

### Python

Listing 2.3: Reading an analog voltage (analogIn.py)

```

1  #!/usr/bin/env python3
2  #////////////////////////////////////
3  #      analogin.py
4  #      Reads the analog value of the light sensor.
5  #////////////////////////////////////
6  import time
7  import os
8
9  pin = "2"          # light sensor, A2, P9_37
10
11  IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw'
12
13  print('Hit ^C to stop')
14
15  f = open(IIOPATH, "r")
16
17  while True:

```

(continues on next page)

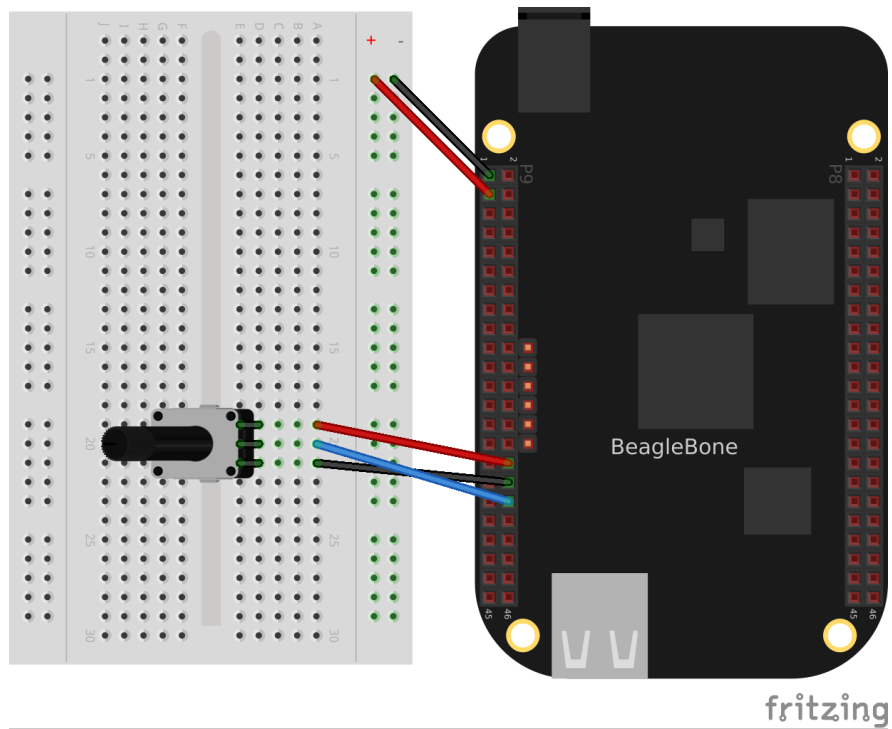


Fig. 2.9: Wiring a 10 kΩ variable resistor (trimpot) to an ADC port

(continued from previous page)

```

18     f.seek(0)
19     x = float(f.read())/4096
20     print('{}: {:.1f}%, {:.3f} V'.format(pin, 100*x, 1.8*x), end = '\r')
21     time.sleep(0.1)
22
23 # // Bone   | Pocket | AIN
24 # // ----- | ----- | ---
25 # // P9_39 | P1_19 | 0
26 # // P9_40 | P1_21 | 1
27 # // P9_37 | P1_23 | 2
28 # // P9_38 | P1_25 | 3
29 # // P9_33 | P1_27 | 4
30 # // P9_36 | P2_35 | 5
31 # // P9_35 | P1_02 | 6
    
```

analogIn.py

### JavaScript

Listing 2.4: Reading an analog voltage (analogIn.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      analogin.js
4  //      Reads the analog value of the light sensor.
5  //////////////////////////////////////
6  const fs = require("fs");
7  const ms = 500; // Time in milliseconds
8
9  const pin = "2"; // light sensor, A2, P9_37
10
    
```

(continues on next page)

(continued from previous page)

```

11 const IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw';
12
13 console.log('Hit ^C to stop');
14
15 // Read every 500ms
16 setInterval(readPin, ms);
17
18 function readPin() {
19     var data = fs.readFileSync(IIOPATH).slice(0, -1);
20     console.log('data = ' + data);
21 }
22 // Bone | Pocket | AIN
23 // ---- | ----- | ---
24 // P9_39 | P1_19 | 0
25 // P9_40 | P1_21 | 1
26 // P9_37 | P1_23 | 2
27 // P9_38 | P1_25 | 3
28 // P9_33 | P1_27 | 4
29 // P9_36 | P2_35 | 5
30 // P9_35 | P1_02 | 6

```

analogIn.js

**Note:** The code above outputs a value between 0 and 4096.

### A variable resistor with two terminals

Some resistive sensors have only two terminals, such as the flex sensor in [Reading a two-terminal flex resistor](#). The resistance between its two terminals changes when it is flexed. In this case, we need to add a fixed resistor in series with the flex sensor. [Reading a two-terminal flex resistor](#) shows how to wire in a 22 k $\Omega$  resistor to give a voltage to measure across the flex sensor.

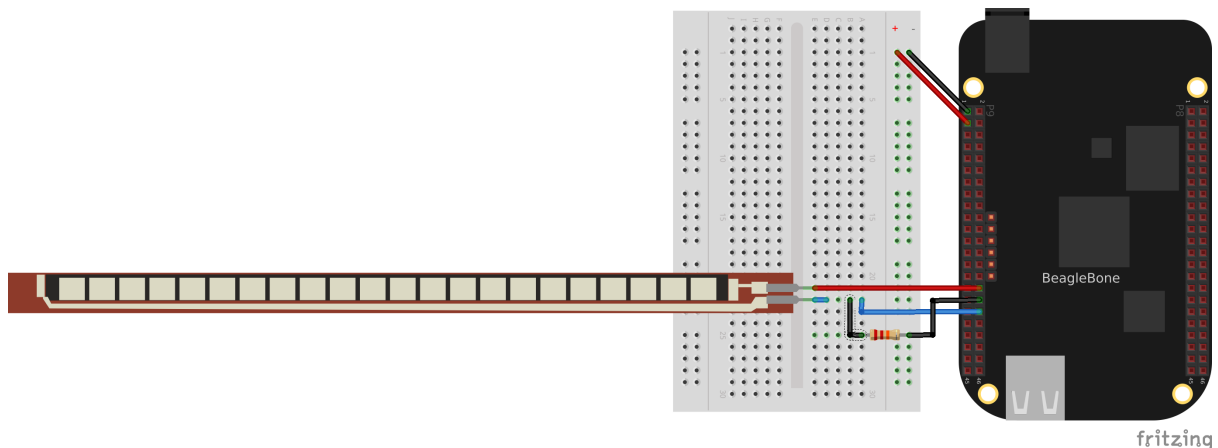


Fig. 2.10: Reading a two-terminal flex resistor

The code in `analogIn.py` also works for this setup.

## 2.6 Reading a Distance Sensor (Analog or Variable Voltage Sensor)

### 2.6.1 Problem

You want to measure distance with a [LV-MaxSonar-EZ1 Sonar Range Finder](#), which outputs a voltage in proportion to the distance.

### 2.6.2 Solution

---

**Note:** The BeagleY-AI doesn't have ADC's, so you can skip this section.

---

To make this recipe, you will need:

- Breadboard and jumper wires.
- LV-MaxSonar-EZ1 Sonar Range Finder

All you have to do is wire the EZ1 to one of the Bone's *analog-in* pins, as shown in [Wiring the LV-MaxSonar-EZ1 Sonar Range Finder to the P9\\_33 analog-in port](#). The device outputs  $\sim 6.4$  mV/in when powered from 3.3 V.

**Warning:** Make sure not to apply more than 1.8 V to the Bone's *analog-in* pins, or you will likely damage them. In practice, this circuit should follow that rule.

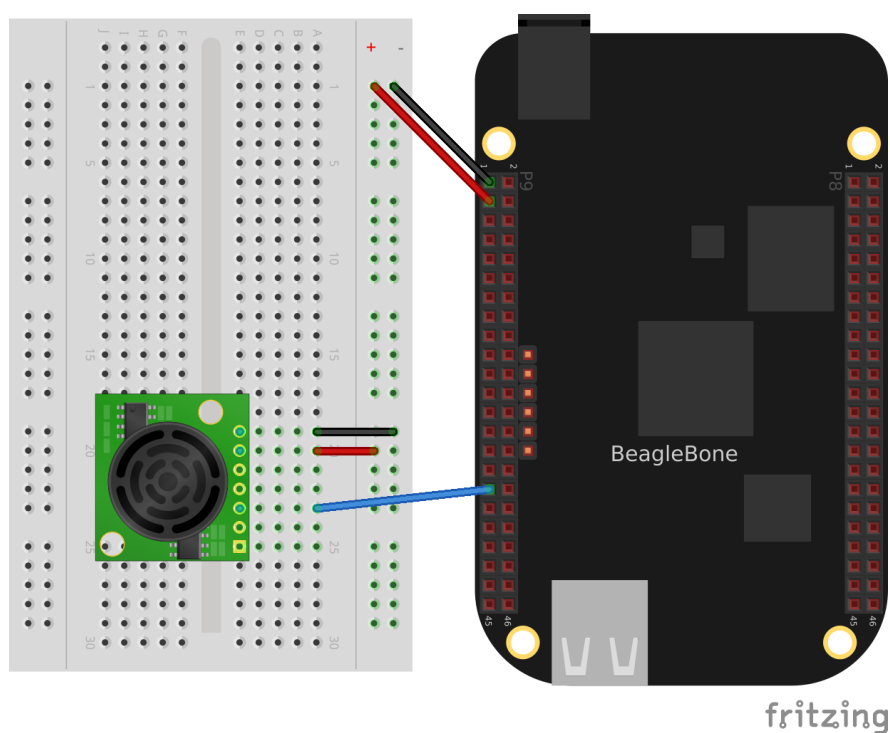


Fig. 2.11: Wiring the LV-MaxSonar-EZ1 Sonar Range Finder to the P9\_33 analog-in port

`ultrasonicRange.py` shows the code that reads the sensor at a fixed interval.

### Python

Listing 2.5: Reading an analog voltage (ultrasonicRange.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      ultrasonicRange.js
4  # //      Reads the analog value of the sensor.
5  # //////////////////////////////////////
6  import time
7  ms = 250; # Time in milliseconds
8
9  pin = "0"      # sensor, A0, P9_39
10
11  IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw'
12
13  print('Hit ^C to stop');
14
15  f = open(IIOPATH, "r")
16  while True:
17      f.seek(0)
18      data = f.read()[:-1]
19      print('data= ' + data)
20      time.sleep(ms/1000)
21
22  # // Bone   | Pocket | AIN
23  # // ----- | ----- | ---
24  # // P9_39 | P1_19 | 0
25  # // P9_40 | P1_21 | 1
26  # // P9_37 | P1_23 | 2
27  # // P9_38 | P1_25 | 3
28  # // P9_33 | P1_27 | 4
29  # // P9_36 | P2_35 | 5
30  # // P9_35 | P1_02 | 6

```

ultrasonicRange.py

### JavaScript

Listing 2.6: Reading an analog voltage (ultrasonicRange.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      ultrasonicRange.js
4  //      Reads the analog value of the sensor.
5  //////////////////////////////////////
6  const fs = require("fs");
7  const ms = 250; // Time in milliseconds
8
9  const pin = "0"; // sensor, A0, P9_39
10
11  const IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw';
12
13  console.log('Hit ^C to stop');
14
15  // Read every ms
16  setInterval(readPin, ms);
17
18  function readPin() {
19      var data = fs.readFileSync(IIOPATH);
20      console.log('data= ' + data);
21  }
22  // Bone   | Pocket | AIN

```

(continues on next page)



```
23 // ----- | ----- | ---
24 // P9_39 | P1_19 | 0
25 // P9_40 | P1_21 | 1
26 // P9_37 | P1_23 | 2
27 // P9_38 | P1_25 | 3
28 // P9_33 | P1_27 | 4
29 // P9_36 | P2_35 | 5
30 // P9_35 | P1_02 | 6
```

ultrasonicRange.js

## 2.7 Reading a Distance Sensor (Variable Pulse Width Sensor)

### 2.7.1 Problem

You want to use a HC-SR04 Ultrasonic Range Sensor with BeagleBone Black.

### 2.7.2 Solution

The HC-SR04 Ultrasonic Range Sensor (shown in [HC-SR04 Ultrasonic range sensor](#)) works by sending a trigger pulse to the *Trigger* input and then measuring the pulse width on the *Echo* output. The width of the pulse tells you the distance.



Fig. 2.12: HC-SR04 Ultrasonic range sensor

To make this recipe, you will need:

- Breadboard and jumper wires.
- 10 k $\Omega$  and 20 k $\Omega$  resistors
- HC-SR04 Ultrasonic Range Sensor.

Wire the sensor as shown in [Wiring an HC-SR04 Ultrasonic Sensor](#). Note that the HC-SR04 is a 5 V device, so the *banded wire* (running from P9\_7 on the Bone to VCC on the range finder) attaches the HC-SR04 to the Bone's 5 V power supply.

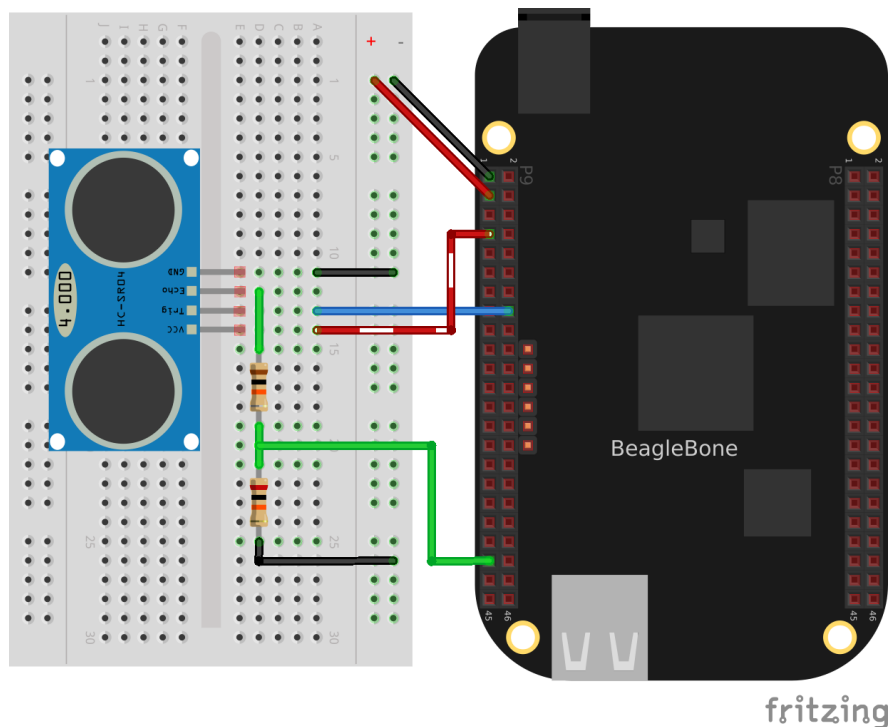


Fig. 2.13: Wiring an HC-SR04 Ultrasonic Sensor

[Driving a HC-SR04 ultrasound sensor \(hc-sr04-ultraSonic.js\)](#) shows BoneScript code used to drive the HC-SR04.

Listing 2.7: Driving a HC-SR04 ultrasound sensor (hc-sr04-ultraSonic.js)

```

1  #!/usr/bin/env node
2
3  // This is an example of reading HC-SR04 Ultrasonic Range Finder
4  // This version measures from the fall of the Trigger pulse
5  //   to the end of the Echo pulse
6
7  var b = require('bonescript');
8
9  var trigger = 'P9_16', // Pin to trigger the ultrasonic pulse
10     echo    = 'P9_41', // Pin to measure to pulse width related to the
    ↳distance
11     ms = 250;          // Trigger period in ms
12
13 var startTime, pulseTime;
14
15 b.pinMode(echo, b.INPUT, 7, 'pulldown', 'fast', doAttach);
16 function doAttach(x) {
17     if(x.err) {
18         console.log('x.err = ' + x.err);
19         return;
20     }
21     // Call pingEnd when the pulse ends
22     b.attachInterrupt(echo, true, b.FALLING, pingEnd);
23 }
24
25 b.pinMode(trigger, b.OUTPUT);

```

(continues on next page)

```

26
27 b.digitalWrite(trigger, 1);      // Unit triggers on a falling edge.
28                               // Set trigger to high so we call pull it_
    ↳ low later
29
30 // Pull the trigger low at a regular interval.
31 setInterval(ping, ms);
32
33 // Pull trigger low and start timing.
34 function ping() {
35     // console.log('ping');
36     b.digitalWrite(trigger, 0);
37     startTime = process.hrtime();
38 }
39
40 // Compute the total time and get ready to trigger again.
41 function pingEnd(x) {
42     if(x.attached) {
43         console.log("Interrupt handler attached");
44         return;
45     }
46     if(startTime) {
47         pulseTime = process.hrtime(startTime);
48         b.digitalWrite(trigger, 1);
49         console.log('pulseTime = ' + (pulseTime[1]/1000000-0.8).toFixed(3));
50     }
51 }

```

hc-sr04-ultraSonic.js

This code is more complex than others in this chapter, because we have to tell the device when to start measuring and time the return pulse.

## 2.8 Accurately Reading the Position of a Motor or Dial

### 2.8.1 Problem

---

**Todo:** Update for BeagleY-AI

---

You have a motor or dial and want to detect rotation using a rotary encoder.

### 2.8.2 Solution

Use a rotary encoder (also called a *quadrature encoder*) connected to one of the Bone's eQEP ports, as shown in [Wiring a rotary encoder using eQEP2](#).

Table 2.1: On the BeagleBone and PocketBeagle the three encoders are:

eQEP0	P9.27 and P9.42 OR P1_33 and P2_34
eQEP1	P9.33 and P9.35
eQEP2	P8.11 and P8.12 OR P2_24 and P2_33

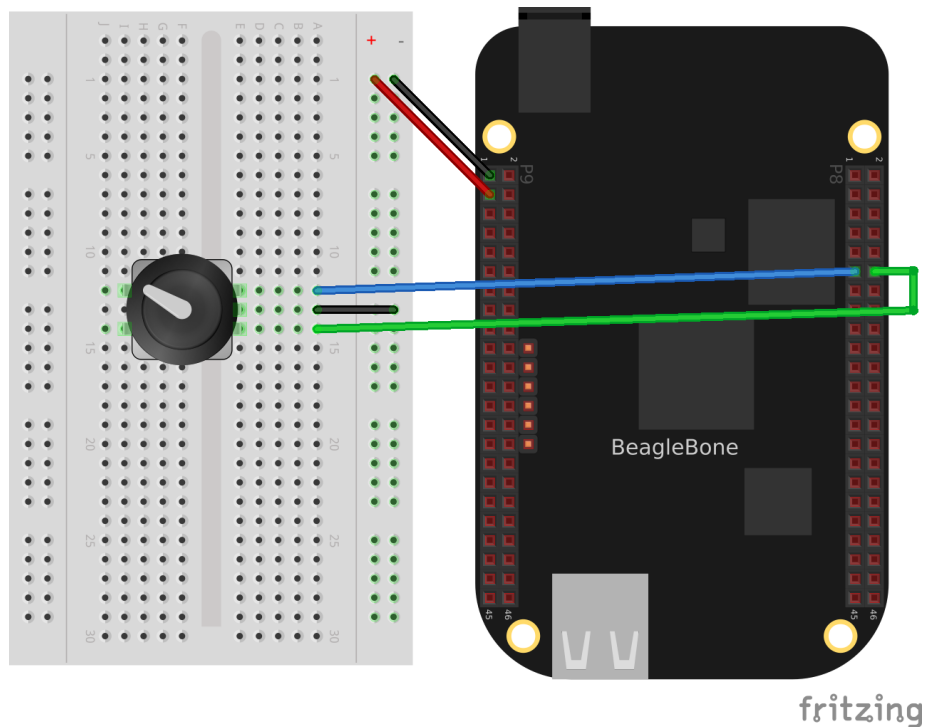


Fig. 2.14: Wiring a rotary encoder using eQEP2

Table 2.2: On the AI it's:

eQEP1	P8.33 and P8.35
eQEP2	P8.11 and P8.12 or P9.19 and P9.41
eQEP3	P8.24 and P8.25 or P9.27 and P9.42

To make this recipe, you will need:

- Breadboard and jumper wires.
- Rotary encoder.

We are using a quadrature rotary encoder, which has two switches inside that open and close in such a manner that you can tell which way the shaft is turning. In this particular encoder, the two switches have a common lead, which is wired to ground. It also has a pushbutton switch wired to the other side of the device, which we aren't using.

Wire the encoder to *P8\_11* and *P8\_12*, as shown in [Wiring a rotary encoder using eQEP2](#).

BeagleBone Black has built-in hardware for reading up to three encoders. Here, we'll use the *eQEP2* encoder via the Linux *count* subsystem.

Then run the following commands:

```
bone$ config-pin P8_11 qep
bone$ config-pin P8_12 qep
bone$ show-pins | grep qep
P8.12      12 fast rx  up   4 qep 2 in A   ocp/P8_12_pinmux (pinmux_P8_12_
→qep_pin)
P8.11      13 fast rx  up   4 qep 2 in B   ocp/P8_11_pinmux (pinmux_P8_11_
→qep_pin)
```

This will enable *eQEP2* on pins *P8\_11* and *P8\_12*. The 2 after the *qep* returned by *show-pins* shows it's *eQEP2*. Finally, add the code below to a file named *rotaryEncoder.py* and run it.

## Python

Listing 2.8: Reading a rotary encoder (rotaryEncoder.py)

```

1  #!/usr/bin/env python
2  # // This uses the eQEP hardware to read a rotary encoder
3  # // bone$ config-pin P8_11 eqep
4  # // bone$ config-pin P8_12 eqep
5  import time
6
7  eQEP = '2'
8  COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0'
9
10 ms = 100          # Time between samples in ms
11 maxCount = '1000000'
12
13 # Set the eEQP maximum count
14 f = open(COUNTERPATH+'/ceiling', 'w')
15 f.write(maxCount)
16 f.close()
17
18 # Enable
19 f = open(COUNTERPATH+'/enable', 'w')
20 f.write('1')
21 f.close()
22
23 f = open(COUNTERPATH+'/count', 'r')
24
25 olddata = -1
26 while True:
27     f.seek(0)
28     data = f.read()[:-1]
29     # Print only if data changes
30     if data != olddata:
31         olddata = data
32         print("data = " + data)
33         time.sleep(ms/1000)
34
35 # Black OR Pocket
36 # eQEP0:          P9.27 and P9.42 OR P1_33 and P2_34
37 # eQEP1:          P9.33 and P9.35
38 # eQEP2:          P8.11 and P8.12 OR P2_24 and P2_33
39
40 # AI
41 # eQEP1:          P8.33 and P8.35
42 # eQEP2:          P8.11 and P8.12 or P9.19 and P9.41
43 # eQEP3:          P8.24 and P8.25 or P9.27 and P9.42

```

rotaryEncoder.py

## JavaScript

Listing 2.9: Reading a rotary encoder (rotaryEncoder.js)

```

1  #!/usr/bin/env node
2  // This uses the eQEP hardware to read a rotary encoder
3  // bone$ config-pin P8_11 eqep
4  // bone$ config-pin P8_12 eqep
5  const fs = require("fs");
6
7  const eQEP = "2";

```

(continues on next page)

(continued from previous page)

```

8  const COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0';
9
10 const ms = 100;           // Time between samples in ms
11 const maxCount = '1000000';
12
13 // Set the eEQP maximum count
14 fs.writeFileSync(COUNTERPATH+'/ceiling', maxCount);
15
16 // Enable
17 fs.writeFileSync(COUNTERPATH+'/enable', '1');
18
19 setInterval(readEncoder, ms);    // Check state every ms
20
21 var olddata = -1;
22 function readEncoder() {
23     var data = parseInt(fs.readFileSync(COUNTERPATH+'/count'));
24     if(data !== olddata) {
25         // Print only if data changes
26         console.log('data = ' + data);
27         olddata = data;
28     }
29 }
30
31 // Black OR Pocket
32 // eQEP0:      P9.27 and P9.42 OR P1_33 and P2_34
33 // eQEP1:      P9.33 and P9.35
34 // eQEP2:      P8.11 and P8.12 OR P2_24 and P2_33
35
36 // AI
37 // eQEP1:      P8.33 and P8.35
38 // eQEP2:      P8.11 and P8.12 or P9.19 and P9.41
39 // eQEP3:      P8.24 and P8.25 or P9.27 and P9.42

```

rotaryEncoder.js

Try rotating the encoder clockwise and counter-clockwise. You'll see an output like this:

```

data = 32
data = 40
data = 44
data = 48
data = 39
data = 22
data = 0
data = 999989
data = 999973
data = 999972
^C

```

The values you get for *data* will depend on which way you are turning the device and how quickly. You will need to press ^C (Ctrl-C) to end.

### 2.8.3 See Also

You can also measure rotation by using a variable resistor (see [Wiring a 10 kΩ variable resistor \(trimpot\) to an ADC port](#)).

## 2.9 Acquiring Data by Using a Smart Sensor over a Serial Connection

### 2.9.1 Problem

You want to connect a smart sensor that uses a built-in microcontroller to stream data, such as a global positioning system (GPS), to the Bone and read the data from it.

### 2.9.2 Solution

The Bone has several serial ports (UARTs) that you can use to read data from an external microcontroller included in smart sensors, such as a GPS. Just wire one up, and you'll soon be gathering useful data, such as your own location.

Here's what you'll need:

- Breadboard and jumper wires.
- GPS receiver

Wire your GPS, as shown in [Wiring a GPS to UART 4](#).

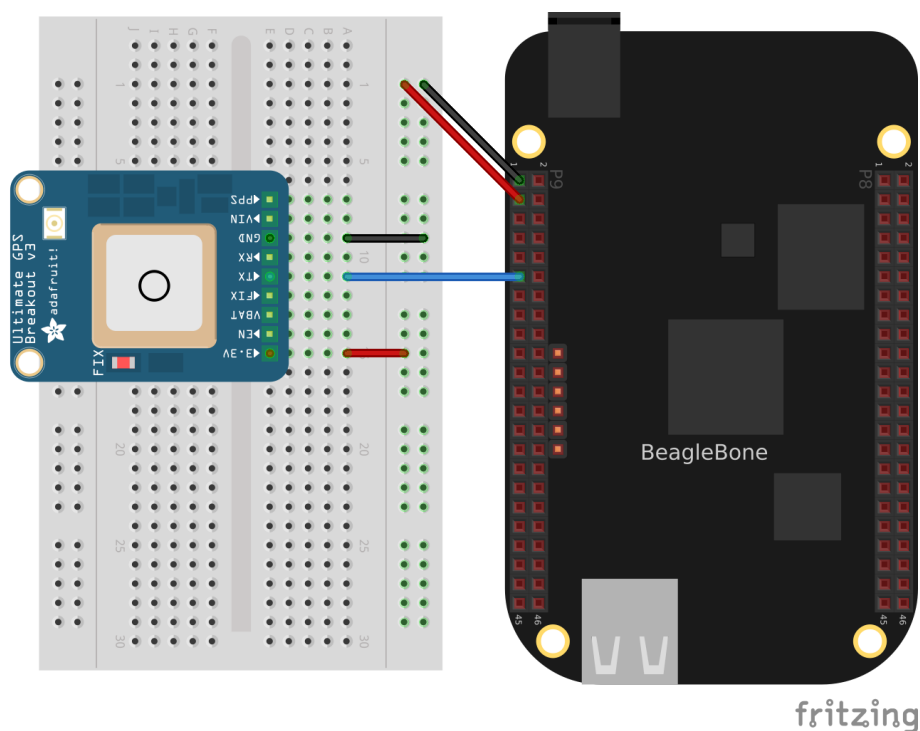


Fig. 2.15: Wiring a GPS to UART 4

The GPS will produce raw National Marine Electronics Association (NMEA) data that's easy for a computer to read, but not for a human. There are many utilities to help convert such sensor data into a human-readable form. For this GPS, run the following command to load a NMEA parser:

```
bone$ npm install -g nmea
```

Running the code in [Talking to a GPS with UART 4 \(GPS.js\)](#) will print the current location every time the GPS outputs it.

Listing 2.10: Talking to a GPS with UART 4 (GPS.js)

```

1 #!/usr/bin/env node
2  // Install with: npm install nmea
3
4  // Need to add exports.serialParsers = m.module.parsers;
5  // to the end of /usr/local/lib/node_modules/bonescript/serial.js
6
7  var b = require('bonescript');
8  var nmea = require('nmea');
9
10 var port = '/dev/ttyO4';
11 var options = {
12     baudrate: 9600,
13     parser: b.serialParsers.readline("\n")
14 };
15
16 b.serialOpen(port, options, onSerial);
17
18 function onSerial(x) {
19     if (x.err) {
20         console.log('***ERROR*** ' + JSON.stringify(x));
21     }
22     if (x.event == 'open') {
23         console.log('***OPENED***');
24     }
25     if (x.event == 'data') {
26         console.log(String(x.data));
27         console.log(nmea.parse(x.data));
28     }
29 }

```

GPS.js

If you don't need the NMEA formatting, you can skip the *npm* part and remove the lines in the code that refer to it.

---

**Note:** If you get an error like this *TypeError: Cannot call method 'readline' of undefined*

add this line to the end of file `/usr/local/lib/node_modules/bonescript/serial.js`:

```
exports.serialParsers = m.module.parsers;
```

---

## 2.10 Measuring a Temperature

### 2.10.1 Problem

You want to measure a temperature using a digital temperature sensor.

### 2.10.2 Solution

The TMP101 sensor is a common digital temperature sensor that uses a standard I<sup>2</sup>C-based serial protocol.

To make this recipe, you will need:

- Breadboard and jumper wires.
- Two 4.7 kΩ resistors.



P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
UART4_RXD	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
UART4_TXD	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
UART1_RTSN	19	20	UART1_CTSN	GPIO_22	19	20	GPIO_63
UART2_TXD	21	22	UART2_RXD	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	UART1_TXD	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	UART1_RXD	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	UART5_CTSN+	31	32	UART5_RTSN
AIN4	33	34	GNDA_ADC	UART4_RTSN	33	34	UART3_RTSN
AIN6	35	36	AIN5	UART4_CTSN	35	36	UART3_CTSN
AIN2	37	38	AIN3	UARR5_TXD+	37	38	UART5_RXD+
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	UART3_TXD	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 2.16: Table of UART outputs

- TMP101 temperature sensor.

### BeagleBone

Wire the TMP101, as shown in [Wiring an I2C TMP101 temperature sensor](#).

There are two I<sup>2</sup>C buses brought out to the headers. [Table of I2C outputs](#) shows that you have wired your device to I<sup>2</sup>C bus 2.

### BeagleY-AI

Running the following on the BeagleY-AI shows it has five i2c buses.

```
bone$ ls /sys/bus/i2c/devices/
2-0030 2-0050 2-0068 4-004c i2c-1 i2c-2 i2c-3 i2c-4 i2c-5
```

But running <https://pinout.beagleboard.io/> show only buses 1 and 4 are exposed on the HAT header. Here we'll use bus 2 whose clock appears on *hat-03* and data on *hat-05*.

Wire your **tmp101** as shown in the table.

Function	hat	tmp101
Ground	09	2
3.3V	01	5
data	03	6
clock	05	1

Once the I<sup>2</sup>C device is wired up, you can use a couple handy I<sup>2</sup>C tools to test the device. Because these are Linux command-line tools, you have to use 2 as the bus number. *i2cdetect*, shown in [I2C tools](#), shows which

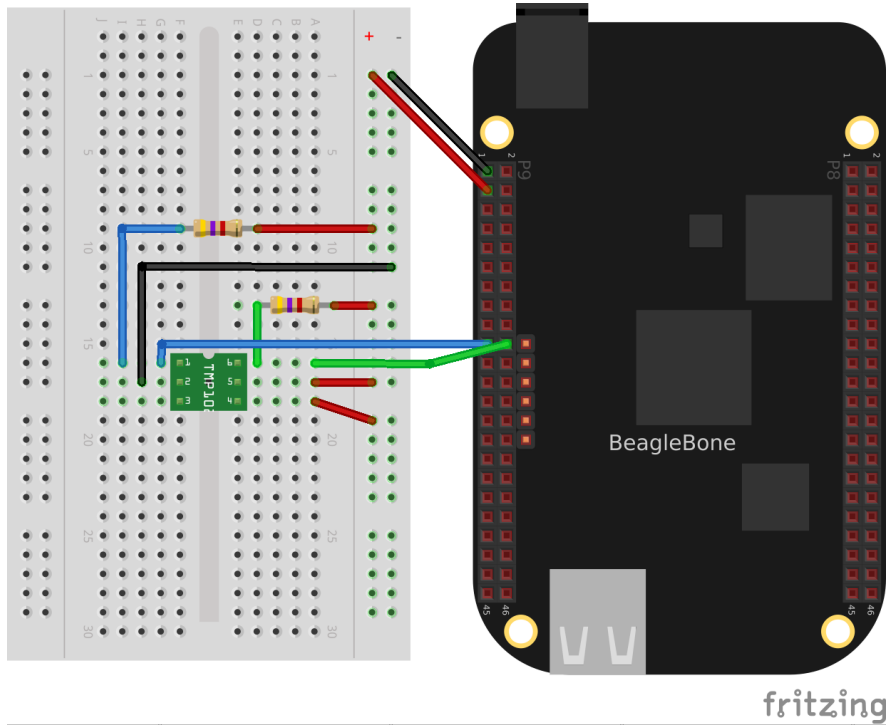


Fig. 2.17: Wiring an I<sup>2</sup>C TMP101 temperature sensor

## 2 I2C ports

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BTN	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
I2C1_SCL	17	18	I2C1_SDA	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	GPIO_22	19	20	GPIO_63
I2C2_SCL	21	22	I2C2_SDA	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	I2C1_SCL	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	I2C1_SDA	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GNDA_ADC	GPIO_9	33	34	GPIO_81
AIN6	35	36	AIN5	GPIO_8	35	36	GPIO_80
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 2.18: Table of I<sup>2</sup>C outputs

I<sup>2</sup>C devices are on the bus. The `-r` flag indicates which bus to use. Our TMP101 is appearing at address `0x49`. You can use the `i2cget` command to read the value. It returns the temperature in hexadecimal and degrees C. In this example, `0x18 = 24{deg}C`, which is `75.2{deg}F`. (Hmmm, the office is a bit warm today.) Try warming up the TMP101 with your finger and running `i2cget` again.

---

**Todo:** fix deg

---

## 2.11 I<sup>2</sup>C tools

One way to see what devices are on a given I<sup>2</sup>C bus is to use `i2cdetect`. Here is bus 2 on the BeagleBone.

```
bone$ i2cdetect -y -r 2
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  49  --  --  --  --  --  --
50:  --  --  --  --  UU  UU  UU  UU  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --

bone$ i2cget -y 2 0x49
0x18
```

Here is bus 1 on the BeagleY-AI.

```
bone$ i2cdetect -y -r 1
 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  49  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --

bone$ i2cget -y 1 0x49
0x18
```

## 2.12 Reading the temperature via the kernel driver

The cleanest way to read the temperature from a TMP101 sensor is to use the kernel driver.

Assuming the TMP101 is on bus 2 (the last digit is the bus number)

---

**Note:** Switch bus 2 to bus 1 if you are using the BeagleY-AI.

---

I<sup>2</sup>C TMP101 via Kernel

```
bone$ cd /sys/class/i2c-adapter/
bone$ ls
i2c-0  i2c-1  i2c-2                                # Three i2c buses (bus 0 is internal)
bone$ cd i2c-2                            # Pick bus 2
```

(continues on next page)

(continued from previous page)

```
bone$ ls -ls
0 --w--w---- 1 root gpio 4096 Jul  1 09:24 delete_device
0 lrwxrwxrwx 1 root gpio  0 Jun 30 16:25 device -> ../../4819c000.i2c
0 drwxrwxr-x 3 root gpio  0 Dec 31  1999 i2c-dev
0 -r--r--r-- 1 root gpio 4096 Dec 31  1999 name
0 --w--w---- 1 root gpio 4096 Jul  1 09:24 new_device
0 lrwxrwxrwx 1 root gpio  0 Jun 30 16:25 of_node -> ../../../../../../../../../../
->/firmware/devicetree/base/ocp/interconnect@48000000/segment@100000/target-
->module@9c000/i2c@0
0 drwxrwxr-x 2 root gpio  0 Dec 31  1999 power
0 lrwxrwxrwx 1 root gpio  0 Jun 30 16:25 subsystem -> ../../../../../../
->../bus/i2c
0 -rw-rw-r-- 1 root gpio 4096 Dec 31  1999 uevent
```

Assuming the TMP101 is at address 0x49

```
bone$ echo tmp101 0x49 > new_device
```

**Note:** If this returns *new\_device: Permission denied*, you will need to run the following first.

```
bone$ sudo chown debian:gpio *
```

This tells the kernel you have a TMP101 sensor at address 0x49. Check the log to be sure.

```
bone$ dmesg -H | tail -3
[ +13.571823] i2c i2c-2: new_device: Instantiated device tmp101 at 0x49
[ +0.043362] lm75 2-0049: supply vs not found, using dummy regulator
[ +0.009976] lm75 2-0049: hwmon0: sensor 'tmp101'
```

Yes, it's there, now see what happened.

```
bone$ ls
2-0049 delete_device device i2c-dev name new_device of_node power ↵
->subsystem uevent
```

Notice a new directory has appeared. It's for i2c bus 2, address 0x49. Look into it.

```
bone$ cd 2-0049/hwmon/hwmon0
bone$ ls -F
device@ name power/ subsystem@ temp1_input temp1_max temp1_max_hyst ↵
->uevent update_interval
bone$ cat temp1_input
24250
```

There is the temperature in milli-degrees C.

Other i2c devices are supported by the kernel. You can try the Linux Kernel Driver Database, <https://cateee.net/lkddb/> to see them.

Once the driver is in place, you can read it via code. `i2cTemp.py` shows how to read the TMP101.

## Python

Listing 2.11: Reading an I<sup>2</sup>C device (`i2cTemp.py`)

```
1 #!/usr/bin/env python
2 # //////////////////////////////////////
3 # //          i2cTemp.py
4 # //          Read a TMP101 sensor on i2c bus 2, address 0x49
```

(continues on next page)

(continued from previous page)

```

5 # //      Wiring:      Attach to i2c as shown in text.
6 # //      Setup:      echo tmp101 0x49 > /sys/class/i2c-adapter/i2c-2/
   ↳new_device
7 # //      See:
8 # //////////////////////////////////////
9 import time
10
11 ms = 1000      # Read time in ms
12 bus = '2'
13 addr = '49'
14 I2CPATH='/sys/class/i2c-adapter/i2c-'+bus+'/'+'+bus+'-00'+addr+'/hwmon/hwmon0';
15
16 f = open(I2CPATH+"/temp1_input", "r")
17
18 while True:
19     f.seek(0)
20     data = f.read()[:-1]      # returns mili-degrees C
21     print("data (C) = " + str(int(data)/1000))
22     time.sleep(ms/1000)

```

i2cTemp.py

## JavaScript

Listing 2.12: Reading an I<sup>2</sup>C device (i2cTemp.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      i2cTemp.js
4  //      Read at TMP101 sensor on i2c bus 2, address 0x49
5  //      Wiring:      Attach to i2c as shown in text.
6  //      Setup:      echo tmp101 0x49 > /sys/class/i2c-adapter/i2c-2/new_
   ↳device
7  //      See:
8  // //////////////////////////////////////
9  const fs = require("fs");
10
11 const ms = 1000;    // Read time in ms
12 const bus = '2';
13 const addr = '49';
14 I2CPATH='/sys/class/i2c-adapter/i2c-'+bus+'/'+'+bus+'-00'+addr+'/hwmon/hwmon0';
15
16 // Read every ms
17 setInterval(readTMP, ms);
18
19 function readTMP() {
20     var data = fs.readFileSync(I2CPATH+"/temp1_input").slice(0, -1);
21     console.log('data (C) = ' + data/1000);
22 }

```

i2cTemp.js

Run the code by using the following command:

```

bone$ ./i2cTemp.js
data (C) = 25.625
data (C) = 27.312
data (C) = 28.187
data (C) = 28.375
^C

```

Notice using the kernel interface gets you more digits of accuracy.

## 2.13 Reading i2c device directly

The TMP102 sensor can be read directly with i2c commands rather than using the kernel driver. First you need to install the i2c module.

```
bone$ sudo apt install python3-smbus
```

Listing 2.13: Reading an I<sup>2</sup>C device (i2cTemp.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      i2ctmp101.py
4  # //      Read at TMP101 sensor on i2c bus 2, address 0x49
5  # //      Wiring:      Attach to i2c as shown in text.
6  # //      Setup:      pip install smbus
7  # //      See:
8  # //////////////////////////////////////
9  import smbus
10 import time
11
12 ms = 1000          # Read time in ms
13 bus = smbus.SMBus(2) # Using i2c bus 2
14 addr = 0x49      # TMP101 is at address 0x49
15
16 while True:
17     data = bus.read_byte_data(addr, 0)
18     print("temp (C) = " + str(data))
19     time.sleep(ms/1000)

```

i2ctmp101.py

This gets only 8 bits for the temperature. See the TMP101 datasheet (<https://www.ti.com/product/TMP101>) for details on how to get up to 12 bits.

## 2.14 Reading Temperature via a Dallas 1-Wire Device

### 2.14.1 Problem

You want to measure a temperature using a Dallas Semiconductor DS18B20 temperature sensor.

### 2.14.2 Solution

---

**Todo:** Update for BeagleY-AI

---

The DS18B20 is an interesting temperature sensor that uses Dallas Semiconductor's 1-wire interface. The data communication requires only one wire! (However, you still need wires from ground and 3.3 V.) You can wire it to any GPIO port.

To make this recipe, you will need:

- Breadboard and jumper wires.
- 4.7 kΩ resistor
- DS18B20 1-wire temperature sensor.

Wire up as shown in [Wiring a Dallas 1-Wire temperature sensor](#).

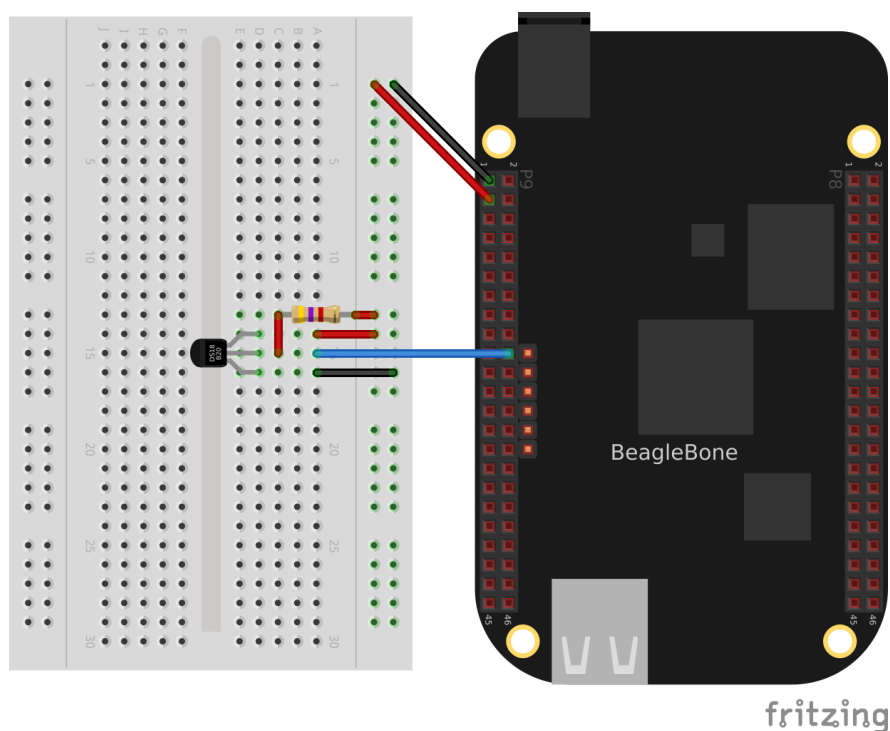


Fig. 2.19: Wiring a Dallas 1-Wire temperature sensor

Edit the file `/boot/uEnt.txt`. Go to about line 19 and edit as shown:

```
17 ###
18 ###Additional custom capes
19 uboot_overlay_addr4=BB-W1-P9.12-00A0.dtbo
20 #uboot_overlay_addr5=<file5>.dtbo
```

Be sure to remove the `#` at the beginning of the line.

Reboot the bone:

```
bone$ reboot
```

Now run the following command to discover the serial number on your device:

```
bone$ ls /sys/bus/w1/devices/
28-00000114ef1b 28-00000128197d w1_bus_master1
```

I have two devices wired in parallel on the same P9\_12 input. This shows the serial numbers for all the devices.

Finally, add the code below in to a file named `w1.py`, edit the path assigned to `w1` so that the path points to your device, and then run it.

## Python

Listing 2.14: Reading a temperature with a DS18B20 (`w1.py`)

```
1 #!/usr/bin/env python
2 # //////////////////////////////////////
3 # //          w1.js
4 # //          Read a Dallas 1-wire device on P9_12
5 # //          Wiring:          Attach gnd and 3.3V and data to P9_12
```

(continues on next page)

(continued from previous page)

```

6 # //      Setup:      Edit /boot/uEnv.txt to include:
7 # //      uboot_overlay_addr4=BB-W1-P9.12-00A0.dtbo
8 # //      See:
9 # //////////////////////////////////////
10 import time
11
12 ms = 500    # Read time in ms
13 # Do ls /sys/bus/w1/devices and find the address of your device
14 addr = '28-00000d459c2c' # Must be changed for your device.
15 W1PATH = '/sys/bus/w1/devices/' + addr
16
17 f = open(W1PATH+'/temperature')
18
19 while True:
20     f.seek(0)
21     data = f.read()[:-1]
22     print("temp (C) = " + str(int(data)/1000))
23     time.sleep(ms/1000)

```

w1.py

## JavaScript

Listing 2.15: Reading a temperature with a DS18B20 (w1.js)

```

1 #!/usr/bin/env node
2 //////////////////////////////////////
3 //      w1.js
4 //      Read a Dallas 1-wire device on P9_12
5 //      Wiring:      Attach gnd and 3.3V and data to P9_12
6 //      Setup:      Edit /boot/uEnv.txt to include:
7 //      uboot_overlay_addr4=BB-W1-P9.12-00A0.dtbo
8 //      See:
9 // //////////////////////////////////////
10 const fs = require("fs");
11
12 const ms = 500    // Read time in ms
13 // Do ls /sys/bus/w1/devices and find the address of your device
14 const addr = '28-00000d459c2c'; // Must be changed for your device.
15 const W1PATH = '/sys/bus/w1/devices/' + addr;
16
17 // Read every ms
18 setInterval(readW1, ms);
19
20 function readW1() {
21     var data = fs.readFileSync(W1PATH+'/temperature').slice(0, -1);
22     console.log('temp (C) = ' + data/1000);
23 }

```

w1.js

```

bone$ ./w1.js
temp (C) = 28.625
temp (C) = 29.625
temp (C) = 30.5
temp (C) = 31.0
^C

```

Each temperature sensor has a unique serial number, so you can have several all sharing the same data line.



## 2.15 Playing and Recording Audio

---

**Todo:** Remove?

---

### 2.15.1 Problem

BeagleBone doesn't have audio built in, but you want to play and record files.

### 2.15.2 Solution

One approach is to buy an audio cape, but another, possibly cheaper approach is to buy a USB audio adapter, such as the one shown in [A USB audio dongle](#).



Fig. 2.20: A USB audio dongle

Drivers for the [Advanced Linux Sound Architecture \(ALSA\)](#) may already be installed on the Bone. If not, run the following:

```
bone$ sudo apt install alsa-utils
```

You can list the recording and playing devices on your Bone by using `aplay` and `arecord`, as shown in [Listing the ALSA audio output and input devices on the Bone](#). BeagleBone Black has audio-out on the HDMI interface. It's listed as `card 0` in [Listing the ALSA audio output and input devices on the Bone](#). `card 1` is my USB audio adapter's audio out.

## 2.16 Listing the ALSA audio output and input devices on the Bone

```
bone$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: Black [TI BeagleBone Black], device 0: HDMI nxp-hdmi-hifi-0 []
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0

bone$ arecord -l
**** List of CAPTURE Hardware Devices ****
card 1: Device [C-Media USB Audio Device], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

In the *aplay* output shown in [Listing the ALSA audio output and input devices on the Bone](#), you can see the USB adapter's audio out. By default, the Bone will send audio to the HDMI. You can change that default by creating a file in your home directory called `~/.asoundrc` and adding the code in [Change the default audio out by putting this in ~/.asoundrc \(audio.asoundrc\)](#) to it.

Listing 2.16: Change the default audio out by putting this in `~/.asoundrc` (audio.asoundrc)

```
1 pcm.!default {
2   type plug
3   slave {
4     pcm "hw:1,0"
5   }
6 }
7 ctl.!default {
8   type hw
9   card 1
10 }
```

audio.asoundrc

You can easily play .wav files with *aplay*:

```
bone$ aplay test.wav
```

You can play other files in other formats by installing *mplayer*:

```
bone$ sudo apt update
bone$ sudo apt install mplayer
bone$ mplayer test.mp3
```

### 2.16.1 Discussion

Adding the simple USB audio adapter opens up a world of audio I/O on the Bone.



## Chapter 3

# Displays and Other Outputs

In this chapter, you will learn how to control physical hardware via BeagleBone Black's general-purpose input/output (GPIO) pins. The Bone has 65 GPIO pins that are brought out on two 46-pin headers, called *P8* and *P9*, as shown in [The P8 and P9 GPIO headers](#).

---

**Note:** All the examples in the book assume you have cloned the Cookbook repository on [git.beagleboard.org](http://git.beagleboard.org). Go here [Cloning the Cookbook Repository](#) for instructions.

---

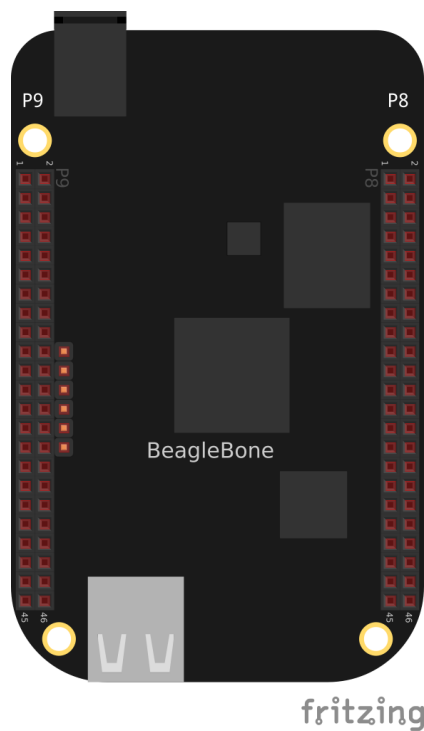


Fig. 3.1: The P8 and P9 GPIO headers

The purpose of this chapter is to give simple examples that show how to use various methods of output. Most solutions require a breadboard and some jumper wires.

All these examples assume that you know how to edit a file ([Editing Code Using Visual Studio Code](#)) and run it, either within Visual Studio Code (VSC) integrated development environment (IDE) or from the command line ([Getting to the Command Shell via SSH](#)).

## 3.1 Toggling an Onboard LED

### 3.1.1 Problem

You want to know how to flash the four LEDs that are next to the Ethernet port on the Bone.

### 3.1.2 Solution

Locate the four onboard LEDs shown in [The four USER LEDs](#). They are labeled *USR0* through *USR3*, but we'll refer to them as the *USER LEDs*.

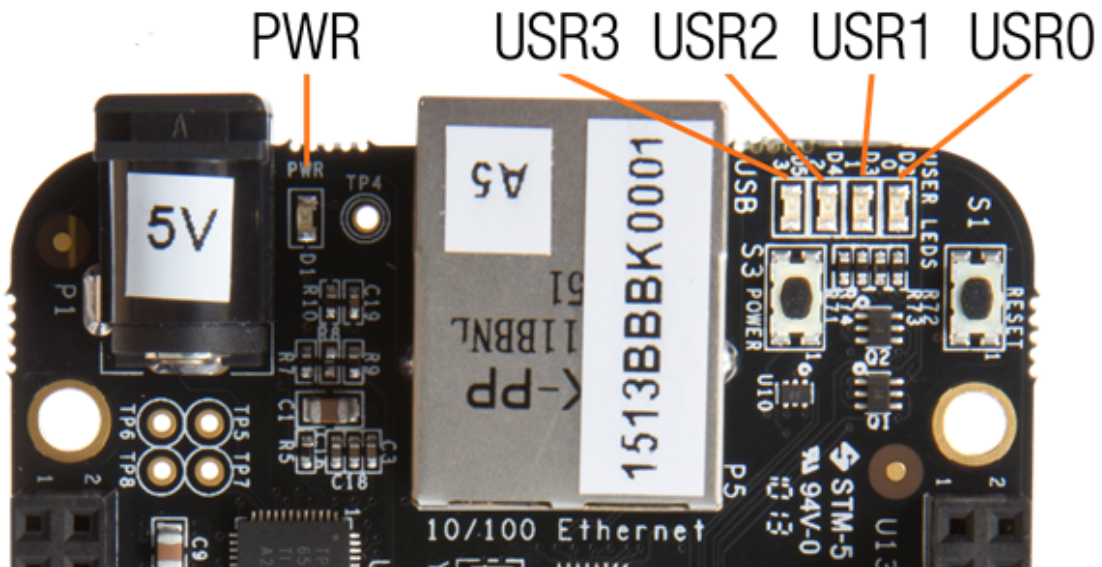


Fig. 3.2: The four *USER LEDs*

Place the code shown in [Using an internal LED \(\*internLED.py\*\)](#) in a file called `internLED.py`. You can do this using VSC to edit files (as shown in [Editing Code Using Visual Studio Code](#)) or with a more traditional editor (as shown in [Editing a Text File from the GNU/Linux Command Shell](#)).

### Python

Listing 3.1: Using an internal LED (`internLED.py`)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  #     internLED.py
4  #     Blinks A USER LED.
5  #     Wiring:
6  #     Setup:
7  #     See:
8  # //////////////////////////////////////
9  import gpiod
10 import time
11
12 LED_CHIP = 'gpiochip1'
13 LED_LINE_OFFSET = [21] # USR0 run: gpioinfo | grep -i -e chip -e usr
14
15 chip = gpiod.Chip(LED_CHIP)

```

(continues on next page)

(continued from previous page)

```

16 lines = chip.get_lines(LED_LINE_OFFSET)
17 lines.request(consumer='internLED.py', type=gpio.LINE_REQ_DIR_OUT)
18
19
20 state = 0      # Start with LED off
21 while True:
22     lines.set_values([state])
23     state = ~state      # Toggle the state
24     time.sleep(0.25)

```

internLED.py

**C**

Listing 3.2: Using an internal LED (internLED.c)

```

1 // // //////////////////////////////////////
2 // #      internLED.c
3 // #      Blinks A USR LED.
4 // #      Wiring:
5 // #      Setup:
6 // #      See:
7 // // //////////////////////////////////////
8 #include <gpio.h>
9 #include <stdio.h>
10 #include <unistd.h>
11
12 #define      CONSUMER      "internLED.c"
13
14 int main(int argc, char **argv)
15 {
16     int chipnumber = 1;
17     unsigned int line_num = 21;      // usr0 LED, run: gpioinfo | grep -
→i -e chip -e usr
18     unsigned int val;
19     struct gpiochip *chip;
20     struct gpiochip_line *line;
21     int i, ret;
22
23     chip = gpiochip_open_by_number(chipnumber);
24     line = gpiochip_get_line(chip, line_num);
25     ret = gpiochip_line_request_output(line, CONSUMER, 0);
26
27     /* Blink */
28     val = 0;
29     while(1) {
30         ret = gpiochip_line_set_value(line, val);
31         // printf("Output %u on line #%u\n", val, line_num);
32         usleep(100000);      // Number of microseconds to
→sleep
33         val = !val;
34     }
35 }

```

internLED.c

In the *bash* command window, enter the following commands:

```

bone$ cd ~/beaglebone-cookbook-code/03displays
bone$ ./internLED.py

```

The *USER0* LED should now be flashing.

## 3.2 Toggling an External LED

### 3.2.1 Problem

You want to connect your own external LED to the Bone.

### 3.2.2 Solution

Connect an LED to one of the GPIO pins using a series resistor to limit the current. To make this recipe, you will need:

- Breadboard and jumper wires.
- 220  $\Omega$  to 470  $\Omega$  resistor.
- LED

**Warning:** The value of the current limiting resistor depends on the LED you are using. The Bone can drive only 4 to 6 mA, so you might need a larger resistor to keep from pulling too much current. A 330  $\Omega$  or 470  $\Omega$  resistor might be better.

*Diagram for using an external LED* shows how you can wire the LED to pin 14 of the *P9* header (*P9\_14*). Every circuit in this book (*Wiring a Breadboard*) assumes you have already wired the rightmost bus to ground (*P9\_1*) and the next bus to the left to the 3.3 V (*P9\_3*) pins on the header. Be sure to get the polarity right on the LED. The *\_short\_* lead always goes to ground.

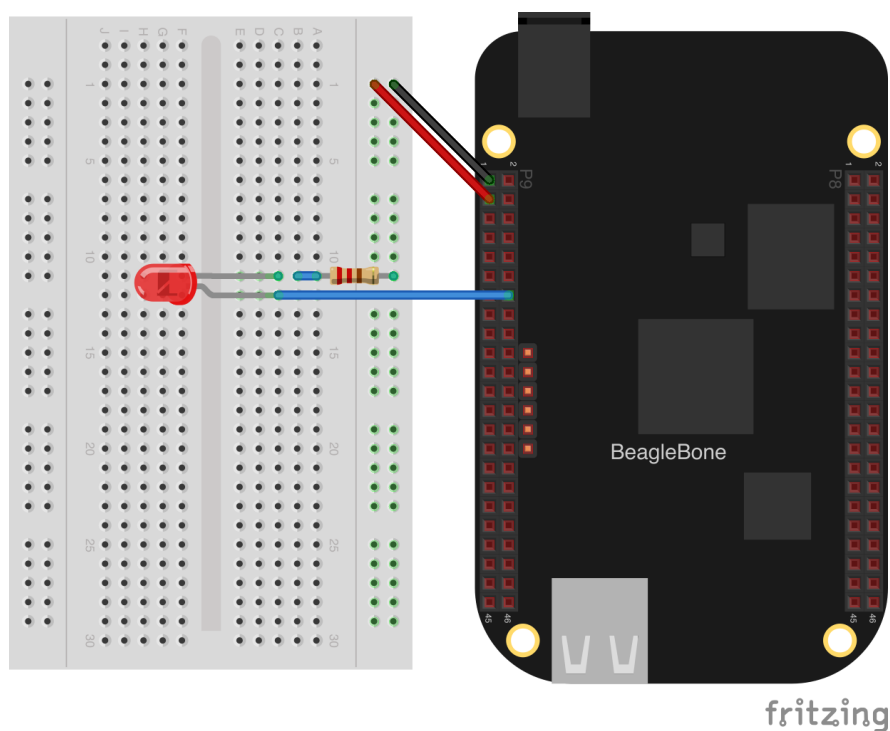


Fig. 3.3: Diagram for using an external LED

After you've wired it, start VSC (see [Editing Code Using Visual Studio Code](#)) and find the code shown in [Code for using an external LED \(externLED.py\)](#). Notice that it looks very similar to the *internLED* code, in fact it only differs in the line number (18 instead of 21). The built-in LEDs use the same GPIO interface as the GPIO pins.

## Python

Listing 3.3: Code for using an external LED (externLED.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  #     externLED.py
4  #     Blinks an external LED wired to P9_14.
5  #     Wiring: P9_14 connects to the plus lead of an LED. The negative_
   ↳lead of the
6  #           LED goes to a 220 Ohm resistor. The other lead of the_
   ↳resistor goes
7  #           to ground
8  #     Setup:
9  #     See:
10 # //////////////////////////////////////
11 import gpiod
12 import time
13
14 LED_CHIP = 'gpiochip1'
15 LED_LINE_OFFSET = [18] # P9_14 run: gpioinfo | grep -i -e chip -e P9_14
16
17 chip = gpiod.Chip(LED_CHIP)
18
19 lines = chip.get_lines(LED_LINE_OFFSET)
20 lines.request(consumer='internLED.py', type=gpiod.LINE_REQ_DIR_OUT)
21
22 state = 0 # Start with LED off
23 while True:
24     lines.set_values([state])
25     state = ~state # Toggle the state
26     time.sleep(0.25)

```

externLED.py

## C

Listing 3.4: Code for using an external LED (externLED.c)

```

1  // // //////////////////////////////////////
2  // #     externLED.c
3  // Blinks an external LED wired to P9_14.
4  // Wiring: P9_14 connects to the plus lead of an LED. The negative lead of_
   ↳the
5  //           LED goes to a 220 Ohm resistor. The other lead of the_
   ↳resistor goes
6  //           to ground
7  //     Setup:
8  //     See:
9  // //////////////////////////////////////
10 #include <gpiod.h>
11 #include <stdio.h>
12 #include <unistd.h>
13
14 #define     CONSUMER     "internLED.c"

```

(continues on next page)



```
15
16 int main(int argc, char **argv)
17 {
18     int chipnumber = 1;
19     unsigned int line_num = 18;           // P9_14, run: gpioinfo | grep -i -
→e chip -e P9_14
20     unsigned int val;
21     struct gpiochip *chip;
22     struct gpiochip_line *line;
23     int i, ret;
24
25     chip = gpiochip_open_by_number(chipnumber);
26     line = gpiochip_get_line(chip, line_num);
27     ret = gpiochip_line_request_output(line, CONSUMER, 0);
28
29     /* Blink */
30     val = 0;
31     while(1) {
32         ret = gpiochip_line_set_value(line, val);
33         // printf("Output %u on line #%u\n", val, line_num);
34         usleep(100000);           // Number of microseconds to
→sleep
35         val = !val;
36     }
37 }
```

externLED.c

Save your file and run the code as before ([Toggling an Onboard LED](#)).

## 3.3 Toggling a High-Voltage External Device

### 3.3.1 Problem

You want to control a device that runs at 120 V.

### 3.3.2 Solution

Working with 120 V can be tricky –even dangerous– if you aren’t careful. Here’s a safe way to do it.

To make this recipe, you will need:

- PowerSwitch Tail II

[Diagram for wiring PowerSwitch Tail II](#) shows how you can wire the PowerSwitch Tail II to pin P9\_14.

After you’ve wired it, because this uses the same output pin as [Toggling an External LED](#), you can run the same code ([Code for using an external LED \(externLED.py\)](#)).

## 3.4 Fading an External LED

### 3.4.1 Problem

You want to change the brightness of an LED from the Bone.

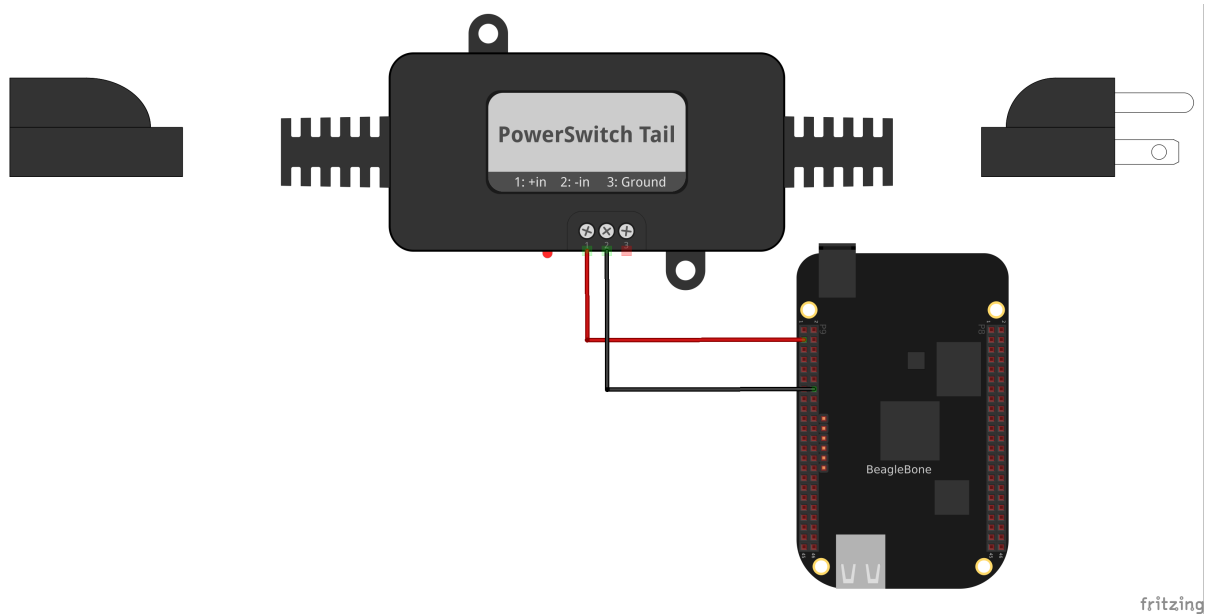


Fig. 3.4: Diagram for wiring PowerSwitch Tail II

### 3.4.2 Solution

Use the Bone's pulse width modulation (PWM) hardware to fade an LED. We'll use the same circuit as before ([Diagram for using an external LED](#)). Find the code in [Code for using an external LED \(fadeLED.py\)](#) Next configure the pins. We are using P9\_14 so run:

```
bone$ config-pin P9_14 pwm
```

Then run it as before.

### Python

Listing 3.5: Code for using an external LED (fadeLED.py)

```

1 #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      fadeLED.py
4  # //      Blinks the P9_14 pin
5  # //      Wiring:
6  # //      Setup:  config-pin P9_14 pwm
7  # //      See:
8  # //////////////////////////////////////
9  import time
10 ms = 20; # Fade time in ms
11
12 pwmPeriod = 1000000 # Period in ns
13 pwm      = '1' # pwm to use
14 channel = 'a' # channel to use
15 PWMPATH='/dev/bone/pwm/'+pwm+'/' + channel
16 step = 0.02 # Step size
17 min = 0.02 # dimmest value
18 max = 1 # brightest value
19 brightness = min # Current brightness
20
21 f = open(PWMPATH+'/period', 'w')

```

(continues on next page)

(continued from previous page)

```

22 f.write(str(pwmPeriod))
23 f.close()
24
25 f = open(PWMPATH+'/enable', 'w')
26 f.write('1')
27 f.close()
28
29 f = open(PWMPATH+'/duty_cycle', 'w')
30 while True:
31     f.seek(0)
32     f.write(str(round(pwmPeriod*brightness)))
33     brightness += step
34     if(brightness >= max or brightness <= min):
35         step = -1 * step
36     time.sleep(ms/1000)
37
38 # | Pin   | pwm | channel
39 # | P9_31 | 0   | a
40 # | P9_29 | 0   | b
41 # | P9_14 | 1   | a
42 # | P9_16 | 1   | b
43 # | P8_19 | 2   | a
44 # | P8_13 | 2   | b

```

fadeLED.py

## JavaScript

Listing 3.6: Code for using an external LED (fadeLED.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////////////////
3  //      fadeLED.js
4  //      Blinks the P9_14 pin
5  //      Wiring:
6  //      Setup:  config-pin P9_14 pwm
7  //      See:
8  //////////////////////////////////////////////////
9  const fs = require("fs");
10 const ms = '20';    // Fade time in ms
11
12 const pwmPeriod = '1000000';    // Period in ns
13 const pwm      = '1';    // pwm to use
14 const channel = 'a';    // channel to use
15 const PWMPATH = '/dev/bone/pwm/'+pwm+'/' + channel;
16 var step = 0.02;    // Step size
17 const min = 0.02,    // dimmest value
18       max = 1;    // brightest value
19 var brightness = min;    // Current brightness;
20
21
22 // Set the period in ns
23 fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
24 fs.writeFileSync(PWMPATH+'/duty_cycle', pwmPeriod/2);
25 fs.writeFileSync(PWMPATH+'/enable', '1');
26
27 setInterval(fade, ms);    // Step every ms
28
29 function fade() {
30     fs.writeFileSync(PWMPATH+'/duty_cycle',

```

(continues on next page)

(continued from previous page)

```

31     parseInt (pwmPeriod*brightness));
32     brightness += step;
33     if(brightness >= max || brightness <= min) {
34         step = -1 * step;
35     }
36 }
37
38 // | Pin   | pwm | channel
39 // | P9_31 | 0   | a
40 // | P9_29 | 0   | b
41 // | P9_14 | 1   | a
42 // | P9_16 | 1   | b
43 // | P8_19 | 2   | a
44 // | P8_13 | 2   | b

```

fadeLED.js

The Bone has several outputs that can be use as pwm's as shown in [Table of PWM outputs](#). There are three *EHRPWM*'s which each has a pair of pwm channels. Each pair must have the same period.

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	TIMER4	7	8	TIMER7
PWR_BUT	9	10	SYS_RESETN	TIMER5	9	10	TIMER6
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	EHRPWM1A	EHRPWM2B	13	14	GPIO_26
GPIO_48	15	16	EHRPWM1B	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	EHRPWM2A	19	20	GPIO_63
EHRPWMOB	21	22	EHRPWMOA	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	GPIO_15	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_14	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	ECAPPWM2	GPIO_86	27	28	GPIO_88
EHRPWMOB	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
EHRPWMOA	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GNDA_ADC	GPIO_9	33	34	EHRPWM1B
AIN6	35	36	AIN5	GPIO_8	35	36	EHRPWM1A
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	ECAPPWMO	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	EHRPWM2A	45	46	EHRPWM2B

Fig. 3.5: Table of PWM outputs

The pwm's are accessed through `/dev/bone/pwm`

```

bone$ cd /dev/bone/pwm
bone$ ls
0 1 2

```

Here we see three pwmchips that can be used, each has two channels. Explore one.

```

bone$ cd 1
bone$ ls
a b

```

(continues on next page)

(continued from previous page)

```
bone$ cd a
bone$ ls
capture  duty_cycle  enable  period  polarity  power  uevent
```

Here is where you can set the period and duty\_cycle (in ns) and enable the pwm. Attach in LED to P9\_14 and if you set the period long enough you can see the LED flash.

```
bone$ echo 1000000000 > period
bone$ echo 500000000 > duty_cycle
bone$ echo 1 > enable
```

Your LED should now be flashing.

[Headers to pwm channel mapping](#) are the mapping I've figured out so far. I don't know how to get to the timers.

Table 3.1: Headers to pwm channel mapping

Pin	pwm	channel
P9_31	0	a
P9_29	0	b
P9_14	1	a
P9_16	1	b
P8_19	2	a
P8_13	2	b

## 3.5 Writing to an LED Matrix

### 3.5.1 Problem

You have an I<sup>2</sup>C-based LED matrix to interface.

### 3.5.2 Solution

There are a number of nice LED matrices that allow you to control several LEDs via one interface. This solution uses an [Adafruit Bicolor 8x8 LED Square Pixel Matrix w/I<sup>2</sup>C Backpack](#).

To make this recipe, you will need:

- Breadboard and jumper wires
- Two 4.7 kΩ resistors.
- I<sup>2</sup>C LED matrix

The LED matrix is a 5 V device, but you can drive it from 3.3 V. Wire, as shown in [Wiring an I<sup>2</sup>C LED matrix](#).

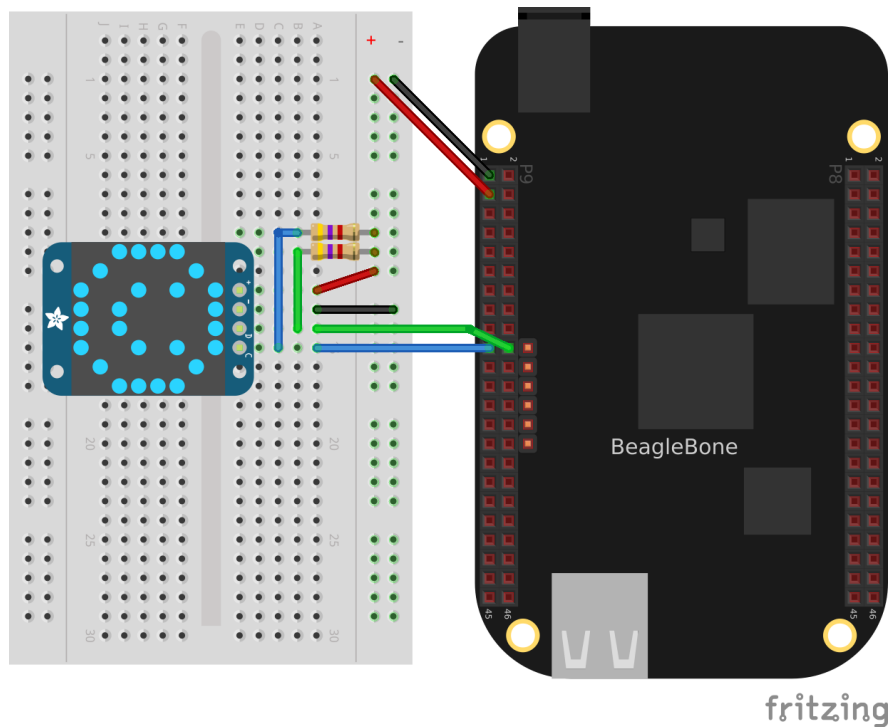
[Measuring a Temperature](#) shows how to use `i2cdetect` to discover the address of an I<sup>2</sup>C device.

Run the `i2cdetect -y -r 2` command to discover the address of the display on I<sup>2</sup>C bus 2, as shown in [Using I<sup>2</sup>C command-line tools to discover the address of the display](#).

## 3.6 Using I<sup>2</sup>C command-line tools to discover the address of the display

```
bone$ i2cdetect -y -r 2
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

(continues on next page)

Fig. 3.6: Wiring an I<sup>2</sup>C LED matrix

(continued from previous page)

```

10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 49 -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

```

Here, you can see a device at `0x49` and `0x70`. I know I have a temperature sensor at `0x49`, so the LED matrix must be at `0x70`.

Find the code in [LED matrix display \(matrixLEDi2c.py\)](#) and run it by using the following command:

```

bone$ pip install smbus # (Do this only once.)
bone$ ./matrixLEDi2c.py

```

### 3.7 LED matrix display (matrixLEDi2c.py)

Listing 3.7: LED matrix display (matrixLEDi2c.py)

```

1 #!/usr/bin/env python
2 # //////////////////////////////////////
3 # //      i2cTemp.py
4 # //      Write an 8x8 Red/Green LED matrix.
5 # //      Wiring:      Attach to i2c as shown in text.
6 # //      Setup:      echo tmp101 0x49 > /sys/class/i2c-adapter/i2c-2/
   ↳new_device
7 # //      See:      https://www.adafruit.com/product/902
8 # //////////////////////////////////////
9 import smbus

```

(continues on next page)

```

10 import time
11
12 bus = smbus.SMBus(2) # Use i2c bus 2
13 matrix = 0x70 # Use address 0x70
14 ms = 1; # Delay between images in ms
15
16 # The first byte is GREEN, the second is RED.
17 smile = [0x00, 0x3c, 0x00, 0x42, 0x28, 0x89, 0x04, 0x85,
18          0x04, 0x85, 0x28, 0x89, 0x00, 0x42, 0x00, 0x3c
19 ]
20 frown = [0x3c, 0x00, 0x42, 0x00, 0x85, 0x20, 0x89, 0x00,
21          0x89, 0x00, 0x85, 0x20, 0x42, 0x00, 0x3c, 0x00
22 ]
23 neutral = [0x3c, 0x3c, 0x42, 0x42, 0xa9, 0xa9, 0x89, 0x89,
24            0x89, 0x89, 0xa9, 0xa9, 0x42, 0x42, 0x3c, 0x3c
25 ]
26
27 bus.write_byte_data(matrix, 0x21, 0) # Start oscillator (p10)
28 bus.write_byte_data(matrix, 0x81, 0) # Disp on, blink off (p11)
29 bus.write_byte_data(matrix, 0xe7, 0) # Full brightness (page 15)
30
31 bus.write_i2c_block_data(matrix, 0, frown) #
32 for fade in range(0xef, 0xe0, -1): #
33     bus.write_byte_data(matrix, fade, 0)
34     time.sleep(ms/10)
35
36 bus.write_i2c_block_data(matrix, 0, neutral)
37 for fade in range(0xe0, 0xef, 1):
38     bus.write_byte_data(matrix, fade, 0)
39     time.sleep(ms/10)
40
41 bus.write_i2c_block_data(matrix, 0, smile)

```

matrixLEDi2c.py

- ① This line states which bus to use. The last digit gives the I<sup>2</sup>C bus number.
- ② This specifies the address of the LED matrix, *0x70* in our case.
- ③ This indicates which LEDs to turn on. The first byte is for the first column of *green* LEDs. In this case, all are turned off. The next byte is for the first column of *red* LEDs. The hex *0x3c* number is *0b00111100* in binary. This means the first two red LEDs are off, the next four are on, and the last two are off. The next byte (*0x00*) says the second column of *green* LEDs are all off, the fourth byte (*0x42 = 0b01000010*) says just two *red* LEDs are on, and so on. Declarations define four different patterns to display on the LED matrix, the last being all turned off.
- ④ Send three commands to the matrix to get it ready to display.
- ⑤ Now, we are ready to display the various patterns. After each pattern is displayed, we sleep a certain amount of time so that the pattern can be seen.
- ⑥ Finally, send commands to the LED matrix to set the brightness. This makes the display fade out and back in again.

## 3.8 Driving a 5 V Device

### 3.8.1 Problem

You have a 5 V device to drive, and the Bone has 3.3 V outputs.

### 3.8.2 Solution

If you are lucky, you might be able to drive a 5 V device from the Bone's 3.3 V output. Try it and see if it works. If not, you need a level translator.

What you will need for this recipe:

- A PCA9306 level translator
- A 5 V power supply (if the Bone's 5 V power supply isn't enough)

The PCA9306 translates signals at 3.3 V to 5 V in both directions. It's meant to work with I<sup>2</sup>C devices that have a pull-up resistor, but it can work with anything needing translation.

[Wiring a PCA9306 level translator to an LED matrix](#) shows how to wire a PCA9306 to an LED matrix. The left is the 3.3 V side and the right is the 5 V side. Notice that we are using the Bone's built-in 5 V power supply.

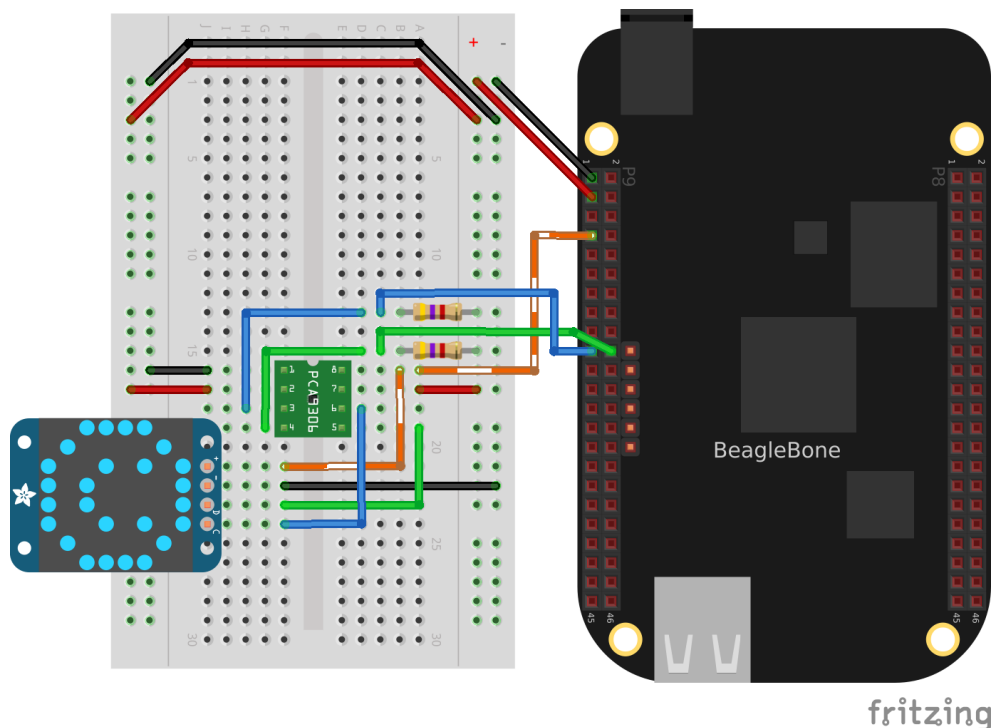


Fig. 3.7: Wiring a PCA9306 level translator to an LED matrix

---

**Note:** If your device needs more current than the Bone's 5 V power supply provides, you can wire in an external power supply.

---

## 3.9 Writing to a NeoPixel LED String Using the PRUs

### 3.9.1 Problem

You have an Adafruit NeoPixel LED string or Adafruit NeoPixel LED matrix and want to light it up.

### 3.9.2 Solution

The PRU Cookbook has a nice discussion ([WS2812 \(NeoPixel\) driver](#)) on driving NeoPixels.



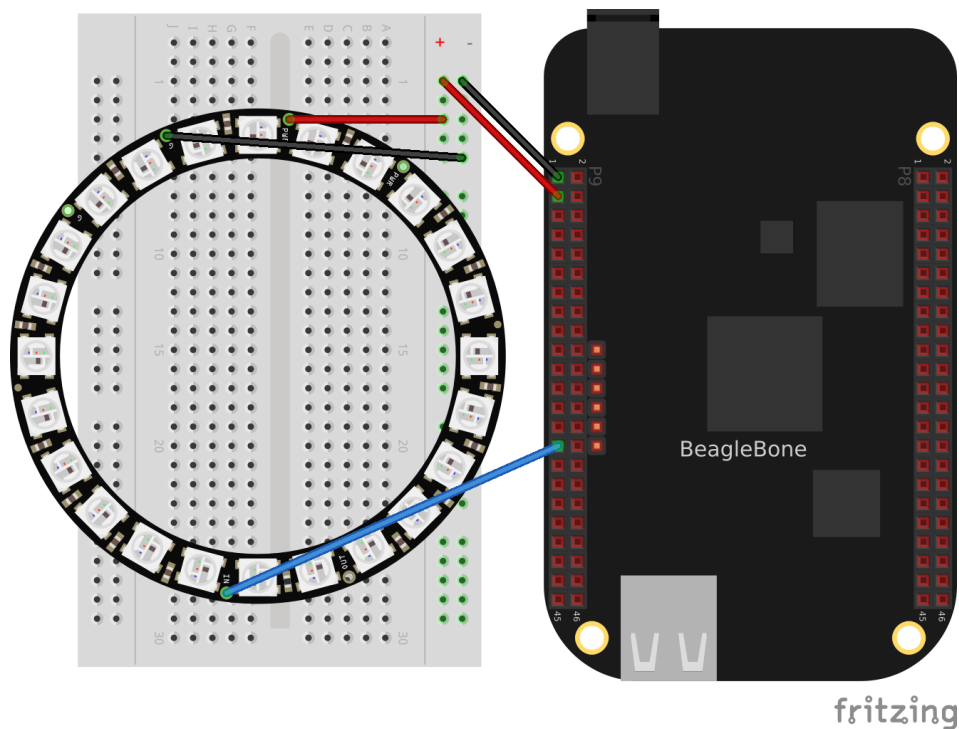


Fig. 3.8: Wiring an Adafruit NeoPixel LED matrix to P9\_29

## 3.10 Writing to a NeoPixel LED String Using LEDscape

## 3.11 Making Your Bone Speak

### 3.11.1 Problem

Your Bone wants to talk.

### 3.11.2 Solution

Just install the `flite` text-to-speech program:

```
bone$ sudo apt install flite
```

Then add the code from [A program that talks \(`speak.js`\)](#) in a file called `speak.js` and run.

Listing 3.8: A program that talks (`speak.js`)

```

1  #!/usr/bin/env node
2
3  var exec = require('child_process').exec;
4
5  function speakForSelf(phrase) {
6  {
7      exec('flite -t "' + phrase + "'", function (error, stdout, stderr) {
8      console.log(stdout);
9      if(error) {
10     console.log('error: ' + error);
11     }
12     if(stderr) {
13     console.log('stderr: ' + stderr);

```

(continues on next page)

(continued from previous page)

```
14     }
15     });
16 }
17
18 speakForSelf("Hello, My name is Borris. " +
19     "I am a BeagleBone Black, " +
20     "a true open hardware, " +
21     "community-supported embedded computer for developers and hobbyists. " +
22     "I am powered by a 1 Giga Hertz Sitara™ ARM® Cortex-A8 processor. " +
23     "I boot Linux in under 10 seconds. " +
24     "You can get started on development in " +
25     "less than 5 minutes with just a single USB cable." +
26     "Bark, bark!"
27 );
```

speak.js

See [Playing and Recording Audio](#) to see how to use a USB audio dongle and set your default audio out.



## Chapter 4

# Motors

One of the many fun things about embedded computers is that you can move physical things with motors. But there are so many different kinds of motors (*servo*, *stepper*, DC), so how do you select the right one?

The type of motor you use depends on the type of motion you want:

- **R/C or hobby servo motor**  
Can be quickly positioned at various absolute angles, but some don't spin. In fact, many can turn only about 180{deg}.
- **Stepper motor**  
Spins and can also rotate in precise relative angles, such as turning 45°. Stepper motors come in two types: *bipolar* (which has four wires) and *unipolar* (which has five or six wires).
- **DC motor**  
Spins either clockwise or counter-clockwise and can have the greatest speed of the three. But a DC motor can't easily be made to turn to a given angle.

When you know which type of motor to use, interfacing is easy. This chapter shows how to interface with each of these motors.

---

**Note:** Motors come in many sizes and types. This chapter presents some of the more popular types and shows how they can interface easily to the Bone. If you need to turn on and off a 120 V motor, consider using something like the PowerSwitch presented in [Toggling a High-Voltage External Device](#).

---

---

**Note:** The Bone has built-in 3.3 V and 5 V supplies, which can supply enough current to drive some small motors. Many motors, however, draw enough current that an external power supply is needed. Therefore, an external 5 V power supply is listed as optional in many of the recipes.

---

---

**Note:** All the examples in the book assume you have cloned the Cookbook repository on [git.beagleboard.org](http://git.beagleboard.org). Go here [Cloning the Cookbook Repository](#) for instructions.

---

## 4.1 Controlling a Servo Motor

### 4.1.1 Problem

You want to use BeagleBone to control the absolute position of a servo motor.

### 4.1.2 Solution

We'll use the pulse width modulation (PWM) hardware of the Bone to control a servo motor.

To make the recipe, you will need:

- Servo motor.
- Breadboard and jumper wires.
- 1 kΩ resistor (optional)
- 5 V power supply (optional)

The 1 kΩ resistor isn't required, but it provides some protection to the general-purpose input/output (GPIO) pin in case the servo fails and draws a large current.

Wire up your servo, as shown in [Driving a servo motor with the 3.3 V power supply](#).

**Note:** There is no standard for how servo motor wires are colored. One of my servos is wired like [Driving a servo motor with the 3.3 V power supply](#) red is 3.3 V, black is ground, and yellow is the control line. I have another servo that has red as 3.3 V and ground is brown, with the control line being orange. Generally, though, the 3.3 V is in the middle. Check the datasheet for your servo before wiring.

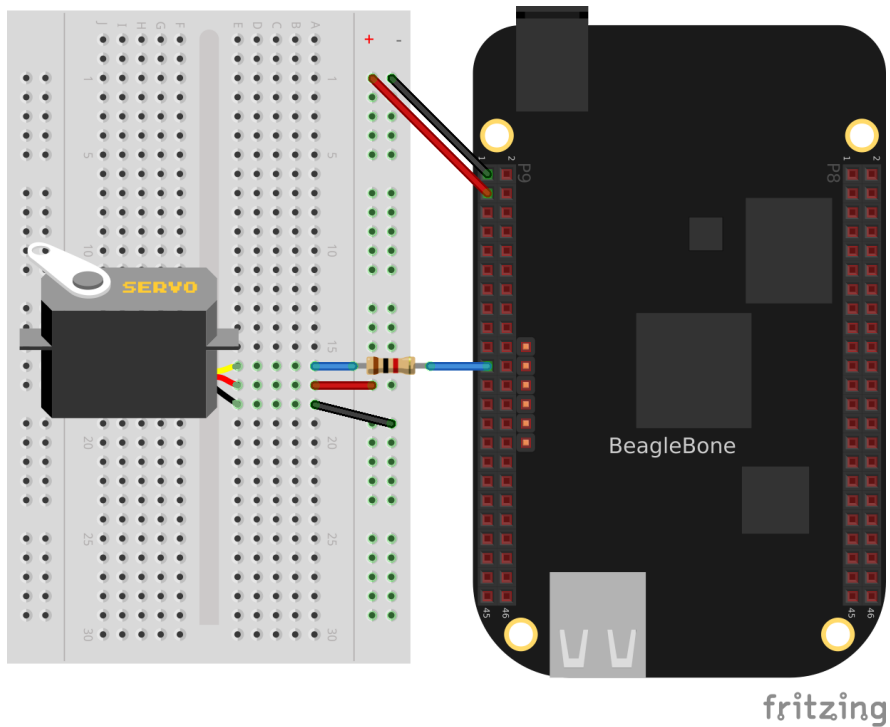


Fig. 4.1: Driving a servo motor with the 3.3 V power supply

The code for controlling the servo motor is in `servoMotor.py`, shown in [Code for driving a servo motor \(servoMotor.py\)](#).

#### Python

Listing 4.1: Code for driving a servo motor (servoMotor.py)

```
1 #!/usr/bin/env python
2 # //////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```

3  # //      servoMotor.py
4  # //      Drive a simple servo motor back and forth on P9_16 pin
5  # //      Wiring:
6  # //      Setup:  config-pin P9_16 pwm
7  # //      See:
8  # //////////////////////////////////////
9  import time
10 import signal
11 import sys
12
13 pwmPeriod = '20000000'    # Period in ns, (20 ms)
14 pwm =      '1' # pwm to use
15 channel = 'b' # channel to use
16 PWMPATH='/dev/bone/pwm/'+pwm+'/' +channel
17 low  = 0.8 # Smallest angle (in ms)
18 hi   = 2.4 # Largest angle (in ms)
19 ms   = 250 # How often to change position, in ms
20 pos  = 1.5 # Current position, about middle ms)
21 step = 0.1 # Step size to next position
22
23 def signal_handler(sig, frame):
24     print('Got SIGINT, turning motor off')
25     f = open(PWMPATH+'/enable', 'w')
26     f.write('0')
27     f.close()
28     sys.exit(0)
29 signal.signal(signal.SIGINT, signal_handler)
30 print('Hit ^C to stop')
31
32 f = open(PWMPATH+'/period', 'w')
33 f.write(pwmPeriod)
34 f.close()
35 f = open(PWMPATH+'/enable', 'w')
36 f.write('1')
37 f.close()
38
39 f = open(PWMPATH+'/duty_cycle', 'w')
40 while True:
41     pos += step    # Take a step
42     if(pos > hi or pos < low):
43         step *= -1
44     duty_cycle = str(round(pos*1000000))    # Convert ms to ns
45     # print('pos = ' + str(pos) + ' duty_cycle = ' + duty_cycle)
46     f.seek(0)
47     f.write(duty_cycle)
48     time.sleep(ms/1000)
49
50 # | Pin   | pwm | channel
51 # | P9_31 | 0   | a
52 # | P9_29 | 0   | b
53 # | P9_14 | 1   | a
54 # | P9_16 | 1   | b
55 # | P8_19 | 2   | a
56 # | P8_13 | 2   | b

```

servoMotor.py

## JavaScript

Listing 4.2: Code for driving a servo motor (servoMotor.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      servoMotor.js
4  //      Drive a simple servo motor back and forth on P9_16 pin
5  //      Wiring:
6  //      Setup:  config-pin P9_16 pwm
7  //      See:
8  //////////////////////////////////////
9  const fs = require("fs");
10
11 const pwmPeriod = '20000000'; // Period in ns, (20 ms)
12 const pwm      = '1'; // pwm to use
13 const channel  = 'b'; // channel to use
14 const PWMPATH = '/dev/bone/pwm/'+pwm+'/' + channel;
15 const low     = 0.8, // Smallest angle (in ms)
16       hi      = 2.4, // Largest angle (in ms)
17       ms      = 250; // How often to change position, in ms
18 var pos      = 1.5, // Current position, about middle ms)
19     step     = 0.1; // Step size to next position
20
21 console.log('Hit ^C to stop');
22 fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
23 fs.writeFileSync(PWMPATH+'/enable', '1');
24
25 var timer = setInterval(sweep, ms);
26
27 // Sweep from low to hi position and back again
28 function sweep() {
29     pos += step; // Take a step
30     if(pos > hi || pos < low) {
31         step *= -1;
32     }
33     var dutyCycle = parseInt(pos*1000000); // Convert ms to ns
34     // console.log('pos = ' + pos + ' duty cycle = ' + dutyCycle);
35     fs.writeFileSync(PWMPATH+'/duty_cycle', dutyCycle);
36 }
37
38 process.on('SIGINT', function() {
39     console.log('Got SIGINT, turning motor off');
40     clearInterval(timer); // Stop the timer
41     fs.writeFileSync(PWMPATH+'/enable', '0');
42 });
43
44 // | Pin   | pwm | channel
45 // | P9_31 | 0   | a
46 // | P9_29 | 0   | b
47 // | P9_14 | 1   | a
48 // | P9_16 | 1   | b
49 // | P8_19 | 2   | a
50 // | P8_13 | 2   | b

```

servoMotor.js

You need to configure the pin for PWM.

### BeagleBone

```
bone$ cd ~/beaglebone-cookbook-code/04motors
bone$ config-pin P9_16 pwm
```

(continues on next page)

(continued from previous page)

```
bone$ ./servoMotor.py
```

### BeagleY-AI

Configuring the PWM on the BeagleY-AI takes a little more effort than on the Bone. First select which PWM you want to use. <https://pinout.beagleboard.io/pinout/pwm> shows you have many to choose from.

3v3 Power	1		2	5v Power
GPIO 2 (I2C1 SDA)	3		4	5v Power
GPIO 3 (I2C1 SCL)	5		6	Ground
GPIO 4	7		8	GPIO 14 (EHRPWM0*)
Ground	9		10	GPIO 15 (EHRPWM0*)
GPIO 17 (ECAP2*)	11		12	GPIO 18 (PWM0)
GPIO 27	13		14	Ground
GPIO 22	15		16	GPIO 23
3v3 Power	17		18	GPIO 24
GPIO 10 (SPI0 MOSI)	19		20	Ground
GPIO 9 (SPI0 MISO)	21		22	GPIO 25
GPIO 11 (SPI0 SCLK)	23		24	GPIO 8 (SPI0 CE0)
Ground	25		26	GPIO 7 (SPI0 CE1)
GPIO 0 (EEPROM SDA)	27		28	GPIO 1 (EEPROM SCL)
GPIO 5 (EHRPWM0*)	29		30	Ground
GPIO 6 (EHRPWM1*)	31		32	GPIO 12 (PWM0)
GPIO 13 (PWM1)	33		34	Ground
GPIO 19 (PWM1)	35		36	GPIO 16 (ECAP1*)
GPIO 26	37		38	GPIO 20 (EHRPWM1*)
Ground	39		40	GPIO 21 (EHRPWM1*)

Fig. 4.2: BeagleY-AI PWMs

Let's use **PWM0** on **GPIO12**. Note this is Hat pin 32 as shown in the figure (**hat-32**). The instructions at [beagle-y-ai-using-pwm](#) give details on how to configure the PWM pin. A shorter version is given here.

To enable any of the PWM Pins, we have to modify the file: `/boot/firmware/extlinux/extlinux.conf`. We can check the available list of Device Tree Overlays using the command:

```
debian@BeagleBone:~$ ls /boot/firmware/overlays/ | grep "beagle-y-ai-pwm"
k3-am67a-beagle-y-ai-pwm-ecap0-gpio12.dtbo
```

(continues on next page)



(continued from previous page)

```
k3-am67a-beagley-ai-pwm-ecap1-gpio16.dtbo
k3-am67a-beagley-ai-pwm-ecap1-gpio21.dtbo
...
```

Add the line shown below to `/boot/firmware/extlinux/extlinux.conf` to load the `gpio12 pwm` device tree overlay:

```
fdtoverlays /overlays/k3-am67a-beagley-ai-pwm-epwm0-ecap0-gpio12.dtbo
```

Your `/boot/firmware/extlinux/extlinux.conf` file should look something like:

```
label microSD (default)
    kernel /Image
    append console=ttyS2,115200n8 root=/dev/mmcblk1p3 ro rootfstype=ext4
    →resume=/dev/mmcblk1p2 rootwait net.ifnames=0 quiet
    fdt_dir /
    fdt /ti/k3-am67a-beagley-ai.dtb
    fdtoverlays /overlays/k3-am67a-beagley-ai-pwm-ecap0-gpio12.dtbo
    initrd /initrd.img
```

Now reboot your BeagleY-AI to load the overlay:

```
beagle$ sudo reboot
```

To configure HAT pin32 (GPIO12) PWM symlink pin using `beagle-pwm-export` execute the command below,

```
beagle$ sudo beagle-pwm-export --pin hat-32
```

We've changed the PWM pin that's being used so we need to modify `servoMotor.py`. Around line 16 you will see:

```
PWMPATH='/dev/bone/pwm/'+pwm+'/' + channel
```

Change it to:

```
PWMPATH='/dev/hat/pwm/GPIO12'
```

Now run your code:

```
beagle$ ./servoMotor.py
```

Running the code causes the motor to move back and forth, progressing to successive positions between the two extremes. You will need to press `^C` (Ctrl-C) to stop the script.

## 4.2 Controlling a Servo with an Rotary Encoder

### 4.2.1 Problem

You have a rotary encoder from from chapter 2 rotary encoder example that you want to use to control a servo motor.

### 4.2.2 Solution

Combine the code from `rotaryEncoder.js` and `servoMotor.js`.

```
bone$ config-pin P9_16 pwm
bone$ config-pin P8_11 eqep
bone$ config-pin P8_12 eqep
bone$ ./servoEncoder.py
```

Listing 4.3: Code for driving a servo motor with a rotary encoder(servoEncoder.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      servoEncoder.py
4  # //      Drive a simple servo motor using rotary encoder via eQEP
5  # //      Wiring: Servo on P9_16, rotary encoder on P8_11 and P8_12
6  # //      Setup:  config-pin P9_16 pwm
7  # //                      config-pin P8_11 eqep
8  # //                      config-pin P8_12 eqep
9  # //      See:
10 # //////////////////////////////////////
11 import time
12 import signal
13 import sys
14
15 # Set up encoder
16 eQEP = '2'
17 COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0'
18 maxCount = '180'
19
20 ms = 100          # Time between samples in ms
21
22 # Set the eEQP maximum count
23 fQEP = open(COUNTERPATH+'/ceiling', 'w')
24 fQEP.write(maxCount)
25 fQEP.close()
26
27 # Enable
28 fQEP = open(COUNTERPATH+'/enable', 'w')
29 fQEP.write('1')
30 fQEP.close()
31
32 fQEP = open(COUNTERPATH+'/count', 'r')
33
34 # Set up servo
35 pwmPeriod = '20000000'      # Period in ns, (20 ms)
36 pwm      = '1'      # pwm to use
37 channel = 'b'      # channel to use
38 PWMPATH='/dev/bone/pwm/'+pwm+'/' + channel
39 low  = 0.6 # Smallest angle (in ms)
40 hi   = 2.5 # Largest angle (in ms)
41 ms   = 250 # How often to change position, in ms
42 pos  = 1.5 # Current position, about middle ms)
43 step = 0.1 # Step size to next position
44
45 def signal_handler(sig, frame):
46     print('Got SIGINT, turning motor off')
47     f = open(PWMPATH+'/enable', 'w')
48     f.write('0')
49     f.close()
50     sys.exit(0)
51 signal.signal(signal.SIGINT, signal_handler)
52
53 f = open(PWMPATH+'/period', 'w')
54 f.write(pwmPeriod)
55 f.close()
56 f = open(PWMPATH+'/duty_cycle', 'w')
57 f.write(str(round(int(pwmPeriod)/2)))
58 f.close()
59 f = open(PWMPATH+'/enable', 'w')

```

(continues on next page)

```

60 f.write('1')
61 f.close()
62
63 print('Hit ^C to stop')
64
65 olddata = -1
66 while True:
67     fQEP.seek(0)
68     data = fQEP.read()[:-1]
69     # Print only if data changes
70     if data != olddata:
71         olddata = data
72         # print("data = " + data)
73         # # map 0-180 to low-hi
74         duty_cycle = -1*int(data)*(hi-low)/180.0 + hi
75         duty_cycle = str(int(duty_cycle*1000000)) # Convert_
        ↳from ms to ns
76         # print('duty_cycle = ' + duty_cycle)
77         f = open(PWMPATH+'/duty_cycle', 'w')
78         f.write(duty_cycle)
79         f.close()
80         time.sleep(ms/1000)
81
82 # Black OR Pocket
83 # eQEP0:      P9.27 and P9.42 OR P1_33 and P2_34
84 # eQEP1:      P9.33 and P9.35
85 # eQEP2:      P8.11 and P8.12 OR P2_24 and P2_33
86
87 # AI
88 # eQEP1:      P8.33 and P8.35
89 # eQEP2:      P8.11 and P8.12 or P9.19 and P9.41
90 # eQEP3:      P8.24 and P8.25 or P9.27 and P9.42
91
92 # | Pin    | pwm | channel
93 # | P9_31 | 0    | a
94 # | P9_29 | 0    | b
95 # | P9_14 | 1    | a
96 # | P9_16 | 1    | b
97 # | P8_19 | 2    | a
98 # | P8_13 | 2    | b

```

servoEncoder.py

## 4.3 Controlling the Speed of a DC Motor

### 4.3.1 Problem

You have a DC motor (or a solenoid) and want a simple way to control its speed, but not the direction.

### 4.3.2 Solution

It would be nice if you could just wire the DC motor to BeagleBone Black and have it work, but it won't. Most motors require more current than the GPIO ports on the Bone can supply. Our solution is to use a transistor to control the current to the bone.

Here we configure the encoder to returns value between 0 and 180 inclusive. This value is then mapped to a value between *min* (0.6 ms) and *max* (2.5 ms). This number is converted from milliseconds and nanoseconds (time 1000000) and sent to the servo motor via the pwm.

Here's what you will need:

- 3 V to 5 V DC motor
- Breadboard and jumper wires.
- 1 k $\Omega$  resistor.
- Transistor 2N3904.
- Diode 1N4001.
- Power supply for the motor (optional)

If you are using a larger motor (more current), you will need to use a larger transistor.

Wire your breadboard as shown in [Wiring a DC motor to spin one direction](#).

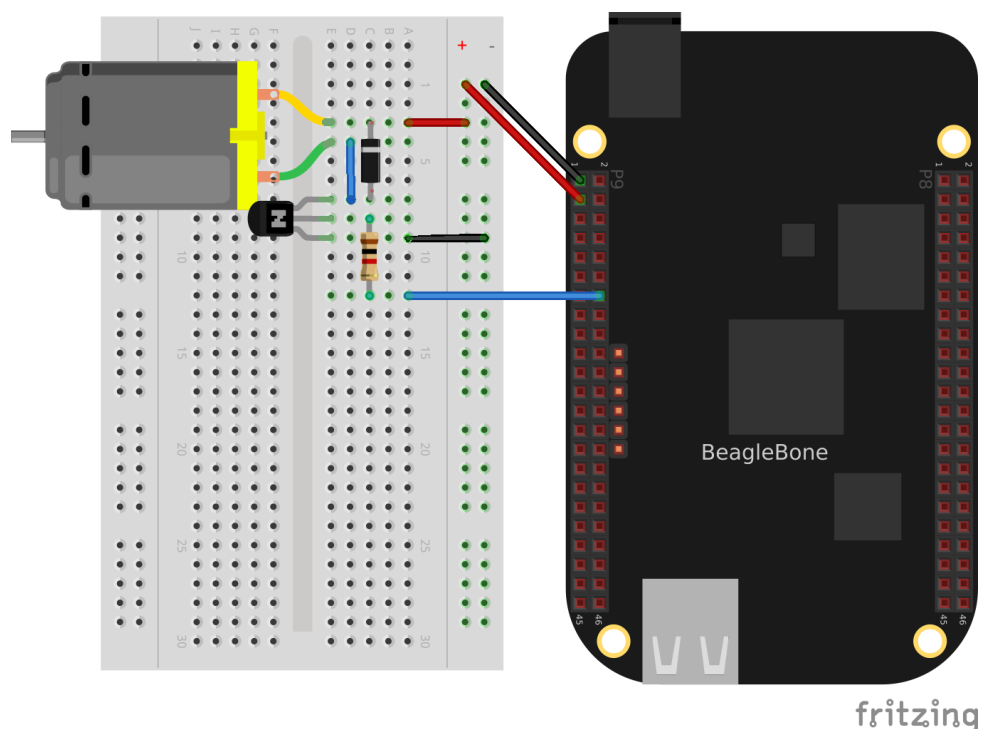


Fig. 4.3: Wiring a DC motor to spin one direction

Use the code in [Driving a DC motor in one direction \(dcMotor.py\)](#) to run the motor.

### Python

Listing 4.4: Driving a DC motor in one direction (dcMotor.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      dcMotor.js
4  # //      This is an example of driving a DC motor
5  # //      Wiring:
6  # //      Setup:  config-pin P9_16 pwm
7  # //      See:
8  # //////////////////////////////////////
9  import time
10 import signal
11 import sys
12

```

(continues on next page)

```

13 def signal_handler(sig, frame):
14     print('Got SIGINT, turning motor off')
15     f = open(PWMPATH+'/enable', 'w')
16     f.write('0')
17     f.close()
18     sys.exit(0)
19 signal.signal(signal.SIGINT, signal_handler)
20
21 pwmPeriod = '1000000'    # Period in ns
22 pwm       = '1'         # pwm to use
23 channel   = 'b'         # channel to use
24 PWMPATH= '/dev/bone/pwm/'+pwm+'/' +channel
25
26 low = 0.05             # Slowest speed (duty cycle)
27 hi  = 1                # Fastest (always on)
28 ms  = 100              # How often to change speed, in ms
29 speed = 0.5           # Current speed
30 step  = 0.05          # Change in speed
31
32 f = open(PWMPATH+'/duty_cycle', 'w')
33 f.write('0')
34 f.close()
35 f = open(PWMPATH+'/period', 'w')
36 f.write(pwmPeriod)
37 f.close()
38 f = open(PWMPATH+'/enable', 'w')
39 f.write('1')
40 f.close()
41
42 f = open(PWMPATH+'/duty_cycle', 'w')
43 while True:
44     speed += step
45     if(speed > hi or speed < low):
46         step *= -1
47     duty_cycle = str(round(speed*1000000))    # Convert ms to ns
48     f.seek(0)
49     f.write(duty_cycle)
50     time.sleep(ms/1000)

```

dcMotor.py

## JavaScript

Listing 4.5: Driving a DC motor in one direction (dcMotor.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      dcMotor.js
4  //      This is an example of driving a DC motor
5  //      Wiring:
6  //      Setup:  config-pin P9_16 pwm
7  //      See:
8  //////////////////////////////////////
9  const fs = require("fs");
10
11 const pwmPeriod = '1000000';    // Period in ns
12 const pwm       = '1';         // pwm to use
13 const channel   = 'b';         // channel to use
14 const PWMPATH= '/dev/bone/pwm/'+pwm+'/' +channel;
15

```

(continues on next page)

(continued from previous page)

```

16 const low = 0.05,      // Slowest speed (duty cycle)
17       hi  = 1,        // Fastest (always on)
18       ms = 100;       // How often to change speed, in ms
19 var   speed = 0.5,    // Current speed;
20       step = 0.05;    // Change in speed
21
22 // fs.writeFileSync(PWMPATH+'/export', pwm); // Export the pwm channel
23 // Set the period in ns, first 0 duty_cycle,
24 fs.writeFileSync(PWMPATH+'/duty_cycle', '0');
25 fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
26 fs.writeFileSync(PWMPATH+'/duty_cycle', pwmPeriod/2);
27 fs.writeFileSync(PWMPATH+'/enable', '1');
28
29 timer = setInterval(sweep, ms);
30
31 function sweep() {
32   speed += step;
33   if(speed > hi || speed < low) {
34     step *= -1;
35   }
36   fs.writeFileSync(PWMPATH+'/duty_cycle', parseInt(pwmPeriod*speed));
37   // console.log('speed = ' + speed);
38 }
39
40 process.on('SIGINT', function() {
41   console.log('Got SIGINT, turning motor off');
42   clearInterval(timer); // Stop the timer
43   fs.writeFileSync(PWMPATH+'/enable', '0');
44 });

```

dcMotor.js

## 4.4 See Also

How do you change the direction of the motor? See [Controlling the Speed and Direction of a DC Motor](#).

## 4.5 Controlling the Speed and Direction of a DC Motor

### 4.5.1 Problem

You would like your DC motor to go forward and backward.

### 4.5.2 Solution

Use an H-bridge to switch the terminals on the motor so that it will run both backward and forward. We'll use the L293D a common, single-chip H-bridge.

Here's what you will need:

- 3 V to 5 V motor.
- Breadboard and jumper wires.
- L293D H-Bridge IC.
- Power supply for the motor (optional)

Lay out your breadboard as shown in [Driving a DC motor with an H-bridge](#). Ensure that the L293D is positioned correctly. There is a notch on one end that should be pointed up.

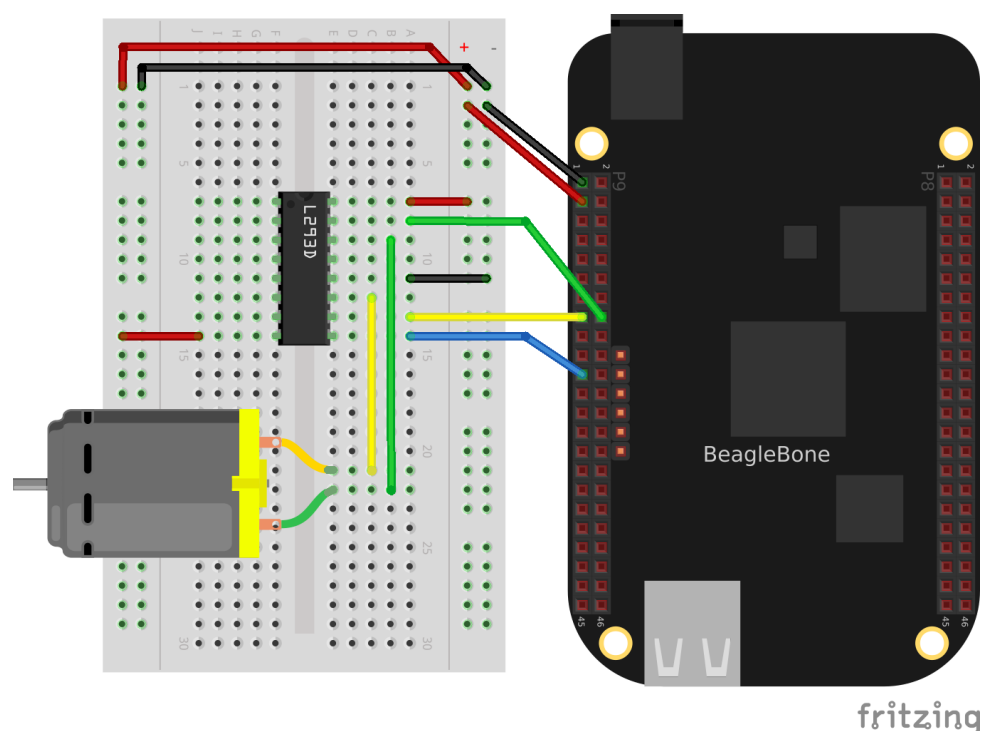


Fig. 4.4: Driving a DC motor with an H-bridge

The code in [Code for driving a DC motor with an H-bridge \(h-bridgeMotor.js\)](#) (`h-bridgeMotor.js`) looks much like the code for driving the DC motor with a transistor ([Driving a DC motor in one direction \(dcMotor.js\)](#)). The additional code specifies which direction to spin the motor.

Listing 4.6: Code for driving a DC motor with an H-bridge (`h-bridgeMotor.js`)

```

1  #!/usr/bin/env node
2
3  // This example uses an H-bridge to drive a DC motor in two directions
4
5  var b = require('bonescript');
6
7  var enable = 'P9_21';    // Pin to use for PWM speed control
8      in1     = 'P9_15',
9      in2     = 'P9_16',
10     step = 0.05,        // Change in speed
11     min  = 0.05,        // Min duty cycle
12     max  = 1.0,         // Max duty cycle
13     ms   = 100,         // Update time, in ms
14     speed = min;        // Current speed;
15
16 b.pinMode(enable, b.ANALOG_OUTPUT, 6, 0, 0, doInterval);
17 b.pinMode(in1, b.OUTPUT);
18 b.pinMode(in2, b.OUTPUT);
19
20 function doInterval(x) {
21     if(x.err) {
22         console.log('x.err = ' + x.err);
23         return;
24     }
25     timer = setInterval(sweep, ms);

```

(continues on next page)

(continued from previous page)

```

26 }
27
28 clockwise();           // Start by going clockwise
29
30 function sweep() {
31     speed += step;
32     if(speed > max || speed < min) {
33         step *= -1;
34         step > 0 ? clockwise() : counterClockwise();
35     }
36     b.analogWrite(enable, speed);
37     console.log('speed = ' + speed);
38 }
39
40 function clockwise() {
41     b.digitalWrite(in1, b.HIGH);
42     b.digitalWrite(in2, b.LOW);
43 }
44
45 function counterClockwise() {
46     b.digitalWrite(in1, b.LOW);
47     b.digitalWrite(in2, b.HIGH);
48 }
49
50 process.on('SIGINT', function() {
51     console.log('Got SIGINT, turning motor off');
52     clearInterval(timer);           // Stop the timer
53     b.analogWrite(enable, 0);       // Turn motor off
54 });

```

h-bridgeMotor.js

## 4.6 Driving a Bipolar Stepper Motor

### 4.6.1 Problem

You want to drive a stepper motor that has four wires.

### 4.6.2 Solution

Use an L293D H-bridge. The bipolar stepper motor requires us to reverse the coils, so we need to use an H-bridge.

Here's what you will need:

- Breadboard and jumper wires.
- 3 V to 5 V bipolar stepper motor.
- L293D H-Bridge IC.

Wire as shown in [Bipolar stepper motor wiring](#).

Use the code in [Driving a bipolar stepper motor \(bipolarStepperMotor.py\)](#) to drive the motor.

Listing 4.7: Driving a bipolar stepper motor (bipolarStepperMotor.py)

```

1 #!/usr/bin/env python
2 import time

```

(continues on next page)



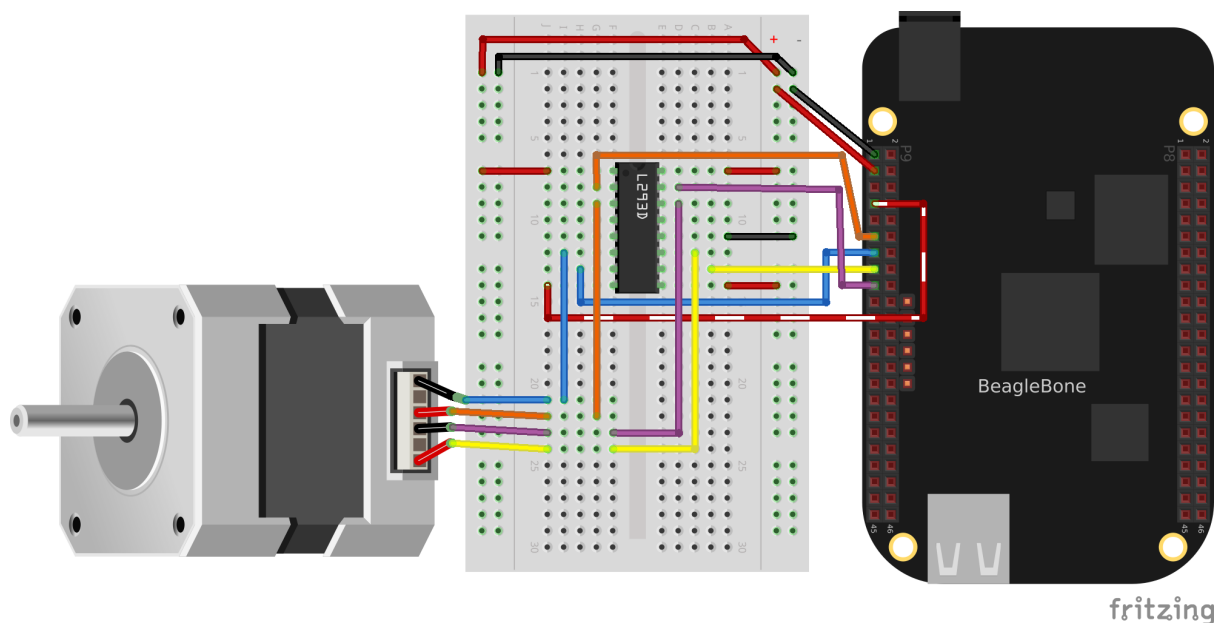


Fig. 4.5: Bipolar stepper motor wiring

(continued from previous page)

```

3 import os
4 import signal
5 import sys
6
7 # Motor is attached here
8 # controller = ["P9_11", "P9_13", "P9_15", "P9_17"];
9 # controller = ["30", "31", "48", "5"]
10 # controller = ["P9_14", "P9_16", "P9_18", "P9_22"];
11 controller = ["50", "51", "4", "2"]
12 states = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
13 statesHiTorque = [[1,1,0,0], [0,1,1,0], [0,0,1,1], [1,0,0,1]]
14 statesHalfStep = [[1,0,0,0], [1,1,0,0], [0,1,0,0], [0,1,1,0],
15                   [0,0,1,0], [0,0,1,1], [0,0,0,1], [1,0,0,1]]
16
17 curState = 0      # Current state
18 ms = 100         # Time between steps, in ms
19 maxStep = 22     # Number of steps to turn before turning around
20 minStep = 0      # minimum step to turn back around on
21
22 CW = 1           # Clockwise
23 CCW = -1
24 pos = 0         # current position and direction
25 direction = CW
26 GPIOPATH="/sys/class/gpio"
27
28 def signal_handler(sig, frame):
29     print('Got SIGINT, turning motor off')
30     for i in range(len(controller)) :
31         f = open(GPIOPATH+"/gpio"+controller[i]+"/value", "w")
32         f.write('0')
33         f.close()
34     sys.exit(0)
35 signal.signal(signal.SIGINT, signal_handler)
36 print('Hit ^C to stop')
37

```

(continues on next page)

(continued from previous page)

```

38 def move():
39     global pos
40     global direction
41     global minStep
42     global maxStep
43     pos += direction
44     print("pos: " + str(pos))
45     # Switch directions if at end.
46     if (pos >= maxStep or pos <= minStep) :
47         direction *= -1
48         rotate(direction)
49
50 # This is the general rotate
51 def rotate(direction) :
52     global curState
53     global states
54     # print("rotate(%d)", direction);
55     # Rotate the state according to the direction of rotation
56     curState += direction
57     if (curState >= len(states)) :
58         curState = 0;
59     elif (curState < 0) :
60         curState = len(states) - 1
61     updateState(states[curState])
62
63 # Write the current input state to the controller
64 def updateState(state) :
65     global controller
66     print(state)
67     for i in range(len(controller)) :
68         f = open(GPIOPATH+"/gpio"+controller[i]+"/value", "w")
69         f.write(str(state[i]))
70         f.close()
71
72 # Initialize motor control pins to be OUTPUTs
73 for i in range(len(controller)) :
74     # Make sure pin is exported
75     if (not os.path.exists(GPIOPATH+"/gpio"+controller[i])):
76         f = open(GPIOPATH+"/export", "w")
77         f.write(pin)
78         f.close()
79     # Make it an output pin
80     f = open(GPIOPATH+"/gpio"+controller[i]+"/direction", "w")
81     f.write("out")
82     f.close()
83
84 # Put the motor into a known state
85 updateState(states[0])
86 rotate(direction)
87
88 # Rotate
89 while True:
90     move()
91     time.sleep(ms/1000)

```

bipolarStepperMotor.py

When you run the code, the stepper motor will rotate back and forth.

## 4.7 Driving a Unipolar Stepper Motor

### 4.7.1 Problem

You want to drive a stepper motor that has five or six wires.

### 4.7.2 Solution

If your stepper motor has five or six wires, it's a unipolar stepper and is wired differently than the bipolar. Here, we'll use a ULN2003 Darlington Transistor Array IC to drive the motor.

Here's what you will need:

- Breadboard and jumper wires.
- 3 V to 5 V unipolar stepper motor.
- ULN2003 Darlington Transistor Array IC.

Wire, as shown in [Unipolar stepper motor wiring](#).

**Note:** The IC in [Unipolar stepper motor wiring](#) is illustrated upside down from the way it is usually displayed.

That is, the notch for pin 1 is on the bottom. This made drawing the diagram much cleaner.

Also, notice the banded wire running the P9\_7 (5 V) to the UL2003A. The stepper motor I'm using runs better at 5 V, so I'm using the Bone's 5 V power supply. The signal coming from the GPIO pins is 3.3 V, but the U2003A will step them up to 5 V to drive the motor.

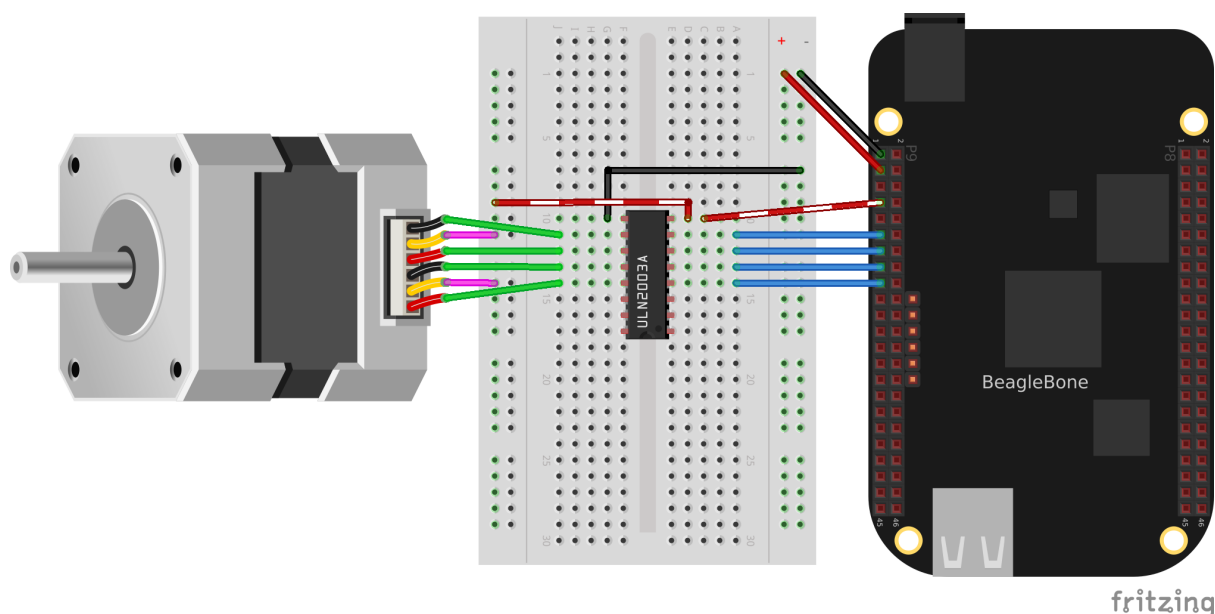


Fig. 4.6: Unipolar stepper motor wiring

The code for driving the motor is in `unipolarStepperMotor.js` however, it is almost identical to the bipolar stepper code ([Driving a bipolar stepper motor \(bipolarStepperMotor.py\)](#)), so [Changes to bipolar code to drive a unipolar stepper motor \(unipolarStepperMotor.js.diff\)](#) shows only the lines that you need to change.

Listing 4.8: Changes to bipolar code to drive a unipolar stepper motor  
(`unipolarStepperMotor.py.diff`)

```
1 # controller = ["P9_11", "P9_13", "P9_15", "P9_17"]
2 controller = ["30", "31", "48", "5"]
```

(continues on next page)

(continued from previous page)

```
3 states = [[1,1,0,0], [0,1,1,0], [0,0,1,1], [1,0,0,1]]
4 curState = 0 // Current state
5 ms = 100 // Time between steps, in ms
6 max = 200 // Number of steps to turn before turning around
```

unipolarStepperMotor.py.diff

Listing 4.9: Changes to bipolar code to drive a unipolar stepper motor  
(unipolarStepperMotor.js.diff)

```
1 # var controller = ["P9_11", "P9_13", "P9_15", "P9_17"];
2 controller = ["30", "31", "48", "5"]
3 var states = [[1,1,0,0], [0,1,1,0], [0,0,1,1], [1,0,0,1]];
4 var curState = 0; // Current state
5 var ms = 100, // Time between steps, in ms
6 max = 200, // Number of steps to turn before turning around
```

unipolarStepperMotor.js.diff

The code in this example makes the following changes:

- The *states* are different. Here, we have two pins high at a time.
- The time between steps (*ms*) is shorter, and the number of steps per direction (*max*) is bigger. The unipolar stepper I'm using has many more steps per rotation, so I need more steps to make it go around.



## Chapter 5

# Beyond the Basics

In *Basics*, you learned how to set up BeagleBone Black, and *Sensors, Displays and Other Outputs*, and *Motors* showed how to interface to the physical world. The remainder of the book moves into some more exciting advanced topics, and this chapter gets you ready for them.

The recipes in this chapter assume that you are running Linux on your host computer (*Selecting an OS for Your Development Host Computer*) and are comfortable with using Linux. We continue to assume that you are logged in as *debian* on your Bone.

### 5.1 Running Your Bone Standalone

#### 5.1.1 Problem

You want to use BeagleBone Black as a desktop computer with keyboard, mouse, and an HDMI display.

#### 5.1.2 Solution

The Bone comes with USB and a microHDMI output. All you need to do is connect your keyboard, mouse, and HDMI display to it.

To make this recipe, you will need:

- Standard HDMI cable and female HDMI-to-male microHDMI adapter, or
- MicroHDMI-to-HDMI adapter cable
- HDMI monitor
- USB keyboard and mouse
- Powered USB hub

---

**Note:** The microHDMI adapter is nice because it allows you to use a regular HDMI cable with the Bone. However, it will block other ports and can damage the Bone if you aren't careful. The microHDMI-to-HDMI cable won't have these problems.

---

---

**Tip:** You can also use an HDMI-to-DVI cable and use your Bone with a DVI-D display.

---

The adapter looks something like *Female HDMI-to-male microHDMI adapter*.

Plug the small end into the microHDMI input on the Bone and plug your HDMI cable into the other end of the adapter and your monitor. If nothing displays on your Bone, reboot.



Fig. 5.1: Female HDMI-to-male microHDMI adapter

If nothing appears after the reboot, edit the `/boot/uEnv.txt` file. Search for the line containing `disable_uboot_overlay_video=1` and make sure it's commented out:

```
###Disable auto loading of virtual capes (emmc/video/wireless/adc)
#disable_uboot_overlay_emmc=1
#disable_uboot_overlay_video=1
```

Then reboot.

The `/boot/uEnv.txt` file contains a number of configuration commands that are executed at boot time. The `#` character is used to add comments; that is, everything to the right of a `+#` is ignored by the Bone and is assumed to be for humans to read. In the previous example, `###Disable auto loading` is a comment that informs us the next line(s) are for disabling things. Two `disable_uboot_overlay` commands follow. Both should be commented-out and won't be executed by the Bone.

Why not just remove the line? Later, you might decide you need more general-purpose input/output (GPIO) pins and don't need the HDMI display. If so, just remove the `#` from the `disable_uboot_overlay_video=1` command. If you had completely removed the line earlier, you would have to look up the details somewhere to re-create it.

When in doubt, comment-out don't delete.

---

**Note:** If you want to re-enable the HDMI audio, just comment-out the line you added.

---

The Bone has only one USB port, so you will need to get either a keyboard with a USB hub or a USB hub. Plug the USB hub into the Bone and then plug your keyboard and mouse in to the hub. You now have a Beagle workstation no host computer is needed.

---

**Tip:** A powered hub is recommended because USB can supply only 500 mA, and you'll want to plug many things into the Bone.

---

This recipe disables the HDMI audio, which allows the Bone to try other resolutions. If this fails, see [BeagleBoneBlack HDMI](#) for how to force the Bone's resolution to match your monitor.

## 5.2 Selecting an OS for Your Development Host Computer

### 5.2.1 Problem

Your project needs a host computer, and you need to select an operating system (OS) for it.

### 5.2.2 Solution

For projects that require a host computer, we assume that you are running [Linux Ubuntu 22.04 LTS](#). You can be running either a native installation, through [Windows Subsystem for Linux](#), via a virtual machine such as [VirtualBox](#), or in the cloud ([Microsoft Azure](#) or [Amazon Elastic Compute Cloud](#), EC2, for example).

Recently I've been preferring [Windows Subsystem for Linux](#).

## 5.3 Getting to the Command Shell via SSH

### 5.3.1 Problem

You want to connect to the command shell of a remote Bone from your host computer.



### 5.3.2 Solution

*Running Python and JavaScript Applications from Visual Studio Code* shows how to run shell commands in the Visual Studio Code *bash* tab. However, the Bone has Secure Shell (SSH) enabled right out of the box, so you can easily connect by using the following command to log in as user *debian*, (note the *\$* at the end of the prompt):

```
host$ ssh debian@192.168.7.2
Warning: Permanently added '192.168.7.2' (ED25519) to the list of known
->hosts.
Debian GNU/Linux 11

BeagleBoard.org Debian Bullseye IoT Image 2023-06-03
Support: https://bbb.io/debian
default username:password is [debian:temppwd]

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

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Jun  8 14:02:40 2023 from 192.168.7.1
bone$
```

### 5.3.3 Default password

*debian* has the default password *temppwd*. It's best to change the password:

```
bone$ password
Changing password for debian.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
password: password updated successfully
```

## 5.4 Removing the *Message of the Day*

### 5.4.1 Problem

Every time you login a long message is displayed that you don't need to see.

### 5.4.2 Solution

The contents of the files */etc/motd*, */etc/issue* and */etc/issue.net* are displayed everytime you long it. You can prevent them from being displayed by moving them elsewhere.

```
bone$ sudo mv /etc/motd /etc/motd.orig
bone$ sudo mv /etc/issue /etc/issue.orig
bone$ sudo mv /etc/issue.net /etc/issue.net.orig
```

Now, the next time you *ssh* in they won't be displayed.

## 5.5 Getting to the Command Shell via the Virtual Serial Port

### 5.5.1 Problem

You want to connect to the command shell of a remote Bone from your host computer without using SSH.

### 5.5.2 Solution

Sometimes, you can't connect to the Bone via SSH, but you have a network working over USB to the Bone. There is a way to access the command line to fix things without requiring extra hardware. ([Viewing and Debugging the Kernel and u-boot Messages at Boot Time](#) shows a way that works even if you don't have a network working over USB, but it requires a special serial-to-USB cable.)

---

**Note:** This method doesn't work with WSL.

---

First, check to ensure that the serial port is there. On the host computer, run the following command:

```
host$ ls -ls /dev/ttyACM0
0 crw-rw---- 1 root dialout 166, 0 Jun 19 11:47 /dev/ttyACM0
```

`/dev/ttyACM0` is a serial port on your host computer that the Bone creates when it boots up. The letters `crw-rw---` show that you can't access it as a normal user. However, you can access it if you are part of `dialout` group. See if you are in the `dialout` group:

```
host$ groups
yoder adm tty uucp dialout cdrom sudo dip plugdev lpadmin sambashare
```

Looks like I'm already in the group, but if you aren't, just add yourself to the group:

```
host$ sudo adduser $USER dialout
```

You have to run `adduser` only once. Your host computer will remember the next time you boot up. Now, install and run the `screen` command:

```
host$ sudo apt install screen
host$ screen /dev/ttyACM0 115200
Debian GNU/Linux 7 beaglebone ttyGS0

default username:password is [debian:temppwd]

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

The IP Address for usb0 is: 192.168.7.2
beaglebone login:
```

The `/dev/ttyACM0` parameter specifies which serial port to connect to, and `115200` tells the speed of the connection. In this case, it's 115,200 bits per second.

## 5.6 Viewing and Debugging the Kernel and u-boot Messages at Boot Time

### 5.6.1 Problem

You want to see the messages that are logged by BeagleBone Black as it comes to life.

## 5.6.2 Solution

There is no network in place when the Bone first boots up, so [Getting to the Command Shell via SSH](#) and [Getting to the Command Shell via the Virtual Serial Port](#) won't work. This recipe uses some extra hardware (FTDI cable) to attach to the Bone's console serial port.

To make this recipe, you will need:

- 3.3 V FTDI cable

**Warning:** Be sure to get a 3.3 V FTDI cable (shown in [FTDI cable](#)), because the 5 V cables won't work.

---

**Tip:** The Bone's Serial Debug J1 connector has Pin 1 connected to ground, Pin 4 to receive, and Pin 5 to transmit. The other pins are not attached.

---



Fig. 5.2: FTDI cable

Look for a small triangle at the end of the FTDI cable ([FTDI connector](#)). It's often connected to the black wire.

### BeagleBone

Next, look for the FTDI pins of the Bone (labeled *J1* on the Bone), shown in [FTDI pins for the FTDI connector](#). They are next to the P9 header and begin near pin 20. There is a white dot near P9\_20.

Plug the FTDI connector into the FTDI pins, being sure to connect the `triangle` pin on the connector to the `white dot` pin of the *FTDI* connector.

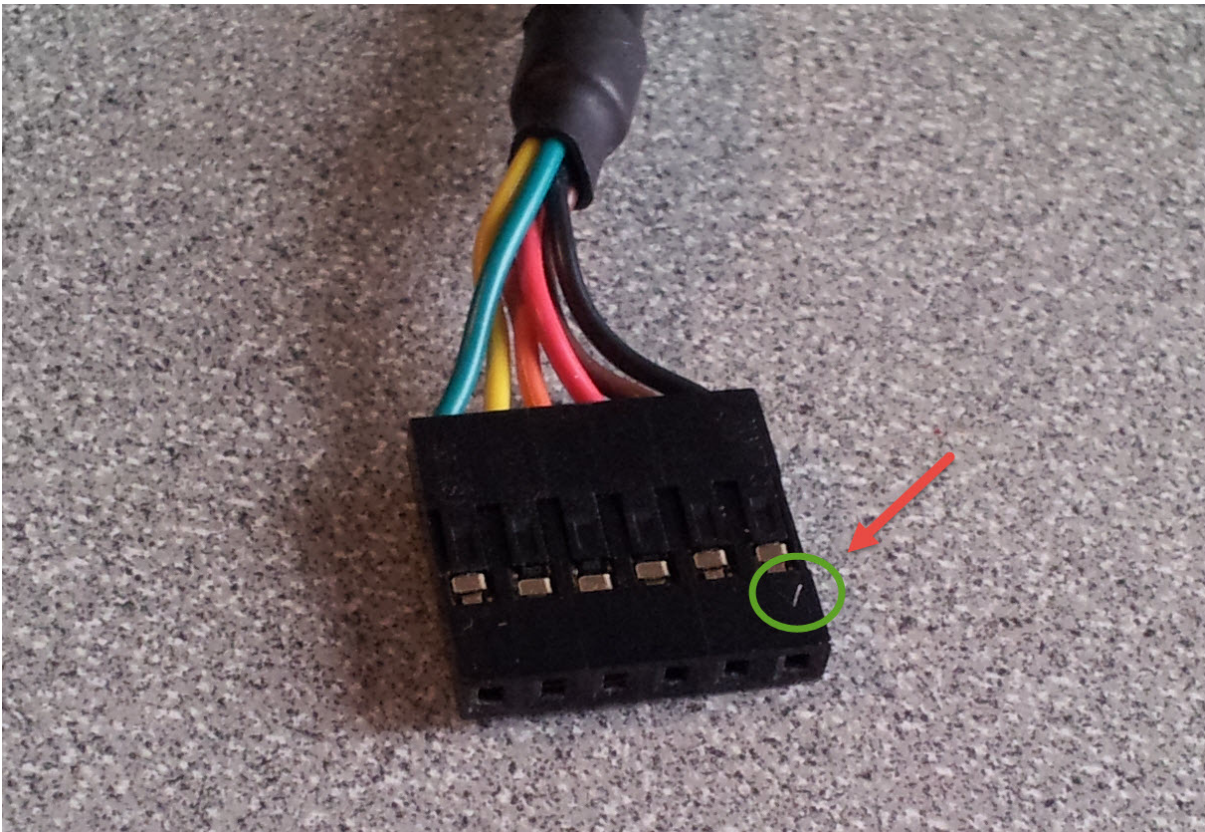


Fig. 5.3: FTDI connector

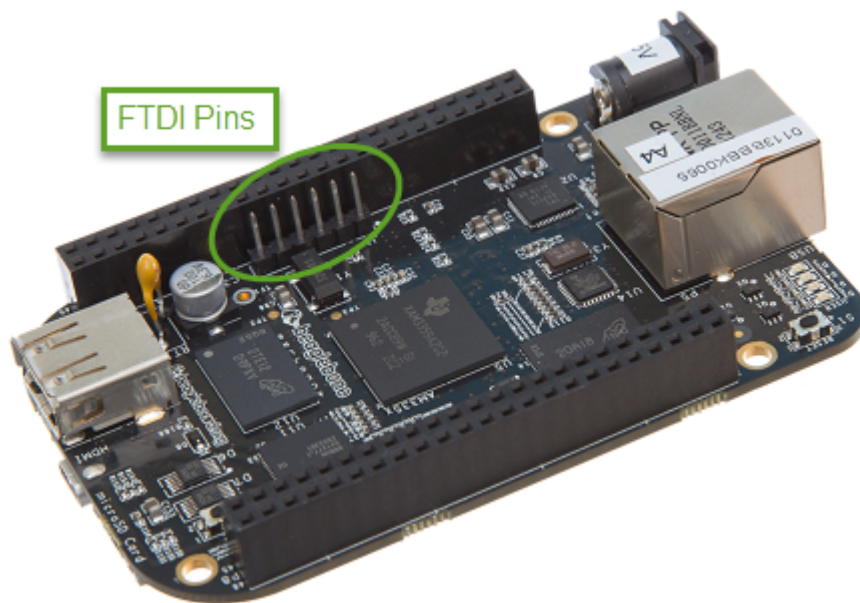


Fig. 5.4: FTDI pins for the FTDI connector

### BeagleY-AI FTDI Cable

When using the BeagleY-AI, if you already have an FTDI cable, all you'll need is a JST SH Compatible 1mm Pitch 3 Pin to Male Headers Cable (<https://www.adafruit.com/product/5755>).

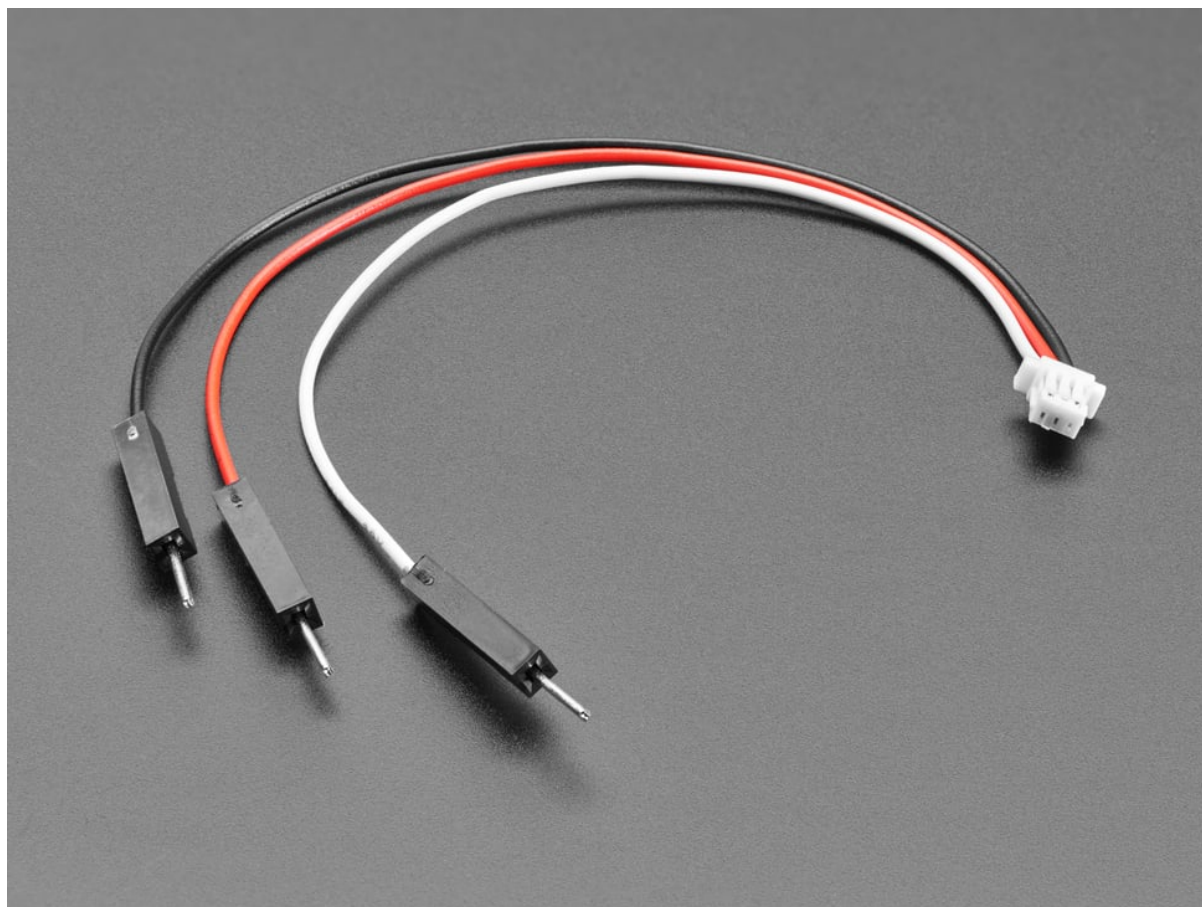


Fig. 5.5: JST SH Compatible 1mm Pitch 3 Pin to Male Headers Cable

Attach the JST cable to the FTDI cable as shown below.

### BeagleY-AI Debug Probe

If you don't have an FTDI cable, you can use a [Raspberry Pi Debug Probe](#) or similar serial (USB to UART) adapter. Connect your UART debug probe to BeagleY-AI as shown in the image below. After making the connection you can use command line utility like `tiO` on Linux or Putty on any operating system. Check [beagle-y-ai-headless](#) for more information.

Now, run the following commands on your host computer:

```
host$ ls -ls /dev/ttyUSB0
0 crw-rw---- 1 root dialout 188, 0 Jun 19 12:43 /dev/ttyUSB0
host$ sudo adduser $USER dialout
host$ screen /dev/ttyUSB0 115200
Debian GNU/Linux 7 beaglebone tty00

default username:password is [debian:tempwd]

Support/FAQ: http://elinux.org/Beagleboard:BeagleBoneBlack\_Debian
```

(continues on next page)



Fig. 5.6: JST to FDTI connection

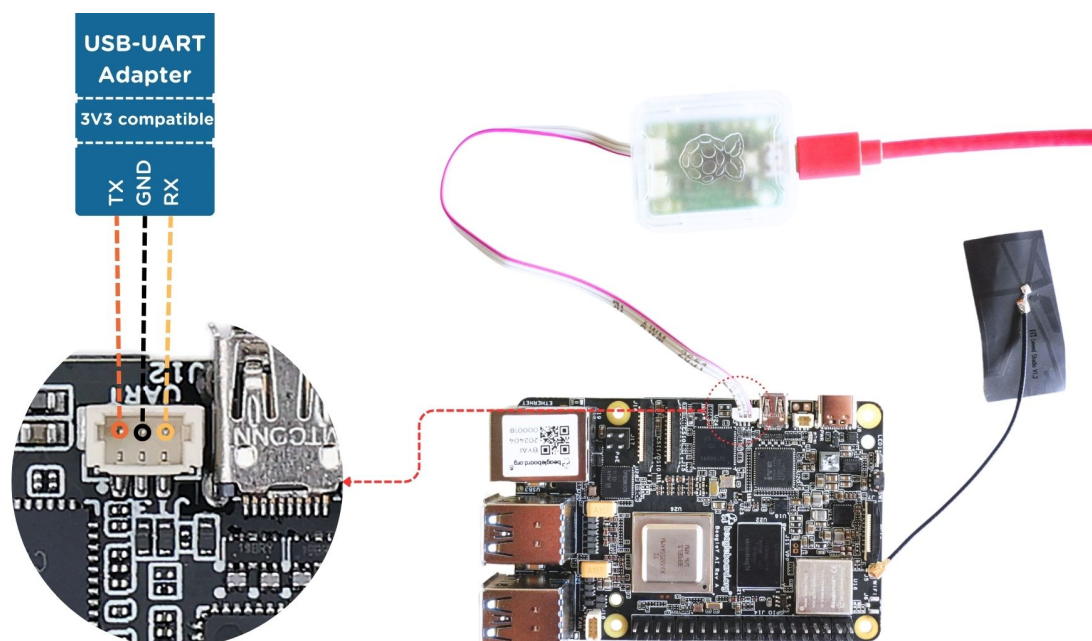


Fig. 5.7: Connecting Raspberry Pi debug probe to BeagleY-AI

(continued from previous page)

```
The IP Address for usb0 is: 192.168.7.2
beaglebone login:
```

**Note:** Your screen might initially be blank. Press Enter a couple of times to see the login prompt.

## 5.7 Verifying You Have the Latest Version of the OS on Your Bone from the Shell

### 5.7.1 Problem

You are logged in to your Bone with a command prompt and want to know what version of the OS you are running.

### 5.7.2 Solution

Log in to your Bone and enter the following command:

```
bone$ cat /etc/dogtag
BeagleBoard.org Debian Bullseye IoT Image 2023-06-03
```

[Verifying You Have the Latest Version of the OS on Your Bone](#) shows how to open the `/etc/dogtag` file to see the OS version. See [Running the Latest Version of the OS on Your Bone](#) if you need to update your OS.

## 5.8 Controlling the Bone Remotely with a VNC

### 5.8.1 Problem

You want to access the BeagleBone's graphical desktop from your host computer.

### 5.8.2 Solution

Install and run a Virtual Network Computing (VNC) server:

```
bone$ sudo apt update
bone$ sudo apt install tightvncserver
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
...
update-alternatives: using /usr/bin/Xtightvnc to provide /usr/bin/Xvnc
→(Xvnc) in auto mode
update-alternatives: using /usr/bin/tightvncpasswd to provide /usr/bin/
→vncpasswd (vncpasswd) in auto mode
Processing triggers for libc-bin (2.31-13+deb11u6) ...

bone$ tightvncserver

You will require a password to access your desktops.

Password:
Verify:
Would you like to enter a view-only password (y/n)? n
xauth: (argv):1: bad display name "beaglebone:1" in "add" command

New 'X' desktop is beaglebone:1

Creating default startup script /home/debian/.vnc/xstartup
Starting applications specified in /home/debian/.vnc/xstartup
Log file is /home/debian/.vnc/beagleboard:1.log
```

To connect to the Bone, you will need to run a VNC client. There are many to choose from. Remmina Remote Desktop Client is already installed on Ubuntu. Start and select the new remote desktop file button ([Creating a new remote desktop file in Remmina Remote Desktop Client](#)).

Give your connection a name, being sure to select "Remmina VNC Plugin" Also, be sure to add `:1` after the server address, as shown in [Configuring the Remmina Remote Desktop Client](#). This should match the `:1` that was displayed when you started `vncserver`.

Click Connect to start graphical access to your Bone, as shown in [The Remmina Remote Desktop Client showing the BeagleBone desktop](#).

---

**Tip:** You might need to resize the VNC screen on your host to see the bottom menu bar on your Bone.

---

**Note:** You need to have X Windows installed and running for the VNC to work. Here's how to install it. This needs some 250M of disk space and 19 minutes to install.

---

```
bone$ sudo apt install bbb.io-xfce4-desktop
bone$ sudo cp /etc/bbb.io/templates/fbdev.xorg.conf /etc/X11/xorg.conf
bone$ startxfce4
/usr/bin/startxfce4: Starting X server
/usr/bin/startxfce4: 122: exec: xinit: not found
```



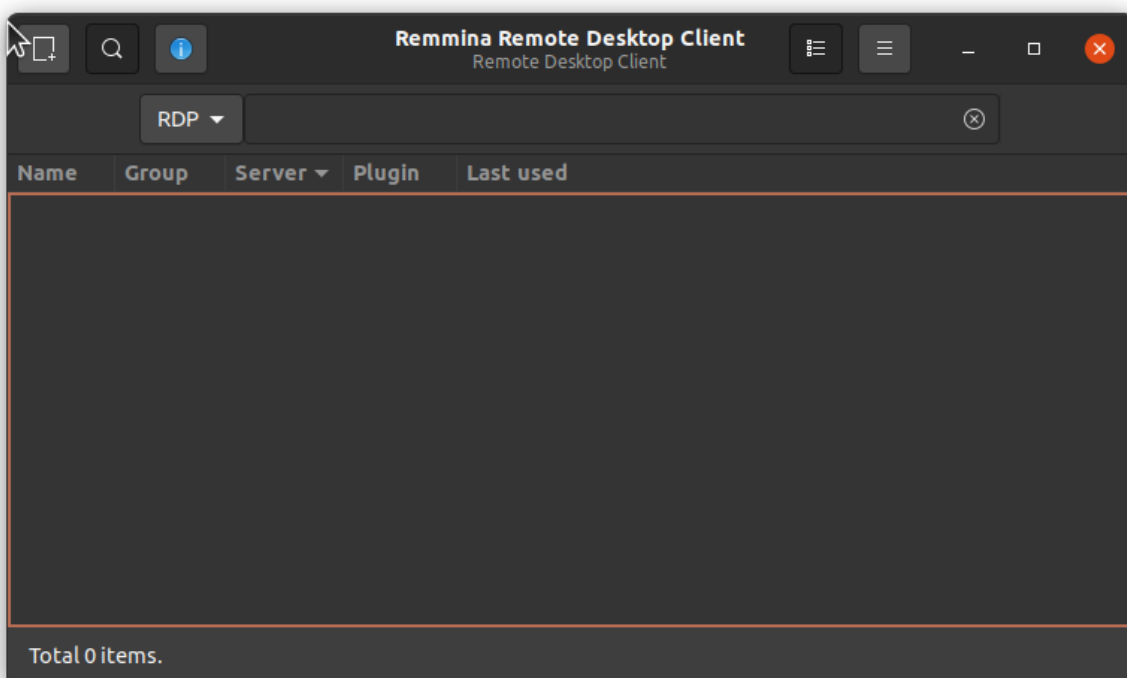


Fig. 5.8: Creating a new remote desktop file in Remmina Remote Desktop Client

## 5.9 Learning Typical GNU/Linux Commands

### 5.9.1 Problem

There are many powerful commands to use in Linux. How do you learn about them?

### 5.9.2 Solution

[Common Linux commands](#) lists many common Linux commands.

Table 5.1: Common Linux commands

Command	Action
pwd	show current directory
cd	change current directory
ls	list directory contents
chmod	change file permissions
chown	change file ownership
cp	copy files
mv	move files
rm	remove files
mkdir	make directory
rmdir	remove directory
cat	dump file contents
less	progressively dump file
vi	edit file (complex)
nano	edit file (simple)
head	trim dump to top

continues on next page

Table 5.1 – continued from previous page

tail	trim dump to bottom
echo	print/dump value
env	dump environment variables
export	set environment variable
history	dump command history
grep	search dump for strings
man	get help on command
apropos	show list of man pages
find	search for files
tar	create/extract file archives
gzip	compress a file
gunzip	decompress a file
du	show disk usage
df	show disk free space
mount	mount disks
tee	write dump to file in parallel
hexdump	readable binary dumps
whereis	locates binary and source files

## 5.10 Editing a Text File from the GNU/Linux Command Shell

### 5.10.1 Problem

You want to run an editor to change a file.

### 5.10.2 Solution

The Bone comes with a number of editors. The simplest to learn is *nano*. Just enter the following command:

```
bone$ nano file
```

You are now in nano (*Editing a file with nano*). You can't move around the screen using the mouse, so use the arrow keys. The bottom two lines of the screen list some useful commands. Pressing ^G (Ctrl-G) will display more useful commands. ^X (Ctrl-X) exits nano and gives you the option of saving the file.

---

**Tip:** By default, the file you create will be saved in the directory from which you opened *nano*.

---

Many other text editors will run on the Bone. *vi*, *vim*, *emacs*, and even *eclipse* are all supported. See [Installing Additional Packages from the Debian Package Feed](#) to learn if your favorite is one of them.

## 5.11 Establishing an Ethernet-Based Internet Connection

### 5.11.1 Problem

You want to connect your Bone to the Internet using the wired network connection.

### 5.11.2 Solution

Plug one end of an Ethernet patch cable into the RJ45 connector on the Bone (see [The RJ45 port on the Bone](#)) and the other end into your home hub/router. The yellow and green link lights on both ends should begin to

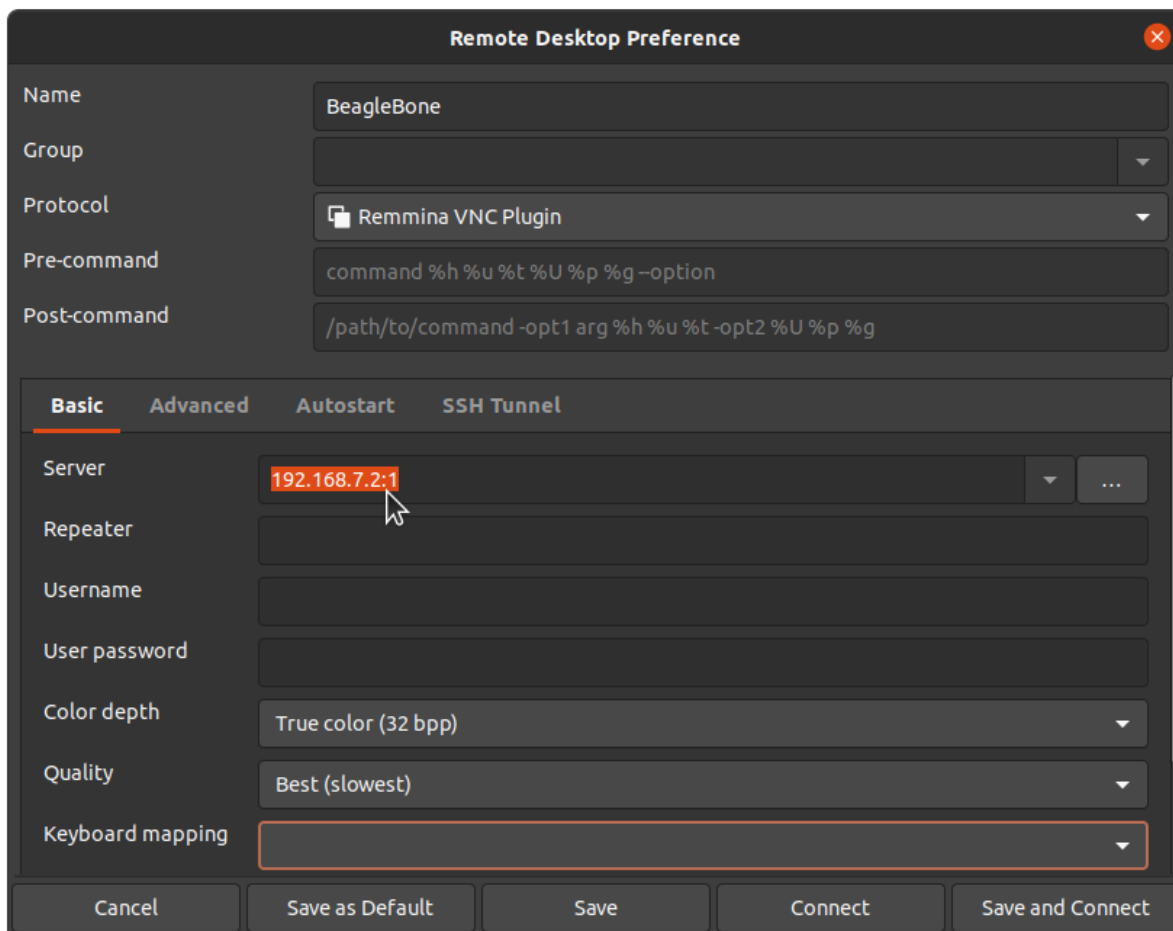


Fig. 5.9: Configuring the Remmina Remote Desktop Client



Fig. 5.10: The Remmina Remote Desktop Client showing the BeagleBone desktop

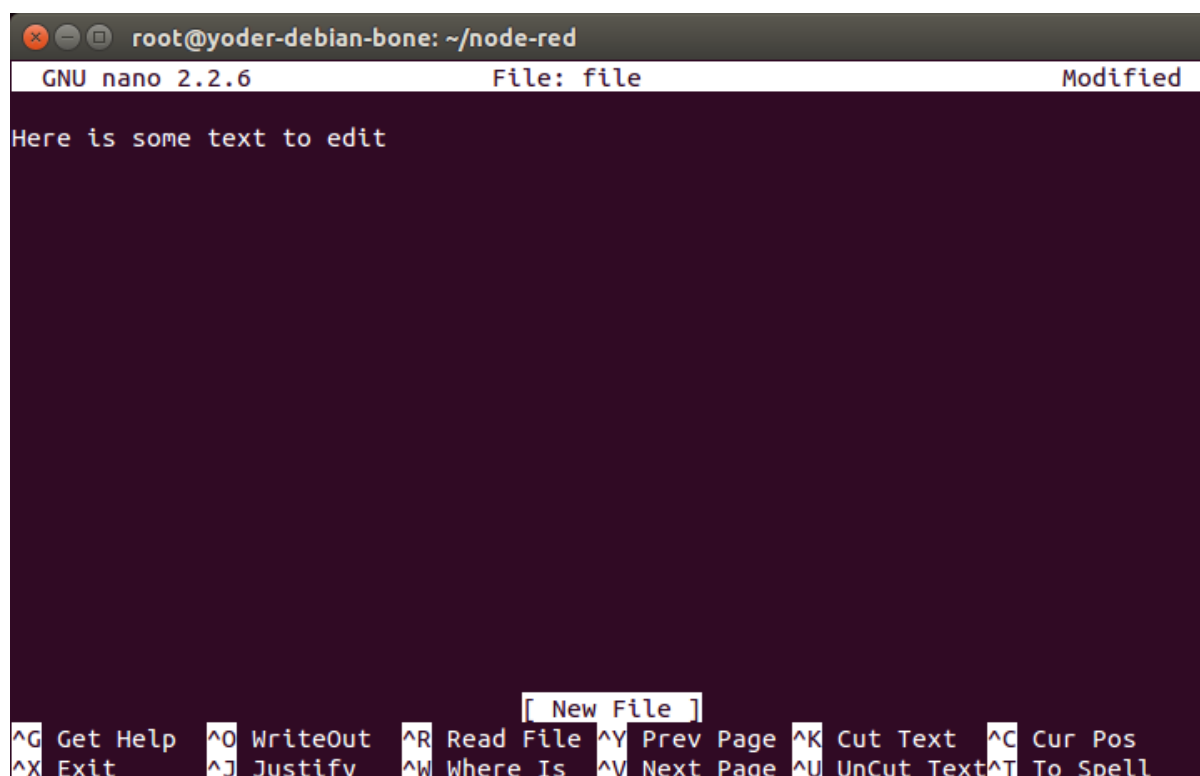


Fig. 5.11: Editing a file with nano

flash.

If your router is already configured to run DHCP (Dynamical Host Configuration Protocol), it will automatically assign an IP address to the Bone.

**Warning:** It might take a minute or two for your router to detect the Bone and assign the IP address.

To find the IP address, open a terminal window and run the *ip* command:

```
bone$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group_
  ↪default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group_
  ↪default qlen 1000
   link/ether c8:a0:30:a6:26:e8 brd ff:ff:ff:ff:ff:ff
   inet 10.0.5.144/24 brd 10.0.5.255 scope global dynamic eth0
       valid_lft 80818sec preferred_lft 80818sec
   inet6 fe80::caa0:30ff:fea6:26e8/64 scope link
       valid_lft forever preferred_lft forever
3: usb0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state_
  ↪UP group default qlen 1000
   link/ether c2:3f:44:bb:41:0f brd ff:ff:ff:ff:ff:ff
   inet 192.168.7.2/24 brd 192.168.7.255 scope global usb0
       valid_lft forever preferred_lft forever
   inet6 fe80::c03f:44ff:febb:410f/64 scope link
```

(continues on next page)

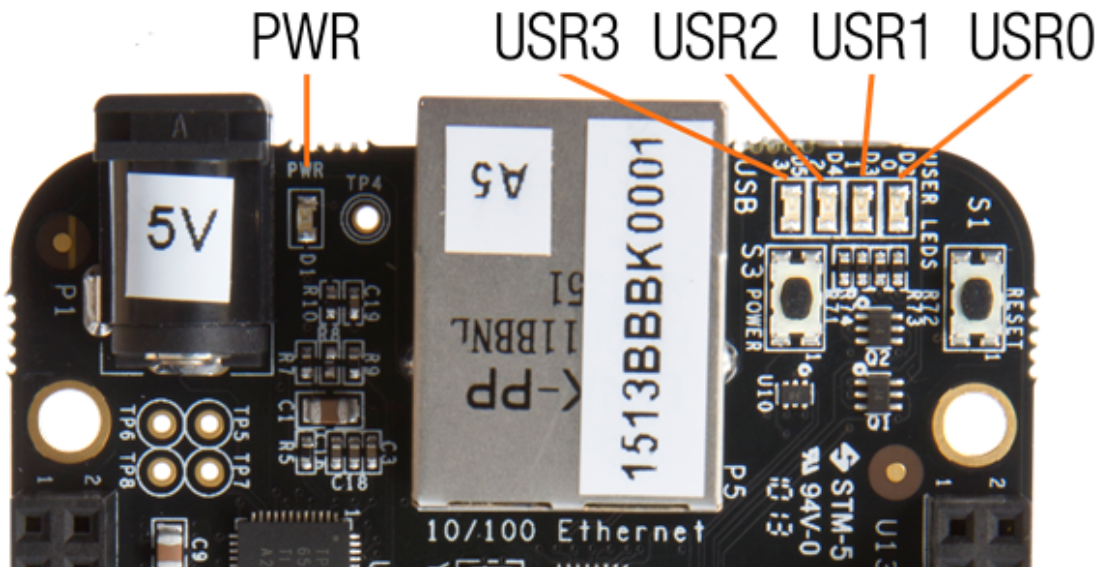


Fig. 5.12: The RJ45 port on the Bone

(continued from previous page)

```

    valid_lft forever preferred_lft forever
4: usb1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state_u
->UP group default qlen 1000
    link/ether 76:7e:49:46:1b:78 brd ff:ff:ff:ff:ff:ff
    inet 192.168.6.2/24 brd 192.168.6.255 scope global usb1
        valid_lft forever preferred_lft forever
    inet6 fe80::747e:49ff:fe46:1b78/64 scope link
        valid_lft forever preferred_lft forever
5: can0: <NOARP,ECHO> mtu 16 qdisc no-op state DOWN group default qlen 10
    link/can
6: can1: <NOARP,ECHO> mtu 16 qdisc no-op state DOWN group default qlen 10
    link/can

```

My Bone is connected to the Internet in two ways: via the RJ45 connection (*eth0*) and via the USB cable (*usb0*). The *inet* field shows that my Internet address is *10.0.5.144* for the RJ45 connector.

On my university campus, you must register your MAC address before any device will work on the network. The *HWaddr* field gives the MAC address. For *eth0*, it's *c8:a0:30:a6:26:e8*.

The IP address of your Bone can change. If it's been assigned by DHCP, it can change at any time. The MAC address, however, never changes; it is assigned to your ethernet device when it's manufactured.

**Warning:** When a Bone is connected to some networks it becomes visible to the world. If you don't secure your Bone, the world will soon find it. See [Default password](#) and [Setting Up a Firewall](#)

On many home networks, you will be behind a firewall and won't be as visible.

## 5.12 Establishing a WiFi-Based Internet Connection

### 5.12.1 Problem

You want BeagleBone Black to talk to the Internet using a USB wireless adapter.

### 5.12.2 Solution

**Tip:** For the correct instructions for the image you are using, go to [latest-images](#) and click on the image you are using.

I'm running Debian 11.x (Bullseye), the top one, on the BeagleBone Black.

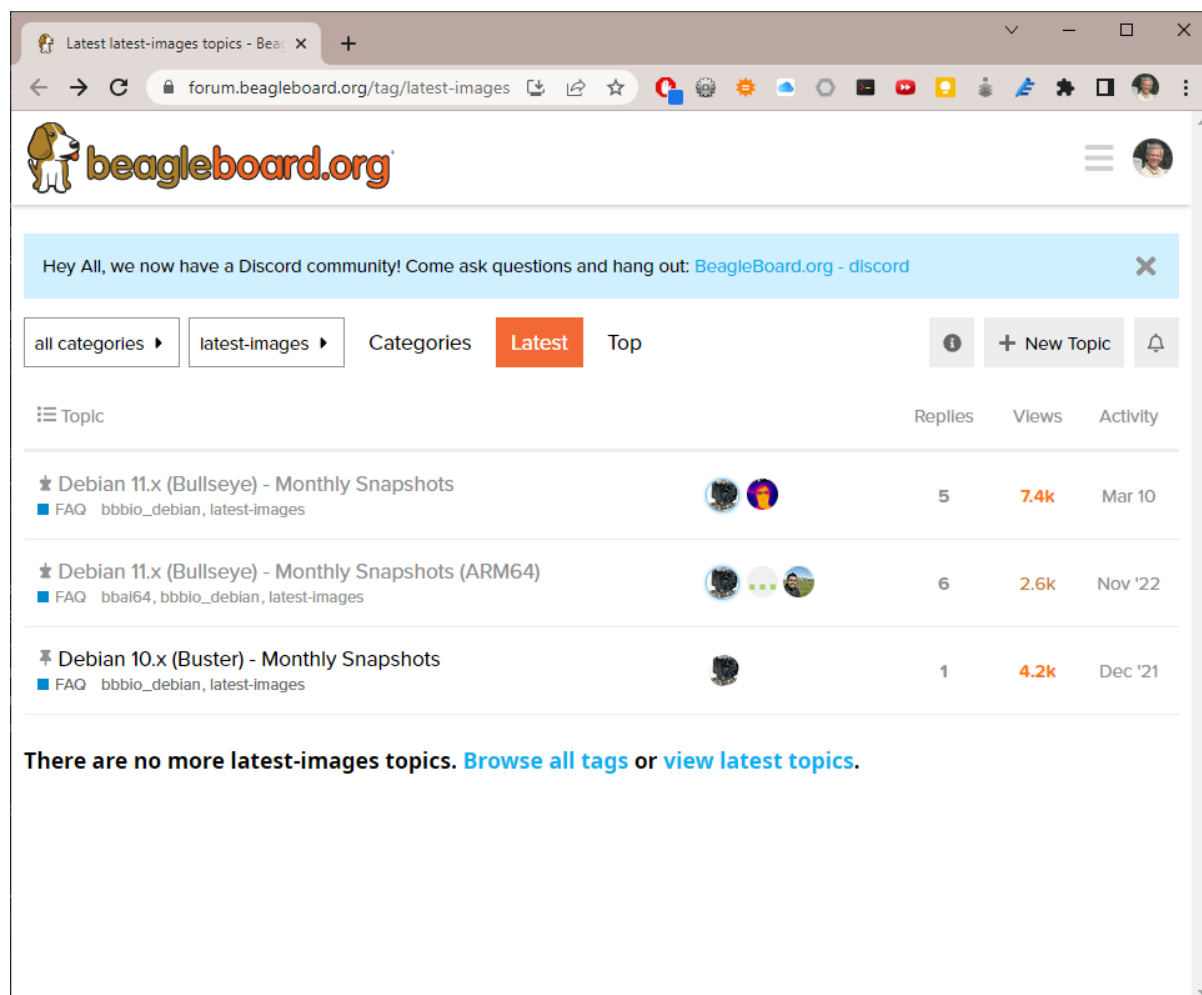


Fig. 5.13: Latest Beagle Images

Scroll to the top of the page and you'll see instructions on setting up Wifi. The instructions here are based on using **networkctl**.

**Todo:** is this up to date?

Several WiFi adapters work with the Bone. Check [WiFi Adapters](#) for the latest list.

To make this recipe, you will need:

- USB Wifi adapter
- 5 V external power supply

Debian 11.x (Bullseye) - Monthly Snapshots

General Discussion bbbio\_debian, latest-images

## network

We migrated from connman to [Debian Systemd-Networkd](#) 28

```

debian@BeagleBone:~$ sudo networkctl
IDX LINK TYPE      OPERATIONAL SETUP
 1 lo  loopback  carrier   unmanaged
 2 eth0 ether     routable  configured
 3 usb0 gadget   no-carrier configuring
 4 usb1 gadget   no-carrier configuring
 5 can0 can       off       unmanaged
 6 can1 can       off       unmanaged

6 links listed.

```

### Configuration files

```

eth0 -> /etc/systemd/network/eth0.network
usb0 (Windows - 192.168.7.x) -> /etc/systemd/network/usb0.network
usb1 (Mac - 192.168.6.x) -> /etc/systemd/network/usb1.network
wlan0 -> /etc/systemd/network/wlan0.network

```

### WiFi Configuration (wpa\_supplicant)

```

sudo nano /etc/wpa_supplicant/wpa_supplicant-wlan0.conf

```

Network

- Configuration files
- WiFi Configuration (wpa\_supplicant)
- WiFi Configuration thru wpa\_cli
- version.sh →
- beagle-version
- Update U-Boot
- eMMC Flasher
- Cloud 9 →
- VSCode port 3000
- NodeRED port 1880
- PRU uio enablement:
- Debian 11.x (Bullseye) Minimal Snapshot
- Debian 11.x (Bullseye) IOT Snapshot

Fig. 5.14: Instructions for setting up your network.



**Warning:** Most adapters need at least 1 A of current to run, and USB supplies only 0.5 A, so be sure to use an external power supply. Otherwise, you will experience erratic behavior and random crashes.

First, plug in the WiFi adapter and the 5 V external power supply and reboot.

Then run *lsusb* to ensure that your Bone found the adapter:

```
bone$ lsusb
Bus 001 Device 002: ID 0bda:8176 Realtek Semiconductor Corp. RTL8188CUS 802.
→11n
WLAN Adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

---

**Note:** There is a well-known bug in the Bone's 3.8 kernel series that prevents USB devices from being discovered when hot-plugged, which is why you should reboot. Newer kernels should address this issue.

---

**Todo:** update

---

Next, run *networkctl* to find your adapter's name. Mine is called *wlan0*, but you might see other names, such as *ra0*.

```
bone$ networkctl
IDX LINK      TYPE          OPERATIONAL SETUP
1 lo          loopback      carrier      unmanaged
2 eth0        ether         no-carrier   configuring
3 usb0        gadget        routable     configured
4 usb1        gadget        routable     configured
5 can0        can           off          unmanaged
6 can1        can           off          unmanaged
7 wlan0       wlan          routable     configured
8 SoftAp0     wlan          routable     configured

8 links listed.
```

If no name appears, try *ip a*:

```
bone$ ip a
...
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state_
→DOWN group default qlen 1000
   link/ether c8:a0:30:a6:26:e8 brd ff:ff:ff:ff:ff:ff
3: usb0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state_
→UP group default qlen 1000
   link/ether c2:3f:44:bb:41:0f brd ff:ff:ff:ff:ff:ff
   inet 192.168.7.2/24 brd 192.168.7.255 scope global usb0
       valid_lft forever preferred_lft forever
   inet6 fe80::c03f:44ff:febb:410f/64 scope link
       valid_lft forever preferred_lft forever
...
7: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group_
→default qlen 1000
   link/ether 64:69:4e:7e:5c:e4 brd ff:ff:ff:ff:ff:ff
   inet 10.0.7.21/24 brd 10.0.7.255 scope global dynamic wlan0
       valid_lft 85166sec preferred_lft 85166sec
   inet6 fe80::6669:4eff:fe7e:5ce4/64 scope link
       valid_lft forever preferred_lft forever
```

(continues on next page)

(continued from previous page)

```
Next edit the configuration file */etc/wpa_supplicant/wpa_supplicant-wlan0.conf*.
```

```
bone$ sudo nano /etc/wpa_supplicant/wpa_supplicant-wlan0.conf
```

In the file you'll see:

```
ctrl_interface=DIR=/run/wpa_supplicant GROUP=netdev
update_config=1
#country=US

network={
    ssid="Your SSID"
    psk="Your Password"
}
```

Change the *ssid* and *psk* entries for your network. Save your file, then run:

```
bone$ sudo systemctl restart systemd-networkd
bone$ ip a
bone$ ping -c2 google.com
PING google.com (142.250.191.206) 56(84) bytes of data.
64 bytes from ord38s31-in-f14.1e100.net (142.250.191.206): icmp_seq=1
    →ttl=115 time=19.5 ms
64 bytes from ord38s31-in-f14.1e100.net (142.250.191.206): icmp_seq=2
    →ttl=115 time=19.4 ms

--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 19.387/19.450/19.513/0.063 ms
```

*wlan0* should now have an ip address and you should be on the network. If not, try rebooting.

## 5.13 Sharing the Host's Internet Connection over USB

---

**Todo:** Test this

---

### 5.13.1 Problem

Your host computer is connected to the Bone via the USB cable, and you want to run the network between the two.

### 5.13.2 Solution

[Establishing an Ethernet-Based Internet Connection](#) shows how to connect BeagleBone Black to the Internet via the RJ45 Ethernet connector. This recipe shows a way to connect without using the RJ45 connector.

A network is automatically running between the Bone and the host computer at boot time using the USB. The host's IP address is *192.168.7.1* and the Bone's is *192.168.7.2*. Although your Bone is talking to your host, it can't reach the Internet in general, nor can the Internet reach it. On one hand, this is good, because those who are up to no good can't access your Bone. On the other hand, your Bone can't reach the rest of the world.

## Letting your bone see the world: setting up IP masquerading

You need to set up IP masquerading on your host and configure your Bone to use it. Here is a solution that works with a host computer running Linux. Add the code in [Code for IP Masquerading \(ipMasquerade.sh\)](#) to a file called `ipMasquerade.sh` on your host computer.

Listing 5.1: Code for IP Masquerading (ipMasquerade.sh)

```

1 #!/bin/bash
2 # These are the commands to run on the host to set up IP
3 # masquerading so the Bone can access the Internet through
4 # the USB connection.
5 # This configures the host, run ./setDNS.sh to configure the Bone.
6 # Inspired by http://thoughtshubham.blogspot.com/2010/03/
7 # internet-over-usb-otg-on-beagleboard.html
8
9 if [ $# -eq 0 ] ; then
10 echo "Usage: $0 interface (such as eth0 or wlan0)"
11 exit 1
12 fi
13
14 interface=$1
15 hostAddr=192.168.7.1
16 beagleAddr=192.168.7.2
17 ip_forward=/proc/sys/net/ipv4/ip_forward
18
19 if [ `cat $ip_forward` == 0 ]
20 then
21     echo "You need to set IP forwarding. Edit /etc/sysctl.conf using:"
22     echo "$ sudo nano /etc/sysctl.conf"
23     echo "and uncomment the line \"net.ipv4.ip_forward=1\""
24     echo "to enable forwarding of packets. Then run the following:"
25     echo "$ sudo sysctl -p"
26     exit 1
27 else
28     echo "IP forwarding is set on host."
29 fi
30 # Set up IP masquerading on the host so the bone can reach the outside world
31 sudo iptables -t nat -A POSTROUTING -s $beagleAddr -o $interface -j_
    ↪MASQUERADE

```

`ipMasquerade.sh`

Then, on your host, run the following commands:

```

host$ chmod +x ipMasquerade.sh
host$ ./ipMasquerade.sh eth0

```

This will direct your host to take requests from the Bone and send them to `eth0`. If your host is using a wireless connection, change `eth0` to `wlan0`.

Now let's set up your host to instruct the Bone what to do. Add the code in [Code for setting the DNS on the Bone \(setDNS.sh\)](#) to `setDNS.sh` on your host computer.

Listing 5.2: Code for setting the DNS on the Bone (setDNS.sh)

```

1 #!/bin/bash
2 # These are the commands to run on the host so the Bone
3 # can access the Internet through the USB connection.
4 # Run ./ipMasquerade.sh the first time. It will set up the host.
5 # Run this script if the host is already set up.
6 # Inspired by http://thoughtshubham.blogspot.com/2010/03/internet-over-usb-
    ↪otg-on-beagleboard.html

```

(continues on next page)

(continued from previous page)

```

7
8 hostAddr=192.168.7.1
9 beagleAddr=${1:-192.168.7.2}
10
11 # Save the /etc/resolv.conf on the Beagle in case we mess things up.
12 ssh root@$beagleAddr "mv -n /etc/resolv.conf /etc/resolv.conf.orig"
13 # Create our own resolv.conf
14 cat - << EOF > /tmp/resolv.conf
15 # This is installed by ./setDNS.sh on the host
16
17 EOF
18
19 TMP=/tmp/nmcli
20 # Look up the nameserver of the host and add it to our resolv.conf
21 # From: http://askubuntu.com/questions/197036/how-to-know-what-dns-am-i-
22   ↳using-in-ubuntu-12-04
23 # Use nmcli dev list for older version nmcli
24 # Use nmcli dev show for newer version nmcli
25 nmcli dev show > $TMP
26 if [ $? -ne 0 ]; then # $? is the return code, if not 0 something bad
27   ↳happened.
28     echo "nmcli failed, trying older 'list' instead of 'show'"
29     nmcli dev list > $TMP
30     if [ $? -ne 0 ]; then
31       echo "nmcli failed again, giving up..."
32       exit 1
33     fi
34 fi
35
36 grep IP4.DNS $TMP | sed 's/IP4.DNS\[.\]:/nameserver/' >> /tmp/resolv.conf
37
38 scp /tmp/resolv.conf root@$beagleAddr:/etc
39
40 # Tell the beagle to use the host as the gateway.
41 ssh root@$beagleAddr "/sbin/route add default gw $hostAddr" || true

```

```
setDNS.sh
```

Then, on your host, run the following commands:

```

host$ chmod +x setDNS.sh
host$ ./setDNS.sh
host$ ssh -X root@192.168.7.2
bone$ ping -c2 google.com
PING google.com (216.58.216.96) 56(84) bytes of data.
64 bytes from ord30s22....net (216.58.216.96): icmp_req=1 ttl=55 time=7.49 ms
64 bytes from ord30s22....net (216.58.216.96): icmp_req=2 ttl=55 time=7.62 ms

--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 7.496/7.559/7.623/0.107 ms

```

This will look up what Domain Name System (DNS) servers your host is using and copy them to the right place on the Bone. The *ping* command is a quick way to verify your connection.

### Letting the world see your bone: setting up port forwarding

Now your Bone can access the world via the USB port and your host computer, but what if you have a web server on your Bone that you want to access from the world? The solution is to use port forwarding from your host. Web servers typically listen to port 80. First, look up the IP address of your host:

---

**Todo:** switch to ip address

---

```
host$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group_
↳default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc mq state UP group_
↳default qlen 1000
   link/ether 00:15:5d:7c:e8:dc brd ff:ff:ff:ff:ff:ff
   inet 172.31.43.210/20 brd 172.31.47.255 scope global eth0
       valid_lft forever preferred_lft forever
   inet6 fe80::215:5dff:fe7c:e8dc/64 scope link
       valid_lft forever preferred_lft forever
```

It's the number following *inet*, which in my case is *172.31.43.210*.

---

**Tip:** If you are on a wireless network, find the IP address associated with *wlan0*.

---

Then run the following, using your host's IP address:

---

**Todo:** check this iptables, convert to ufw

---

```
host$ sudo iptables -t nat -A PREROUTING -p tcp -s 0/0 \
-d 172.31.43.210 --dport 1080 -j DNAT --to 192.168.7.2:80
```

Now browse to your host computer at port *1080*. That is, if your host's IP address is *123.456.789.0*, enter *123.456.789.0:1080*. The *:1080* specifies what port number to use. The request will be forwarded to the server on your Bone listening to port *80*. (I used *1080* here, in case your host is running a web server of its own on port *80*.)

## 5.14 Setting Up a Firewall

### 5.14.1 Problem

You have put your Bone on the network and want to limit which IP addresses can access it.

### 5.14.2 Solution

[How-To Geek](#) has a great posting on how do use *ufw*, the "uncomplicated firewall". Check out [How to Secure Your Linux Server with a UFW Firewall](#). I'll summarize the initial setup here.

First install and check the status:

```
bone$ sudo apt update
bone$ sudo apt install ufw
bone$ sudo ufw status
Status: inactive
```

Now turn off everything coming in and leave on all outgoing. Note, this won't take effect until *ufw* is enabled.

```
bone$ sudo ufw default deny incoming
bone$ sudo ufw default allow outgoing
```

Don't enable yet, make sure *ssh* still has access

```
bone$ sudo ufw allow 22
```

Just to be sure, you can install *nmap* on your host computer to see what ports are currently open.

```
host$ sudo apt update
host$ sudo apt install nmap
host$ nmap 192.168.7.2
Starting Nmap 7.80 ( https://nmap.org ) at 2022-07-09 13:37 EDT
Nmap scan report for bone (192.168.7.2)
Host is up (0.014s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
3000/tcp  open  ppp

Nmap done: 1 IP address (1 host up) scanned in 0.19 seconds
```

Currently there are three ports visible: 22, 80 and 3000 (visual studio code). Now turn on the firewall and see what happens.

```
bone$ sudo ufw enable
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup

host$ nmap 192.168.7.2
Starting Nmap 7.80 ( https://nmap.org ) at 2022-07-09 13:37 EDT
Nmap scan report for bone (192.168.7.2)
Host is up (0.014s latency).
Not shown: 999 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh

Nmap done: 1 IP address (1 host up) scanned in 0.19 seconds
```

Only port 22 (*ssh*) is accessible now.

The firewall will remain on, even after a reboot. Disable it now if you don't want it on.

```
bone$ sudo ufw disable
Firewall stopped and disabled on system startup
```

See the [How-To Geek](#) article for more examples.

## 5.15 Installing Additional Packages from the Debian Package Feed

### 5.15.1 Problem

You want to do more cool things with your BeagleBone by installing more programs.

## 5.15.2 Solution

**Warning:** Your Bone needs to be on the network for this to work. See [Establishing an Ethernet-Based Internet Connection](#), [Establishing a WiFi-Based Internet Connection](#), or [Sharing the Host's Internet Connection over USB](#).

The easiest way to install more software is to use **apt**:

```
bone$ sudo apt update
bone$ sudo apt install "name of software"
```

A *sudo* is necessary since you aren't running as *root*. The first command downloads package lists from various repositories and updates them to get information on the newest versions of packages and their dependencies. (You need to run it only once a week or so.) The second command fetches the software and installs it and all packages it depends on.

How do you find out what software you can install? Try running this:

```
bone$ apt-cache pkgnames | sort > /tmp/list
bone$ wc /tmp/list
  67974   67974 1369852 /tmp/list
bone$ less /tmp/list
```

The first command lists all the packages that *apt* knows about and sorts them and stores them in */tmp/list*. The second command shows why you want to put the list in a file. The *wc* command counts the number of lines, words, and characters in a file. In our case, there are over 67,000 packages from which we can choose! The *less* command displays the sorted list, one page at a time. Press the space bar to go to the next page. Press **q** to quit.

Suppose that you would like to install an online dictionary (*dict*). Just run the following command:

```
bone$ sudo apt install dict
```

Now you can run *dict*.

## 5.16 Removing Packages Installed with apt

### 5.16.1 Problem

You've been playing around and installing all sorts of things with *apt* and now you want to clean things up a bit.

### 5.16.2 Solution

*apt* has a *remove* option, so you can run the following command:

```
bone$ sudo apt remove dict
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer
↳required:
libmaa3 librecode0 recode
Use 'apt autoremove' to remove them.
The following packages will be REMOVED:
dict
```

(continues on next page)

(continued from previous page)

```
0 upgraded, 0 newly installed, 1 to remove and 27 not upgraded.
After this operation, 164 kB disk space will be freed.
Do you want to continue [Y/n]? y
```

## 5.17 Copying Files Between the Onboard Flash and the microSD Card

### 5.17.1 Problem

You want to move files between the onboard flash and the microSD card.

### 5.17.2 Solution

First, make sure your Beagle has eMMC. Run `lsblk`.

```
beagle:~$ lsblk
NAME                MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
mmcblk1             179:0    0   3.6G  0 disk
└─mmcblk1p1         179:1    0   3.6G  0 part
mmcblk1boot0        179:256  0     2M  1 disk
mmcblk1boot1        179:512  0     2M  1 disk
mmcblk0             179:768  0   7.4G  0 disk
└─mmcblk0p1         179:769  0   7.4G  0 part /
```

If the results show `mmcblk0` and `mmcblk1` like above, you have eMMC and can do the rest of this recipe. If your results are like below, you don't have eMMC.

```
beagle:~$ lsblk
NAME                MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
mmcblk1             179:0    0   7.5G  0 disk
├─mmcblk1p1         179:1    0  256M  0 part /boot/firmware
└─mmcblk1p2         179:2    0   7.3G  0 part /
```

If you booted from the microSD card, run the following command:

```
bone$ df -h
Filesystem      Size  Used Avail Use% Mounted on
rootfs          7.2G  2.0G  4.9G  29% /
udev            10M    0   10M   0% /dev
tmpfs           100M  1.9M   98M   2% /run
/dev/mmcblk0p2  7.2G  2.0G  4.9G  29% /
tmpfs           249M    0  249M   0% /dev/shm
tmpfs           249M    0  249M   0% /sys/fs/cgroup
tmpfs           5.0M    0   5.0M   0% /run/lock
tmpfs           100M    0  100M   0% /run/user

bone$ ls /dev/mmcblk*
/dev/mmcblk0    /dev/mmcblk0p2  /dev/mmcblk1boot0  /dev/mmcblk1p1
/dev/mmcblk0p1 /dev/mmcblk1    /dev/mmcblk1boot1
```

The `df` command shows what partitions are already mounted. The line `/dev/mmcblk0p2 7.2G 2.0G 4.9G 29% /` shows that `mmcblk0` partition `p2` is mounted as `/`, the root file system. The general rule is that the media you're booted from (either the onboard flash or the microSD card) will appear as `mmcblk0`. The second partition (`p2`) is the root of the file system.

The `ls` command shows what devices are available to mount. Because `mmcblk0` is already mounted, `/dev/mmcblk1p1` must be the other media that we need to mount. Run the following commands to mount it:



---

**Todo:** update

---

```
bone$ cd /mnt
bone$ sudo mkdir onboard
bone$ ls onboard
bone$ sudo mount /dev/mmcblk1p1 onboard/
bone$ ls onboard
bin    etc      lib      mnt      proc    sbin     sys      var
boot  home    lost+found  nfs-uEnv.txt  root    selinux  tmp
dev   ID.txt  media    opt      run     srv      usr
```

The `cd` command takes us to a place in the file system where files are commonly mounted. The `mkdir` command creates a new directory (`onboard`) to be a mount point. The `ls` command shows there is nothing in `onboard`. The `mount` command makes the contents of the onboard flash accessible. The next `ls` shows there now are files in `onboard`. These are the contents of the onboard flash, which can be copied to and from like any other file.

This same process should also work if you have booted from the onboard flash. When you are done with the onboard flash, you can unmount it by using this command:

```
bone$ sudo umount /mnt/onboard
```

## 5.18 Freeing Space on the Onboard Flash or microSD Card

### 5.18.1 Problem

You are starting to run out of room on your microSD card (or onboard flash) and have removed several packages you had previously installed ([Removing Packages Installed with apt](#)), but you still need to free up more space.

### 5.18.2 Solution

To free up space, you can remove preinstalled packages or discover big files to remove.

#### Removing preinstalled packages

You might not need a few things that come preinstalled in the Debian image, including such things as OpenCV, the Chromium web browser, and some documentation.

---

**Note:** The Chromium web browser is the open source version of Google's Chrome web browser. Unless you are using the Bone as a desktop computer, you can probably remove it.

---

Here's how you can remove these:

```
bone$ sudo apt remove bb-node-red-installer (171M)
bone$ sudo apt autoremove
bone$ sudo -rf /usr/share/doc (116M)
bone$ sudo -rf /usr/share/man (19M)
```

#### Discovering big files

The `du` (disk usage) command offers a quick way to discover big files:

```
bone$ sudo du -shx /*
12M  /bin
160M /boot
0    /dev
23M  /etc
835M /home
4.0K /ID.txt
591M /lib
16K  /lost+found
4.0K /media
8.0K /mnt
664M /opt
du: cannot access '/proc/1454/task/1454/fd/4': No such file or directory
du: cannot access '/proc/1454/task/1454/fdinfo/4': No such file or directory
du: cannot access '/proc/1454/fd/3': No such file or directory
du: cannot access '/proc/1454/fdinfo/3': No such file or directory
0    /proc
1.4M /root
1.4M /run
13M  /sbin
4.0K /srv
0    /sys
48K  /tmp
1.6G /usr
1.9G /var
```

If you booted from the microSD card, *du* lists the usage of the microSD. If you booted from the onboard flash, it lists the onboard flash usage.

The *-s* option summarizes the results rather than displaying every file. *-h* prints it in *\_human\_* form—that is, using *M* and *K* postfixes rather than showing lots of digits. The */\** specifies to run it on everything in the top-level directory. It looks like a couple of things disappeared while the command was running and thus produced some error messages.

---

**Tip:** For more help, try *du -help*.

---

The */var* directory appears to be the biggest user of space at 1.9 GB. You can then run the following command to see what's taking up the space in */var*:

```
bone$ sudo du -sh /var/*
4.0K /var/backups
76M  /var/cache
93M  /var/lib
4.0K /var/local
0    /var/lock
751M /var/log
4.0K /var/mail
4.0K /var/opt
0    /var/run
16K  /var/spool
987M /var/swap
28K  /var/tmp
16K  /var/www
```

A more interactive way to explore your disk usage is by installing *ncdu* (ncurses disk usage):

```
bone$ sudo apt install ncdu
bone$ ncdu /
```

After a moment, you'll see the following:

```
ncdu 1.15.1 ~ Use the arrow keys to navigate, press ? for help
--- / -----
.   1.9 GiB [#####] /var
.   1.5 GiB [##### ] /usr
835.0 MiB [####  ] /home
663.5 MiB [###   ] /opt
590.9 MiB [###   ] /lib
159.0 MiB [      ] /boot
.   22.8 MiB [      ] /etc
.   12.5 MiB [      ] /sbin
.   11.1 MiB [      ] /bin
.    1.4 MiB [      ] /run
.   40.0 KiB [      ] /tmp
!   16.0 KiB [      ] /lost+found
.    8.0 KiB [      ] /mnt
e    4.0 KiB [      ] /srv
!    4.0 KiB [      ] /root
e    4.0 KiB [      ] /media
.    4.0 KiB [      ] ID.txt
.    0.0  B [      ] /sys
.    0.0  B [      ] /proc
.    0.0  B [      ] /dev

Total disk usage:   5.6 GiB  Apparent size:   5.5 GiB  Items: 206148
```

`ncdu` is a character-based graphics interface to `du`. You can now use your arrow keys to navigate the file structure to discover where the big unused files are. Press `?` for help.

**Warning:** Be careful not to press the `d` key, because it's used to delete a file or directory.

## 5.19 Using C to Interact with the Physical World

### 5.19.1 Problem

You want to use C on the Bone to talk to the world.

### 5.19.2 Solution

The C solution isn't as simple as the JavaScript or Python solution, but it does work and is much faster. The approach is the same, write to the `/sys/class/gpio` files.

Listing 5.3: Use C to blink an LED (blinkLED.c)

```
1 ///////////////////////////////////////////////////////////////////
2 //      blinkLED.c
3 //      Blinks the P9_14 pin
4 //      Wiring:
5 //      Setup:
6 //      See:
7 ///////////////////////////////////////////////////////////////////
8 #include <stdio.h>
9 #include <string.h>
10 #include <unistd.h>
11 #define MAXSTR 100
12 // Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1, line 18.
   ↪maps to 50
```

(continues on next page)

(continued from previous page)

```

13 int main() {
14     FILE *fp;
15     char pin[] = "50";
16     char GPIOPATH[] = "/sys/class/gpio";
17     char path[MAXSTR] = "";
18
19     // Make sure pin is exported
20     snprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", pin);
21     if (!access(path, F_OK) == 0) {
22         snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
23         fp = fopen(path, "w");
24         fprintf(fp, "%s", pin);
25         fclose(fp);
26     }
27
28     // Make it an output pin
29     snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", pin, "/direction");
30     fp = fopen(path, "w");
31     fprintf(fp, "out");
32     fclose(fp);
33
34     // Blink every .25 sec
35     int state = 0;
36     snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", pin, "/value");
37     fp = fopen(path, "w");
38     while (1) {
39         fseek(fp, 0, SEEK_SET);
40         if (state) {
41             fprintf(fp, "1");
42         } else {
43             fprintf(fp, "0");
44         }
45         state = ~state;
46         usleep(250000); // sleep time in microseconds
47     }
48 }

```

blinkLED.c

Here, as with JavaScript and Python, the gpio pins are referred to by the Linux gpio number. [Mapping from header pin to internal GPIO number](#) shows how the P8 and P9 Headers numbers map to the gpio number. For this example P9\_14 is used, which the table shows in gpio 50.

Compile and run the code:

```

bone$ gcc -o blinkLED blinkLED.c
bone$ ./blinkLED
^C

```

Hit ^C to stop the blinking.

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BTN	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	GPIO_22	19	20	GPIO_63
GPIO_3	21	22	GPIO_2	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	GPIO_15	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_14	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GND_A_ADC	GPIO_9	33	34	GPIO_81
AIN6	35	36	AIN5	GPIO_8	35	36	GPIO_80
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 5.15: Mapping from header pin to internal GPIO number

## Chapter 6

# Internet of Things

You can easily connect BeagleBone Black to the Internet via a wire ([Establishing an Ethernet-Based Internet Connection](#)), wirelessly ([Establishing a WiFi-Based Internet Connection](#)), or through the USB to a host and then to the Internet ([Sharing the Host's Internet Connection over USB](#)). Either way, it opens up a world of possibilities for the “Internet of Things” (IoT).

Now that you're online, this chapter offers various things to do with your connection.

### 6.1 Accessing Your Host Computer's Files on the Bone

#### 6.1.1 Problem

You want to access a file on a Linux host computer that's attached to the Bone.

#### 6.1.2 Solution

If you are running Linux on a host computer attached to BeagleBone Black, it's not hard to mount the Bone's files on the host or the host's files on the Bone by using *sshfs*. Suppose that you want to access files on the host from the Bone. First, install *sshfs*:

```
bone$ sudo apt install sshfs
```

Now, mount the files to an empty directory (substitute your username on the host computer for *username* and the IP address of the host for *192.168.7.1*):

```
bone$ mkdir host
bone$ sshfs username@$192.168.7.1:. host
bone$ cd host
bone$ ls
```

The *ls* command will now list the files in your home directory on your host computer. You can edit them as if they were local to the Bone. You can access all the files by substituting *:/* for the *./* following the IP address.

You can go the other way, too. Suppose that you are on your Linux host computer and want to access files on your Bone. Install *sshfs*:

```
host$ sudo apt install sshfs
```

and then access:

```
host$ mkdir /mnt/bone
host$ sshfs debian@$192.168.7.2:/ /mnt/bone
host$ cd /mnt/bone
host$ ls
```

Here, we are accessing the files on the Bone as *debian*. We've mounted the entire file system, starting with `/`, so you can access any file. Of course, with great power comes great responsibility, so be careful.

The `sshfs` command gives you easy access from one computer to another. When you are done, you can unmount the files by using the following commands:

```
host$ umount /mnt/bone
bone$ umount home
```

## 6.2 Serving Web Pages from the Bone

### 6.2.1 Problem

You want to use BeagleBone Black as a web server.

### 6.2.2 Solution

BeagleBone Black already has the *nginx* web server running.

When you point your browser to `192.168.7.2`, you are using the *nginx* web server. The web pages are served from `/var/www/html/`. Add the HTML in [A sample web page \(test.html\)](#) to a file called `/var/www/html/test.html`, and then point your browser to `192.168.7.2/test.html`.

Listing 6.1: A sample web page (test.html)

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h1>My First Heading</h1>
6
7 <p>My first paragraph.</p>
8
9 </body>
10 </html>
```

test.html

You will see the web page shown in [test.html as served by nginx](#).

## 6.3 Interacting with the Bone via a Web Browser

### 6.3.1 Problem

BeagleBone Black is interacting with the physical world nicely and you want to display that information on a web browser.

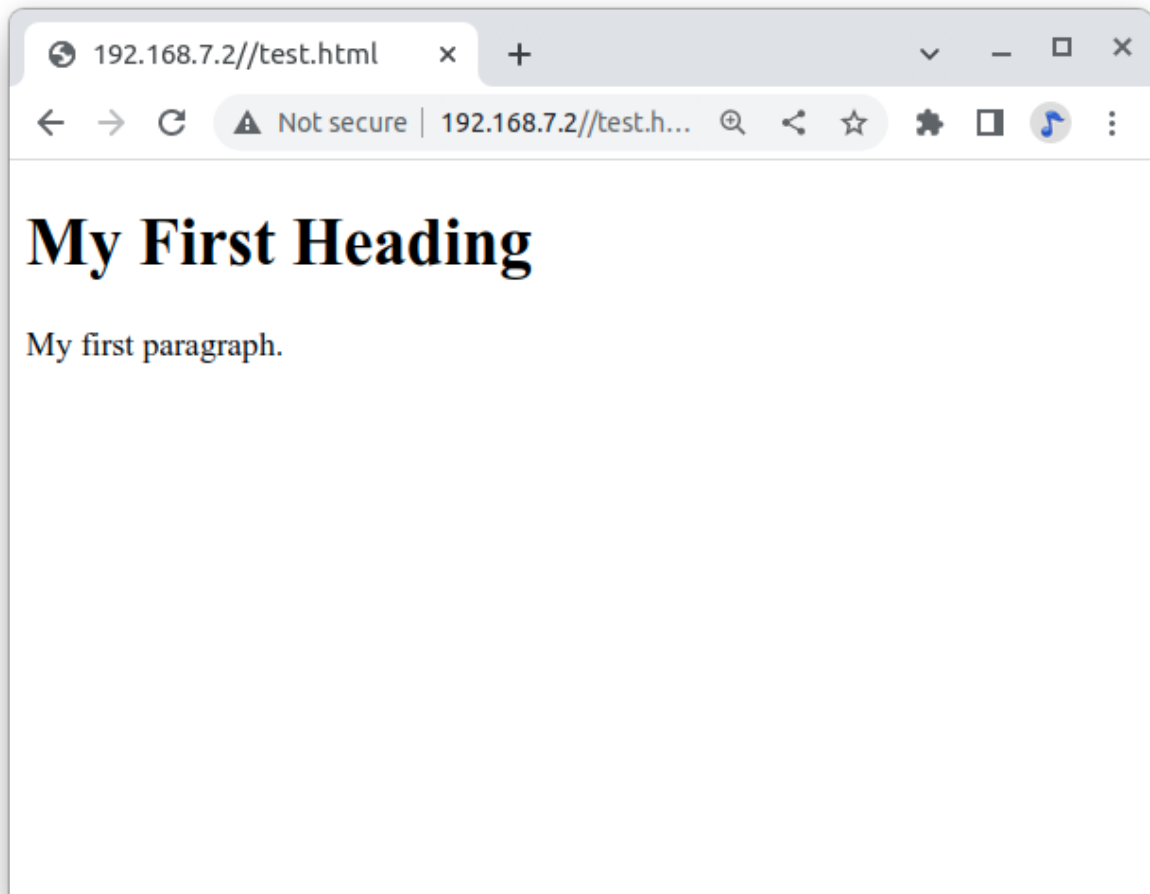


Fig. 6.1: test.html as served by nginx



### 6.3.2 Solution

Flask is a Python web framework built with a small core and easy-to-extend philosophy. [Serving Web Pages from the Bone](#) shows how to use nginx, the web server that's already running. This recipe shows how easy it is to build your own server. This is an adaptation of [Python WebServer With Flask and Raspberry Pi](#).

First, install flask:

```
bone$ sudo apt update
bone$ sudo apt install python3-flask
```

All the code in is the Cookbook repo:

```
bone$ git clone https://git.beagleboard.org/beagleboard/beaglebone-cookbook-
→code
bone$ cd beaglebone-cookbook-code/06iot/flask
```

## 6.4 First Flask - hello, world

Our first example is *helloWorld.py*

Listing 6.2: Python code for flask hello world (helloWorld.py)

```
1 #!/usr/bin/env python
2 # From: https://towardsdatascience.com/python-webserver-with-flask-and-
→raspberrypi-398423cc6f5d
3
4 from flask import Flask
5 app = Flask(__name__)
6 @app.route('/')
7 def index():
8     return 'hello, world'
9 if __name__ == '__main__':
10     app.run(debug=True, port=8080, host='0.0.0.0')
```

helloWorld.py

1. The first line loads the Flask module into your Python script.
2. The second line creates a Flask object called `app`.
3. The third line is where the action is, it says to run the `index()` function when someone accesses the root URL (`/`) of the server. In this case, send the text “hello, world” to the client’s web browser via `return`.
4. The last line says to “listen” on port 8080, reporting any errors.

Now on your host computer, browse to `192.168.7.2:8080 flask` and you should see.

## 6.5 Adding a template

Let’s improve our “hello, world” application, by using an HTML template and a CSS file for styling our page. Note: these have been created for you in the “templates” sub-folder. So, we will create a file named *index1.html*, that has been saved in */templates*.

Here’s what’s in *templates/index1.html*:

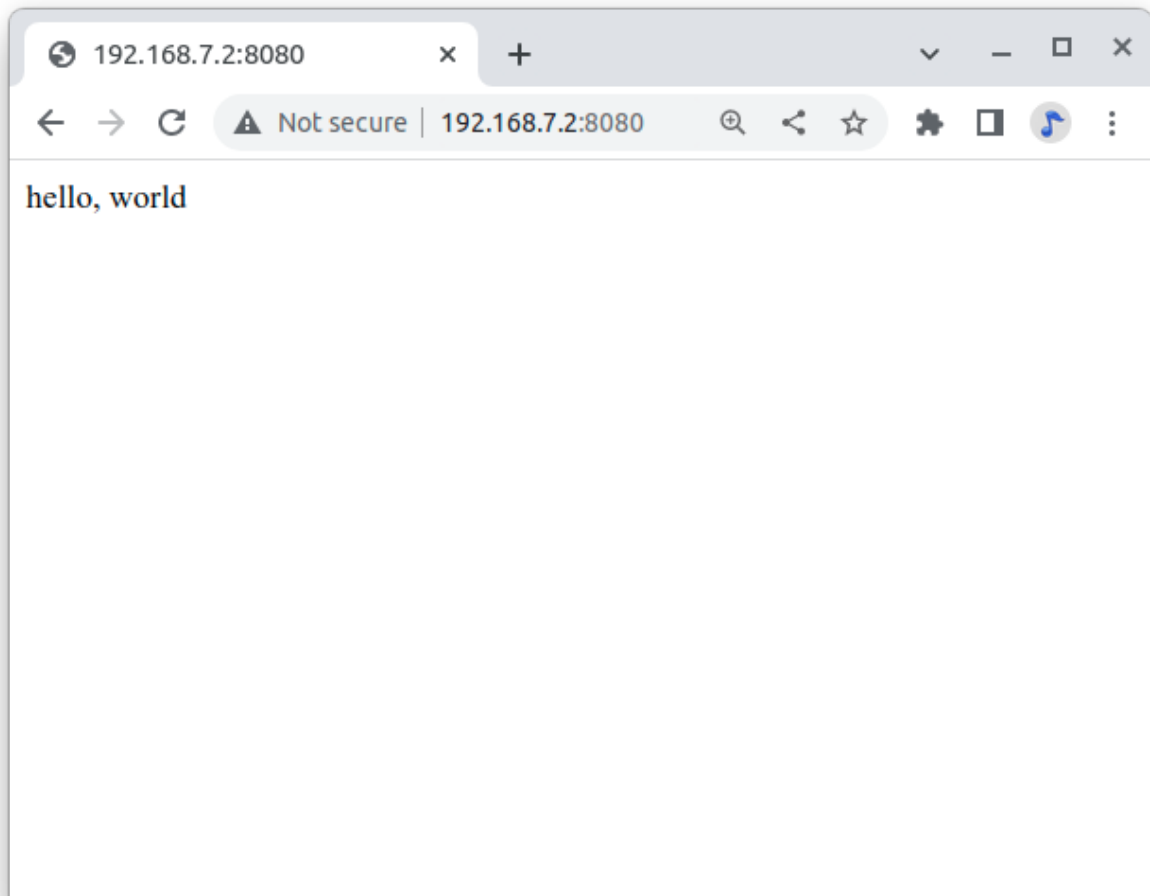


Fig. 6.2: Test page served by our custom flask server

Listing 6.3: index1.html

```
1 <!DOCTYPE html>
2   <head>
3     <title>{{ title }}</title>
4   </head>
5   <body>
6     <h1>Hello, World!</h1>
7     <h2>The date and time on the server is: {{ time }}</h2>
8   </body>
9 </html>
```

index1.html

Note: a style sheet (style.css) is also included. This will be populated later.

Observe that anything in double curly braces within the HTML template is interpreted as a variable that would be passed to it from the Python script via the `render_template` function. Now, let's create a new Python script. We will name it `app1.py`:

Listing 6.4: app1.py

```
1 #!/usr/bin/env python
2 # From: https://towardsdatascience.com/python-webserver-with-flask-and-
3     ↳raspberrypi-398423cc6f5d
4
5 '''
6 Code created by Matt Richardson
7 for details, visit: http://mattrichardson.com/Raspberry-Pi-Flask/inde...
8 '''
9 from flask import Flask, render_template
10 import datetime
11 app = Flask(__name__)
12 @app.route("/")
13 def hello():
14     now = datetime.datetime.now()
15     timeString = now.strftime("%Y-%m-%d %H:%M")
16     templateData = {
17         'title' : 'HELLO!',
18         'time': timeString
19     }
20     return render_template('index1.html', **templateData)
21 if __name__ == "__main__":
22     app.run(host='0.0.0.0', port=8080, debug=True)
```

app1.py

Note that we create a formatted string (“timeString”) using the date and time from the “now” object, that has the current time stored on it.

Next important thing on the above code, is that we created a dictionary of variables (a set of keys, such as the title that is associated with values, such as HELLO!) to pass into the template. On “return”, we will return the `index1.html` template to the web browser using the variables in the `templateData` dictionary.

Execute the Python script:

```
bone$ .\app.py
```

Open any web browser and browse to 192.168.7.2:8080. You should see:

Note that the page's content changes dynamically any time that you refresh it with the actual variable data passed by Python script. In our case, “title” is a fixed value, but “time” changes every minute.

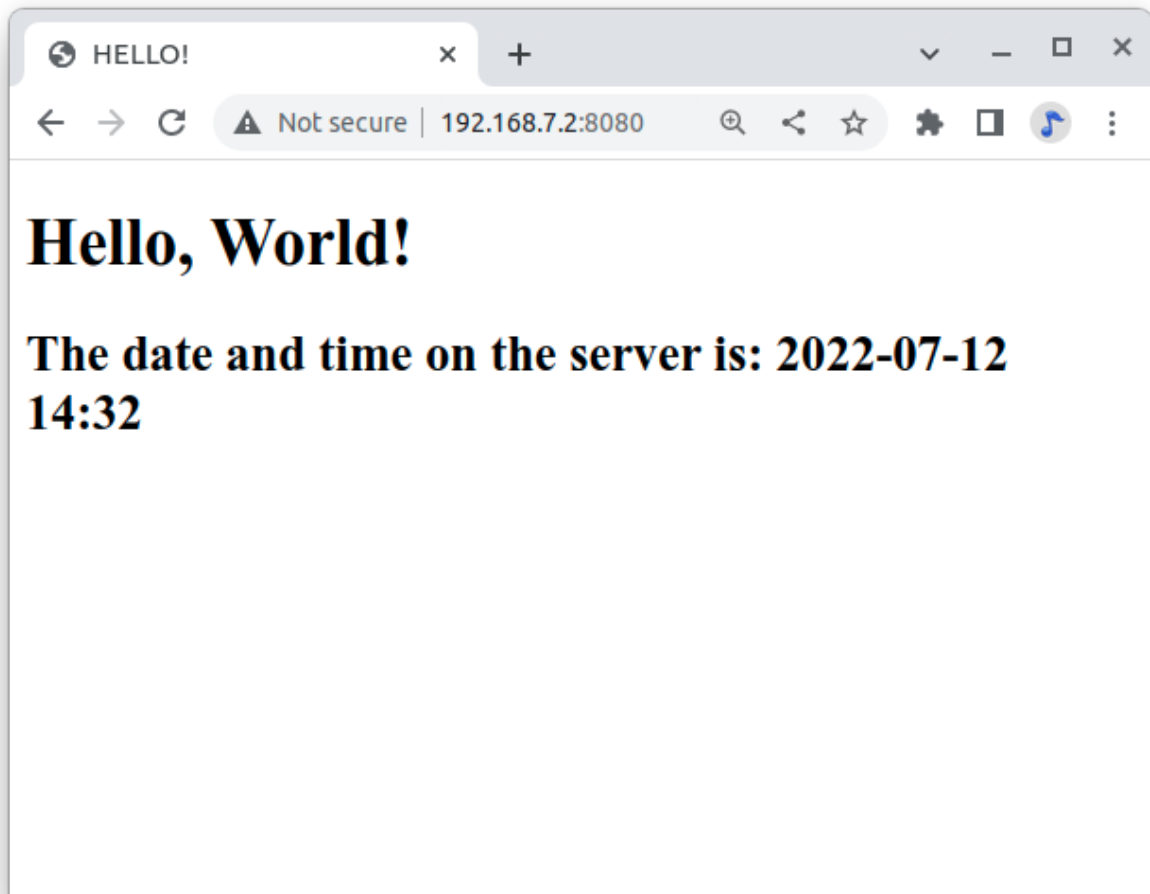


Fig. 6.3: Test page served by app1.py

## 6.6 Displaying GPIO Status in a Web Browser - reading a button

### 6.6.1 Problem

You want a web page to display the status of a GPIO pin.

### 6.6.2 Solution

This solution builds on the Flask-based web server solution in [Interacting with the Bone via a Web Browser](#).

To make this recipe, you will need:

- Breadboard and jumper wires.
- Pushbutton switch.

Wire your pushbutton as shown in [Diagram for wiring a pushbutton and magnetic reed switch input](#). Wire a button to P9\_11 and have the web page display the value of the button.

Let's use a new Python script named `app2.py`.

Listing 6.5: A simple Flask-based web server to read a GPIO (`app2.py`)

```
1  #!/usr/bin/env python
2  # From: https://towardsdatascience.com/python-webserver-with-flask-and-
   ↳raspberrypi-398423cc6f5d
3  import os
4  from flask import Flask, render_template
5  app = Flask(__name__)
6
7  pin = '30' # P9_11 is gpio 30
8  GPIOPATH="/sys/class/gpio"
9  buttonSts = 0
10
11 # Make sure pin is exported
12 if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
13     f = open(GPIOPATH+"/export", "w")
14     f.write(pin)
15     f.close()
16
17 # Make it an input pin
18 f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
19 f.write("in")
20 f.close()
21
22 @app.route("/")
23 def index():
24     # Read Button Status
25     f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
26     buttonSts = f.read()[:-1]
27     f.close()
28
29     # buttonSts = GPIO.input(button)
30     templateData = {
31         'title' : 'GPIO input Status!',
32         'button' : buttonSts,
33     }
34     return render_template('index2.html', **templateData)
35 if __name__ == "__main__":
36     app.run(host='0.0.0.0', port=8080, debug=True)
```

`app2.py`

What we are doing is defining the button on *P9\_11* as input, reading its value and storing it in *buttonSts*. Inside the function *index()*, we will pass that value to our web page through “button” that is part of our variable dictionary: *templateData*.

Let’s also see the new *index2.html* to show the GPIO status:

Listing 6.6: A simple Flask-based web server to read a GPIO (*index2.html*)

```

1 <!DOCTYPE html>
2 <head>
3   <title>{{ title }}</title>
4   <link rel="stylesheet" href='../static/style.css' />
5 </head>
6 <body>
7   <h1>{{ title }}</h1>
8   <h2>Button pressed:  {{ button }}</h2>
9 </body>
10</html>

```

*index2.html*

Now, run the following command:

```
bone$ ./app2.py
```

Point your browser to *http://192.168.7.2:8080*, and the page will look like [Status of a GPIO pin on a web page](#).

Currently, the *0* shows that the button isn’t pressed. Try refreshing the page while pushing the button, and you will see *1* displayed.

It’s not hard to assemble your own HTML with the GPIO data. It’s an easy extension to write a program to display the status of all the GPIO pins.

## 6.7 Controlling GPIOs

### 6.7.1 Problem

You want to control an LED attached to a GPIO pin.

### 6.7.2 Solution

Now that we know how to “read” GPIO Status, let’s change them. What we will do will control the LED via the web page. We have an LED connected to *P9\_14*. Controlling remotely we will change its status from LOW to HIGH and vice-versa.

Create a new Python script and name it *app3.py*.

Listing 6.7: A simple Flask-based web server to read a GPIO (*app3.py*)

```

1 #!/usr/bin/env python
2 # From: https://towardsdatascience.com/python-webserver-with-flask-and-
   ↳raspberrypi-398423cc6f5d
3 # import Adafruit_BBIO.GPIO as GPIO
4 import os
5 from flask import Flask, render_template, request
6 app = Flask(__name__)
7 #define LED GPIO
8 ledRed = "P9_14"
9 pin = '50' # P9_14 is gpio 50

```

(continues on next page)

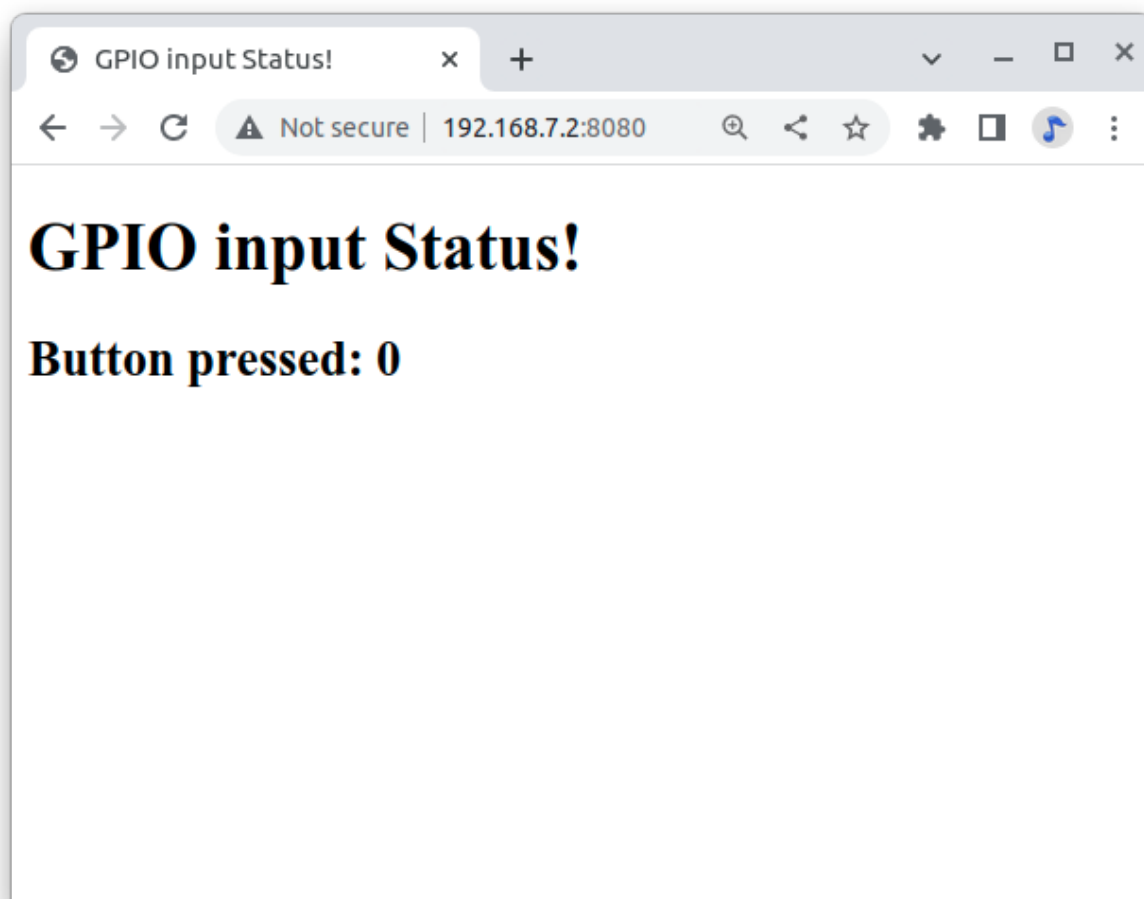


Fig. 6.4: Status of a GPIO pin on a web page

(continued from previous page)

```

10 GPIOPATH="/sys/class/gpio"
11
12 #initialize GPIO status variable
13 ledRedSts = 0
14 # Make sure pin is exported
15 if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
16     f = open(GPIOPATH+"/export", "w")
17     f.write(pin)
18     f.close()
19 # Define led pin as output
20 f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
21 f.write("out")
22 f.close()
23 # turn led OFF
24 f = open(GPIOPATH+"/gpio"+pin+"/value", "w")
25 f.write("0")
26 f.close()
27
28 @app.route("/")
29 def index():
30     # Read Sensors Status
31     f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
32     ledRedSts = f.read()
33     f.close()
34     templateData = {
35         'title' : 'GPIO output Status!',
36         'ledRed' : ledRedSts,
37     }
38     return render_template('index3.html', **templateData)
39
40 @app.route("/<deviceName>/<action>")
41 def action(deviceName, action):
42     if deviceName == 'ledRed':
43         actuator = ledRed
44         f = open(GPIOPATH+"/gpio"+pin+"/value", "w")
45         if action == "on":
46             f.write("1")
47         if action == "off":
48             f.write("0")
49         f.close()
50
51         f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
52         ledRedSts = f.read()
53         f.close()
54
55         templateData = {
56             'ledRed' : ledRedSts,
57         }
58         return render_template('index3.html', **templateData)
59 if __name__ == "__main__":
60     app.run(host='0.0.0.0', port=8080, debug=True)

```

app3.py

What we have new on above code is the new "route":

```
@app.route("/<deviceName>/<action>")
```

From the webpage, calls will be generated with the format:

<http://192.168.7.2:8081/ledRed/on>

or



<http://192.168.7.2:8081/ledRed/off>

For the above example, *ledRed* is the “deviceName” and *on* or *off* are examples of possible “action”. Those routes will be identified and properly “worked”. The main steps are:

- Convert the string “ledRED”, for example, on its equivalent GPIO pin. The integer variable *ledRed* is equivalent to P9\_14. We store this value on variable “actuator”
- For each actuator, we will analyze the “action”, or “command” and act properly. If “action = on” for example, we must use the command: `f.write("1")`
- Update the status of each actuator
- Return the data to `index.html`

Let’s now create an *index.html* to show the GPIO status of each actuator and more importantly, create “buttons” to send the commands:

Listing 6.8: A simple Flask-based web server to write a GPIO (`index3.html`)

```

1 <!DOCTYPE html>
2 <head>
3   <title>GPIO Control</title>
4   <link rel="stylesheet" href='../static/style.css' />
5 </head>
6 <body>
7     <h2>Actuators</h2>
8     <h3> Status </h3>
9       RED LED ==>  {{ ledRed  }}
10    <br>
11    <h3> Commands </h3>
12      RED LED Ctrl ==>
13      <a href="/ledRed/on" class="button">TURN ON</a>
14      <a href="/ledRed/off" class="button">TURN OFF</a>
15
16 </body>
</html>

```

`index3.html`

```
bone$ ./app3.py
```

Point your browser as before and you will see:

Status of a GPIO pin on a web page

Try clicking the “TURN ON” and “TURN OFF” buttons and your LED will respond.

`app4.py` and `app5.py` combine the previous apps. Try them out.

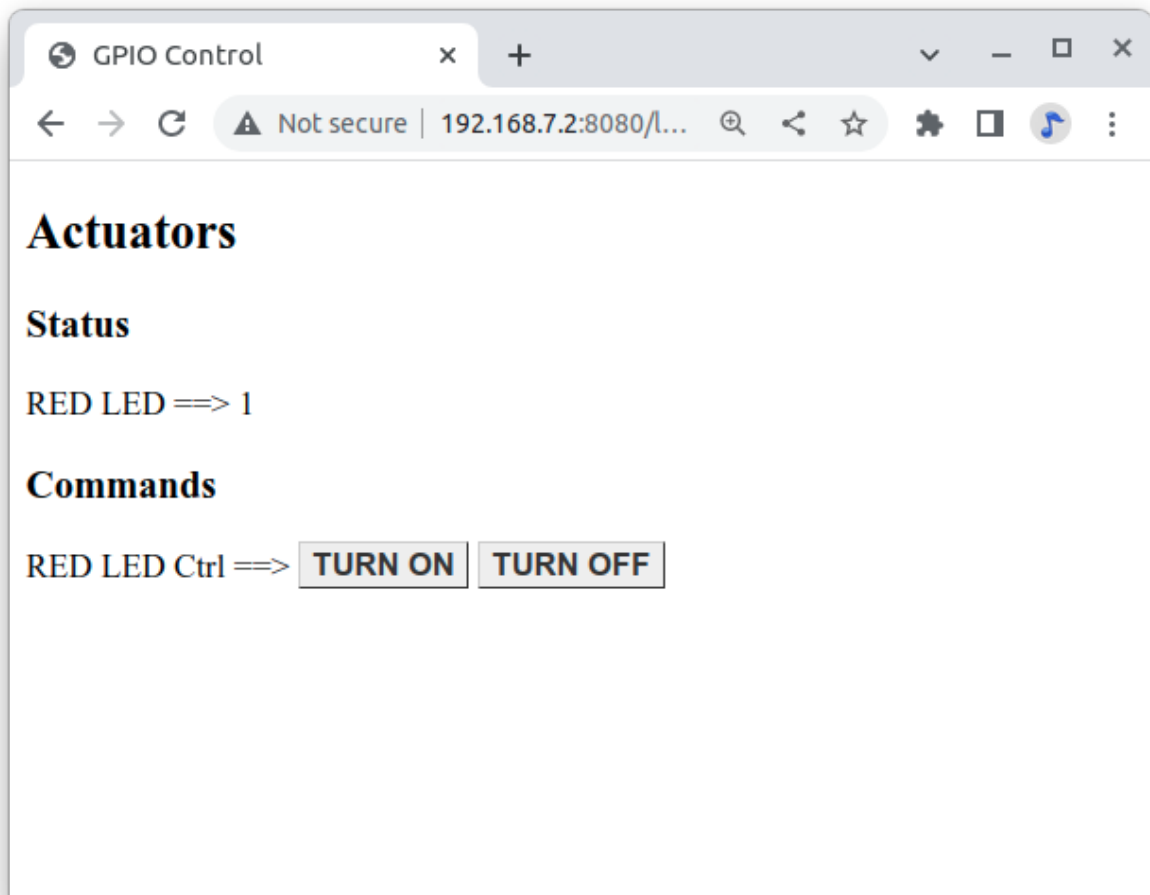
```
app4.py app5.py
```

## 6.8 Plotting Data

### 6.8.1 Problem

You have live, continuous, data coming into your Bone via one of the Analog Ins, and you want to plot it.

### 6.8.2 Solution



## Analog in - Continuous

(This is based on information at: [http://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational\\_Components/Kernel/Kernel\\_Drivers/ADC.html#Continuous%20Mode](http://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/Kernel/Kernel_Drivers/ADC.html#Continuous%20Mode))

Reading a continuous analog signal requires some set up. First go to the iio devices directory.

```
bone$ cd /sys/bus/iio/devices/iio:device0
bone$ ls -F
buffer/  in_voltage0_raw  in_voltage2_raw  in_voltage4_raw  in_voltage6_raw  ↵
↪name    power/           subsystem@
dev      in_voltage1_raw  in_voltage3_raw  in_voltage5_raw  in_voltage7_raw  ↵
↪of_node@ scan_elements/  uevent
```

Here you see the files used to read the one shot values. Look in `scan_elements` to see how to enable continuous input.

```
bone$ ls scan_elements
in_voltage0_en      in_voltage1_index  in_voltage2_type   in_voltage4_en      ↵
↪in_voltage5_index  in_voltage6_type
in_voltage0_index   in_voltage1_type   in_voltage3_en     in_voltage4_index   ↵
↪in_voltage5_type   in_voltage7_en
in_voltage0_type    in_voltage2_en     in_voltage3_index  in_voltage4_type    ↵
↪in_voltage6_en     in_voltage7_index
in_voltage1_en      in_voltage2_index  in_voltage3_type   in_voltage5_en      ↵
↪in_voltage6_index  in_voltage7_type
```

Here you see three values for each analog input, `_en (enable)`, `_index` (index of this channel in the buffer's chunks) and `_type` (how the ADC stores its data). (See the link above for details.) Let's use the input at `P9.40` which is `AIN1`. To enable this input:

```
bone$ echo 1 > scan_elements/in_voltage1_en
```

Next set the buffer size.

```
bone$ ls buffer
data_available  enable  length  watermark
```

Let's use a 512 sample buffer. You might need to experiment with this.

```
bone$ echo 512 > buffer/length
```

Then start it running.

```
bone$ echo 1 > buffer/enable
```

Now, just `read` from `*/dev/iio:device0*`.

An example Python program that does the above and reads and plots the buffer is **`analogInContinuous.py`**.

Listing 6.9: Code to read and plot a continuous analog input(`analogInContinuous.py`)

```
1  #!/usr/bin/python
2  #////////////////////////////////////
3  #      analogInContinuous.py
4  #      Read analog data via IIO continuous mode and plots it.
5  #////////////////////////////////////
6  # From: https://stackoverflow.com/questions/20295646/python-ascii-plots-in-
   ↪terminal
7  # https://github.com/dkogan/gnuplotlib
8  # https://github.com/dkogan/gnuplotlib/blob/master/guide/guide.org
```

(continues on next page)

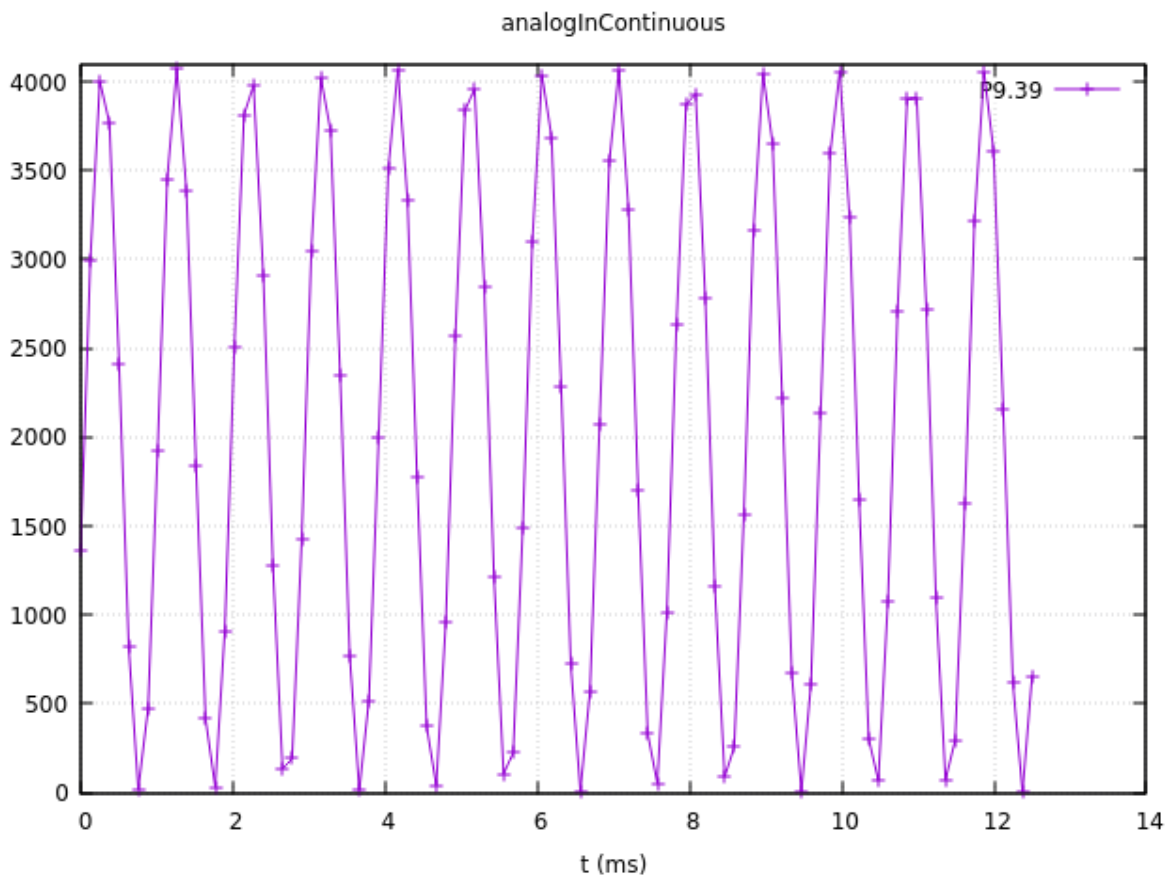


Fig. 6.5: 1KHz sine wave sampled at 8KHz

(continued from previous page)

```
9 # sudo apt install gnuplot (10 minute to install)
10 # sudo apt install libatlas-base-dev
11 # pip3 install gnuplotlib
12 # This uses X11, so when connecting to the bone from the host use: ssh -X.
   ↳bone
13
14 # See https://elinux.org/index.php?title=EBC_Exercise_10a_Analog_In#Analog_
   ↳in_-_Continuous.2C_Change_the_sample_rate
15 # for instructions on changing the sampling rate. Can go up to 200KHz.
16
17 fd = open(IIODEV, "r")
18 import numpy as np
19 import gnuplotlib as gp
20 import time
21 # import struct
22
23 IIOPATH='/sys/bus/iio/devices/iio:device0'
24 IIODEV='/dev/iio:device0'
25 LEN = 100
26 SAMPLERATE=8000
27 AIN='2'
28
29 # Setup IIO for Continous reading
30 # Enable AIN
31 try:
32     file1 = open(IIOPATH+'/scan_elements/in_voltage'+AIN+'_en', 'w')
33     file1.write('1')
34     file1.close()
35 except: # carry on if it's already enabled
36     pass
37 # Set buffer length
38 file1 = open(IIOPATH+'/buffer/length', 'w')
39 file1.write(str(2*LEN)) # I think LEN is in 16-bit values, but here we
   ↳pass bytes
40 file1.close()
41 # Enable continuous
42 file1 = open(IIOPATH+'/buffer/enable', 'w')
43 file1.write('1')
44 file1.close()
45
46 x = np.linspace(0, 1000*LEN/SAMPLERATE, LEN)
47 # Do a dummy plot to give time of the fonts to load.
48 gp.plot(x, x)
49 print("Waiting for fonts to load")
50 time.sleep(10)
51
52 print('Hit ^C to stop')
53
54 fd = open(IIODEV, "r")
55
56 try:
57     while True:
58         y = np.fromfile(fd, dtype='uint16', count=LEN)*1.8/4096
59         # print(y)
60         gp.plot(x, y,
61                xlabel = 't (ms)',
62                ylabel = 'volts',
63                _yrange = [0, 2],
64                title = 'analogInContinuous',
65                legend = np.array( ("P9.39", ), ),
66                # ascii=1,
```

(continues on next page)

(continued from previous page)

```

67         # terminal="xterm",
68         # legend = np.array( ("P9.40", "P9.38"), ),
69         # _with = 'lines'
70     )
71
72 except KeyboardInterrupt:
73     print("Turning off input.")
74     # Disable continuous
75     file1 = open(IIOPATH+'/buffer/enable', 'w')
76     file1.write('0')
77     file1.close()
78
79     file1 = open(IIOPATH+'/scan_elements/in_voltage'+AIN+'_en', 'w')
80     file1.write('0')
81     file1.close()
82
83 # // Bone | Pocket | AIN
84 # // ---- | ----- | ---
85 # // P9_39 | P1_19 | 0
86 # // P9_40 | P1_21 | 1
87 # // P9_37 | P1_23 | 2
88 # // P9_38 | P1_25 | 3
89 # // P9_33 | P1_27 | 4
90 # // P9_36 | P2_35 | 5
91 # // P9_35 | P1_02 | 6

```

analogInContinuous.py

Be sure to read the installation instructions in the comments. Also note this uses X windows and you need to `ssh -X 192.168.7.2` for X to know where the display is.

Run it:

```

host$ ssh -X bone
bone$ cd beaglebone-cookbook-code/06iot
bone$ ./analogInContinuous.py
Hit ^C to stop

```

**Todo:** verify this works. fonts are taking too long to load

*1KHz sine wave sampled at 8KHz* is the output of a 1KHz sine wave.

It's a good idea to disable the buffer when done.

```
bone$ echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable
```

### Analog in - Continuous, Change the sample rate

The built in ADCs sample at 8k samples/second by default. They can run as fast as 200k samples/second by editing a device tree.

```
bone$ cd /opt/source/bb.org-overlays
bone$ make
```

This will take a while the first time as it compiles all the device trees.

```
bone$ vi src/arm/src/arm/BB-ADC-00A0.dts
```

Around line 57 you'll see

```
Line   Code
57     // For each step, number of adc clock cycles to wait between setting_
      ↳up muxes and sampling.
58     // range: 0 .. 262143
59     // optional, default is 152 (XXX but why?!)
60     ti,chan-step-opedelay = <152 152 152 152 152 152 152 152>;
61     //`
62     // XXX is there any purpose to set this nonzero other than to fine-
      ↳tune the sample rate?
63
64
65     // For each step, how many times it should sample to average.
66     // range: 1 .. 16, must be power of two (i.e. 1, 2, 4, 8, or 16)
67     // optional, default is 16
68     ti,chan-step-avg = <16 16 16 16 16 16 16 16>;
```

The comments give lots of details on how to adjust the device tree to change the sample rate. Line 68 says for every sample returned, average 16 values. This will give you a cleaner signal, but if you want to go fast, change the 16's to 1's. Line 60 says to delay 152 cycles between each sample. Set this to 0 to get as fast as possible.

```
ti,chan-step-avg = <1 1 1 1 1 1 1 1>;
ti,chan-step-opedelay = <0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00>;
```

Now compile it.

```
bone$ make
      DTC      src/arm/BB-ADC-00A0.dtbo
gcc -o config-pin ./tools/pmunts_muntsos/config-pin.c
```

It knows to only recompile the file you just edited. Now install and reboot.

```
bone$ sudo make install
...
'src/arm/AM335X-PRU-UIO-00A0.dtbo' -> '/lib/firmware/AM335X-PRU-UIO-00A0.dtbo
↳'
'src/arm/BB-ADC-00A0.dtbo' -> '/lib/firmware/BB-ADC-00A0.dtbo'
'src/arm/BB-BBBMINI-00A0.dtbo' -> '/lib/firmware/BB-BBBMINI-00A0.dtbo'
...
bone$ reboot
```

A number of files get installed, including the ADC file. Now try rerunning.

```
bone$ cd beaglebone-cookbook-code/06iot
bone$ ./analogInContinuous.py
Hit ^C to stop
```

Here's the output of a 10KHz triangle wave.

---

**Todo:** Is this true: (The plot is wrong, but eLinux won't let me fix it.)

---

It's still a good idea to disable the buffer when done.

```
bone$ echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable
```

## 6.9 Sending an Email

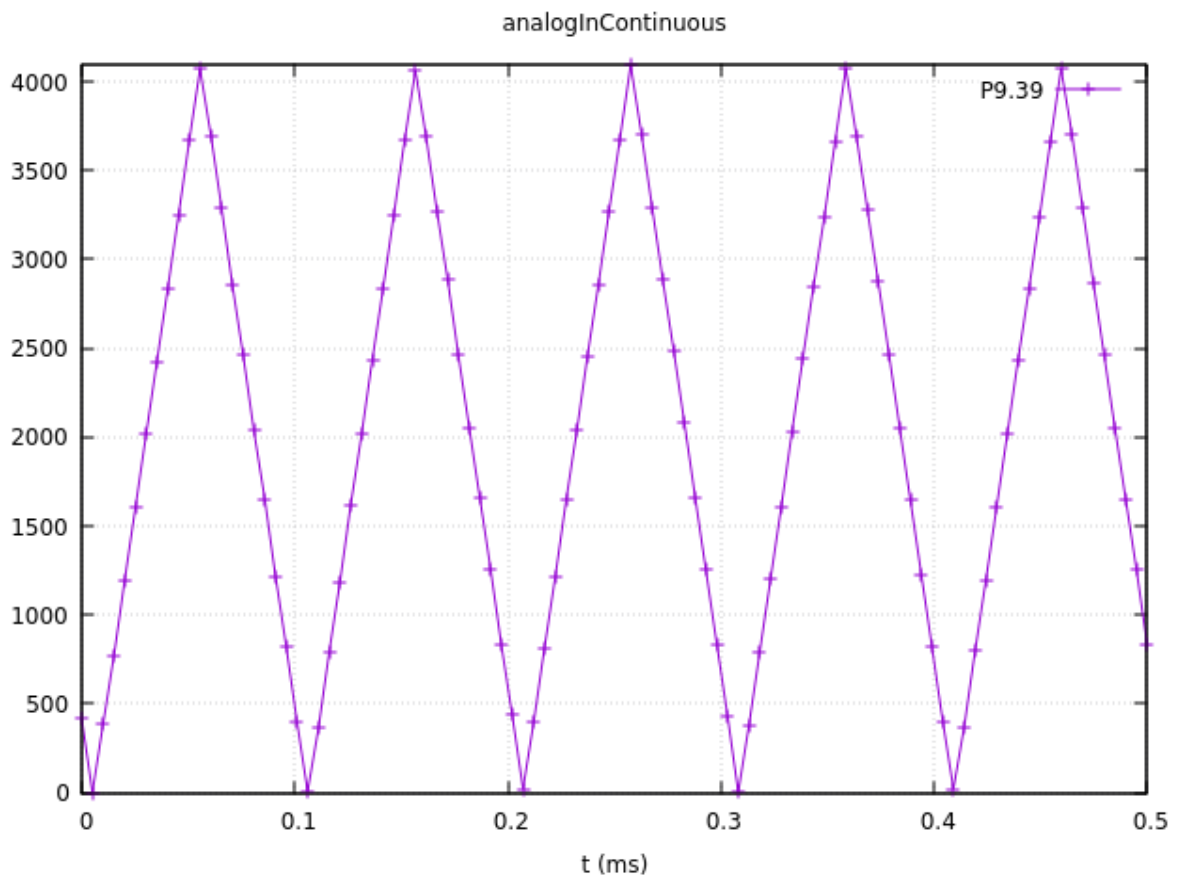


Fig. 6.6: 10KHz triangle wave sampled at 200KHz



### 6.9.1 Problem

You want to send an email via Gmail from the Bone.

### 6.9.2 Solution

This example came from <https://realpython.com/python-send-email/>. First, you need to set up a Gmail account, if you don't already have one. Then add the code in *Sending email using nodemailer (emailTest.py)* to a file named `emailTest.py`. Substitute your own Gmail username. For the password:

- Go to: <https://myaccount.google.com/security>
- Go to *2-Step Verification* and at the bottom, select App password.
- Generate your own 16 char password and copy it into `emailTest.py`.
- Be sure to delete password when done <https://myaccount.google.com/apppasswords>.

Listing 6.10: Sending email using nodemailer (emailTest.py)

```
1  #!/usr/bin/env python
2  # From: https://realpython.com/python-send-email/
3  import smtplib, ssl
4
5  port = 587 # For starttls
6  smtp_server = "smtp.gmail.com"
7  sender_email = "from_account@gmail.com"
8  receiver_email = "to_account@gmail.com"
9  # Go to: https://myaccount.google.com/security
10 # Select App password
11 # Generate your own 16 char password, copy here
12 # Delete password when done
13 password = "cftqhcejjdjfdwjh"
14 message = """\
15 Subject: Testing email
16
17 This message is sent from Python.
18
19 """
20 context = ssl.create_default_context()
21 with smtplib.SMTP(smtp_server, port) as server:
22     server.starttls(context=context)
23     server.login(sender_email, password)
24     server.sendmail(sender_email, receiver_email, message)
```

`emailTest.py`

Then run the script to send the email:

```
bone$ chmod *x emailTest.py
bone$ ./emailTest.py
```

**Warning:** This solution requires your Gmail password to be in plain text in a file, which is a security problem. Make sure you know who has access to your Bone. Also, if you remove the microSD card, make sure you know who has access to it. Anyone with your microSD card can read your Gmail password.

Be careful about putting this into a loop. Gmail presently limits you to 500 emails per day and 10 MB per message.

See <https://realpython.com/python-send-email/> for an example that sends an attached file.

## 6.10 Sending an SMS Message

**Todo:** My twilio account is suspended, using [yoder@rose-hulman.edu](mailto:yoder@rose-hulman.edu).

### 6.10.1 Problem

You want to send a text message from BeagleBone Black.

### 6.10.2 Solution

There are a number of SMS services out there. This recipe uses Twilio because you can use it for free, but you will need to [verify the number](#) to which you are texting. First, go to [Twilio's home page](#) and set up an account. Note your account SID and authorization token. If you are using the free version, be sure to [verify your numbers](#).

Next, install Trilio by using the following command for python:

```
bone$ sudo apt install python-pip
bone$ sudo pip install twilio
```

or for Javascript:

```
bone$ npm install -g twilio
```

Finally, add the code in [Sending SMS messages using Twilio \(twilioTest.py\)](#) to a file named `twilioTest.py` and run it. Your text will be sent.

### Python

Listing 6.11: Sending SMS messages using Twilio (twilioTest.py)

```
1  #!/usr/bin/env python
2  # Download the helper library from https://www.twilio.com/docs/python/install
3  import os
4  from twilio.rest import Client
5
6
7  # Find your Account SID and Auth Token at twilio.com/console
8  # and set the environment variables. See http://twil.io/secure
9  account_sid = os.environ['TWILIO_ACCOUNT_SID']
10 auth_token = os.environ['TWILIO_AUTH_TOKEN']
11 client = Client(account_sid, auth_token)
12
13 message = client.messages \
14     .create(
15         body="Join Earth's mightiest heroes. Like Kevin Bacon.",
16         from_='+18122333219',
17         to='+18122333219'
18     )
19
20 print(message.sid)
```

twilioTest.py

## JavaScript

Listing 6.12: Sending SMS messages using Twilio (twilio-test.js)

```

1  #!/usr/bin/env node
2  // From: http://twilio.github.io/twilio-node/
3  // Twilio Credentials
4  var accountSid = '';
5  var authToken = '';
6
7  //require the Twilio module and create a REST client
8  var client = require('twilio')(accountSid, authToken);
9
10 client.messages.create({
11     to: "812555121",
12     from: "+2605551212",
13     body: "This is a test",
14 }, function(err, message) {
15     console.log(message.sid);
16 });
17
18 // https://github.com/twilio/twilio-node/blob/master/LICENSE

```

twilio-test.js

Twilio allows a small number of free text messages, enough to test your code and to play around some.

## 6.11 Displaying the Current Weather Conditions

### 6.11.1 Problem

You want to display the current weather conditions.

### 6.11.2 Solution

Because your Bone is on the network, it's not hard to access the current weather conditions from a weather API.

- Go to <https://openweathermap.org/> and create an account.
- Go to [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys) and get your API key.
- Store your key in the *bash* variable *APPID*.

```
bash$ export APPID="Your key"
```

- Then add the code in [Code for getting current weather conditions \(weather.py\)](#) to a file named `weather.py`.
- Run the python script.

Listing 6.13: Code for getting current weather conditions (weather.py)

```

1  #!/usr/bin/env python3
2  # Displays current weather and forecast
3  import os
4  import sys
5  from datetime import datetime

```

(continues on next page)

(continued from previous page)

```

6 import requests      # For getting weather
7
8 # http://api.openweathermap.org/data/2.5/onecall
9 params = {
10     'appid': os.environ['APPID'],
11     # 'city': 'brazil,indiana',
12     'exclude': "minutely,hourly",
13     'lat': '39.52',
14     'lon': '-87.12',
15     'units': 'imperial'
16 }
17 urlWeather = "http://api.openweathermap.org/data/2.5/onecall"
18
19 print("Getting weather")
20
21 try:
22     r = requests.get(urlWeather, params=params)
23     if(r.status_code==200):
24         # print("headers: ", r.headers)
25         # print("text: ", r.text)
26         # print("json: ", r.json())
27         weather = r.json()
28         print("Temp: ", weather['current']['temp'])           # ☒
29         print("Humid:", weather['current']['humidity'])
30         print("Low: ", weather['daily'][1]['temp']['min'])
31         print("High: ", weather['daily'][0]['temp']['max'])
32         day = weather['daily'][0]['sunrise']-weather['timezone_offset']
33         print("sunrise: " + datetime.datetime.fromtimestamp(day).strftime('%Y-%m-
→%d %H:%M:%S'))
34         # print("Day: " + datetime.datetime.fromtimestamp(day).strftime('%a'))
35         # print("weather: ", weather['daily'][1])           # ☒
36         # print("weather: ", weather)                       # ☒
37         # print("icon: ", weather['current']['weather'][0]['icon'])
38         # print()
39
40     else:
41         print("status_code: ", r.status_code)
42 except IOError:
43     print("File not found: " + tmp101)
44     print("Have you run setup.sh?")
45 except:
46     print("Unexpected error:", sys.exc_info())

```

weather.py

1. Prints current conditions.
2. Prints the forecast for the next day.
3. Prints everything returned by the weather site.

Uncomment what you want to be displayed.

Run this by using the following commands:

```

bone$ ./weather.py
Getting weather
Temp: 73.72
Humid: 31
Low: 54.21
High: 75.47
sunrise: 2023-06-09 14:21:07

```

The weather API returns lots of information. Use Python to extract the information you want.

## 6.12 Sending and Receiving Tweets

### 6.12.1 Problem

You want to send and receive tweets (Twitter posts) with your Bone.

### 6.12.2 Solution

Twitter has a whole [git repo](#) of sample code for interacting with Twitter. Here I'll show how to create a tweet and then how to delete it.

## 6.13 Creating a Project and App

- Follow the [directions here](#) to create a project and app.
- Be sure to give your app Read and Write permission.
- Then go to the [developer portal](#) and select your app by clicking on the gear icon to the right of the app name.
- Click on the *Keys and tokens* tab. Here you can get to all your keys and tokens.

---

**Tip:** Be sure to record them, you can't get them later.

---

- Open the file `twitterKeys.sh` and record your keys in it.

```
export API_KEY='XXX'
export API_SECRET_KEY='XXX'
export BEARER_TOKEN='XXX'
export TOKEN='XXXX'
export TOKEN_SECRET='XXX'
```

- Next, source the file so the values will appear in your bash session.

```
bash$ source twitterKeys.sh
```

You'll need to do this every time you open a new *bash* window.

## 6.14 Creating a tweet

Add the code in [Create a Tweet \(twitter\\_create\\_tweet.py\)](#) to a file called `twitter_create_tweet.py` and run it to see your timeline.

Listing 6.14: Create a Tweet (`twitter_create_tweet.py`)

```
1 #!/usr/bin/env python
2 # From: https://github.com/twitterdev/Twitter-API-v2-sample-code/blob/main/
   ↳ Manage-Tweets/create_tweet.py
3 from requests_oauthlib import OAuth1Session
4 import os
5 import json
6
7 # In your terminal please set your environment variables by running the
   ↳ following lines of code.
8 # export 'API_KEY'='<your_consumer_key>'
```

(continues on next page)

(continued from previous page)

```

9 # export 'API_SECRET_KEY'='<your_consumer_secret>'
10
11 consumer_key = os.environ.get("API_KEY")
12 consumer_secret = os.environ.get("API_SECRET_KEY")
13
14 # Be sure to add replace the text of the with the text you wish to Tweet.
15 ↳ You can also add parameters to post polls, quote Tweets, Tweet with reply,
16 ↳ settings, and Tweet to Super Followers in addition to other features.
17 payload = {"text": "Hello world!"}
18
19 # Get request token
20 request_token_url = "https://api.twitter.com/oauth/request_token?oauth_
21 ↳ callback=oob&x_auth_access_type=write"
22 oauth = OAuth1Session(consumer_key, client_secret=consumer_secret)
23
24 try:
25     fetch_response = oauth.fetch_request_token(request_token_url)
26 except ValueError:
27     print(
28         "There may have been an issue with the consumer_key or consumer_
29 ↳ secret you entered."
30     )
31
32 resource_owner_key = fetch_response.get("oauth_token")
33 resource_owner_secret = fetch_response.get("oauth_token_secret")
34 print("Got OAuth token: %s" % resource_owner_key)
35
36 # Get authorization
37 base_authorization_url = "https://api.twitter.com/oauth/authorize"
38 authorization_url = oauth.authorization_url(base_authorization_url)
39 print("Please go here and authorize: %s" % authorization_url)
40 verifier = input("Paste the PIN here: ")
41
42 # Get the access token
43 access_token_url = "https://api.twitter.com/oauth/access_token"
44 oauth = OAuth1Session(
45     consumer_key,
46     client_secret=consumer_secret,
47     resource_owner_key=resource_owner_key,
48     resource_owner_secret=resource_owner_secret,
49     verifier=verifier,
50 )
51 oauth_tokens = oauth.fetch_access_token(access_token_url)
52
53 access_token = oauth_tokens["oauth_token"]
54 access_token_secret = oauth_tokens["oauth_token_secret"]
55
56 # Make the request
57 oauth = OAuth1Session(
58     consumer_key,
59     client_secret=consumer_secret,
60     resource_owner_key=access_token,
61     resource_owner_secret=access_token_secret,
62 )
63
64 # Making the request
65 response = oauth.post(
66     "https://api.twitter.com/2/tweets",
67     json=payload,
68 )

```

(continues on next page)

(continued from previous page)

```

66 if response.status_code != 201:
67     raise Exception(
68         "Request returned an error: {} {}".format(response.status_code,
69         ↳response.text)
69     )
70
71 print("Response code: {}".format(response.status_code))
72
73 # Saving the response as JSON
74 json_response = response.json()
75 print(json.dumps(json_response, indent=4, sort_keys=True))

```

twitter\_create\_tweet.py

Run the code and you'll have to authorize.

```

bash$ ./twitter_create_tweet.py
Got OAuth token: tWBldQAAAAAAWBJgAAABggJt7qg
Please go here and authorize: https://api.twitter.com/oauth/authorize?oauth_
↳token=tWBldQAAAAAAWBJgAAABggJt7qg
Paste the PIN here: 4859044
Response code: 201
{
  "data": {
    "id": "1547963178700533760",
    "text": "Hello world!"
  }
}

```

Check your twitter account and you'll see the new tweet. Record the *id* number and we'll use it next to delete the tweet.

## 6.15 Deleting a tweet

Use the code in [Code to delete a tweet \(twitter\\_delete\\_tweet.py\)](#) to delete a tweet. Around line 15 is the *id* number. Paste in the value returned above.

Listing 6.15: Code to delete a tweet (twitter\_delete\_tweet.py)

```

1  #!/usr/bin/env python
2  # From: https://github.com/twitterdev/Twitter-API-v2-sample-code/blob/main/
↳Manage-Tweets/delete_tweet.py
3  from requests_oauthlib import OAuth1Session
4  import os
5  import json
6
7  # In your terminal please set your environment variables by running the
↳following lines of code.
8  # export 'API_KEY'='<your_consumer_key>'
9  # export 'API_SECRET_KEY'='<your_consumer_secret>'
10
11 consumer_key = os.environ.get("API_KEY")
12 consumer_secret = os.environ.get("API_SECRET_KEY")
13
14 # Be sure to replace tweet-id-to-delete with the id of the Tweet you wish to
↳delete. The authenticated user must own the list in order to delete
15 id = "1547963178700533760"
16

```

(continues on next page)

(continued from previous page)

```

17 # Get request token
18 request_token_url = "https://api.twitter.com/oauth/request_token?oauth_
   ↳callback=oob&x_auth_access_type=write"
19 oauth = OAuth1Session(consumer_key, client_secret=consumer_secret)
20
21 try:
22     fetch_response = oauth.fetch_request_token(request_token_url)
23 except ValueError:
24     print(
25         "There may have been an issue with the consumer_key or consumer_
   ↳secret you entered."
26     )
27
28 resource_owner_key = fetch_response.get("oauth_token")
29 resource_owner_secret = fetch_response.get("oauth_token_secret")
30 print("Got OAuth token: %s" % resource_owner_key)
31
32 # Get authorization
33 base_authorization_url = "https://api.twitter.com/oauth/authorize"
34 authorization_url = oauth.authorization_url(base_authorization_url)
35 print("Please go here and authorize: %s" % authorization_url)
36 verifier = input("Paste the PIN here: ")
37
38 # Get the access token
39 access_token_url = "https://api.twitter.com/oauth/access_token"
40 oauth = OAuth1Session(
41     consumer_key,
42     client_secret=consumer_secret,
43     resource_owner_key=resource_owner_key,
44     resource_owner_secret=resource_owner_secret,
45     verifier=verifier,
46 )
47 oauth_tokens = oauth.fetch_access_token(access_token_url)
48
49 access_token = oauth_tokens["oauth_token"]
50 access_token_secret = oauth_tokens["oauth_token_secret"]
51
52 # Make the request
53 oauth = OAuth1Session(
54     consumer_key,
55     client_secret=consumer_secret,
56     resource_owner_key=access_token,
57     resource_owner_secret=access_token_secret,
58 )
59
60 # Making the request
61 response = oauth.delete("https://api.twitter.com/2/tweets/{}".format(id))
62
63 if response.status_code != 200:
64     raise Exception(
65         "Request returned an error: {} {}".format(response.status_code,
   ↳response.text)
66     )
67
68 print("Response code: {}".format(response.status_code))
69
70 # Saving the response as JSON
71 json_response = response.json()
72 print(json_response)

```

twitter\_delete\_tweet.py



---

**Todo:** Start Here. Update for python.

---

The code in [Tweet when a button is pushed \(twitterPushbutton.js\)](#) sends a tweet whenever a button is pushed.

Listing 6.16: Tweet when a button is pushed (twitterPushbutton.js)

```
1  #!/usr/bin/env node
2  // From: https://www.npmjs.org/package/node-twitter
3  // Tweets with attached image media (JPG, PNG or GIF) can be posted
4  // using the upload API endpoint.
5  var Twitter = require('node-twitter');
6  var b = require('bonescript');
7  var key = require('./twitterKeys');
8  var gpio = "P9_42";
9  var count = 0;
10
11 b.pinMode(gpio, b.INPUT);
12 b.attachInterrupt(gpio, sendTweet, b.FALLING);
13
14 var twitterRestClient = new Twitter.RestClient(
15     key.API_KEY, key.API_SECRET,
16     key.TOKEN,   key.TOKEN_SECRET
17 );
18
19 function sendTweet() {
20     console.log("Sending...");
21     count++;
22
23     twitterRestClient.statusesUpdate(
24         {'status': 'Posting tweet ' + count + ' via my BeagleBone Black', },
25         function(error, result) {
26             if (error) {
27                 console.log('Error: ' +
28                     (error.code ? error.code + ' ' + error.message : error.
29 →message));
30             }
31
32             if (result) {
33                 console.log(result);
34             }
35         });
36     }
37
38 // node-twitter is made available under terms of the BSD 3-Clause License.
39 // http://www.opensource.org/licenses/BSD-3-Clause
```

twitterPushbutton.js

To see many other examples, go to [Twitter for Node.js on NPMJS.com](#).

This opens up many new possibilities. You can read a temperature sensor and tweet its value whenever it changes, or you can turn on an LED whenever a certain hashtag is used. What are you going to tweet?

## 6.16 Wiring the IoT with Node-RED

### 6.16.1 Problem

You want BeagleBone to interact with the Internet, but you want to program it graphically.

## 6.16.2 Solution

Node-RED is a visual tool for wiring the IoT. It makes it easy to turn on a light when a certain hashtag is tweeted, or spin a motor if the forecast is for hot weather.

## 6.17 Starting Node-RED

Node-RED is already installed, to run Node-RED, use the following command to start.

```
bone$ sudo systemctl start nodered
```

Or run the following to have Node-RED start everytime you reboot.

```
bone$ sudo systemctl enable --now nodered
```

Node-RED is listening on port 1880. Point your browser to <http://192.168.7.2:1880>, and you will see the screen shown in [The Node-RED web page](#).

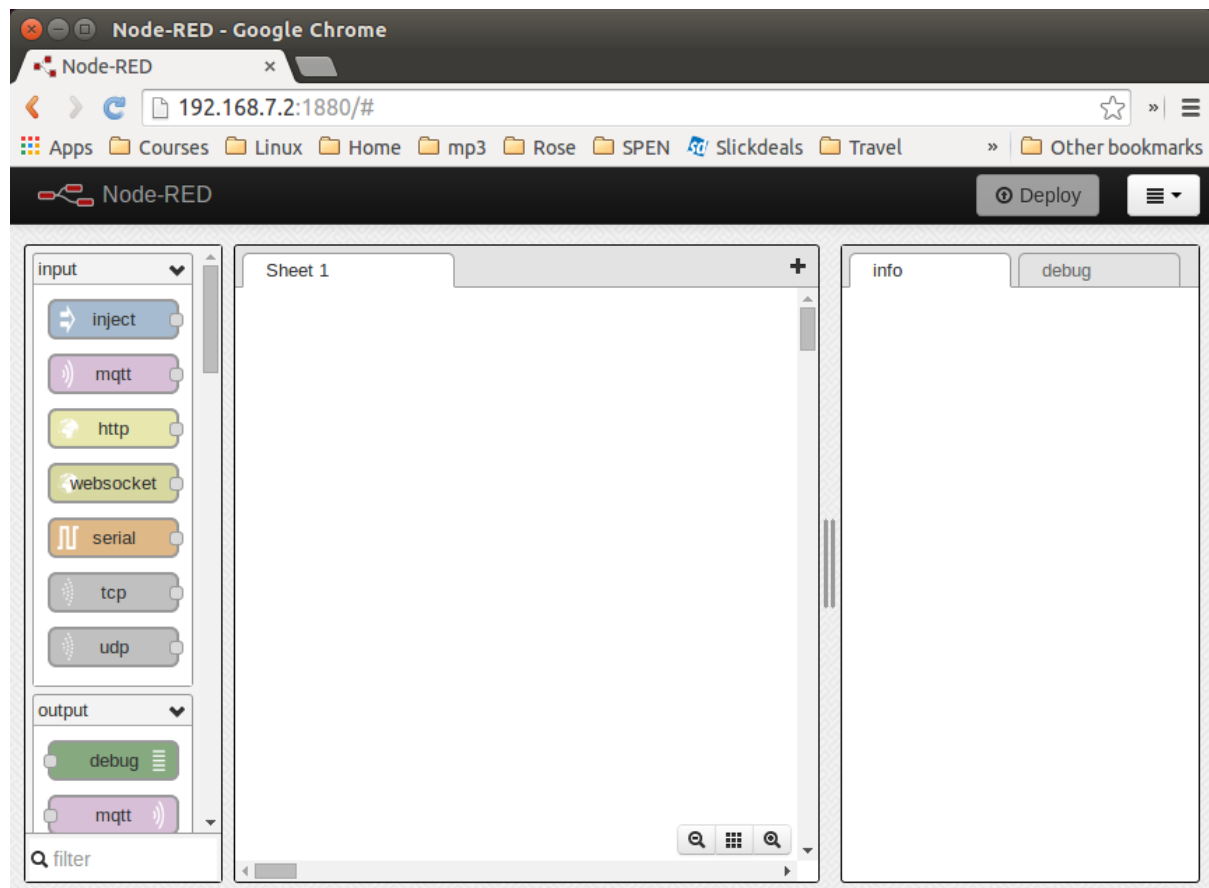


Fig. 6.7: The Node-RED web page

## 6.18 Building a Node-RED Flow

The example in this recipe builds a Node-RED flow that will toggle an LED whenever a certain hashtag is tweeted. But first, you need to set up the Node-RED flow with the *twitter* node:

- On the Node-RED web page, scroll down until you see the *social* nodes on the left side of the page.

- Drag the *twitter* node to the canvas, as shown in [Node-RED twitter node](#).

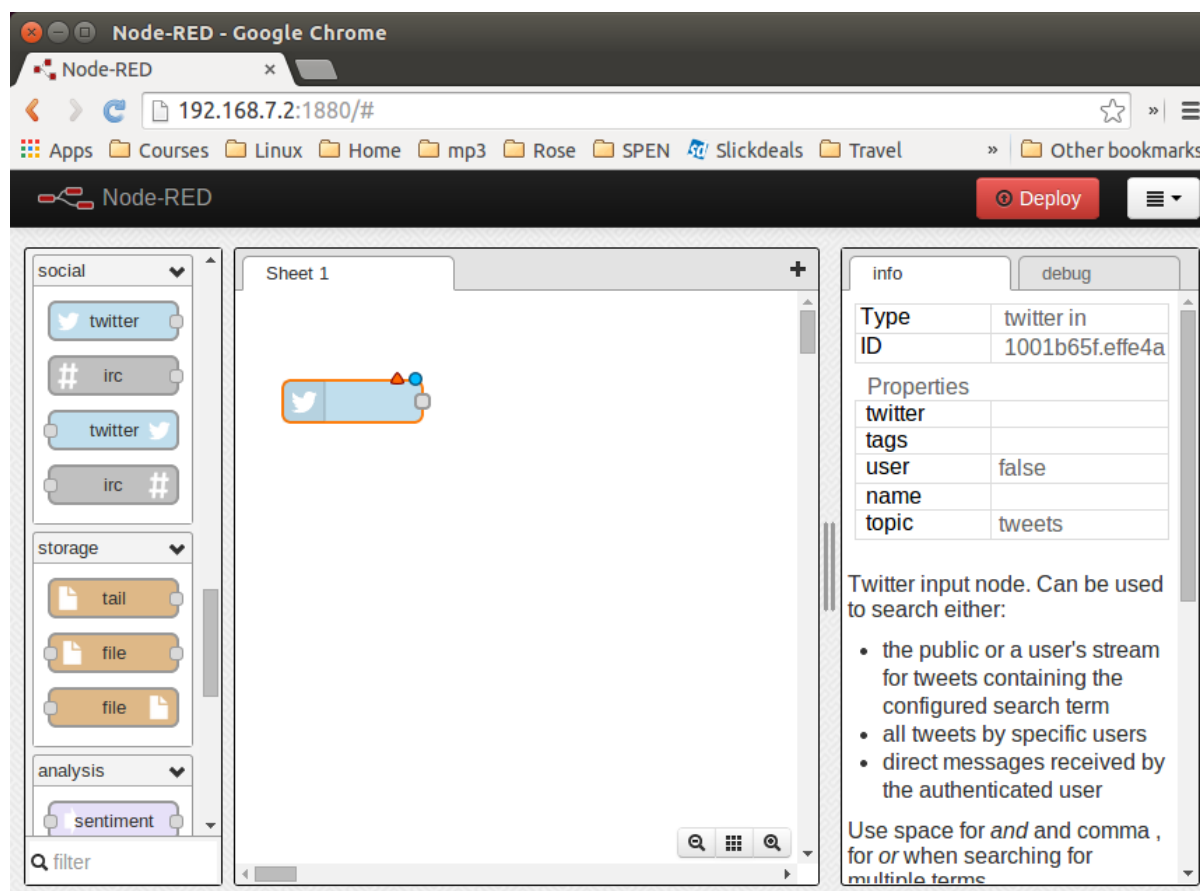


Fig. 6.8: Node-RED twitter node

Authorize Twitter by double-clicking the *twitter* node. You'll see the screen shown in [Node-RED Twitter authorization, step 1](#).

Click the pencil button to bring up the dialog box shown in [Node-RED twitter authorization, step 2](#).

- Click the "here" link, as shown in [Node-RED twitter authorization, step 2](#), and you'll be taken to Twitter to authorize Node-RED.
- Log in to Twitter and click the "Authorize app" button ([Node-RED Twitter site authorization](#)).
- When you're back to Node-RED, click the Add button, add your Twitter credentials, enter the hashtags to respond to ([Node-RED adding the #BeagleBone hashtag](#)), and then click the Ok button.
- Go back to the left panel, scroll up to the top, and then drag the *debug* node to the canvas- (*debug* is in the *output* section.)
- Connect the two nodes by clicking and dragging ([Node-RED Twitter adding debug node and connecting](#)).
- In the right panel, in the upper-right corner, click the "debug" tab.
- Finally, click the Deploy button above the "debug" tab.

Your Node-RED flow is now running on the Bone. Test it by going to Twitter and tweeting something with the hashtag *#BeagleBone*. Your Bone is now responding to events happening out in the world.

## 6.19 Adding an LED Toggle

Now, we're ready to add the LED toggle:

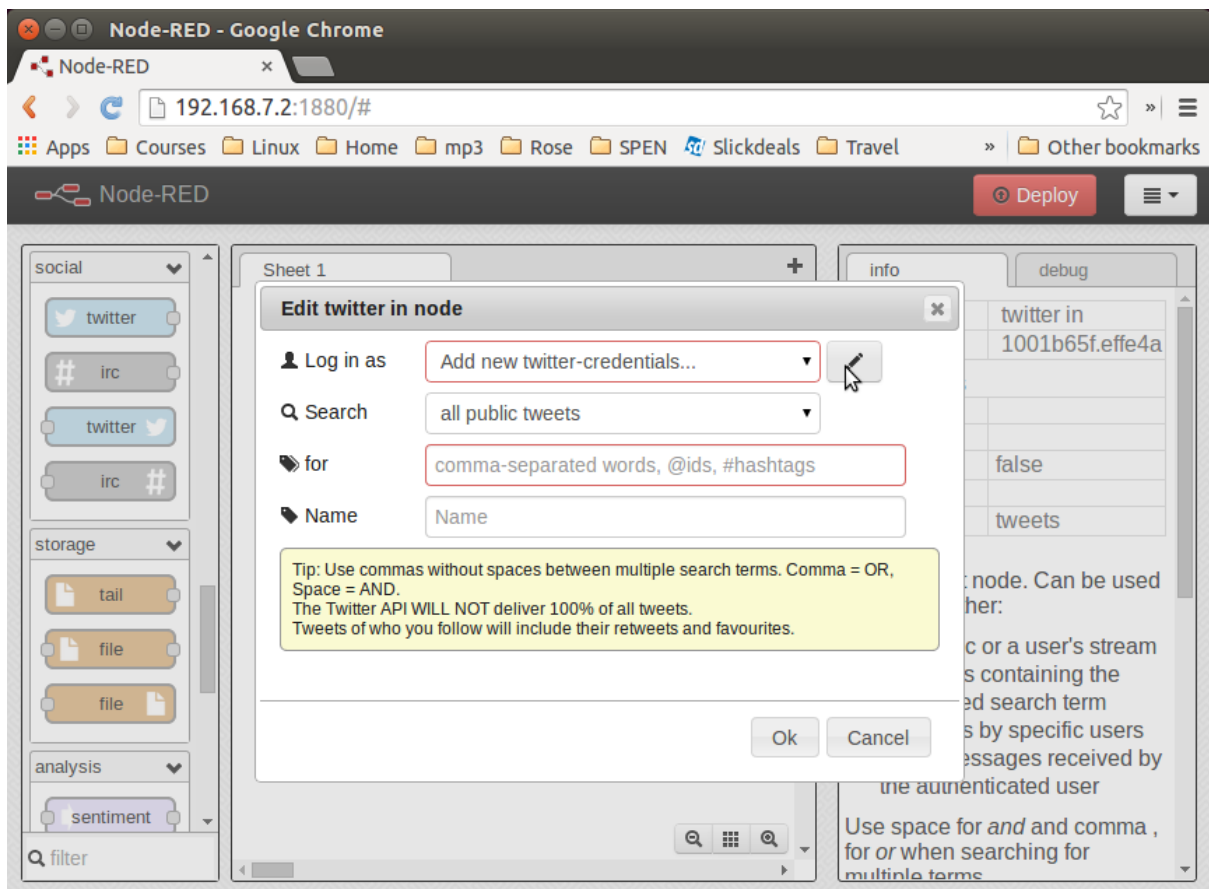


Fig. 6.9: Node-RED Twitter authorization, step 1

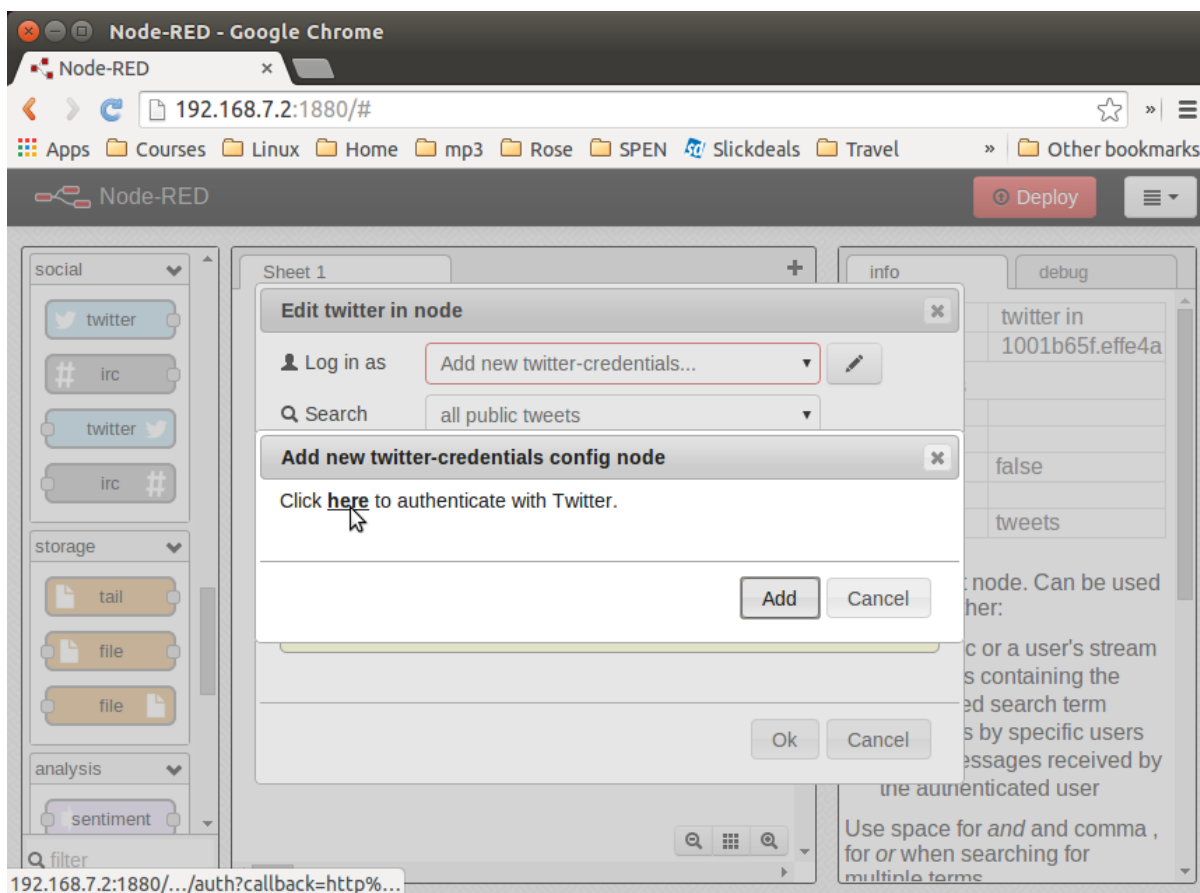


Fig. 6.10: Node-RED twitter authorization, step 2

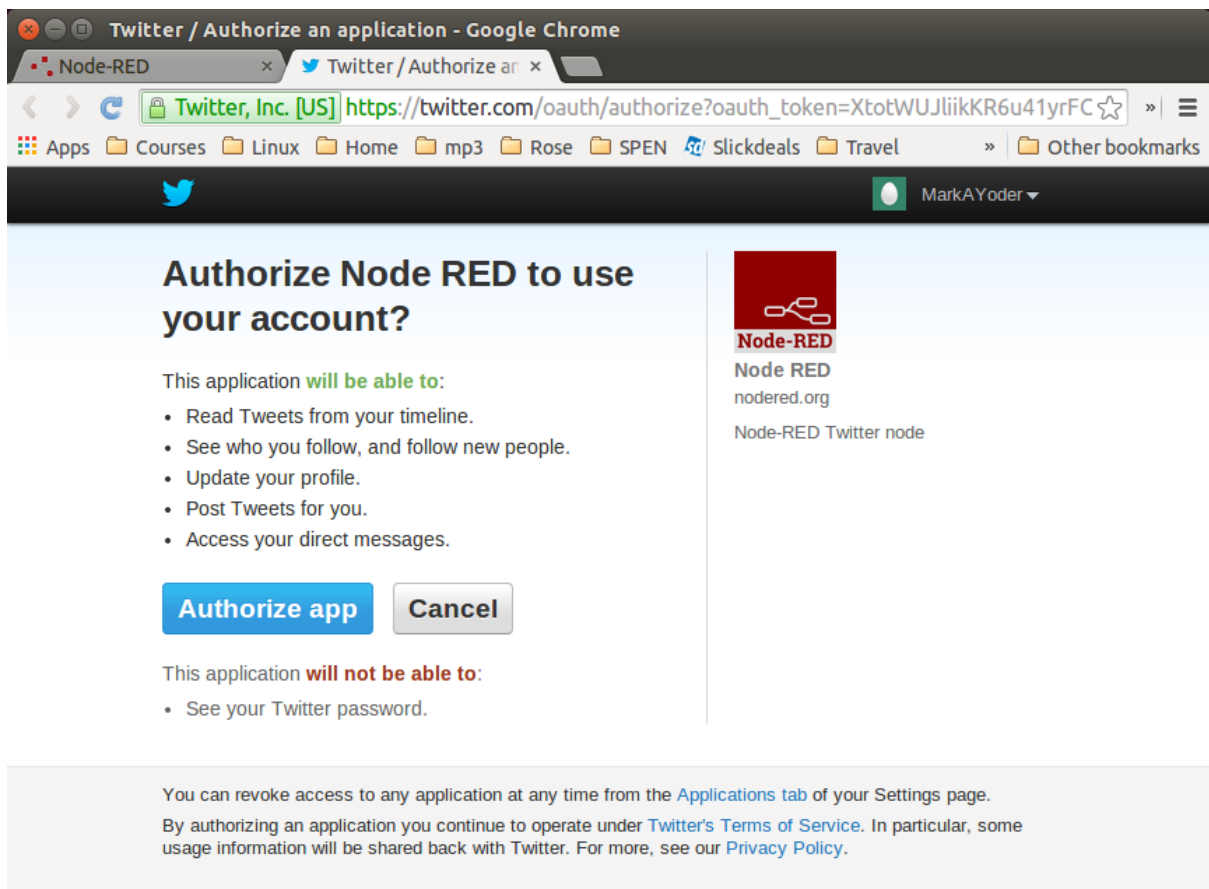


Fig. 6.11: Node-RED Twitter site authorization

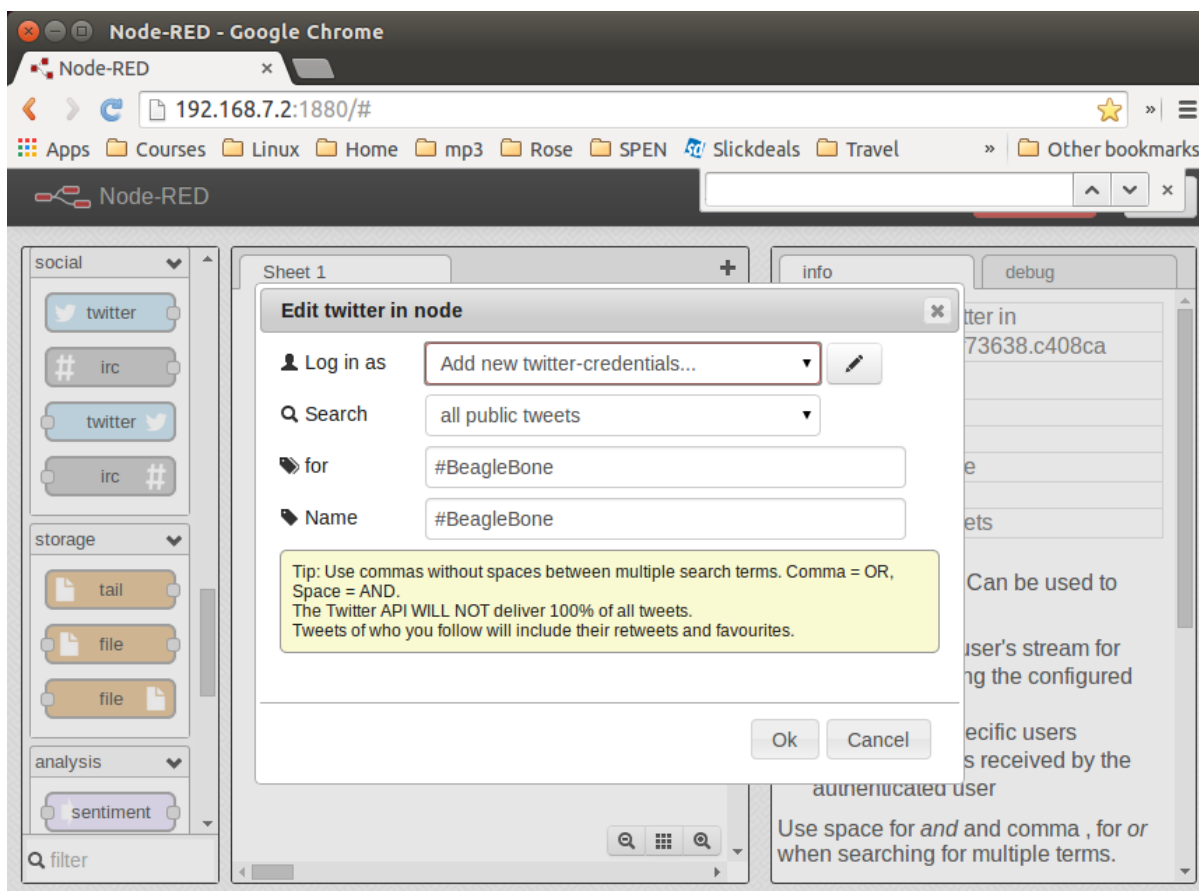


Fig. 6.12: Node-RED adding the #BeagleBone hashtag

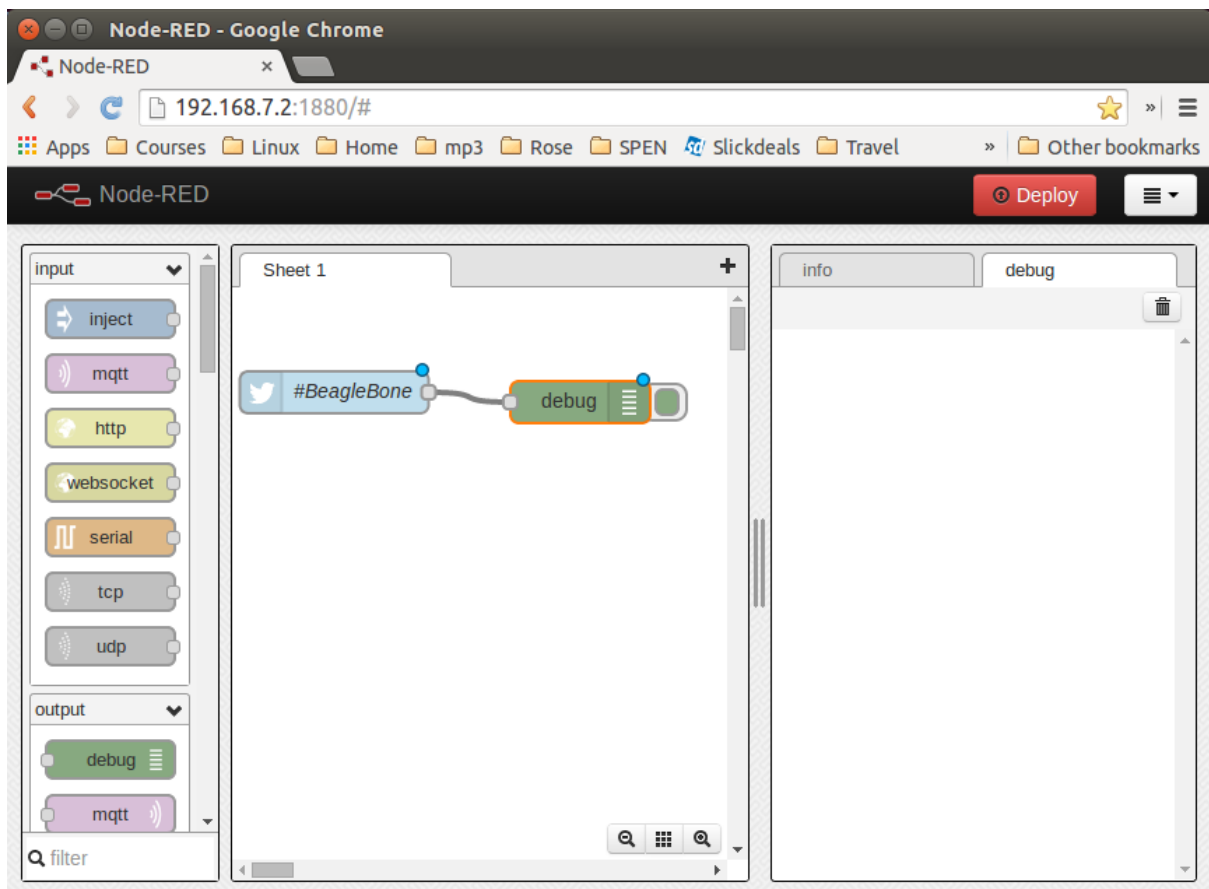


Fig. 6.13: Node-RED Twitter adding *debug* node and connecting



- Wire up an LED as shown in [Toggling an External LED](#). Mine is wired to P9\_14.
- Scroll to the bottom of the left panel and drag the *bbb-discrete-out* node (second from the bottom of the *bbb* nodes) to the canvas and wire it ([Node-RED adding bbb-discrete-out node](#)).

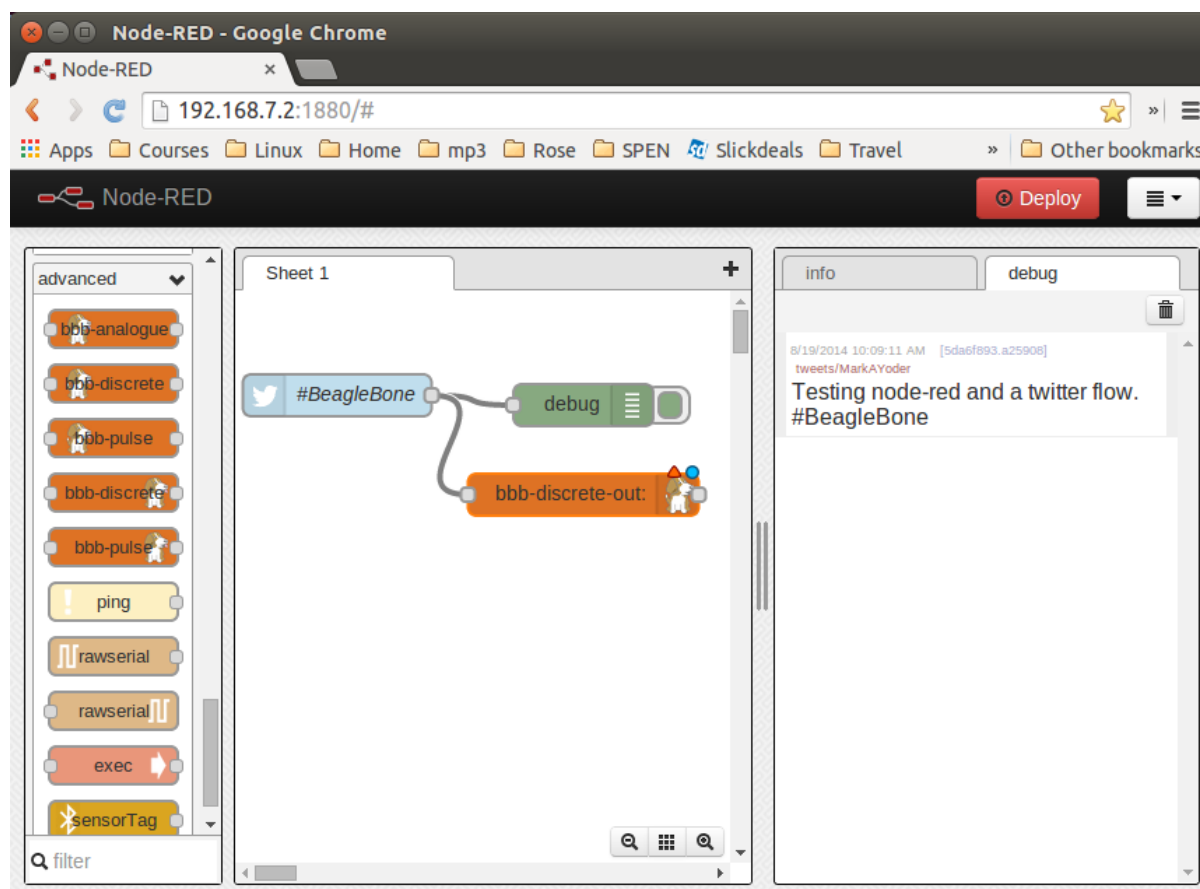


Fig. 6.14: Node-RED adding bbb-discrete-out node

Double-click the node, select your GPIO pin and “Toggle state,” and then set “Startup as” to 1 ([Node-RED adding bbb-discrete-out configuration](#)).

Click Ok and then Deploy.

Test again. The LED will toggle every time the hashtag *#BeagleBone* is tweeted. With a little more exploring, you should be able to have your Bone ringing a bell or spinning a motor in response to tweets.

## 6.20 Communicating over a Serial Connection to an Arduino or LaunchPad

### 6.20.1 Problem

You would like your Bone to talk to an Arduino or LaunchPad.

### 6.20.2 Solution

The common serial port (also known as a UART) is the simplest way to talk between the two. Wire it up as shown in [Wiring a LaunchPad to a Bone via the common serial port](#).

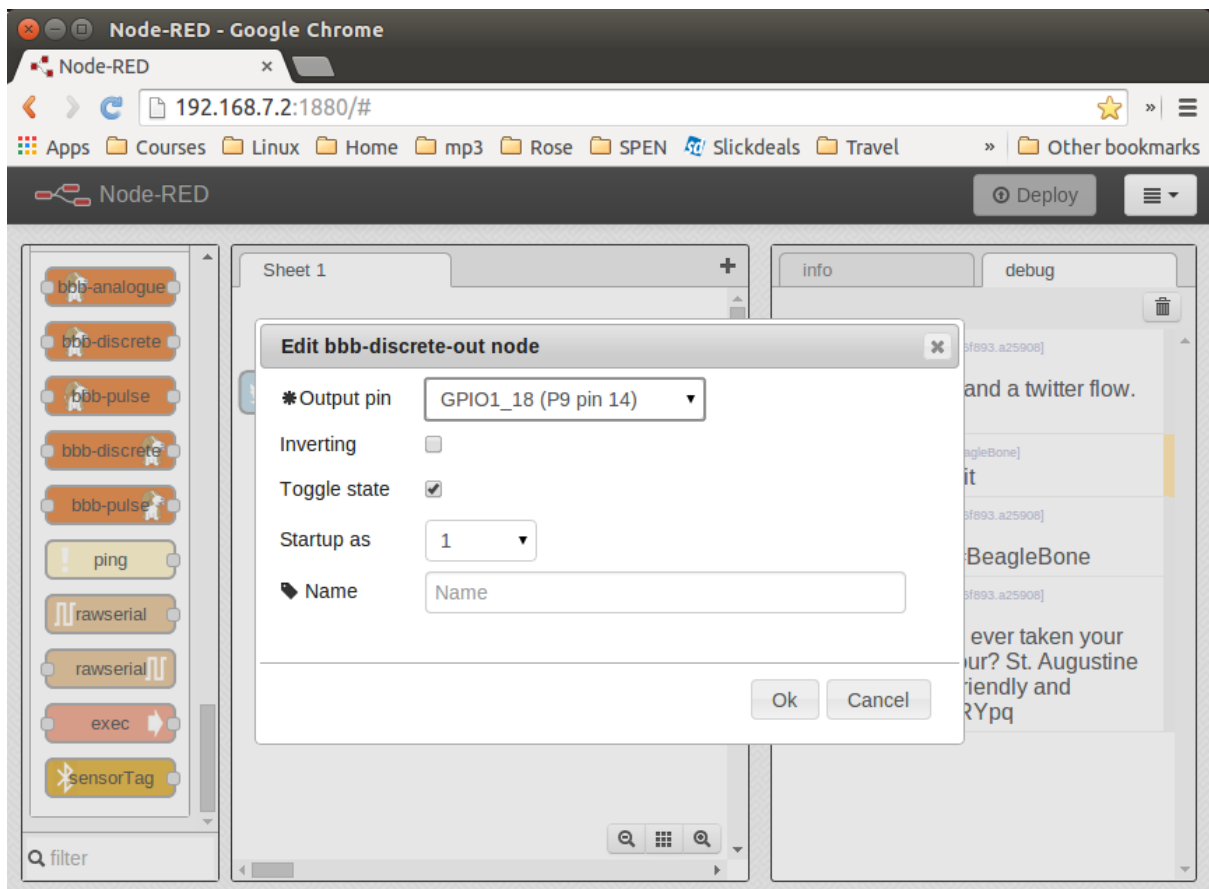


Fig. 6.15: Node-RED adding bbb-discrete-out configuration

**Warning:** BeagleBone Black runs at 3.3 V. When wiring other devices to it, ensure that they are also 3.3 V. The LaunchPad I'm using is 3.3 V, but many Arduinos are 5.0 V and thus won't work. Or worse, they might damage your Bone.

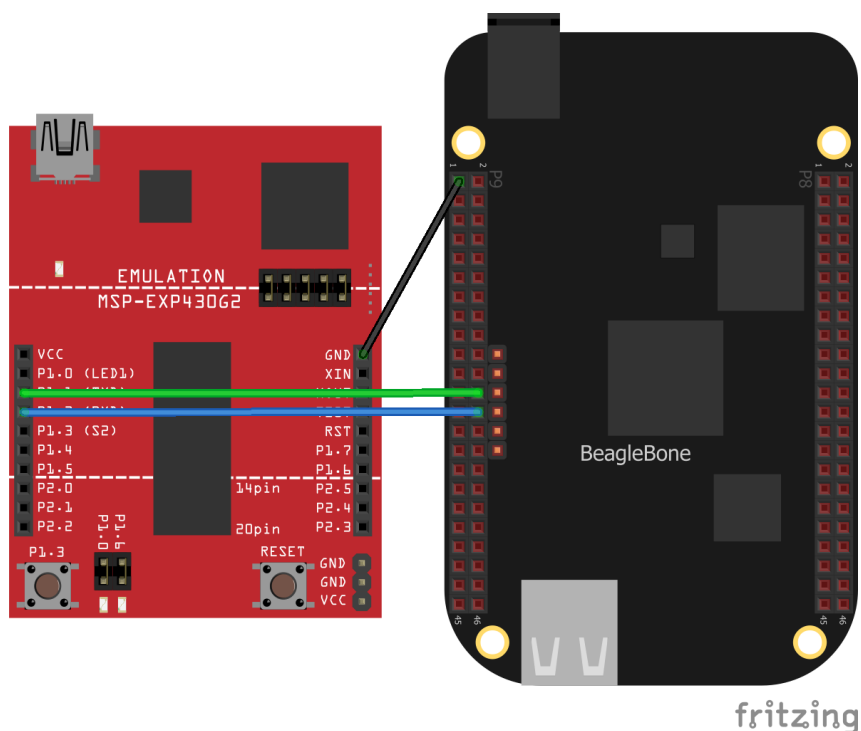


Fig. 6.16: Wiring a LaunchPad to a Bone via the common serial port

Add the code (or sketch, as it's called in Arduino-speak) in [LaunchPad code for communicating via the UART \(launchPad.ino\)](#) to a file called `launchPad.ino` and run it on your LaunchPad.

Listing 6.17: LaunchPad code for communicating via the UART (launchPad.ino)

```

1  /*
2   Tests connection to a BeagleBone
3   Mark A. Yoder
4   Waits for input on Serial Port
5   g - Green toggle
6   r - Red toggle
7  */
8  char inChar = 0; // incoming serial byte
9  int red = 0;
10 int green = 0;
11
12 void setup()
13 {
14 // initialize the digital pin as an output.
15 pinMode(RED_LED, OUTPUT); // ?
16 pinMode(GREEN_LED, OUTPUT);
17 // start serial port at 9600 bps:
18 Serial.begin(9600); // ?
19 Serial.print("Command (r, g): "); // ?
20
21 digitalWrite(GREEN_LED, green); // ?
22 digitalWrite( RED_LED, red);
23 }

```

(continues on next page)

(continued from previous page)

```

24
25 void loop()
26 {
27   if(Serial.available() > 0 ) {           // ?
28     inChar = Serial.read();
29     switch(inChar) {                     // ?
30       case 'g':
31         green = ~green;
32         digitalWrite(GREEN_LED, green);
33         Serial.println("Green");
34         break;
35       case 'r':
36         red = ~red;
37         digitalWrite(RED_LED, red);
38         Serial.println("Red");
39         break;
40     }
41     Serial.print("Command (r, g): ");
42   }
43 }
44

```

launchPad.ino

- ① Set the mode for the built-in red and green LEDs.
- ② Start the serial port at 9600 baud.
- ③ Prompt the user, which in this case is the Bone.
- ④ Set the LEDs to the current values of the *red* and *green* variables.
- ⑤ Wait for characters to arrive on the serial port.
- ⑥ After the characters are received, read it and respond to it.

On the Bone, add the script in [Code for communicating via the UART \(launchPad.js\)](#) to a file called *launchPad.js* and run it.

Listing 6.18: Code for communicating via the UART (launchPad.js)

```

1  #!/usr/bin/env node
2  // Need to add exports.serialParsers = m.module.parsers;
3  // to /usr/local/lib/node_modules/bonescript/serial.js
4  var b = require('bonescript');
5
6  var port = '/dev/ttyO1';           // ?
7  var options = {
8     baudrate: 9600,                 // ?
9     parser: b.serialParsers.readline("\n") // ?
10 };
11
12 b.serialOpen(port, options, onSerial); // ?
13
14 function onSerial(x) {             // ?
15   console.log(x.event);
16   if (x.err) {
17     console.log('***ERROR*** ' + JSON.stringify(x));
18   }
19   if (x.event == 'open') {
20     console.log('***OPENED***');
21     setInterval(sendCommand, 1000); // ?
22   }
23   if (x.event == 'data') {

```

(continues on next page)

(continued from previous page)

```

24     console.log(String(x.data));
25   }
26 }
27
28 var command = ['r', 'g'];           // ?
29 var commIdx = 1;
30
31 function sendCommand() {
32   // console.log('Command: ' + command[commIdx]);
33   b.serialWrite(port, command[commIdx++]); // ?
34   if(commIdx >= command.length) {     // ?
35     commIdx = 0;
36   }
37 }

```

launchPad.js

- ① Select which serial port to use. [Table of UART outputs](#) shows what's available. We've wired P9\_24 and P9\_26, so we are using serial port `/dev/ttyO1`. (Note that's the letter O and not the number zero.)
- ② Set the baudrate to 9600, which matches the setting on the LaunchPad.
- ③ Read one line at a time up to the newline character (`n`).
- ④ Open the serial port and call `onSerial()` whenever there is data available.
- ⑤ Determine what event has happened on the serial port and respond to it.
- ⑥ If the serial port has been *opened*, start calling `sendCommand()` every 1000 ms.
- ⑦ These are the two commands to send.
- ⑧ Write the character out to the serial port and to the LaunchPad.
- ⑨ Move to the next command.

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
UART4_RXD	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
UART4_TXD	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
UART1_RTSN	19	20	UART1_CTSN	GPIO_22	19	20	GPIO_63
UART2_TXD	21	22	UART2_RXD	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	UART1_TXD	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	UART1_RXD	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	UART5_CTSN+	31	32	UART5_RTSN
AIN4	33	34	GNDA_ADC	UART4_RTSN	33	34	UART3_RTSN
AIN6	35	36	AIN5	UART4_CTSN	35	36	UART3_CTSN
AIN2	37	38	AIN3	UARR5_TXD+	37	38	UART5_RXD+
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	UART3_TXD	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 6.17: Table of UART outputs

### 6.20.3 Discussion

When you run the script in [Code for communicating via the UART \(launchPad.js\)](#), the Bone opens up the serial port and every second sends a new command, either *r* or *g*. The LaunchPad waits for the command, when it arrives, responds by toggling the corresponding LED.



## Chapter 7

# The Kernel

The kernel is the heart of the Linux operating system. It's the software that takes the low-level requests, such as reading or writing files, or reading and writing general-purpose input/output (GPIO) pins, and maps them to the hardware. When you install a new version of the OS ([Verifying You Have the Latest Version of the OS on Your Bone](#)), you get a certain version of the kernel.

You usually won't need to mess with the kernel, but sometimes you might want to try something new that requires a different kernel. This chapter shows how to switch kernels. The nice thing is you can have multiple kernels on your system at the same time and select from among them which to boot up.

### 7.1 Updating the Kernel

#### 7.1.1 Problem

You have an out-of-date kernel and want to make it current.

#### 7.1.2 Solution

Use the following command to determine which kernel you are running:

```
bone$ uname -a
Linux beaglebone 5.10.168-ti-r62 #1bullseye SMP PREEMPT Tue May 23 20:15:00
->UTC 2023 armv7l GNU/Linux
GNU/Linux
```

The `5.10.168-ti-r62` string is the kernel version.

To update to the current kernel, ensure that your Bone is on the Internet ([Sharing the Host's Internet Connection over USB](#) or [Establishing an Ethernet-Based Internet Connection](#)) and then run the following commands:

```
bone$ apt-cache pkgnames | grep linux-image | sort | less
...
linux-image-5.10.162-ti-r59
linux-image-5.10.162-ti-rt-r56
linux-image-5.10.162-ti-rt-r57
linux-image-5.10.162-ti-rt-r58
linux-image-5.10.162-ti-rt-r59
linux-image-5.10.168-armv7-lpae-x71
linux-image-5.10.168-armv7-rt-x71
linux-image-5.10.168-armv7-x71
linux-image-5.10.168-bone71
linux-image-5.10.168-bone-rt-r71
```

(continues on next page)



(continued from previous page)

```
linux-image-5.10.168-ti-r60
linux-image-5.10.168-ti-r61
linux-image-5.10.168-ti-r62
linux-image-5.10.168-ti-rt-r60
linux-image-5.10.168-ti-rt-r61
linux-image-5.10.168-ti-rt-r62
...

bone$ sudo apt install linux-image-5.10.162-ti-rt-r59
bone$ sudo reboot

bone$ uname -a
Linux beaglebone 5.10.162-ti-rt-r59 #1 SMP PREEMPT Wed Nov 19 21:11:08 UTC
↳2014 armv7l
GNU/Linux
```

The first command lists the versions of the kernel that are available. The second command installs one. After you have rebooted, the new kernel will be running.

If the current kernel is doing its job adequately, you probably don't need to update, but sometimes a new software package requires a more up-to-date kernel. Fortunately, precompiled kernels are available and ready to download.

### Seeing which kernels are installed

You can have multiple kernels install at the same time. They are saved in **/boot**

```
bone$ cd /boot
bone$ ls
config-5.10.168-ti-r62      initrd.img-5.10.168-ti-r63  uboot
↳ vmlinuz-5.10.168-ti-r63
config-5.10.168-ti-r63    SOC.sh                      uEnv.txt
dtbs                     System.map-5.10.168-ti-r62  uEnv.txt.orig
initrd.img-5.10.168-ti-r62 System.map-5.10.168-ti-r63  vmlinuz-5.10.168-ti-
↳r62
```

Here I have two kernel versions installed.

### Bone

On the Bone (Not the Play) the file **uEnv.txt** tells which kernel to use on the next reboot. Here are the first few lines:

```
Line
1 #Docs: http://elinux.org/Beagleboard:U-boot\_partitioning\_layout\_2.0
2
3 # uname_r=4.14.108-ti-r137
4 uname_r=4.19.94-ti-r50
5 # uname_r=5.4.52-ti-r17
6 #uuid=
```

Lines 3-5 list the various kernels, and the uncommented one on line 4 is the one that will be used next time. You will have to add your own `uname_r`'s. Get the names from the files in `/boot`. Be careful, if you mistype the name your Bone won't boot.

### Play

On the Play you can see which version of the kernel will boot next by:

```
play$ cat /boot/firmware/kversion
5.10.168-ti-arm64-r106
```

If you want to change the version run:

```
bone$ sudo apt install linux-image-5.10.168-ti-arm64-r105 --reinstall
```

## 7.2 Building and Installing Kernel Modules

### 7.2.1 Problem

You need to use a peripheral for which there currently is no driver, or you need to improve the performance of an interface previously handled in user space.

### 7.2.2 Solution

The solution is to run in kernel space by building a kernel module. There are entire [books on writing Linux Device Drivers](#). This recipe assumes that the driver has already been written and shows how to compile and install it. After you've followed the steps for this simple module, you will be able to apply them to any other module.

For our example module, add the code in [Simple Kernel Module \(hello.c\)](#) to a file called `hello.c`.

Listing 7.1: Simple Kernel Module (hello.c)

```

1  #include <linux/module.h>          /* Needed by all modules */
2  #include <linux/kernel.h>        /* Needed for KERN_INFO */
3  #include <linux/init.h>          /* Needed for the macros */
4
5  static int __init hello_start(void)
6  {
7      printk(KERN_INFO "Loading hello module...\n");
8      printk(KERN_INFO "Hello, World!\n");
9      return 0;
10 }
11
12 static void __exit hello_end(void)
13 {
14     printk(KERN_INFO "Goodbye Boris\n");
15 }
16
17 module_init(hello_start);
18 module_exit(hello_end);
19
20 MODULE_AUTHOR("Boris Houndleroy");
21 MODULE_DESCRIPTION("Hello World Example");
22 MODULE_LICENSE("GPL");
```

`hello.c`

When compiling on the Bone, all you need to do is load the Kernel Headers for the version of the kernel you're running:

```
bone$ sudo apt install linux-headers-`uname -r`
```

**Note:** The quotes around `uname -r` are backtick characters. On a United States keyboard, the backtick key is to the left of the 1 key.

This took a little more than three minutes on my Bone. The `uname -r` part of the command looks up what version of the kernel you are running and loads the headers for it.

**Note:** If you don't have a network connection you can get the headers from the running kernel with the following.

```
sudo modprobe kheaders
rm -rf $HOME/headers
mkdir -p $HOME/headers
tar -xvf /sys/kernel/kheaders.tar.xz -C $HOME/headers > /dev/null
cd my-kernel-module
make -C $HOME/headers M=$(pwd) modules
sudo rmmod kheaders
```

The `modprobe kheaders` makes the `/sys/kernel/kheaders.tar.xz` appear.

---

Next, add the code in [Simple Kernel Module \(Makefile\)](#) to a file called `Makefile`.

Listing 7.2: Simple Kernel Module (Makefile)

```
1 obj-m := hello.o
2 KDIR := /lib/modules/$(shell uname -r)/build
3
4 all:
5 <TAB>make -C $(KDIR) M=$$PWD
6
7 clean:
8 <TAB>rm hello.mod.c hello.o modules.order hello.mod.o Module.symvers
```

`Makefile.display`

**Note:** Replace the two instances of `<TAB>` with a tab character (the key left of the Q key on a United States keyboard). The tab characters are very important to makefiles and must appear as shown.

---

Now, compile the kernel module by using the `make` command:

```
bone$ make
make -C /lib/modules/5.10.168-ti-r62/build M=$PWD
make[1]: Entering directory '/usr/src/linux-headers-5.10.168-ti-r62'
CC [M] /home/debian/docs.beagleboard.io/books/beaglebone-cookbook/code/
↳07kernel/hello.o
MODPOST /home/debian/docs.beagleboard.io/books/beaglebone-cookbook/code/
↳07kernel/Module.symvers
CC [M] /home/debian/docs.beagleboard.io/books/beaglebone-cookbook/code/
↳07kernel/hello.mod.o
LD [M] /home/debian/host/BeagleBoard/docs.beagleboard.io/books/beaglebone-
↳cookbook/code/07kernel/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.10.168-ti-r62'

bone$ ls
Makefile      hello.c      hello.mod.c  hello.o
Module.symvers hello.ko     hello.mod.o  modules.order
```

Notice that several files have been created. `hello.ko` is the one you want. Try a couple of commands with it:

```
bone$ modinfo hello.ko
filename:      /home/debian/host/BeagleBoard/docs.beagleboard.io/books/
↳beaglebone-cookbook/code/07kernel/hello.ko
license:      GPL
```

(continues on next page)

(continued from previous page)

```

description:   Hello World Example
author:       Boris Houndleroy
depends:
name:         hello
vermagic:     5.10.168-ti-r62 SMP preempt mod_unload modversions ARMv7 p2v8

bone$ sudo insmod hello.ko
bone$ dmesg | tail -4
[ 377.944777] lm75 1-004a: hwmon1: sensor 'tmp101'
[ 377.944976] i2c i2c-1: new_device: Instantiated device tmp101 at 0x4a
[85819.772666] Loading hello module...
[85819.772687] Hello, World!

```

The first command displays information about the module. The `insmod` command inserts the module into the running kernel. If all goes well, nothing is displayed, but the module does print something in the kernel log. The `dmesg` command displays the messages in the log, and the `tail -4` command shows the last four messages. The last two messages are from the module. It worked!

## 7.3 Compiling the Kernel

### 7.3.1 Problem

You need to download, patch, and compile the kernel from its source code.

### 7.3.2 Solution

This is easier than it sounds, thanks to some very powerful scripts.

**Warning:** Be sure to run this recipe on your host computer. The Bone has enough computational power to compile a module or two, but compiling the entire kernel takes lots of time and resources.

## 7.4 Downloading and Compiling the Kernel

To download and compile the kernel, follow these steps:

```

host$ git clone https://git.beagleboard.org/RobertCNelson/ti-linux-kernel-
->dev # [?]
host$ cd ti-linux-kernel-dev
host$ git checkout ti-linux-5.10.y # [?]
host$ ./build_deb.sh # [?]

```

**Note:** If you are using a 64 bit Bone, **git checkout ti-linux-arm64-5.10.y**

- ① The first command clones a repository with the tools to build the kernel for the Bone.
- ② When you know which kernel to try, use `git checkout` to check it out. This command checks out branch `ti-linux-5.10.y`.
- ③ `build_deb.sh` is the master builder. If needed, it will download the cross compilers needed to compile the kernel (`gcc` is the current cross compiler). If there is a kernel at `~/linux-dev`, it will use it; otherwise, it will download a copy to `ti-linux-kernel-dev/ignore/linux-src`. It will then patch the kernel so that it will run on the Bone.

**Note:** `build_deb.sh` may ask you to install additional files. Just run `sudo apt install *files*` to install them.

After the kernel is patched, you'll see a screen similar to [Kernel configuration menu](#), on which you can configure the kernel.

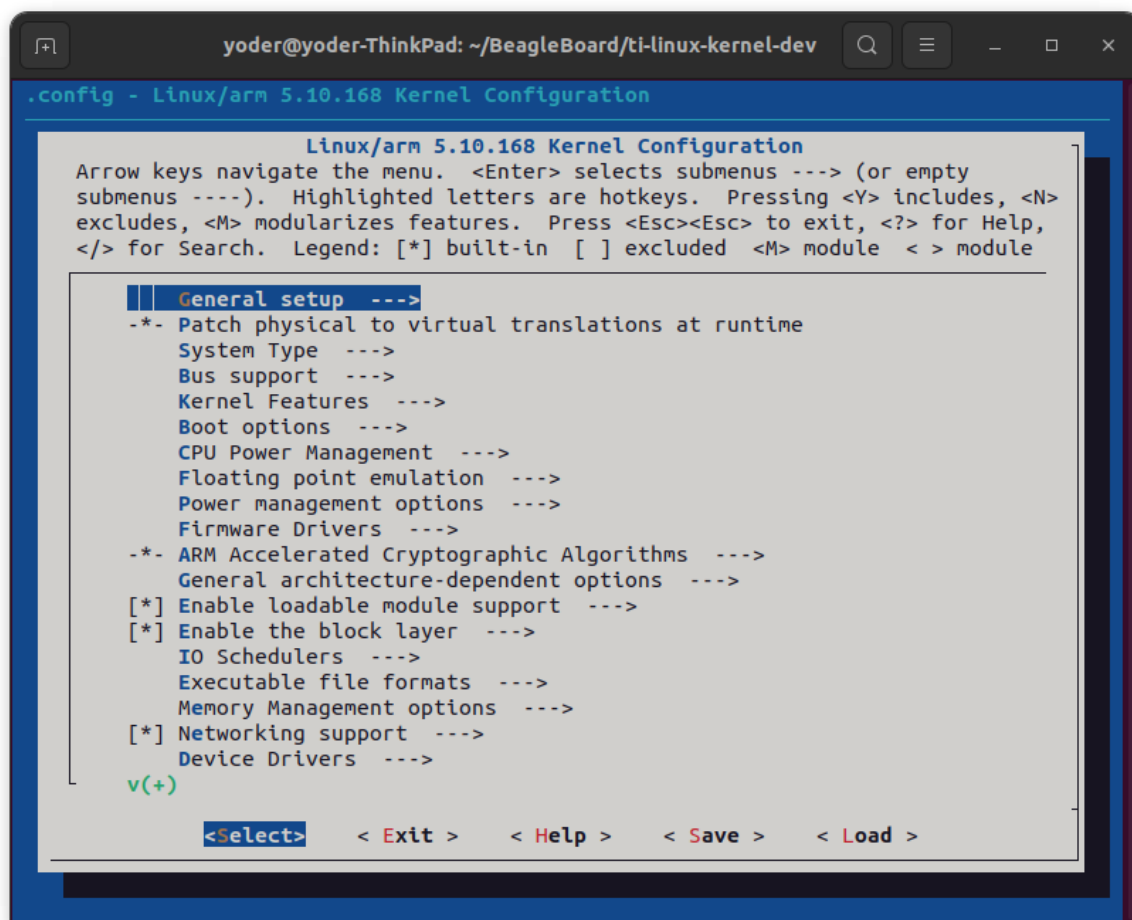


Fig. 7.1: Kernel configuration menu

You can use the arrow keys to navigate. No changes need to be made, so you can just press the right arrow and Enter to start the kernel compiling. The entire process took about 25 minutes on my 8-core host.

The `ti-linux-kernel-dev/KERNEL` directory contains the source code for the kernel. The `ti-linux-kernel-dev/deploy` directory contains the compiled kernel and the files needed to run it.

## 7.5 Installing the Kernel on the Bone

The `./build_deb.sh` script creates a single `.deb` file that contains all the files needed for the new kernel. You find it here:

```

host$ cd ti-linux-kernel-dev/deploy
host$ ls -sh
total 40M
7.7M linux-headers-5.10.168-ti-r62_1xross_armhf.deb  8.0K linux-upstream_

```

(continues on next page)

(continued from previous page)

```

↪1xross_armhf.buildinfo
33M linux-image-5.10.168-ti-r62_1xross_armhf.deb      4.0K linux-upstream_
↪1xross_armhf.changes
1.1M linux-libc-dev_1xross_armhf.deb

```

The **linux-image-** file is the one we want. It contains over 3000 files.

```

host$ dpkg -c linux-image-5.10.168-ti-r62_1xross_armhf.deb | wc
3251  19506  379250

```

The **dpkg** command lists all the files in the .deb file and the **wc** counts all the lines in the output. You can see those files with:

```

host$ dpkg -c linux-image-5.10.168-ti-r62_1xross_armhf.deb | less
drwxr-xr-x root/root      0 2023-06-12 12:57 ./
drwxr-xr-x root/root      0 2023-06-12 12:57 ./boot/
-rw-r--r-- root/root 4763113 2023-06-12 12:57 ./boot/System.map-5.10.168-
↪ti-r62
-rw-r--r-- root/root  191331 2023-06-12 12:57 ./boot/config-5.10.168-ti-r62
drwxr-xr-x root/root      0 2023-06-12 12:57 ./boot/dtbs/
drwxr-xr-x root/root      0 2023-06-12 12:57 ./boot/dtbs/5.10.168-ti-r62/
-rwxr-xr-x root/root  90644 2023-06-12 12:57 ./boot/dtbs/5.10.168-ti-r62/
↪am335x-baltos-ir2110.dtb
-rwxr-xr-x root/root  91362 2023-06-12 12:57 ./boot/dtbs/5.10.168-ti-r62/
↪am335x-baltos-ir3220.dtb
-rwxr-xr-x root/root  91633 2023-06-12 12:57 ./boot/dtbs/5.10.168-ti-r62/
↪am335x-baltos-ir5221.dtb
-rwxr-xr-x root/root  88684 2023-06-12 12:57 ./boot/dtbs/5.10.168-ti-r62/
↪am335x-base0033.dtb

```

You can see it's putting things in the **/boot** directory.

**Note:** You can also look into the other two .deb files and see what they install.

Move the **linux-image-** file to your Bone.

```

host$ scp linux-image-5.10.168-ti-r62_1xross_armhf.deb bone:.

```

You might have to use **debian@192.168.7.2** for bone if you haven't set everything up.

Now ssh to the bone.

```

host$ ssh bone
bone$ ls -sh
bin  exercises  linux-image-5.10.168-ti-r62_1xross_armhf.deb

```

Now install it.

```

bone$ sudo dpkg --install linux-image-5.10.168-ti-r62_1xross_armhf.deb

```

Wait a while. (Mine took almost 2 minutes.) Once done check **/boot**.

```

bone$ ls -sh /boot
total 40M
160K config-4.19.94-ti-r50      4.0K SOC.sh      4.0K uEnv.
↪txt.orig
180K config-5.10.168-ti-r62    3.5M System.map-4.19.94-ti-r50  9.7M
↪vmlinuz-4.19.94-ti-r50
4.0K dtbs                      4.1M System.map-5.10.168-ti-r62  8.6M
↪vmlinuz-5.10.168-ti-r62
6.4M initrd.img-4.19.94-ti-r50  4.0K uboot
6.8M initrd.img-5.10.168-ti-r62  4.0K uEnv.txt

```

You see the new kernel files along with the old files. Check uEnv.txt.

```
bone$ head /boot/uEnv.txt
#Docs: http://elinux.org/Beagleboard:U-boot_partitioning_layout_2.0
# uname_r=4.19.94-ti-r50
uname_r=5.10.168-ti-r62
```

I added the commented out `uname_r` line to make it easy to switch between versions of the kernel.

Reboot and test out the new kernel.

```
bone$ sudo reboot
```

## 7.6 Install a Cross Compiler

### 7.6.1 Problem

You want to compile on your host computer and run on the Beagle.

### 7.6.2 Solution

Run the following:

#### 32-bit

```
host$ sudo apt install gcc-arm-linux-gnueabi
```

#### 64-bit

```
host$ sudo apt install gcc-aarch64-linux-gnu
```

---

**Note:** From now on use **arm** if you are using a 32-bit machine and **aarch64** if you are using a 64-bit machine.

---

This installs a cross compiler, but you need to set up a couple of things so that it can be found. At the command prompt, enter **arm-<TAB><TAB>** to see what was installed.

```
host$ arm-<TAB><TAB>
arm-linux-gnueabi-addr2line      arm-linux-gnueabi-gcc-nm      arm-
↳ linux-gnueabi-ld.bfd
arm-linux-gnueabi-ar             arm-linux-gnueabi-gcc-nm-11   arm-
↳ linux-gnueabi-ld.gold
arm-linux-gnueabi-as            arm-linux-gnueabi-gcc-ranlib  arm-
↳ linux-gnueabi-lto-dump-11
arm-linux-gnueabi-c++filt       arm-linux-gnueabi-gcc-ranlib-11 arm-
↳ linux-gnueabi-nm
arm-linux-gnueabi-cpp           arm-linux-gnueabi-gcov        arm-
↳ linux-gnueabi-objcopy
arm-linux-gnueabi-cpp-11       arm-linux-gnueabi-gcov-11     arm-
↳ linux-gnueabi-objdump
arm-linux-gnueabi-dwp           arm-linux-gnueabi-gcov-dump   arm-
↳ linux-gnueabi-ranlib
arm-linux-gnueabi-elfedit       arm-linux-gnueabi-gcov-dump-11 arm-
↳ linux-gnueabi-readelf
```

(continues on next page)

(continued from previous page)

```

arm-linux-gnueabi-hf-gcc          arm-linux-gnueabi-hf-gcov-tool    arm-
↳linux-gnueabi-hf-size
arm-linux-gnueabi-hf-gcc-11      arm-linux-gnueabi-hf-gcov-tool-11 arm-
↳linux-gnueabi-hf-strings
arm-linux-gnueabi-hf-gcc-ar      arm-linux-gnueabi-hf-gprof        arm-
↳linux-gnueabi-hf-strip
arm-linux-gnueabi-hf-gcc-ar-11   arm-linux-gnueabi-hf-ld

```

What you see are all the cross-development tools.

## 7.7 Setting Up Variables

Now, set up a couple of variables to know which compiler you are using:

```

host$ export ARCH=arm
host$ export CROSS_COMPILE=arm-linux-gnueabi-hf-

```

These lines set up the standard environmental variables so that you can determine which cross-development tools to use. Test the cross compiler by adding [Simple helloWorld.c to test cross compiling \(helloWorld.c\)](#) to a file named `_helloWorld.c_`.

Listing 7.3: Simple helloWorld.c to test cross compiling (helloWorld.c)

```

1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     printf("Hello, World! \n");
5 }

```

helloWorld.c

You can then cross-compile by using the following commands:

```

host$ ${CROSS_COMPILE}gcc helloWorld.c
host$ file a.out
a.out: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.31,
BuildID[sha1]=0x10182364352b9f3cb15d1aa61395aeede11a52ad, not stripped

```

The `file` command shows that `a.out` was compiled for an ARM processor.

## 7.8 Applying Patches

### 7.8.1 Problem

You have a patch file that you need to apply to the kernel.

### 7.8.2 Solution

[Simple kernel patch file \(hello.patch\)](#) shows a patch file that you can use on the kernel.



Listing 7.4: Simple kernel patch file (hello.patch)

```

1 From eaf4f7ea7d540bc8bb57283a8f68321ddb4401f4 Mon Sep 17 00:00:00 2001
2 From: Jason Kridner <jdk@ti.com>
3 Date: Tue, 12 Feb 2013 02:18:03 +0000
4 Subject: [PATCH] hello: example kernel modules
5
6 ---
7  hello/Makefile      |      7 ++++++
8  hello/hello.c       |     18 ++++++
9  2 files changed, 25 insertions(+), 0 deletions(-)
10 create mode 100644 hello/Makefile
11 create mode 100644 hello/hello.c
12
13 diff --git a/hello/Makefile b/hello/Makefile
14 new file mode 100644
15 index 0000000..4b23da7
16 --- /dev/null
17 +++ b/hello/Makefile
18 @@ -0,0 +1,7 @@
19 +obj-m := hello.o
20 +
21 +PWD := $(shell pwd)
22 +KDIR := ${PWD}/..
23 +
24 +default:
25 +    make -C $(KDIR) SUBDIRS=$(PWD) modules
26 diff --git a/hello/hello.c b/hello/hello.c
27 new file mode 100644
28 index 0000000..157d490
29 --- /dev/null
30 +++ b/hello/hello.c
31 @@ -0,0 +1,22 @@
32 +#include <linux/module.h>          /* Needed by all modules */
33 +#include <linux/kernel.h>        /* Needed for KERN_INFO */
34 +#include <linux/init.h>          /* Needed for the macros */
35 +
36 +static int __init hello_start(void)
37 +{
38 +    printk(KERN_INFO "Loading hello module...\n");
39 +    printk(KERN_INFO "Hello, World!\n");
40 +    return 0;
41 +}
42 +
43 +static void __exit hello_end(void)
44 +{
45 +    printk(KERN_INFO "Goodbye Boris\n");
46 +}
47 +
48 +module_init(hello_start);
49 +module_exit(hello_end);
50 +
51 +MODULE_AUTHOR("Boris Houndleroy");
52 +MODULE_DESCRIPTION("Hello World Example");
53 +MODULE_LICENSE("GPL");

```

hello.patch

Here's how to use it:

- Install the kernel sources ([Compiling the Kernel](#)).
- Change to the kernel directory (+cd ti-linux-kernel-dev/KERNEL+).
- Add [Simple kernel patch file \(hello.patch\)](#) to a file named hello.patch in the

ti-linux-kernel-dev/KERNEL directory.

- Run the following commands:

```
host$ cd ti-linux-kernel-dev/KERNEL
host$ patch -p1 < hello.patch
patching file hello/Makefile
patching file hello/hello.c
```

The output of the `patch` command appries you of what it's doing. Look in the `hello` directory to see what was created:

```
host$ cd hello
host$ ls
hello.c  Makefile
```

[Building and Installing Kernel Modules](#) shows how to build and install a module, and [Creating Your Own Patch File](#) shows how to create your own patch file.

## 7.9 Creating Your Own Patch File

### 7.9.1 Problem

You made a few changes to the kernel, and you want to share them with your friends.

### 7.9.2 Solution

Create a patch file that contains just the changes you have made. Before making your changes, check out a new branch:

```
host$ cd ti-linux-kernel-dev/KERNEL
host$ git status
# On branch master
nothing to commit (working directory clean)
```

Good, so far no changes have been made. Now, create a new branch:

```
host$ git checkout -b hello1
host$ git status
# On branch hello1
nothing to commit (working directory clean)
```

You've created a new branch called `hello1` and checked it out. Now, make whatever changes to the kernel you want. I did some work with a simple character driver that we can use as an example:

```
host$ cd ti-linux-kernel-dev/KERNEL/drivers/char/
host$ git status
# On branch hello1
# Changes not staged for commit:
#   (use "git add file..." to update what will be committed)
#   (use "git checkout -- file..." to discard changes in working directory)
#
#   modified:   Kconfig
#   modified:   Makefile
#
# Untracked files:
#   (use "git add file..." to include in what will be committed)
#
```

(continues on next page)

(continued from previous page)

```
# examples/  
no changes added to commit (use "git add" and/or "git commit -a")
```

Add the files that were created and commit them:

```
host$ git add Kconfig Makefile examples  
host$ git status  
# On branch hello1  
# Changes to be committed:  
#   (use "git reset HEAD file..." to unstage)  
#  
#   modified:   Kconfig  
#   modified:   Makefile  
#   new file:   examples/Makefile  
#   new file:   examples/hello1.c  
#  
host$ git commit -m "Files for hello1 kernel module"  
[hello1 99346d5] Files for hello1 kernel module  
4 files changed, 33 insertions(+)  
create mode 100644 drivers/char/examples/Makefile  
create mode 100644 drivers/char/examples/hello1.c
```

Finally, create the patch file:

```
host$ git format-patch master --stdout &gt; hello1.patch
```

## Chapter 8

# Real-Time I/O

Sometimes, when BeagleBone Black interacts with the physical world, it needs to respond in a timely manner. For example, your robot has just detected that one of the driving motors needs to turn a bit faster. Systems that can respond quickly to a real event are known as `real-time` systems. There are two broad categories of real-time systems: soft and hard.

In a `soft real-time` system, the real-time requirements should be met `most` of the time, where `most` depends on the system. A video playback system is a good example. The goal might be to display 60 frames per second, but it doesn't matter much if you miss a frame now and then. In a 100 percent `hard real-time` system, you can never fail to respond in time. Think of an airbag deployment system on a car. You can't even be 50 ms late.

Systems running Linux generally can't do 100 percent hard real-time processing, because Linux gets in the way. However, the Bone has an ARM processor running Linux and two additional 32-bit programmable real-time units (PRUs [Ti AM33XX PRUSSv2](#)) available to do real-time processing. Although the PRUs can achieve 100 percent hard real-time, they take some effort to use.

This chapter shows several ways to do real-time input/output (I/O), starting with the effortless, yet slower JavaScript and moving up with increasing speed (and effort) to using the PRUs.

---

**Note:** In this chapter, as in the others, we assume that you are logged in as `debian` (as indicated by the `bone$` prompt). This gives you quick access to the general-purpose input/output (GPIO) ports but you may have to use `sudo` some times.

---

## 8.1 I/O with Python and JavaScript

### 8.1.1 Problem

You want to read an input pin and write it to the output as quickly as possible with JavaScript.

### 8.1.2 Solution

[Reading the Status of a Pushbutton or Magnetic Switch \(Passive On/Off Sensor\)](#) shows how to read a pushbutton switch and [Toggling an External LED](#) controls an external LED. This recipe combines the two to read the switch and turn on the LED in response to it. To make this recipe, you will need:

- Breadboard and jumper wires
- Pushbutton switch
- 220R resistor

- LED

Wire up the pushbutton and LED as shown in [Diagram for wiring a pushbutton and LED with the LED attached to P9\\_14](#).

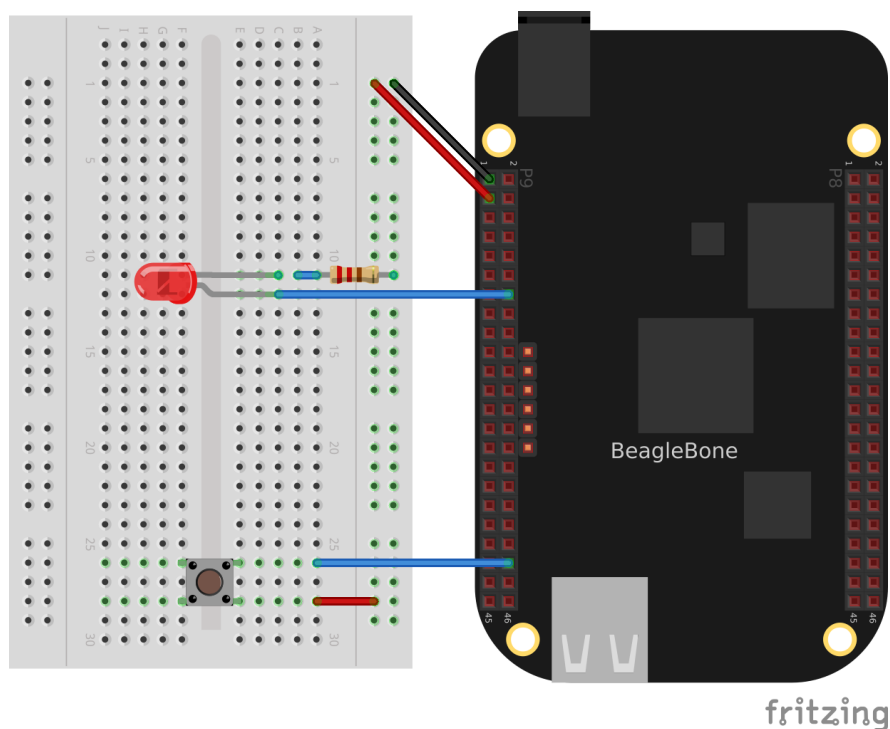


Fig. 8.1: Diagram for wiring a pushbutton and LED with the LED attached to P9\_14

The code in [Monitoring a pushbutton \(pushLED.py\)](#) reads GPIO port P9\_42, which is attached to the pushbutton, and turns on the LED attached to P9\_12 when the button is pushed.

### Python

Listing 8.1: Monitoring a pushbutton (pushLED.py)

```

1  #!/usr/bin/env python
2  # //////////////////////////////////////
3  # //      pushLED.py
4  # //      Blinks an LED attached to P9_12 when the button at P9_42 is
   ↪ pressed
5  # //      Wiring:
6  # //      Setup:
7  # //      See:
8  # //////////////////////////////////////
9  import time
10 import os
11
12 ms = 50    # Read time in ms
13
14 LED="50"   # Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1,
   ↪ line 18 maps to 50
15 button="7" # P9_42 maps to 7
16
17 GPIOPATH="/sys/class/gpio/"
18
19 # Make sure LED is exported

```

(continues on next page)

(continued from previous page)

```

20 if (not os.path.exists(GPIOPATH+"gpio"+LED)):
21     f = open(GPIOPATH+"export", "w")
22     f.write(LED)
23     f.close()
24
25 # Make it an output pin
26 f = open(GPIOPATH+"gpio"+LED+"/direction", "w")
27 f.write("out")
28 f.close()
29
30 # Make sure button is exported
31 if (not os.path.exists(GPIOPATH+"gpio"+button)):
32     f = open(GPIOPATH+"export", "w")
33     f.write(button)
34     f.close()
35
36 # Make it an output pin
37 f = open(GPIOPATH+"gpio"+button+"/direction", "w")
38 f.write("in")
39 f.close()
40
41 # Read every ms
42 fin = open(GPIOPATH+"gpio"+button+"/value", "r")
43 fout = open(GPIOPATH+"gpio"+LED+"/value", "w")
44
45 while True:
46     fin.seek(0)
47     fout.seek(0)
48     fout.write(fin.read())
49     time.sleep(ms/1000)

```

pushLED.py

c

Listing 8.2: Code for reading a switch and blinking an LED (pushLED.c)

```

1  //////////////////////////////////////
2  //      blinkLED.c
3  //      Blinks the P9_14 pin based on the P9_42 pin
4  //      Wiring:
5  //      Setup:
6  //      See:
7  //////////////////////////////////////
8  #include <stdio.h>
9  #include <string.h>
10 #include <unistd.h>
11 #define MAXSTR 100
12
13 int main() {
14     FILE *fpbutton, *fpLED;
15     char LED[] = "50"; // Look up P9.14 using gpioinfo | grep -e chip -e P9.
16     ↪14. chip 1, line 18 maps to 50
17     char button[] = "7"; // Look up P9.42 using gpioinfo | grep -e chip -e P9.
18     ↪42. chip 0, line 7 maps to 7
19     char GPIOPATH[] = "/sys/class/gpio";
20     char path[MAXSTR] = "";
21
22     // Make sure LED is exported
23     sprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", LED);

```

(continues on next page)

(continued from previous page)

```

22  if (!access(path, F_OK) == 0) {
23      snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
24      fpLED = fopen(path, "w");
25      fprintf(fpLED, "%s", LED);
26      fclose(fpLED);
27  }
28
29  // Make it an output LED
30  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", LED, "/direction");
31  fpLED = fopen(path, "w");
32  fprintf(fpLED, "out");
33  fclose(fpLED);
34
35  // Make sure bbuttonutton is exported
36  snprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", button);
37  if (!access(path, F_OK) == 0) {
38      snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
39      fpbutton = fopen(path, "w");
40      fprintf(fpbutton, "%s", button);
41      fclose(fpbutton);
42  }
43
44  // Make it an input button
45  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", button, "/direction
→");
46  fpbutton = fopen(path, "w");
47  fprintf(fpbutton, "in");
48  fclose(fpbutton);
49
50  // I don't know why I can open the LED outside the loop and use fseek_
→before
51  // each read, but I can't do the same for the button. It appears it needs
52  // to be opened every time.
53  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", LED, "/value");
54  fpLED = fopen(path, "w");
55
56  char state = '0';
57
58  while (1) {
59      snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", button, "/value");
60      fpbutton = fopen(path, "r");
61      fseek(fpLED, 0L, SEEK_SET);
62      fscanf(fpbutton, "%c", &state);
63      printf("state: %c\n", state);
64      fprintf(fpLED, "%c", state);
65      fclose(fpbutton);
66      usleep(250000); // sleep time in microseconds
67  }
68 }

```

```

bone$ gcc -o pushLED pushLED.c -lgiopid
bone$ ./pushLED

```

```

1
1
0
0
0
1
^C

```

pushLED.c

## JavaScript

Listing 8.3: Monitoring a pushbutton (pushLED.js)

```

1  #!/usr/bin/env node
2  //////////////////////////////////////
3  //      pushLED.js
4  //      Blinks an LED attached to P9_12 when the button at P9_42 is pressed
5  //      Wiring:
6  //      Setup:
7  //      See:
8  //////////////////////////////////////
9  const fs = require("fs");
10
11  const ms = 500 // Read time in ms
12
13  const LED="50"; // Look up P9.14 using gpioinfo | grep -e chip -e P9.14. ↵
14  ↵chip 1, line 18 maps to 50
15  const button="7"; // P9_42 maps to 7
16
17  GPIOPATH="/sys/class/gpio/";
18
19  // Make sure LED is exported
20  if(!fs.existsSync(GPIOPATH+"gpio"+LED)) {
21      fs.writeFileSync(GPIOPATH+"export", LED);
22  }
23  // Make it an output pin
24  fs.writeFileSync(GPIOPATH+"gpio"+LED+"/direction", "out");
25
26  // Make sure button is exported
27  if(!fs.existsSync(GPIOPATH+"gpio"+button)) {
28      fs.writeFileSync(GPIOPATH+"export", button);
29  }
30  // Make it an input pin
31  fs.writeFileSync(GPIOPATH+"gpio"+button+"/direction", "in");
32
33  // Read every ms
34  setInterval(flashLED, ms);
35
36  function flashLED() {
37      var data = fs.readFileSync(GPIOPATH+"gpio"+button+"/value").slice(0, -1);
38      console.log('data = ' + data);
39      fs.writeFileSync(GPIOPATH+"gpio"+LED+"/value", data);
40  }

```

pushLED.js

Add the code to a file named pushLED.py and run it by using the following commands:

```

bone$ chmod *x pushLED.py
bone$ ./pushLED.py
Hit ^C to stop
0
0
1
1
^C

```

Press ^C (Ctrl-C) to stop the code.

## 8.2 I/O with devmem2



### 8.2.1 Problem

Your C code isn't responding fast enough to the input signal. You want to read the GPIO registers directly.

### 8.2.2 Solution

The solution is to use a simple utility called *devmem2*, with which you can read and write registers from the command line.

**Warning:** This solution is much more involved than the previous ones. You need to understand binary and hex numbers and be able to read the [AM335x Technical Reference Manual](#).

First, download and install *devmem2*:

```
bone$ wget http://bootlin.com/pub/mirror/devmem2.c
bone$ gcc -o devmem2 devmem2.c
bone$ sudo mv devmem2 /usr/bin
```

This solution will read a pushbutton attached to *P9\_42* and flash an LED attached to *P9\_13*. Note that this is a change from the previous solutions that makes the code used here much simpler. Wire up your Bone as shown in [Diagram for wiring a pushbutton and LED with the LED attached to P9\\_13](#).

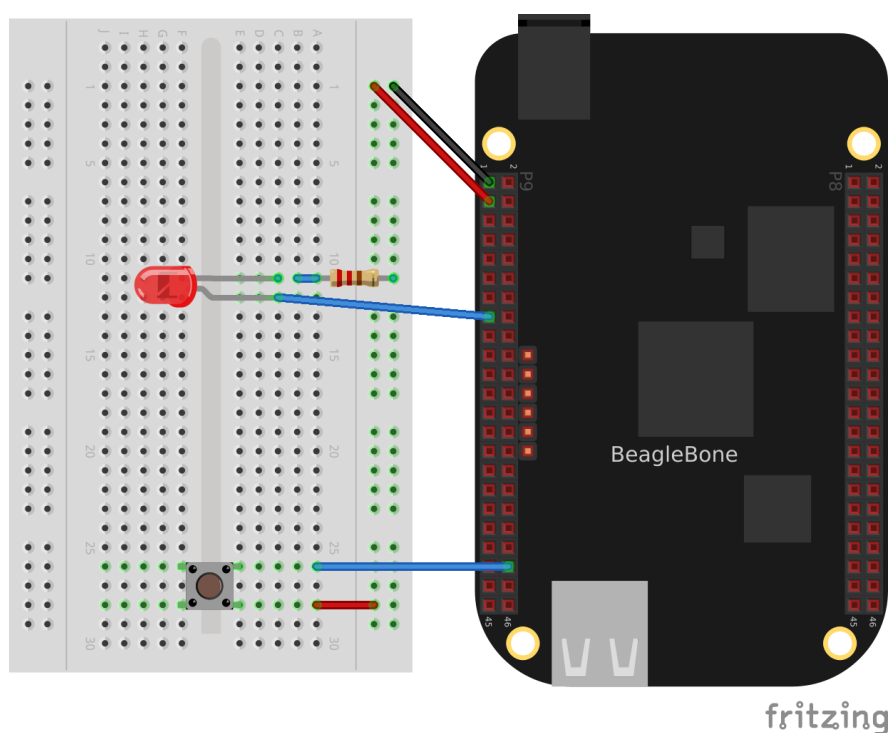


Fig. 8.2: Diagram for wiring a pushbutton and LED with the LED attached to P9\_13

Now, flash the LED attached to *P9\_13* using the Linux *sysfs* interface ([Controlling GPIOs by Using SYSFS Entries](#)). To do this, first look up which GPIO number *P9\_13* is attached to by referring to [Mapping from header pin to internal GPIO number](#). Finding *P9\_13* at GPIO 31, export GPIO 31 and make it an output:

```
bone$ cd /sys/class/gpio/
bone$ echo 31 > export
bone$ cd gpio31
bone$ echo out > direction
```

(continues on next page)

(continued from previous page)

```
bone$ echo 1 > value
bone$ echo 0 > value
```

The LED will turn on when *1* is echoed into *value* and off when *0* is echoed.

Now that you know the LED is working, look up its memory address. This is where things get very detailed. First, download the [AM335x Technical Reference Manual](#). Look up *GPIO* in the Memory Map chapter (sensors). Table 2-2 indicates that *GPIO* starts at address *0x44E0\_7000*. Then go to Section 25.4.1, “GPIO Registers.” This shows that *GPIO\_DATAIN* has an offset of *0x138*, *GPIO\_CLEARDATAOUT* has an offset of *0x190*, and *GPIO\_SETDATAOUT* has an offset of *0x194*.

This means you read from address  $0x44E0_7000 + 0x138 = 0x44E0_7138$  to see the status of the LED:

```
bone$ sudo devmem2 0x44E07138
/dev/mem opened.
Memory mapped at address 0xb6f8e000.
Value at address 0x44E07138 (0xb6f8e138): 0xC000C404
```

The returned value *0xC000C404* (*1100 0000 0000 0000 1100 0100 0000 0100* in binary) has bit 31 set to *1*, which means the LED is on. Turn the LED off by writing *0x80000000* (*1000 0000 0000 0000 0000 0000 0000 0000* binary) to the *GPIO\_CLEARDATA* register at  $0x44E0_7000 + 0x190 = 0x44E0_7190$ :

```
bone$ sudo devmem2 0x44E07190 w 0x80000000
/dev/mem opened.
Memory mapped at address 0xb6fd7000.
Value at address 0x44E07190 (0xb6fd7190): 0x80000000
Written 0x80000000; readback 0x0
```

The LED is now off.

You read the pushbutton switch in a similar way. [Mapping from header pin to internal GPIO number](#) says *P9\_42* is GPIO 7, which means bit 7 is the state of *P9\_42*. The *devmem2* in this example reads *0x0*, which means all bits are *0*, including GPIO 7. Section 25.4.1 of the Technical Reference Manual instructs you to use offset *0x13C* to read *GPIO\_DATAOUT*. Push the pushbutton and run *devmem2*:

```
bone$ sudo devmem2 0x44e07138
/dev/mem opened.
Memory mapped at address 0xb6fe2000.
Value at address 0x44E07138 (0xb6fe2138): 0x4000C484
```

Here, bit 7 is set in *0x4000C484*, showing the button is pushed.

This is much more tedious than the previous methods, but it’s what’s necessary if you need to minimize the time to read an input. [I/O with C and mmap\(\)](#) shows how to read and write these addresses from C.

## 8.3 I/O with C and mmap()

### 8.3.1 Problem

Your C code isn’t responding fast enough to the input signal.

### 8.3.2 Solution

In smaller processors that aren’t running an operating system, you can read and write a given memory address directly from C. With Linux running on Bone, many of the memory locations are hardware protected, so you can’t accidentally access them directly.

This recipe shows how to use *mmap()* (memory map) to map the GPIO registers to an array in C. Then all you need to do is access the array to read and write the registers.

**Warning:** This solution is much more involved than the previous ones. You need to understand binary and hex numbers and be able to read the AM335x Technical Reference Manual.

This solution will read a pushbutton attached to *P9\_42* and flash an LED attached to *P9\_13*. Note that this is a change from the previous solutions that makes the code used here much simpler.

**Tip:** See *I/O with devmem2* for details on mapping the GPIO numbers to memory addresses.

Add the code in *Memory address definitions (pushLEDmmap.h)* to a file named `pushLEDmmap.h`.

Listing 8.4: Memory address definitions (pushLEDmmap.h)

```

1 // From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
2 // -through-dev-mem
3 // user contributions licensed under cc by-sa 3.0 with attribution required
4 // http://creativecommons.org/licenses/by-sa/3.0/
5 // http://blog.stackoverflow.com/2009/06/attribution-required/
6 // Author: madscientist159 (http://stackoverflow.com/users/3000377/
  →madscientist159)
7
8 #ifndef _BEAGLEBONE_GPIO_H_
9 #define _BEAGLEBONE_GPIO_H_
10
11 #define GPIO0_START_ADDR 0x44e07000
12 #define GPIO0_END_ADDR 0x44e08000
13 #define GPIO0_SIZE (GPIO0_END_ADDR - GPIO0_START_ADDR)
14
15 #define GPIO1_START_ADDR 0x4804C000
16 #define GPIO1_END_ADDR 0x4804D000
17 #define GPIO1_SIZE (GPIO1_END_ADDR - GPIO1_START_ADDR)
18
19 #define GPIO2_START_ADDR 0x41A4C000
20 #define GPIO2_END_ADDR 0x41A4D000
21 #define GPIO2_SIZE (GPIO2_END_ADDR - GPIO2_START_ADDR)
22
23 #define GPIO3_START_ADDR 0x41A4E000
24 #define GPIO3_END_ADDR 0x41A4F000
25 #define GPIO3_SIZE (GPIO3_END_ADDR - GPIO3_START_ADDR)
26
27 #define GPIO_DATAIN 0x138
28 #define GPIO_SETDATAOUT 0x194
29 #define GPIO_CLEARDATAOUT 0x190
30
31 #define GPIO_03 (1<<3)
32 #define GPIO_07 (1<<7)
33 #define GPIO_31 (1<<31)
34 #define GPIO_60 (1<<28)
35 #endif

```

`pushLEDmmap.h`

Add the code in *Code for directly reading memory addresses (pushLEDmmap.c)* to a file named `pushLEDmmap.c`.

Listing 8.5: Code for directly reading memory addresses (pushLEDmmap.c)

```

1 // From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
2 // -through-dev-mem
3 // user contributions licensed under cc by-sa 3.0 with attribution required

```

(continues on next page)

(continued from previous page)

```

4 // http://creativecommons.org/licenses/by-sa/3.0/
5 // http://blog.stackoverflow.com/2009/06/attribution-required/
6 // Author: madscientist159 (http://stackoverflow.com/users/3000377/
  ↪madscientist159)
7 //
8 // Read one gpio pin and write it out to another using mmap.
9 // Be sure to set -O3 when compiling.
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <sys/mman.h>
13 #include <fcntl.h>
14 #include <signal.h> // Defines signal-handling functions (i.e. trap Ctrl-
  ↪C)
15 #include "pushLEDmmap.h"
16
17 // Global variables
18 int keepgoing = 1; // Set to 0 when Ctrl-c is pressed
19
20 // Callback called when SIGINT is sent to the process (Ctrl-C)
21 void signal_handler(int sig) {
22     printf( "\nCtrl-C pressed, cleaning up and exiting...\n" );
23     keepgoing = 0;
24 }
25
26 int main(int argc, char *argv[]) {
27     volatile void *gpio_addr;
28     volatile unsigned int *gpio_datain;
29     volatile unsigned int *gpio_setdataout_addr;
30     volatile unsigned int *gpio_cleardataout_addr;
31
32     // Set the signal callback for Ctrl-C
33     signal(SIGINT, signal_handler);
34
35     int fd = open("/dev/mem", O_RDWR);
36
37     printf("Mapping %X - %X (size: %X)\n", GPIO0_START_ADDR, GPIO0_END_ADDR,
38           GPIO0_SIZE);
39
40     gpio_addr = mmap(0, GPIO0_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
41                    GPIO0_START_ADDR);
42
43     gpio_datain = gpio_addr + GPIO_DATAIN;
44     gpio_setdataout_addr = gpio_addr + GPIO_SETDATAOUT;
45     gpio_cleardataout_addr = gpio_addr + GPIO_CLEARDATAOUT;
46
47     if(gpio_addr == MAP_FAILED) {
48         printf("Unable to map GPIO\n");
49         exit(1);
50     }
51     printf("GPIO mapped to %p\n", gpio_addr);
52     printf("GPIO SETDATAOUTADDR mapped to %p\n", gpio_setdataout_addr);
53     printf("GPIO CLEARDATAOUT mapped to %p\n", gpio_cleardataout_addr);
54
55     printf("Start copying GPIO_07 to GPIO_31\n");
56     while(keepgoing) {
57         if(*gpio_datain & GPIO_07) {
58             *gpio_setdataout_addr= GPIO_31;
59         } else {
60             *gpio_cleardataout_addr = GPIO_31;
61         }
62         //usleep(1);

```

(continues on next page)

```
63     }
64
65     munmap((void *)gpio_addr, GPIO0_SIZE);
66     close(fd);
67     return 0;
68 }
```

pushLEDmmap.c

Now, compile and run the code:

```
bone$ gcc -O3 pushLEDmmap.c -o pushLEDmmap
bone$ sudo ./pushLEDmmap
Mapping 44E07000 - 44E08000 (size: 1000)
GPIO mapped to 0xb6fac000
GPIO SETDATAOUTADDR mapped to 0xb6fac194
GPIO CLEARDATAOUT mapped to 0xb6fac190
Start copying GPIO_07 to GPIO_31
^C
Ctrl-C pressed, cleaning up and exiting...
```

The code is in a tight *while* loop that checks the status of GPIO 7 and copies it to GPIO 31.

## 8.4 Tighter Delay Bounds with the PREEMPT\_RT Kernel

### 8.4.1 Problem

You want to run real-time processes on the Beagle, but the OS is slowing things down.

### 8.4.2 Solution

The Kernel can be compiled with PREEMPT\_RT enabled which reduces the delay from when a thread is scheduled to when it runs.

Switching to a PREEMPT\_RT kernel is rather easy, but be sure to follow the steps in the Discussion to see how much the latencies are reduced.

- First see which kernel you are running:

```
bone$ uname -a
Linux breadboard-home 5.10.120-ti-r47 #1bullseye SMP PREEMPT Tue Jul 12
↪18:59:38 UTC 2022 armv7l GNU/Linux
```

I'm running a 5.10 kernel. Remember the whole string, *5.10.120-ti-r47*, for later.

- Go to [kernel update](#) and look for *5.10*.

In *The regular and RT kernels* you see the regular kernel on top and the RT below.

- We want the RT one.

```
bone$ sudo apt update
bone$ sudo apt install bbb.io-kernel-5.10-ti-rt-am335x
```

---

**Note:** Use the *am57xx* if you are using the BeagleBoard AI or AI64.

---

- Before rebooting, edit */boot/uEnv.txt* to start with:

## v5.10.x-ti branch:

```
bbb.io-kernel-5.10-ti-am335x - BeagleBoard.org 5.10-ti for am335x
bbb.io-kernel-5.10-ti-am57xx - BeagleBoard.org 5.10-ti for am57xx
```

## v5.10.x-ti-rt branch:

```
bbb.io-kernel-5.10-ti-rt-am335x - BeagleBoard.org 5.10-ti-rt for am335x
bbb.io-kernel-5.10-ti-rt-am57xx - BeagleBoard.org 5.10-ti-rt for am57xx
```

Fig. 8.3: The regular and RT kernels

```
#Docs: http://elinux.org/Beagleboard:U-boot_partitioning_layout_2.0

# uname_r=5.10.120-ti-r47
uname_r=5.10.120-ti-rt-r47
#uuid=
#dtb=
```

`uname_r` tells the boot loader which kernel to boot. Here we've commented out the regular kernel and left in the RT kernel. Next time you boot you'll be running the RT kernel. Don't reboot just yet. Let's gather some latency data first.

Bootlin's [preempt\\_rt workshop](#) looks like a good workshop on PREEMPT RT. Their slides say:

- One way to implement a multi-task Real-Time Operating System is to have a preemptible system
- Any task can be interrupted at any point so that higher priority tasks can run
- Userspace preemption already exists in Linux
- The Linux Kernel also supports real-time scheduling policies
- However, code that runs in kernel mode isn't fully preemptible
- The Preempt-RT patch aims at making all code running in kernel mode preemptible

The workshop goes into many details on how to get real-time performance on Linux. Checkout their [slides](#) and [labs](#). Though you can skip the first lab since we present a simpler way to get the RT kernel running.

## 8.5 Cyclictest

`cyclictest` is one tool for measuring the latency from when a thread is scheduled and when it runs. The `code/rt` directory in the git repo has some scripts for gathering latency data and plotting it. Here's how to run the scripts.

- First look in [rt/install.sh](#) to see what to install.

Listing 8.6: `rt/install.sh`

```
1 sudo apt install rt-tests
2 # You can run gnuplot on the host
3 sudo apt install gnuplot
```

```
rt/install.sh
```

- Open up another window and start something that will create a load on the Bone, then run the following:

```
bone$ time sudo ./hist.gen > nort.hist
```

*hist.gen* shows what's being run. It defaults to 100,000 loops, so it takes a while. The data is saved in *nort.hist*, which stands for no RT histogram.

Listing 8.7: hist.gen

```
1 #!/bin/sh
2 # This code is from Julia Cartwright julia@kernel.org
3
4 cyclicttest -m -S -p 90 -h 400 -l "${1:-100000}"
```

```
rt/hist.gen
```

---

**Note:** If you get an error:

Unable to change scheduling policy! Probably missing capabilities, either run as root or increase RLIMIT\_RTPRIO limits

try running *./setup.sh*. If that doesn't work try:

```
bone$ sudo bash
bone# ulimit -r unlimited
bone# ./hist.gen > nort.hist
bone# exit
```

- Now you are ready to reboot into the RT kernel and run the test again.

```
bone$ reboot
```

- After rebooting:

```
bone$ uname -a
Linux breadboard-home 5.10.120-ti-rt-r47 #1bullseye SMP PREEMPT RT Tue Jul
↪12 18:59:38 UTC 2022 armv7l GNU/Linux
```

Congratulations you are running the RT kernel.

---

**Note:** If the Beagle appears to be running (the LEDs are flashing) but you are having trouble connecting via *ssh 192.168.7.2*, you can try connecting using the approach shown in [Viewing and Debugging the Kernel and u-boot Messages at Boot Time](#).

Now run the script again (note it's being saved in *rt.hist* this time.)

```
bone$ time sudo ./hist.gen > rt.hist
```

---

**Note:** At this point you can edit */boot/uEnt.txt* to boot the non RT kernel and reboot.

Now it's time to plot the results.

```
bone$ gnuplot hist.plt
```

This will generate the file *cyclicttest.png* which contains your plot. It should look like:

Notice the NON-RT data have much longer latencies. They may not happen often (fewer than 10 times in each bin), but they are occurring and may be enough to miss a real-time deadline.

The PREEMPT-RT times are all under a 150 us.

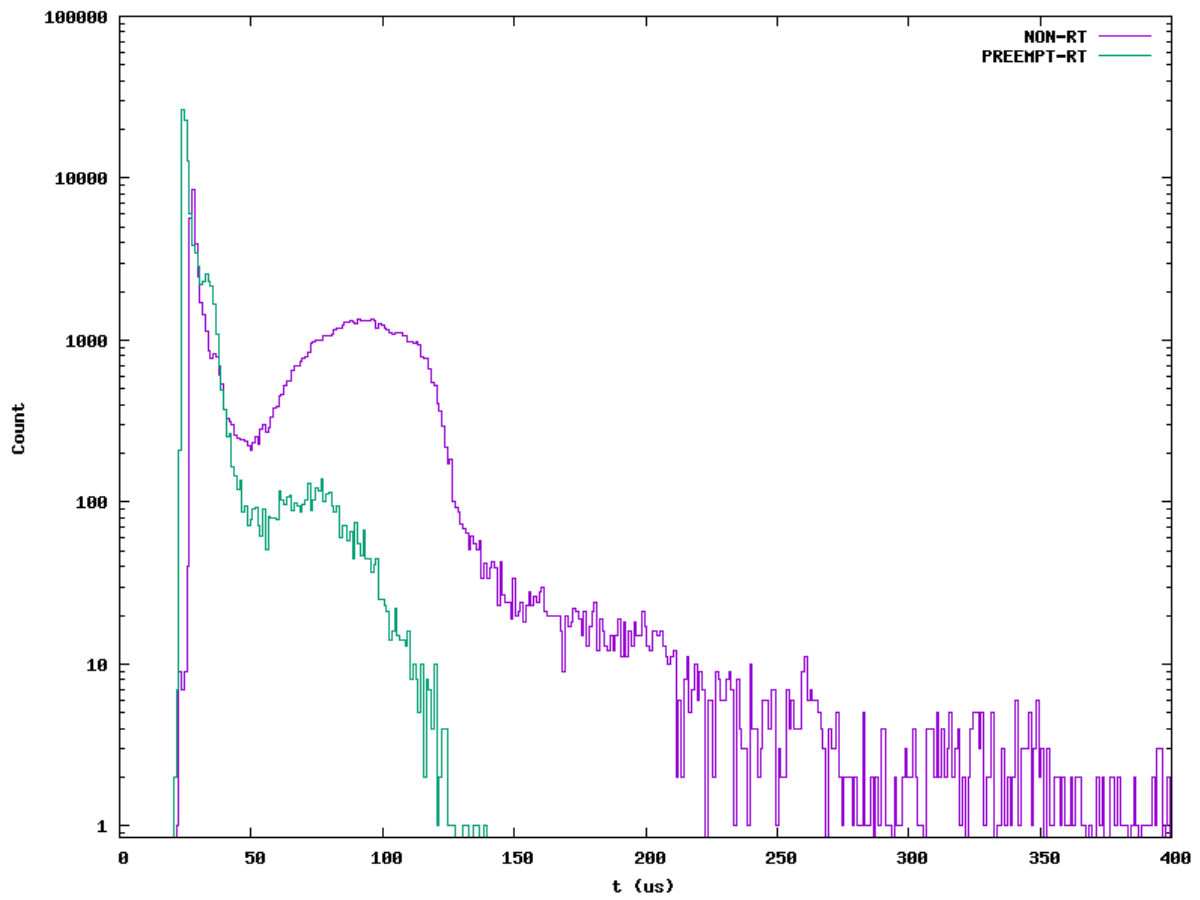


Fig. 8.4: Histogram of Non-RT and RT kernels running cyclictest



## 8.6 I/O with simpPRU

### 8.6.1 Problem

You require better timing than running C on the ARM can give you.

### 8.6.2 Solution

The AM335x processor on the Bone has an ARM processor that is running Linux, but it also has two 32-bit PRUs that are available for processing I/O. It takes a fair amount of understanding to program the PRU. Fortunately, [simpPRU](#) is an intuitive language for PRU which compiles down to PRU C. This solution shows how to use it.

## 8.7 Background

simpPRU

## Chapter 9

# Capes

Previous chapters of this book show a variety of ways to interface BeagleBone Black to the physical world by using a breadboard and wiring to the +P8+ and +P9+ headers. This is a great approach because it's easy to modify your circuit to debug it or try new things. At some point, though, you might want a more permanent solution, either because you need to move the Bone and you don't want wires coming loose, or because you want to share your hardware with the masses.

You can easily expand the functionality of the Bone by adding a *cape*. A cape is simply a board—often a printed circuit board (PCB) that connects to the **P8** and **P9** headers and follows a few standard pin usages. You can stack up to four capes onto the Bone. Capes can range in size covering a few pins to much larger than the Bone.

---

**Todo:** Add cape examples of various sizes

---

This chapter shows how to attach a couple of capes, move your design to a protoboard, then to a PCB, and finally on to mass production.

---

**Todo:** Update display cape example

---

### 9.1 Connecting Multiple Capes

#### 9.1.1 Problem

You want to use more than one cape at a time.

#### 9.1.2 Solution

First, look at each cape that you want to stack mechanically. Are they all using stacking headers like the ones shown in [Stacking headers](#)? No more than one should be using non-stacking headers.

Note that larger LCD panels might provide expansion headers, such as the ones shown in [LCD Backside](#), rather than the stacking headers, and that those can also be used for adding additional capes.

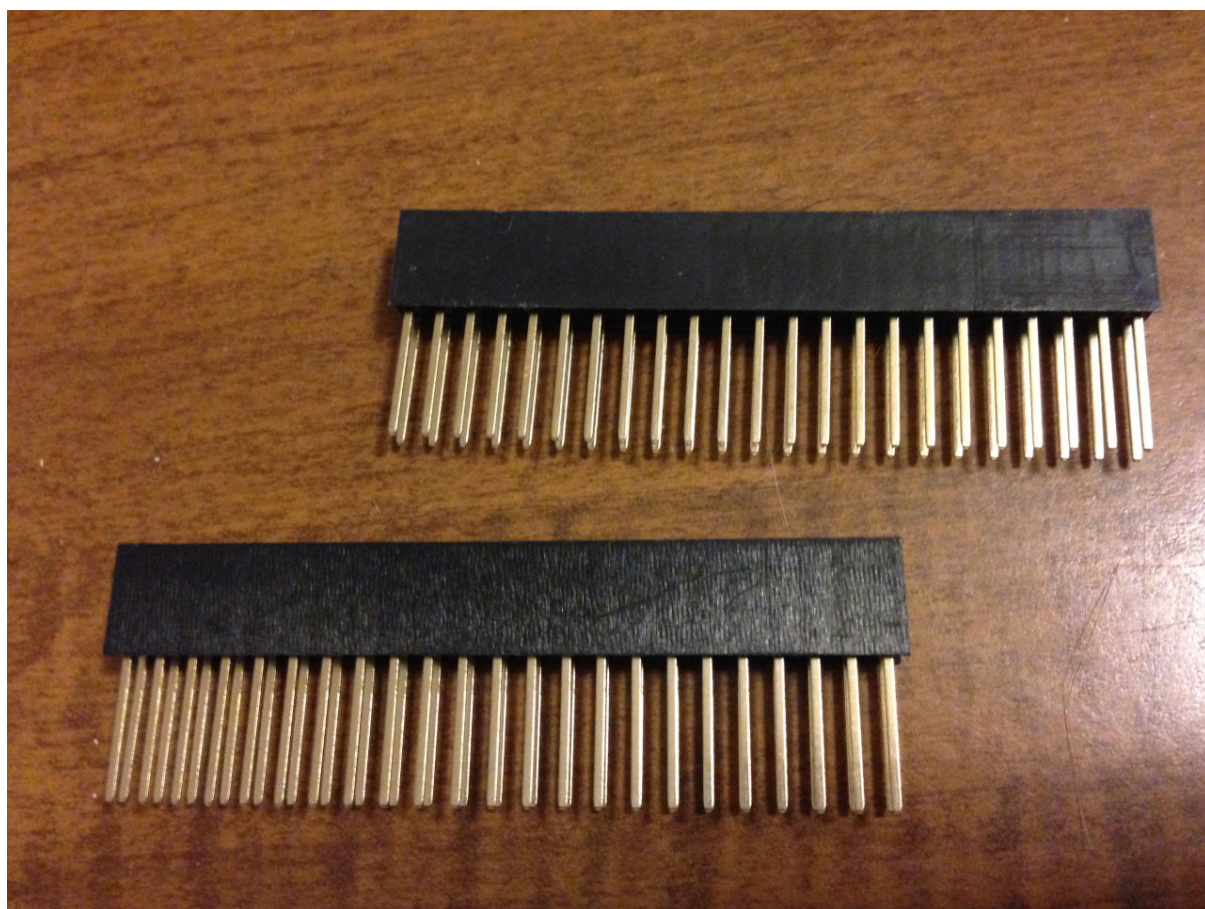


Fig. 9.1: Stacking headers

---

## 9.2 LCD Backside

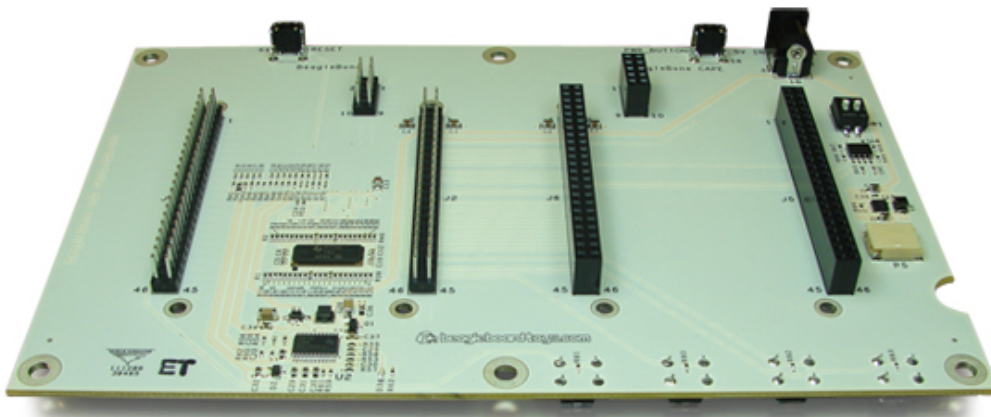
---

**Note:** Back side of LCD7 cape, *LCD Backside* was originally posted by CircuitCo at <http://elinux.org/File:BeagleBone-LCD-Backside.jpg> under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

---

**Note:** #TODO# One of the 4D Systems LCD capes would make a better example for an LCD cape. The CircuitCo cape is no longer available.

---



Next, take a note of each pin utilized by each cape. The *BeagleBone Capes catalog* provides a graphical representation for the pin usage of most capes, as shown in *Audio cape pins* for the Circuitco Audio Cape.

---

**Note:** #TODO# Bela would make a better example for an audio cape. The CircuitCo cape is no longer available.

---

## 9.3 Audio cape pins

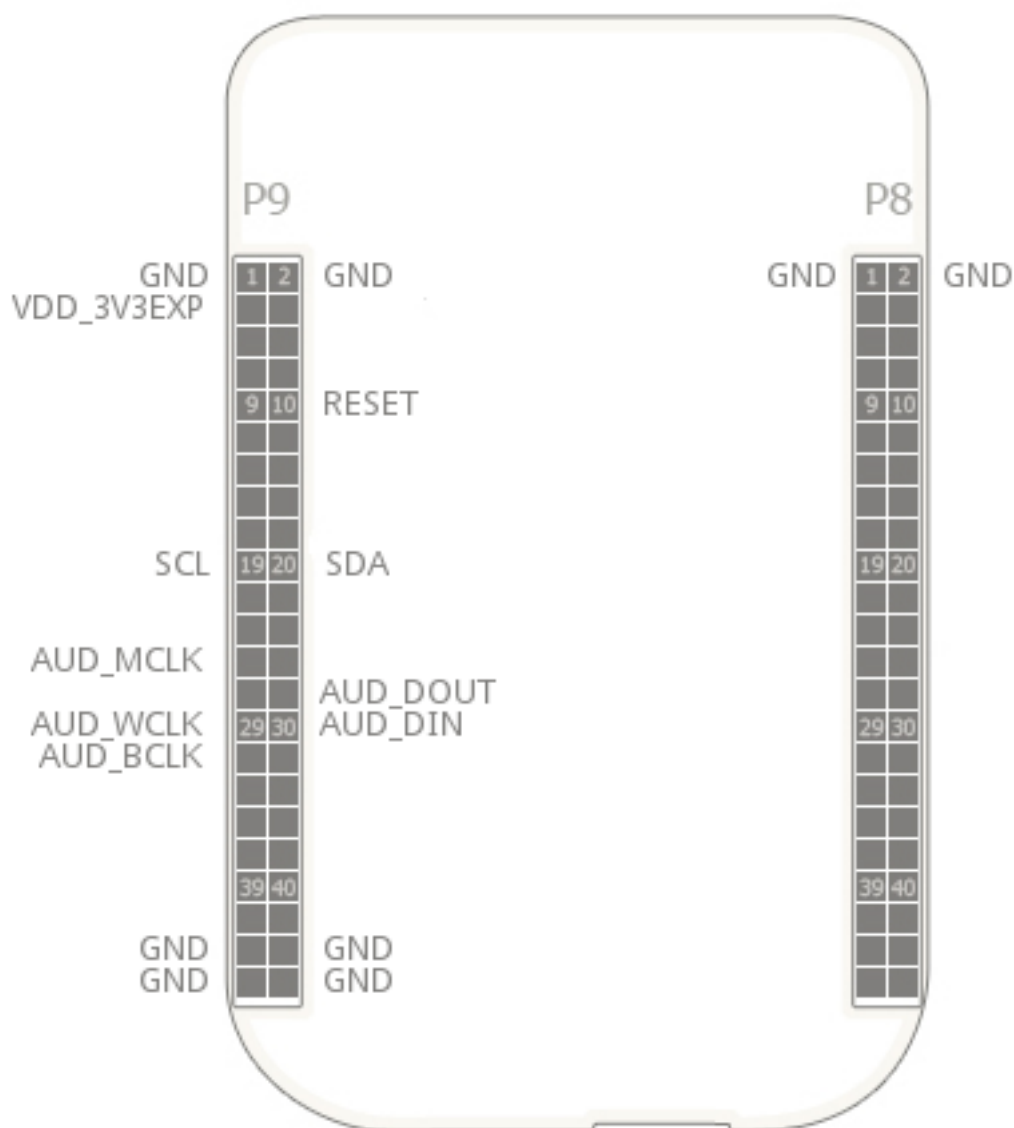
---

**Note:** Pins utilized by CircuitCo Audio Cape, *Audio cape pins* was originally posted by Djackson at [http://elinux.org/File:Audio\\_pins\\_revb.png](http://elinux.org/File:Audio_pins_revb.png) under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

---

In most cases, the same pin should never be used on two different capes, though in some cases, pins can be shared. Here are some exceptions:

- **GND**



- The ground (*GND*) pins should be shared between the capes, and there's no need to worry about consumed resources on those pins.

- **VDD\_3V3**

- The 3.3 V power supply (*VDD\_3V3*) pins can be shared by all capes to supply power, but the total combined consumption of all the capes should be less than 500 mA (250 mA per *VDD\_3V3* pin).

- **VDD\_5V**

The 5.0 V power supply (*VDD\_5V*) pins can be shared by all capes to supply power, but the total combined consumption of all the capes should be less than 2 A (1 A per +*VDD\_5V*+ pin). It is possible for one, and only one, of the capes to *provide* power to this pin rather than consume it, and it should provide at least 3 A to ensure proper system function. Note that when no voltage is applied to the DC connector, nor from a cape, these pins will not be powered, even if power is provided via USB.

- **SYS\_5V**

The regulated 5.0 V power supply (*SYS\_5V*) pins can be shared by all capes to supply power, but the total combined consumption of all the capes should be less than 500 mA (250 mA per *SYS\_5V* pin).

- **VADC and AGND**

- The ADC reference voltage pins can be shared by all capes.

- **I2C2\_SCL and I2C2\_SDA**

- I<sup>2</sup>C is a shared bus, and the *I2C2\_SCL* and *I2C2\_SDA* pins default to having this bus enabled for use by cape expansion ID EEPROMs.

## 9.4 Moving from a Breadboard to a Protoboard

### 9.4.1 Problem

You have your circuit working fine on the breadboard, but you want a more reliable solution.

### 9.4.2 Solution

Solder your components to a protoboard.

To make this recipe, you will need:

- Protoboard
- Soldering iron
- Your other components

Many places make premade circuit boards that are laid out like the breadboard we have been using. The [Adafruit Proto Cape Kit](#) is one protoboard option.

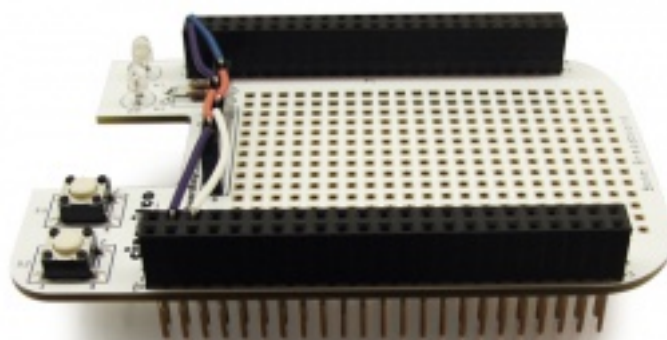
### BeagleBone Breadboard

---

**Note:** This was originally posted by William Traynor at <http://elinux.org/File:BeagleBone-Breadboard.jpg> under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#)

---

You just solder your parts on the protoboard as you had them on the breadboard.



## 9.5 Creating a Prototype Schematic

### 9.5.1 Problem

You've wired up a circuit on a breadboard. How do you turn that prototype into a schematic others can read and that you can import into other design tools?

### 9.5.2 Solution

In [Fritzing tips](#), we introduced Fritzing as a useful tool for drawing block diagrams. Fritzing can also do circuit schematics and printed-circuit layout. For example, [A simple robot controller diagram \(quickBot.fzz\)](#) shows a block diagram for a simple robot controller (quickBot.fzz is the name of the Fritzing file used to create the diagram).

The controller has an H-bridge to drive two DC motors ([Controlling the Speed and Direction of a DC Motor](#)), an IR range sensor, and two headers for attaching analog encoders for the motors. Both the IR sensor and the encoders have analog outputs that exceed 1.8 V, so each is run through a voltage divider (two resistors) to scale the voltage to the correct range (see [Reading a Distance Sensor \(Variable Pulse Width Sensor\)](#) for a voltage divider example).

[Automatically generated schematic](#) shows the schematic automatically generated by Fritzing. It's a mess. It's up to you to fix it.

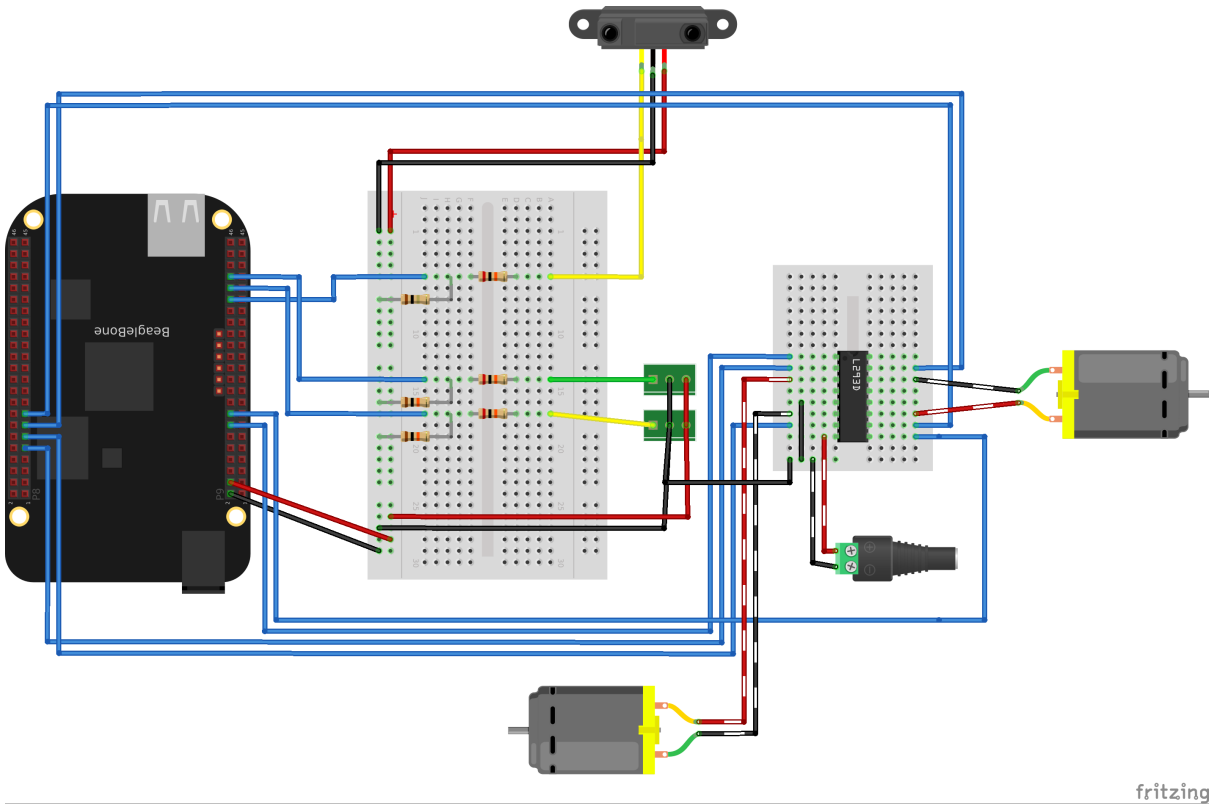
[Cleaned-up schematic](#) shows my cleaned-up schematic. I did it by moving the parts around until it looked better.

You might find that you want to create your design in a more advanced design tool, perhaps because it has the library components you desire, it integrates better with other tools you are using, or it has some other feature (such as simulation) of which you'd like to take advantage.

## 9.6 Verifying Your Cape Design

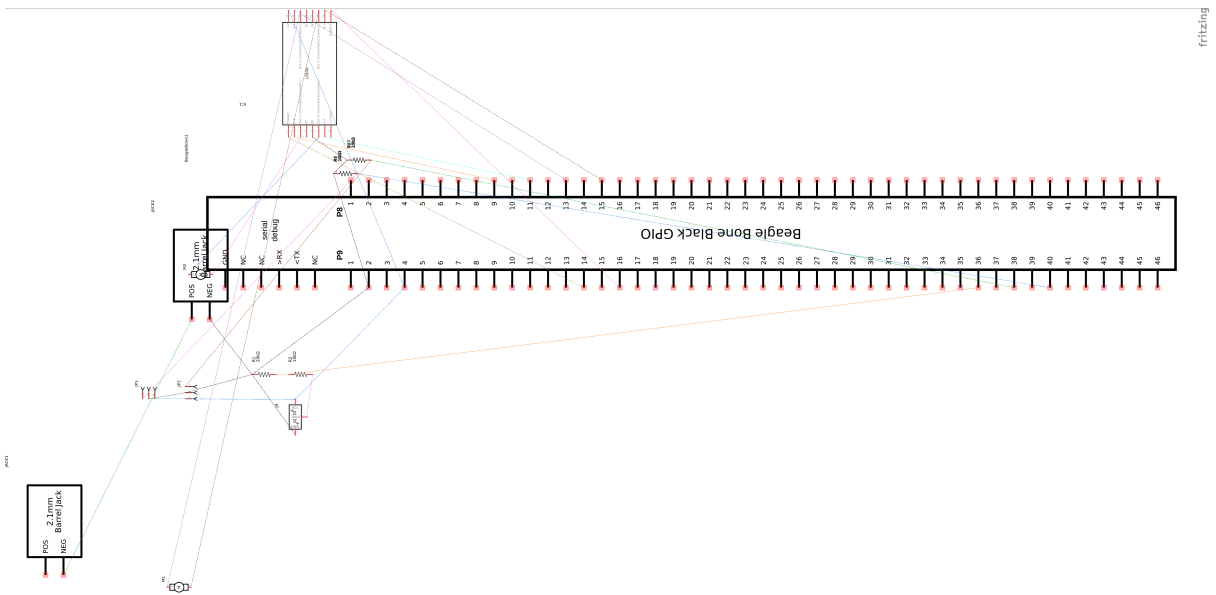
### 9.6.1 Problem

You've got a design. How do you quickly verify that it works?



fritzing

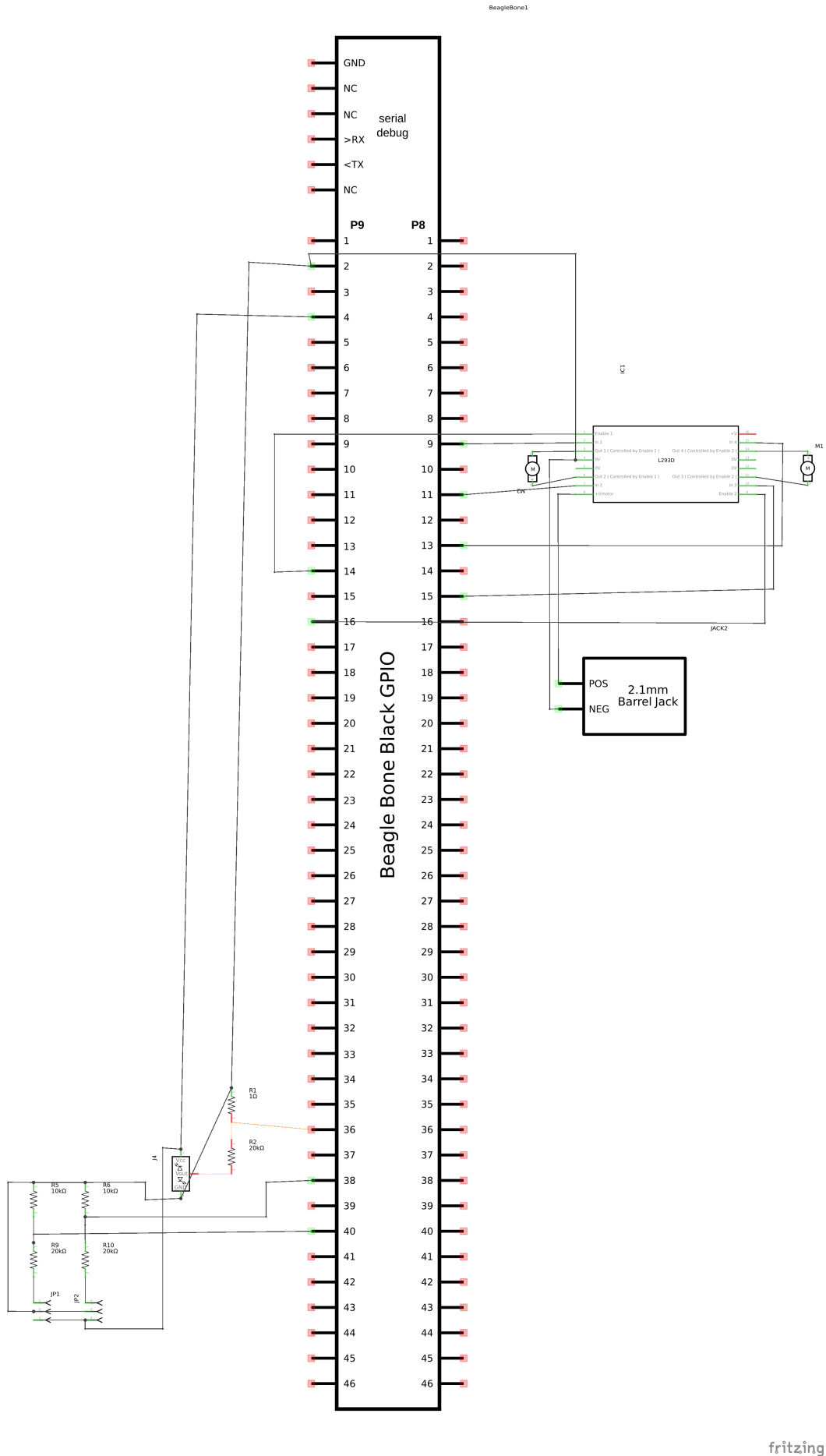
Fig. 9.2: A simple robot controller diagram (quickBot.fzz)



fritzing

Fig. 9.3: Automatically generated schematic





fritzing

Fig. 9.4: Cleaned-up schematic

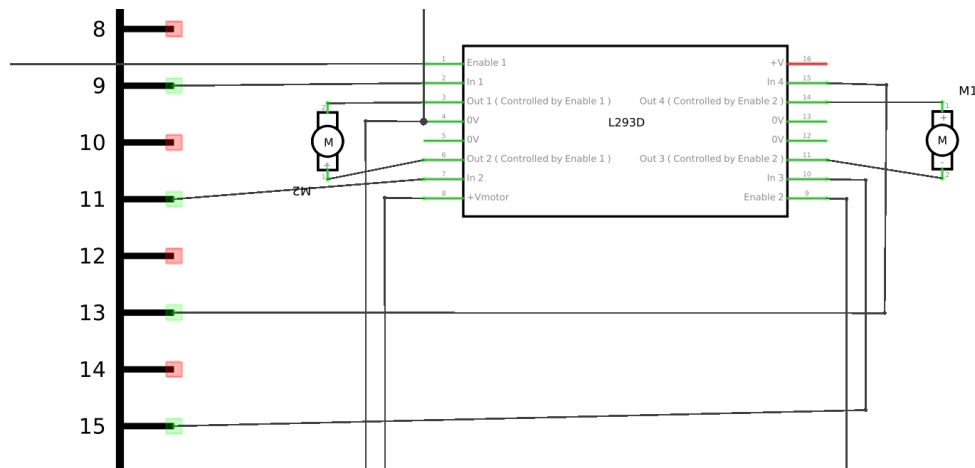


Fig. 9.5: Zoomed-in schematic

### 9.6.2 Solution

To make this recipe, you will need:

- An oscilloscope

Break down your design into functional subcomponents and write tests for each. Use components you already know are working, such as the onboard LEDs, to display the test status with the code in [Testing the quickBot motors interface \(quickBot\\_motor\\_test.js\)](#).

## 9.7 Testing the quickBot motors interface (quickBot\_motor\_test.js)

```
#!/usr/bin/env node
var b = require('bonescript');
var M1_SPEED = 'P9_16'; // ⓧ
var M1_FORWARD = 'P8_15';
var M1_BACKWARD = 'P8_13';
var M2_SPEED = 'P9_14';
var M2_FORWARD = 'P8_9';
var M2_BACKWARD = 'P8_11';
var freq = 50; // ⓧ
var fast = 0.95;
var slow = 0.7;
var state = 0; // ⓧ

b.pinMode(M1_FORWARD, b.OUTPUT); // ⓧ
b.pinMode(M1_BACKWARD, b.OUTPUT);
b.pinMode(M2_FORWARD, b.OUTPUT);
b.pinMode(M2_BACKWARD, b.OUTPUT);
b.analogWrite(M1_SPEED, 0, freq); // ⓧ
b.analogWrite(M2_SPEED, 0, freq);

updateMotors(); // ⓧ

function updateMotors() {
  //console.log("Setting state = " + state); // ⓧ
  updateLEDs(state);
  switch(state) { // ⓧ
    case 0:
    default:
      M1_set(0); // ⓧ
  }
}
```

(continues on next page)

(continued from previous page)

```
        M2_set(0);
        state = 1; // 1
        break;
    case 1:
        M1_set(slow);
        M2_set(slow);
        state = 2;
        break;
    case 2:
        M1_set(slow);
        M2_set(-slow);
        state = 3;
        break;
    case 3:
        M1_set(-slow);
        M2_set(slow);
        state = 4;
        break;
    case 4:
        M1_set(fast);
        M2_set(fast);
        state = 0;
        break;
    }
    setTimeout(updateMotors, 2000); // 2
}

function updateLEDs(state) { // 3
    switch(state) {
    case 0:
        b.digitalWrite("USR0", b.LOW);
        b.digitalWrite("USR1", b.LOW);
        b.digitalWrite("USR2", b.LOW);
        b.digitalWrite("USR3", b.LOW);
        break;
    case 1:
        b.digitalWrite("USR0", b.HIGH);
        b.digitalWrite("USR1", b.LOW);
        b.digitalWrite("USR2", b.LOW);
        b.digitalWrite("USR3", b.LOW);
        break;
    case 2:
        b.digitalWrite("USR0", b.LOW);
        b.digitalWrite("USR1", b.HIGH);
        b.digitalWrite("USR2", b.LOW);
        b.digitalWrite("USR3", b.LOW);
        break;
    case 3:
        b.digitalWrite("USR0", b.LOW);
        b.digitalWrite("USR1", b.LOW);
        b.digitalWrite("USR2", b.HIGH);
        b.digitalWrite("USR3", b.LOW);
        break;
    case 4:
        b.digitalWrite("USR0", b.LOW);
        b.digitalWrite("USR1", b.LOW);
        b.digitalWrite("USR2", b.LOW);
        b.digitalWrite("USR3", b.HIGH);
        break;
    }
}
```

(continues on next page)

(continued from previous page)

```

function M1_set (speed) { // ?
  speed = (speed > 1) ? 1 : speed; // ?
  speed = (speed < -1) ? -1 : speed;
  b.digitalWrite (M1_FORWARD, b.LOW);
  b.digitalWrite (M1_BACKWARD, b.LOW);
  if (speed > 0) {
    b.digitalWrite (M1_FORWARD, b.HIGH);
  } else if (speed < 0) {
    b.digitalWrite (M1_BACKWARD, b.HIGH);
  }
  b.analogWrite (M1_SPEED, Math.abs (speed), freq); // ?
}

function M2_set (speed) {
  speed = (speed > 1) ? 1 : speed;
  speed = (speed < -1) ? -1 : speed;
  b.digitalWrite (M2_FORWARD, b.LOW);
  b.digitalWrite (M2_BACKWARD, b.LOW);
  if (speed > 0) {
    b.digitalWrite (M2_FORWARD, b.HIGH);
  } else if (speed < 0) {
    b.digitalWrite (M2_BACKWARD, b.HIGH);
  }
  b.analogWrite (M2_SPEED, Math.abs (speed), freq);
}

```

- ① Define each pin as a variable. This makes it easy to change to another pin if you decide that is necessary.
  - ② Make other simple parameters variables. Again, this makes it easy to update them. When creating this test, I found that the PWM frequency to drive the motors needed to be relatively low to get over the kickback shown in [quickBot motor test showing kickback](#). I also found that I needed to get up to about 70 percent duty cycle for my circuit to reliably start the motors turning.
  - ③ Use a simple variable such as *state* to keep track of the test phase. This is used in a *switch* statement to jump to the code to configure for that test phase and updated after configuring for the current phase in order to select the next phase. Note that the next phase isn't entered until after a two-second delay, as specified in the call to *setTimeout()*.
  - ④ Perform the initial setup of all the pins.
  - ⑤ The first time a PWM pin is used, it is configured with the update frequency. It is important to set this just once to the right frequency, because other PWM channels might use the same PWM controller, and attempts to reset the PWM frequency might fail. The *pinMode()* function doesn't have an argument for providing the update frequency, so use the *analogWrite()* function, instead. You can review using the PWM in [Controlling a Servo Motor](#).
  - ⑥ *updateMotors()* is the test function for the motors and is defined after all the setup and initialization code. The code calls this function every two seconds using the *setTimeout()* JavaScript function. The first call is used to prime the loop.
  - ⑦ The call to *console.log()* was initially here to observe the state transitions in the debug console, but it was replaced with the *updateLEDs()* call. Using the *USER* LEDs makes it possible to note the state transitions without having visibility of the debug console. *updateLEDs()* is defined later.
  - ⑧ The *M1\_set()* and *M2\_set()* functions are defined near the bottom and do the work of configuring the motor drivers into a particular state. They take a single argument of *speed*, as defined between *-1* (maximum reverse), *0* (stop), and *1* (maximum forward).
  - ⑨ Perform simple bounds checking to ensure that speed values are between *-1* and *1*.
  - ⑩ The *analogWrite()* call uses the absolute value of *speed*, making any negative numbers a positive magnitude.
- Using the solution in [Basics](#), you can untether from your coding station to test your design at your lab workbench, as shown in [quickBot motor test code under scope](#).

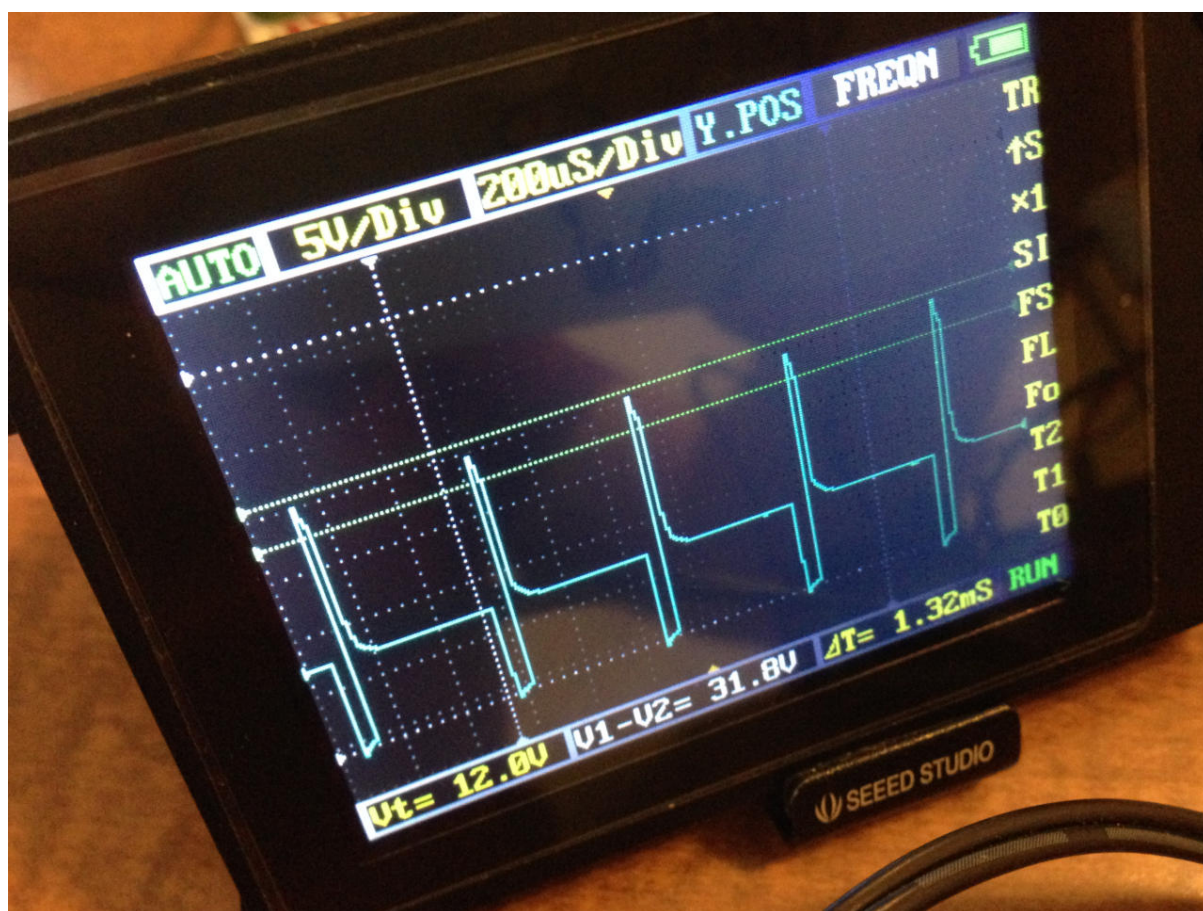


Fig. 9.6: quickBot motor test showing kickback

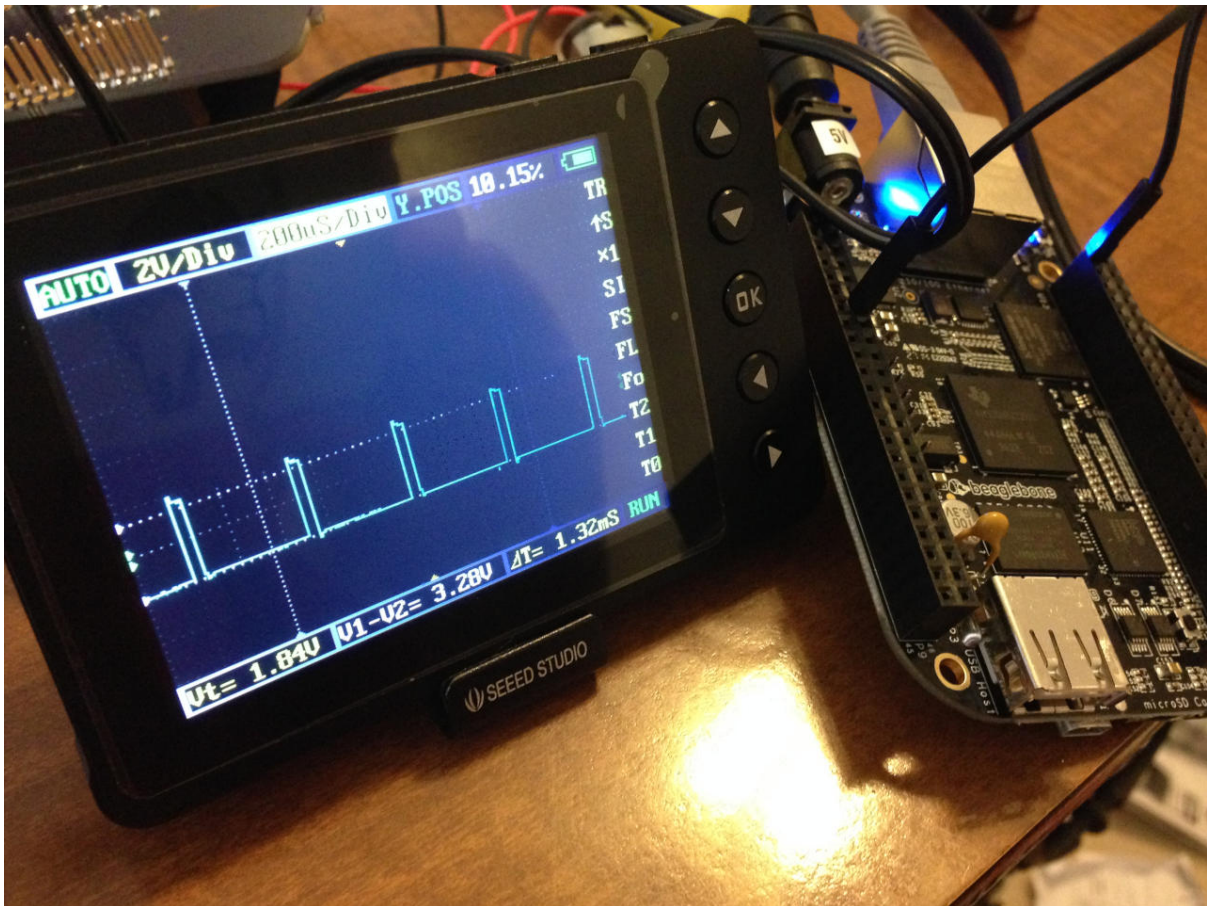


Fig. 9.7: quickBot motor test code under scope

SparkFun provides a [useful guide to using an oscilloscope](#). You might want to check it out if you've never used an oscilloscope before. Looking at the stimulus you'll generate *before* you connect up your hardware will help you avoid surprises.

## 9.8 Laying Out Your Cape PCB

### 9.8.1 Problem

You've generated a diagram and schematic for your circuit and verified that they are correct. How do you create a PCB?

### 9.8.2 Solution

If you've been using Fritzing, all you need to do is click the PCB tab, and there's your board. Well, almost. Much like the schematic view shown in [Creating a Prototype Schematic](#), you need to do some layout work before it's actually usable. I just moved the components around until they seemed to be grouped logically and then clicked the Autoroute button. After a minute or two of trying various layouts, Fritzing picked the one it determined to be the best. [Simple robot PCB](#) shows the results.

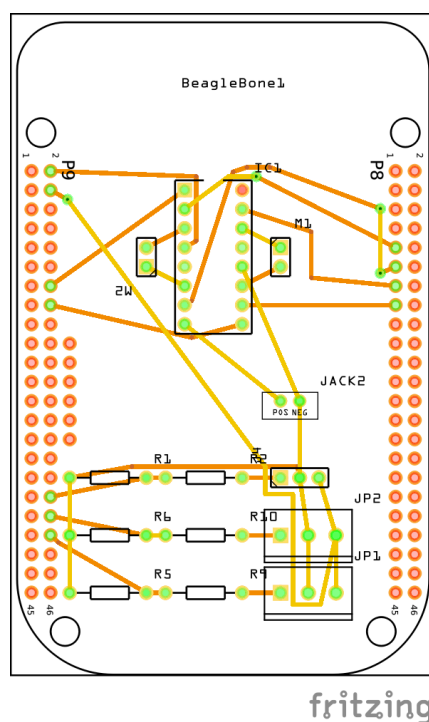


Fig. 9.8: Simple robot PCB

The [Fritzing pre-fab web page](#) has a few helpful hints, including checking the widths of all your traces and cleaning up any questionable routing created by the autorouter.

The PCB in [Simple robot PCB](#) is a two-sided board. One color (or shade of gray in the printed book) represents traces on one side of the board, and the other color (or shade of gray) is the other side. Sometimes, you'll see a trace come to a small circle and then change colors. This is where it is switching sides of the board through what's called a *via*. One of the goals of PCB design is to minimize the number of vias.

[Simple robot PCB](#) wasn't my first try or my last. My approach was to see what was needed to hook where and move the components around to make it easier for the autorouter to carry out its job.

**Note:** There are entire books and websites dedicated to creating PCB layouts. Look around and see what you can find. [SparkFun's guide to making PCBs](#) is particularly useful.

## 9.9 Customizing the Board Outline

One challenge that slipped my first pass review was the board outline. The part we installed in [Fritzing tips](#) is meant to represent BeagleBone Black, not a cape, so the outline doesn't have the notch cut out of it for the Ethernet connector.

The [Fritzing custom PCB outline page](#) describes how to create and use a custom board outline. Although it is possible to use a drawing tool like Inkscape, I chose to use the SVG path command directly to create [Outline SVG for BeagleBone cape \(beaglebone\\_cape\\_boardoutline.svg\)](#).

Listing 9.1: Outline SVG for BeagleBone cape (beaglebone\_cape\_boardoutline.svg)

```

1 <?xml version='1.0' encoding='UTF-8' standalone='no'?>
2 <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
3   width="306" height="193.5"> <!-- -->
4   <g id="board"> <!-- -->
5     <path fill="#338040" id="boardoutline" d="M 22.5,0 l 0,56 L 72,56
6       q 5,0 5,5 l 0,53.5 q 0,5 -5,5 L 0,119.5 L 0,171 Q 0,193.5 22.5,193.5
7       l 238.5,0 c 24.85281,0 45,-20.14719 45,-45 L 306,45
8       C 306,20.14719 285.85281,0 261,0 z"/> <!-- -->
9   </g>
10 </svg>

```

① This is a standard SVG header. The width and height are set based on the BeagleBone outline provided in the Adafruit library.

② Fritzing requires the element to be within a layer called *board*

③ Fritzing requires the color to be #338040 and the layer to be called *boardoutline*. The units end up being 1/90 of an inch. That is, take the numbers in the SVG code and divide by 90 to get the numbers from the System Reference Manual.

The measurements are taken from the beagleboneblack-mechanical section of the BeagleBone Black System Reference Manual, as shown in [Cape dimensions](#).

You can observe the rendered output of [Outline SVG for BeagleBone cape \(beaglebone\\_cape\\_boardoutline.svg\)](#) quickly by opening the file in a web browser, as shown in [Rendered cape outline in Chrome](#).

## 9.10 Fritzing tips

After you have the SVG outline, you'll need to select the PCB in Fritzing and select a custom shape in the Inspector box. Begin with the original background, as shown in [PCB with original board, without notch for Ethernet connector](#).

Hide all but the Board Layer ([PCB with all but the Board Layer hidden](#)).

Select the PCB1 object and then, in the Inspector pane, scroll down to the "load image file" button ([Clicking :load image file: with PCB1 selected](#)).

Navigate to the [beaglebone\\_cape\\_boardoutline.svg](#) file created in [Outline SVG for BeagleBone cape \(beaglebone\\_cape\\_boardoutline.svg\)](#), as shown in [Selecting the .svg file](#).

Turn on the other layers and line up the Board Layer with the rest of the PCB, as shown in [PCB Inspector](#).

Now, you can save your file and send it off to be made, as described in [Producing a Prototype](#).



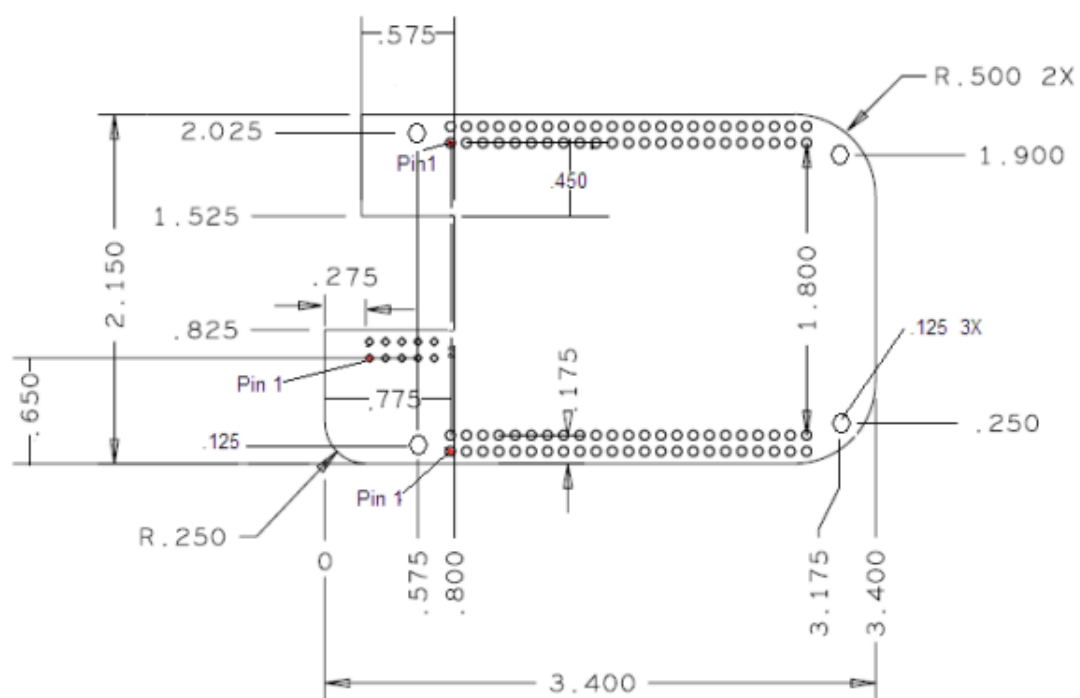


Figure 70. Cape Board Dimensions

Fig. 9.9: Cape dimensions

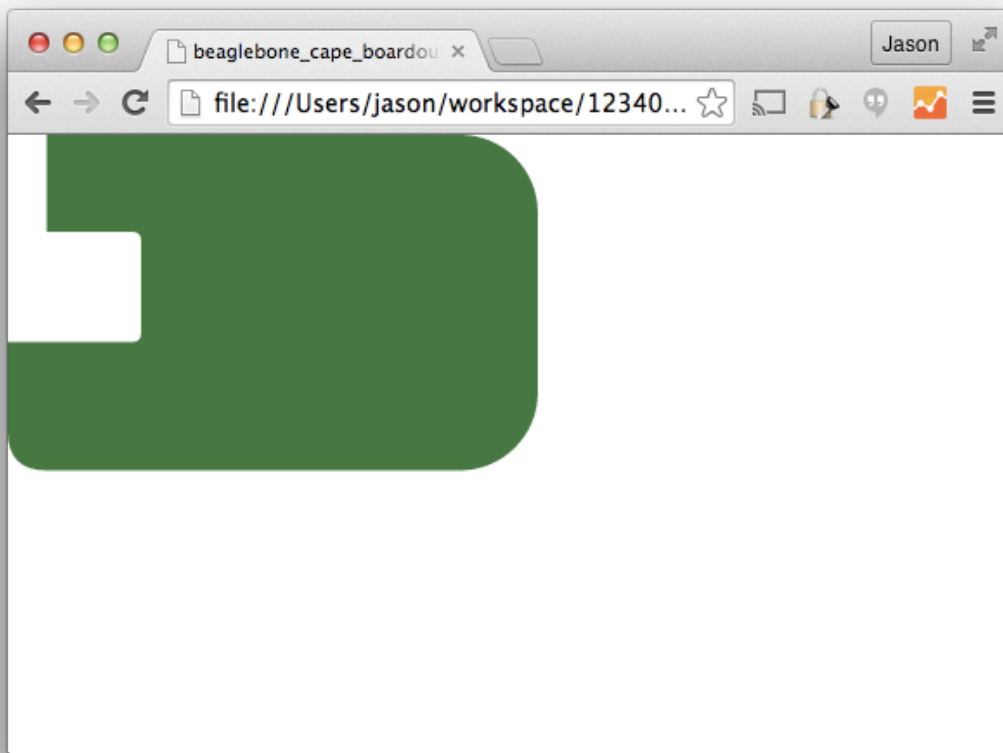


Fig. 9.10: Rendered cape outline in Chrome

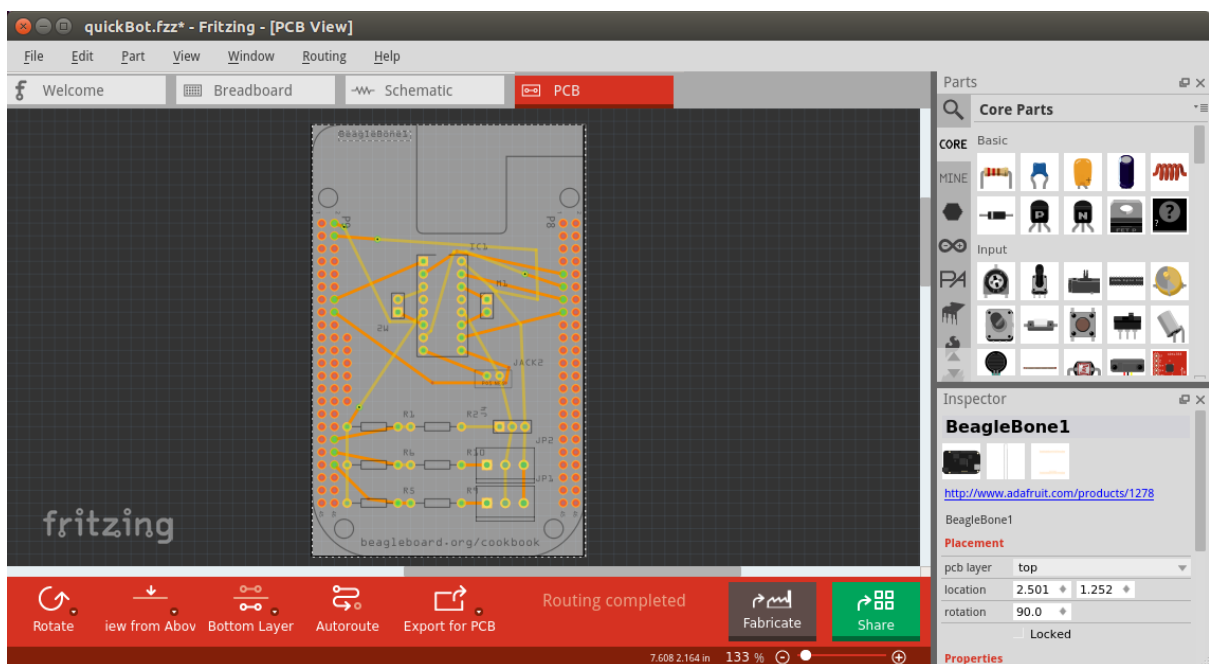


Fig. 9.11: PCB with original board, without notch for Ethernet connector

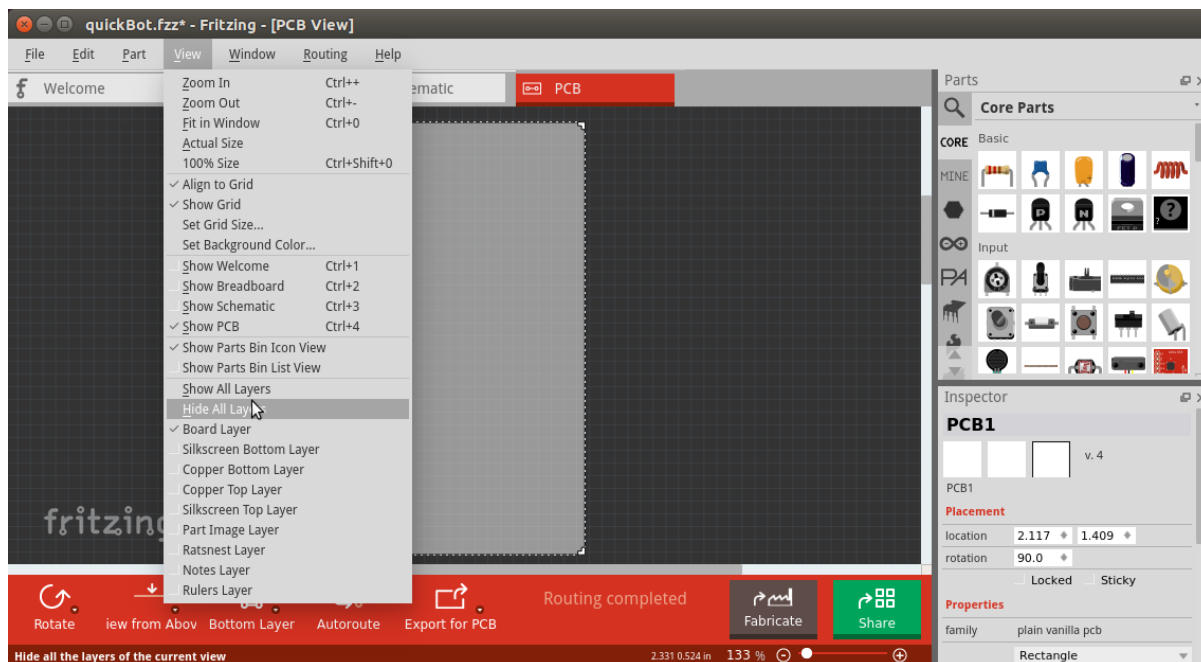


Fig. 9.12: PCB with all but the Board Layer hidden

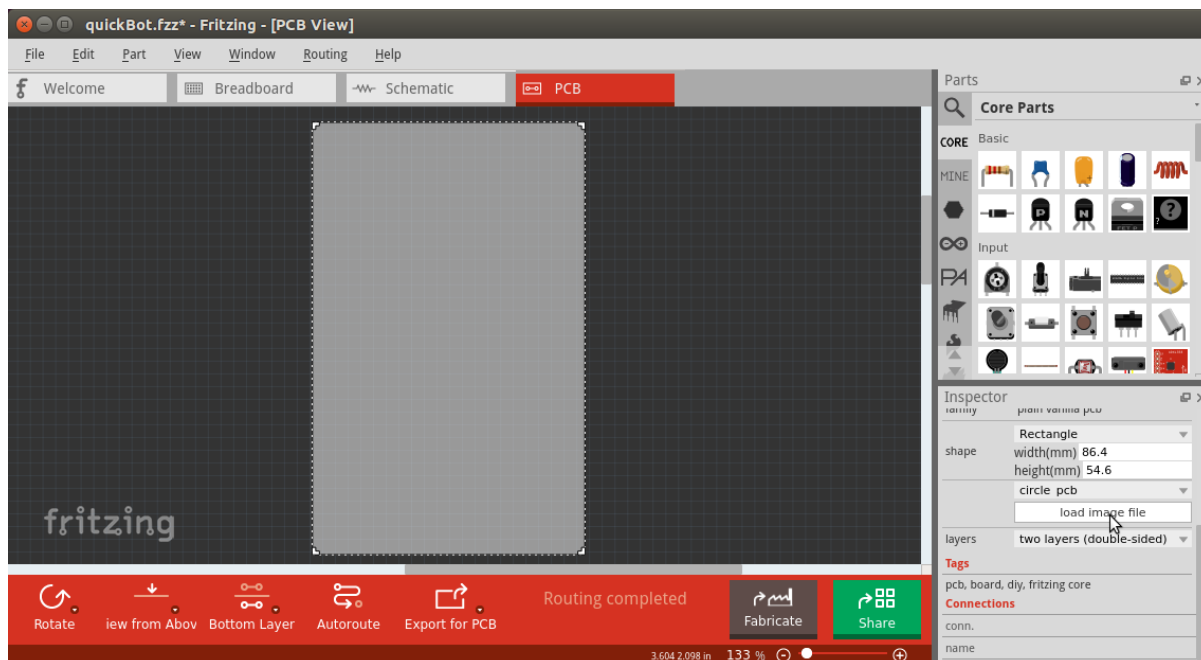


Fig. 9.13: Clicking :load image file: with PCB1 selected

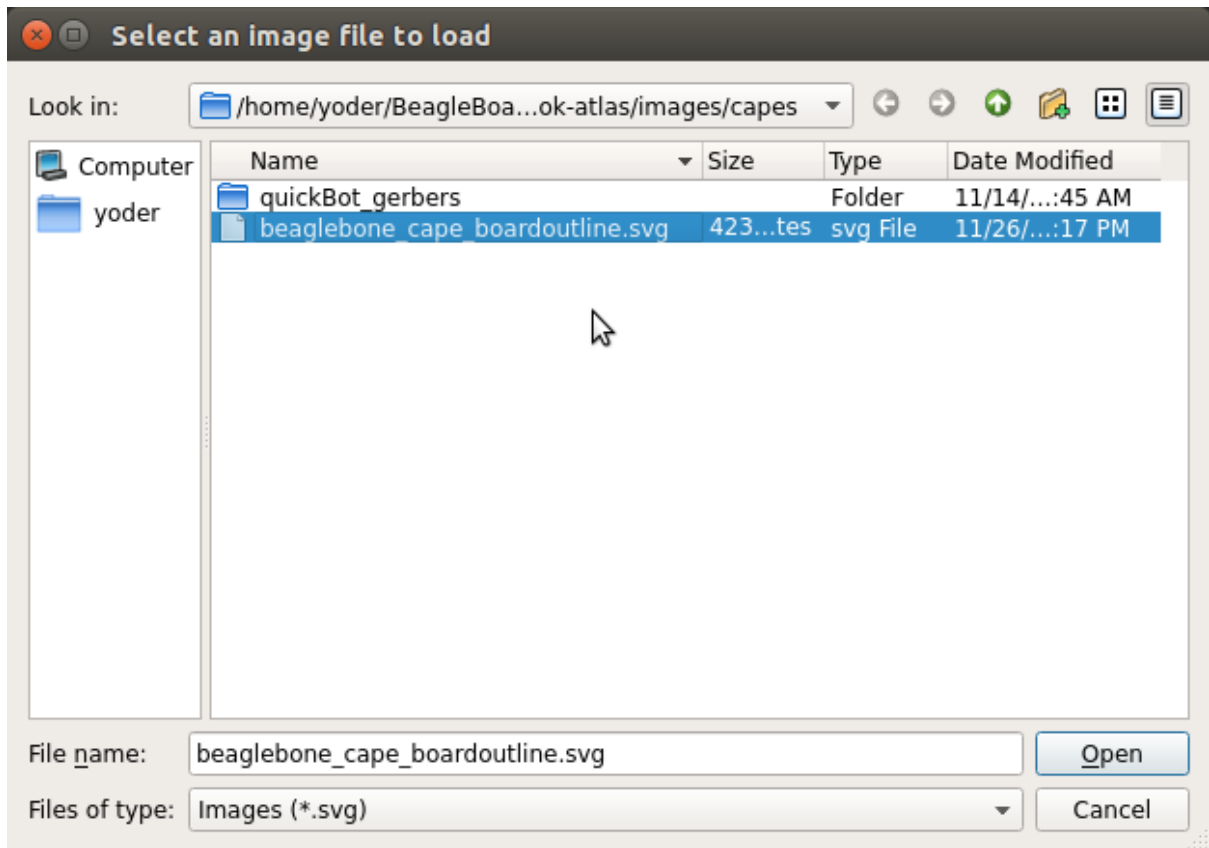


Fig. 9.14: Selecting the .svg file

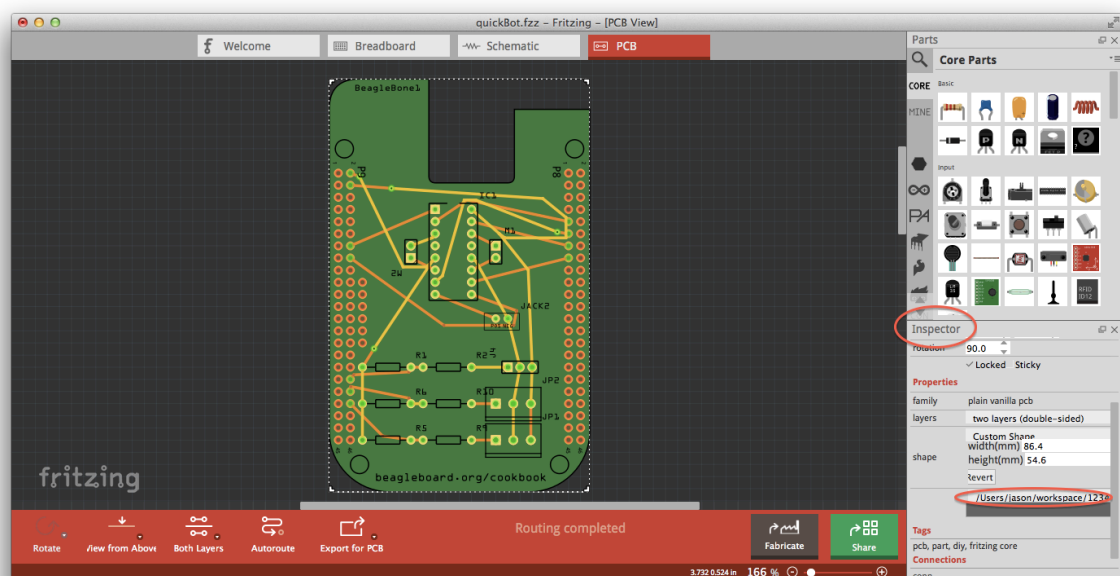


Fig. 9.15: PCB Inspector

## 9.11 PCB Design Alternatives

There are other free PCB design programs. Here are a few.

### 9.11.1 EAGLE

Eagle PCB and DesignSpark PCB are two popular design programs. Many capes (and other PCBs) are designed with Eagle PCB, and the files are available. For example, the MiniDisplay cape has the schematic shown in *Schematic for the MiniDisplay cape* and PCB shown in *PCB for MiniDisplay cape*.

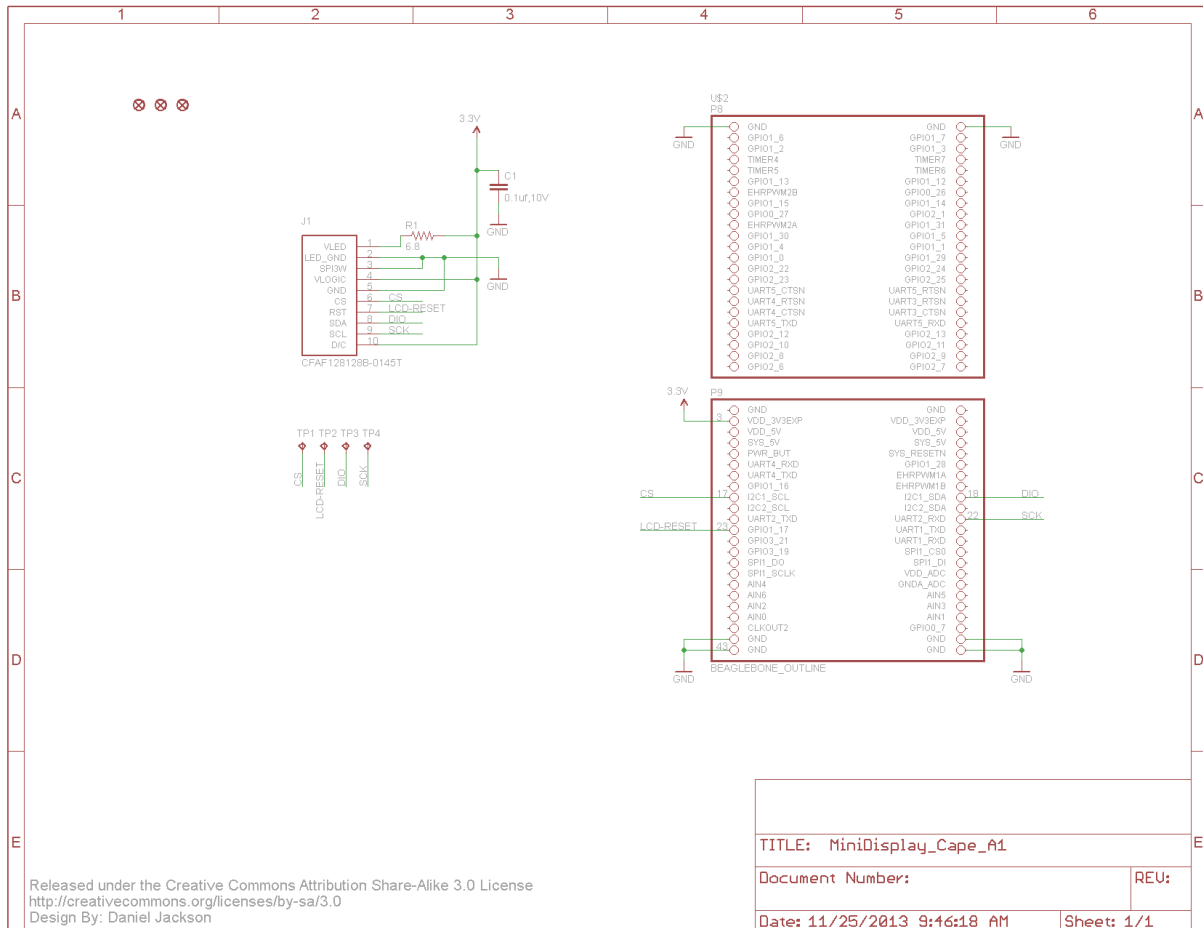


Fig. 9.16: Schematic for the MiniDisplay cape

**Note:** #TODO#: The MiniDisplay cape is not currently available, so this example should be updated.

A good starting point is to take the PCB layout for the MiniDisplay and edit it for your project. The connectors for **P8** and **P9** are already in place and ready to go.

Eagle PCB is a powerful system with many good tutorials online. The free version works on Windows, Mac, and Linux, but it has three limitations:

- The usable board area is limited to 100 x 80 mm (4 x 3.2 inches).
- You can use only two signal layers (Top and Bottom).
- The schematic editor can create only one sheet.

You can install Eagle PCB on your Linux host by using the following command:

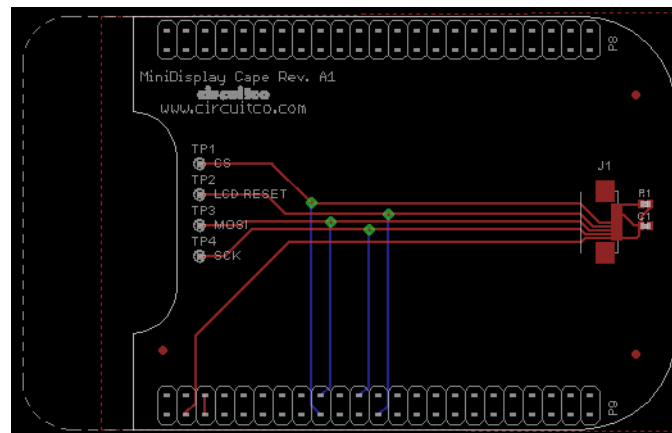


Fig. 9.17: PCB for MiniDisplay cape

```

host$ sudo apt install eagle
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
Setting up eagle (6.5.0-1) ...
Processing triggers for libc-bin (2.19-0ubuntu6.4) ...
host$ eagle
  
```

You'll see the startup screen shown in [Eagle PCB startup screen](#).

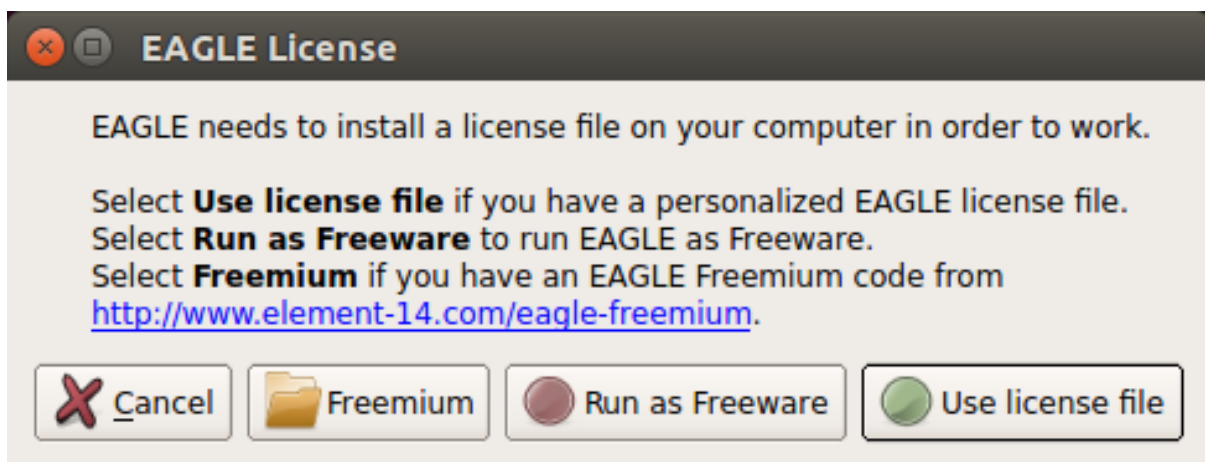


Fig. 9.18: Eagle PCB startup screen

Click “Run as Freeware.” When my Eagle started, it said it needed to be updated. To update on Linux, follow the link provided by Eagle and download *eagle-lin-7.2.0.run* (or whatever version is current.). Then run the following commands:

```

host$ chmod +x eagle-lin-7.2.0.run
host$ ./eagle-lin-7.2.0.run
  
```

A series of screens will appear. Click Next. When you see a screen that looks like [The Eagle installation destination directory](#), note the Destination Directory.

Continue clicking Next until it's installed. Then run the following commands (where *~/eagle-7.2.0* is the path you noted in [The Eagle installation destination directory](#)):



Fig. 9.19: The Eagle installation destination directory

```

host$ cd /usr/bin
host$ sudo rm eagle
host$ sudo ln -s ~/eagle-7.2.0/bin/eagle .
host$ cd
host$ eagle

```

The `ln` command links `eagle` in `/usr/bin`, so you can run `+eagle+` from any directory. After `eagle` starts, you'll see the start screen shown in [The Eagle start screen](#).

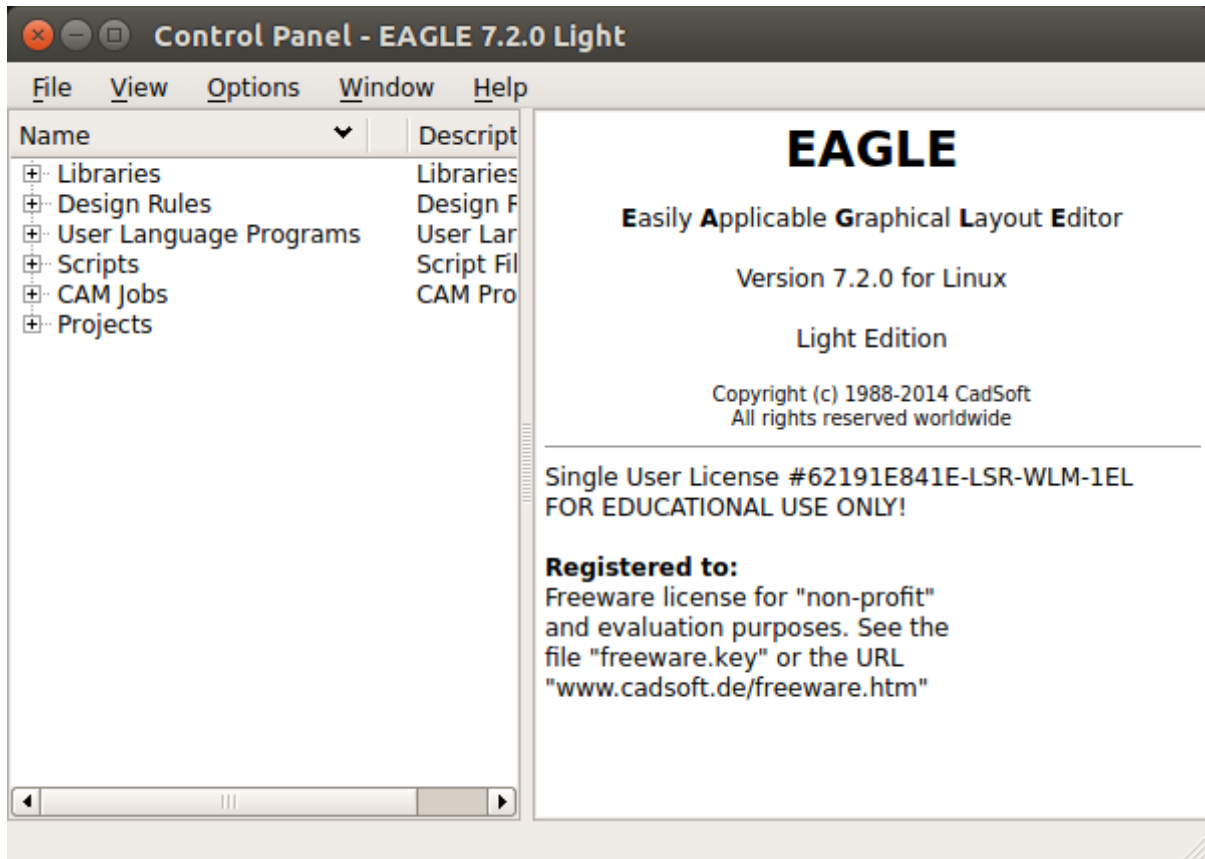


Fig. 9.20: The Eagle start screen

Ensure that the correct version number appears.

If you are moving a design from Fritzing to Eagle, see [Migrating a Fritzing Schematic to Another Tool](#) for tips on converting from one to the other.

### 9.11.2 DesignSpark PCB

The free [DesignSpark](#) doesn't have the same limitations as Eagle PCB, but it runs only on Windows. Also, it doesn't seem to have the following of Eagle at this time.

### 9.11.3 Upverter

In addition to free solutions you run on your desktop, you can also work with a browser-based tool called [Upverter](#). With Upverter, you can collaborate easily, editing your designs from anywhere on the Internet. It also provides many conversion options and a PCB fabrication service.



**Note:** Don't confuse Upverter with Upconverter ([Migrating a Fritzing Schematic to Another Tool](#)). Though their names differ by only three letters, they differ greatly in what they do.

---

### 9.11.4 Kicad

Unlike the previously mentioned free (no-cost) solutions, Kicad is open source and provides some features beyond those of Fritzing. Notably, [CircuitHub](#) site (discussed in [Putting Your Cape Design into Production](#)) provides support for uploading Kicad designs.

## 9.12 Migrating a Fritzing Schematic to Another Tool

### 9.12.1 Problem

You created your schematic in Fritzing, but it doesn't integrate with everything you need. How can you move the schematic to another tool?

### 9.12.2 Solution

Use the `Upverter schematic-file-converter` Python script. For example, suppose that you want to convert the Fritzing file for the diagram shown in [A simple robot controller diagram \(quickBot.fzz\)](#). First, install Upverter.

I found it necessary to install `+libfreetype6+` and `+freetype-py+` onto my system, but you might not need this first step:

```
host$ sudo apt install libfreetype6
Reading package lists... Done
Building dependency tree
Reading state information... Done
libfreetype6 is already the newest version.
0 upgraded, 0 newly installed, 0 to remove and 154 not upgraded.
host$ sudo pip install freetype-py
Downloading/unpacking freetype-py
Running setup.py egg_info for package freetype-py

Installing collected packages: freetype-py
Running setup.py install for freetype-py

Successfully installed freetype-py
Cleaning up...
```

---

**Note:** All these commands are being run on the Linux-based host computer, as shown by the `host$` prompt. Log in as a normal user, not `+root+`.

---

Now, install the `schematic-file-converter` tool:

```
host$ git clone git@github.com:upverter/schematic-file-converter.git
Cloning into 'schematic-file-converter'...
remote: Counting objects: 22251, done.
remote: Total 22251 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (22251/22251), 39.45 MiB | 7.28 MiB/s, done.
Resolving deltas: 100% (14761/14761), done.
Checking connectivity... done.
Checking out files: 100% (16880/16880), done.
```

(continues on next page)

(continued from previous page)

```

host$ cd schematic-file-converter
host$ sudo python setup.py install
.
.
.
Extracting python_upconvert-0.8.9-py2.7.egg to \
  /usr/local/lib/python2.7/dist-packages
Adding python-upconvert 0.8.9 to easy-install.pth file

Installed /usr/local/lib/python2.7/dist-packages/python_upconvert-0.8.9-py2.
→7.egg
Processing dependencies for python-upconvert==0.8.9
Finished processing dependencies for python-upconvert==0.8.9
host$ cd ..
host$ python -m upconvert.upconverter -h
usage: upconverter.py [-h] [-i INPUT] [-f TYPE] [-o OUTPUT] [-t TYPE]
                    [-s SYMDIRS [SYMDIRS ...]] [--unsupported]
                    [--raise-errors] [--profile] [-v] [--formats]

optional arguments:
-h, --help            show this help message and exit
-i INPUT, --input INPUT
                    read INPUT file in
-f TYPE, --from TYPE read input file as TYPE
-o OUTPUT, --output OUTPUT
                    write OUTPUT file out
-t TYPE, --to TYPE   write output file as TYPE
-s SYMDIRS [SYMDIRS ...], --sym-dirs SYMDIRS [SYMDIRS ...]
                    specify SYMDIRS to search for .sym files (for gEDA
                    only)
--unsupported        run with an unsupported python version
--raise-errors      show tracebacks for parsing and writing errors
--profile           collect profiling information
-v, --version       print version information and quit
--formats           print supported formats and quit

```

At the time of this writing, Upverter supports the following file types:

File type	Support
openjson	i/o
kicad	i/o
geda	i/o
eagle	i/o
eaglexml	i/o
fritzing	in only schematic only
gerber	i/o
specctra	i/o
image	out only
ncdrill	out only
bom (csv)	out only
netlist (csv)	out only

After Upverter is installed, run the file (`quickBot.fzz`) that generated [A simple robot controller diagram \(quickBot.fzz\)](#) through Upverter:

```

host$ python -m upconvert.upconverter -i quickBot.fzz \
-f fritzing -o quickBot-eaglexml.sch -t eaglexml --unsupported
WARNING: RUNNING UNSUPPORTED VERSION OF PYTHON (2.7 > 2.6)
DEBUG:main:parsing quickBot.fzz in format fritzing
host$ ls -l
total 188
-rw-rw-r-- 1 ubuntu 63914 Nov 25 19:47 quickBot-eaglexml.sch

```

(continues on next page)

(continued from previous page)

```
-rw-r--r-- 1 ubuntu 122193 Nov 25 19:43 quickBot.fzz  
drwxrwxr-x 9 ubuntu 4096 Nov 25 19:42 schematic-file-converter
```

*Output of Upverter conversion* shows the output of the conversion.

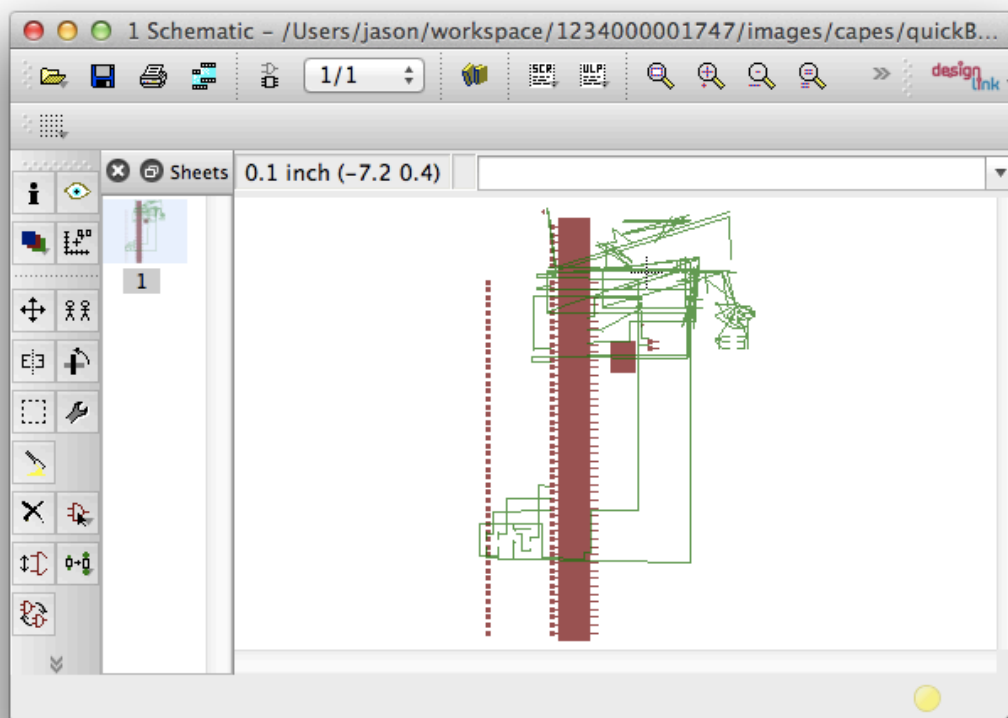


Fig. 9.21: Output of Upverter conversion

No one said it would be pretty!

I found that Eagle was more generous at reading in the **eaglexml** format than the **eagle** format. This also made it easier to hand-edit any translation issues.

## 9.13 Producing a Prototype

### 9.13.1 Problem

You have your PCB all designed. How do you get it made?

### 9.13.2 Solution

To make this recipe, you will need:

- A completed design
- Soldering iron

- Oscilloscope
- Multimeter
- Your other components

Upload your design to [OSH Park](#) and order a few boards. [The OSH Park QuickBot Cape shared project page](#) shows a resulting [shared project page](#) for the quickBot cape created in [Laying Out Your Cape PCB](#). We'll proceed to break down how this design was uploaded and shared to enable ordering fabricated PCBs.

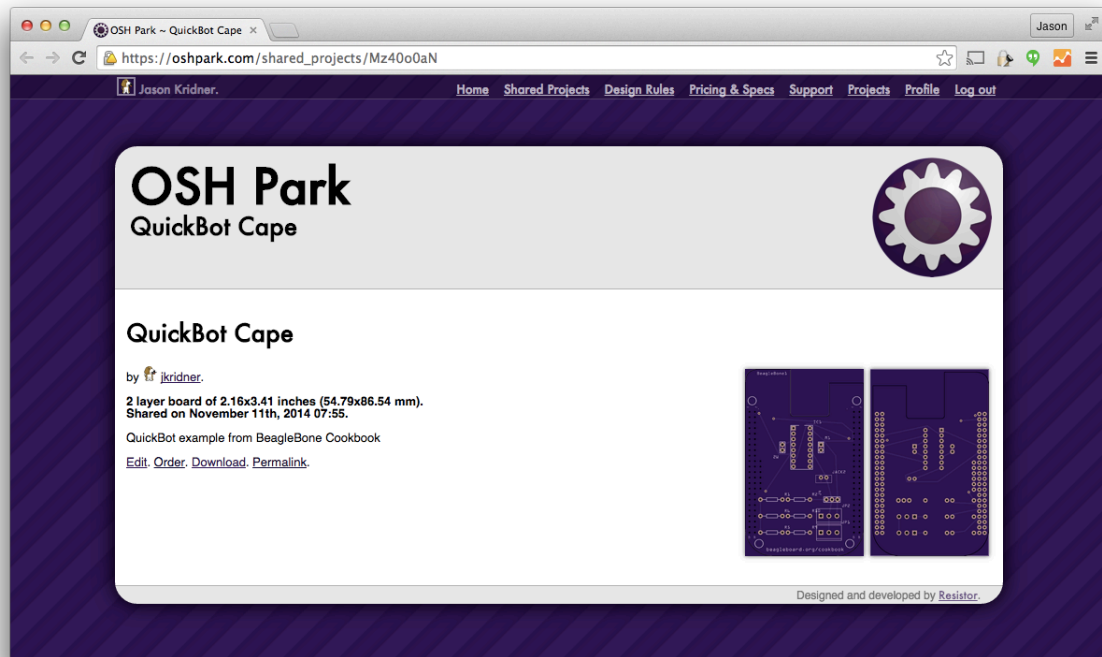


Fig. 9.22: The OSH Park QuickBot Cape shared project page

Within Fritzing, click the menu next to “Export for PCB” and choose “Extended Gerber,” as shown in [Choosing “Extended Gerber” in Fritzing](#). You’ll need to choose a directory in which to save them and then compress them all into a Zip file. The [WikiHow article on creating Zip files](#) might be helpful if you aren’t very experienced at making these.

Things on the [OSH Park website](#) are reasonably self-explanatory. You’ll need to create an account and upload the Zip file containing the [Gerber files](#) you created. If you are a cautious person, you might choose to examine the Gerber files with a Gerber file viewer first. The [Fritzing fabrication FAQ](#) offers several suggestions, including [gerbv](#) for Windows and Linux users.

When your upload is complete, you’ll be given a quote, shown images for review, and presented with options for accepting and ordering. After you have accepted the design, your [list of accepted designs](#) will also include the option of enabling sharing of your designs so that others can order a PCB, as well. If you are looking to make some money on your design, you’ll want to go another route, like the one described in [Putting Your Cape Design into Production](#). [QuickBot PCB](#) shows the resulting PCB that arrives in the mail.

Now is a good time to ensure that you have all of your components and a soldering station set up as in [Moving from a Breadboard to a Protoboard](#), as well as an oscilloscope, as used in [Verifying Your Cape Design](#).

When you get your board, it is often informative to “buzz out” a few connections by using a multimeter. If you’ve never used a multimeter before, the [SparkFun](#) or [Adafruit](#) tutorials might be helpful. Set your meter to continuity testing mode and probe between points where the headers are and where they should be connecting to your components. This would be more difficult and less accurate after you solder down your components, so it is a good idea to keep a bare board around just for this purpose.

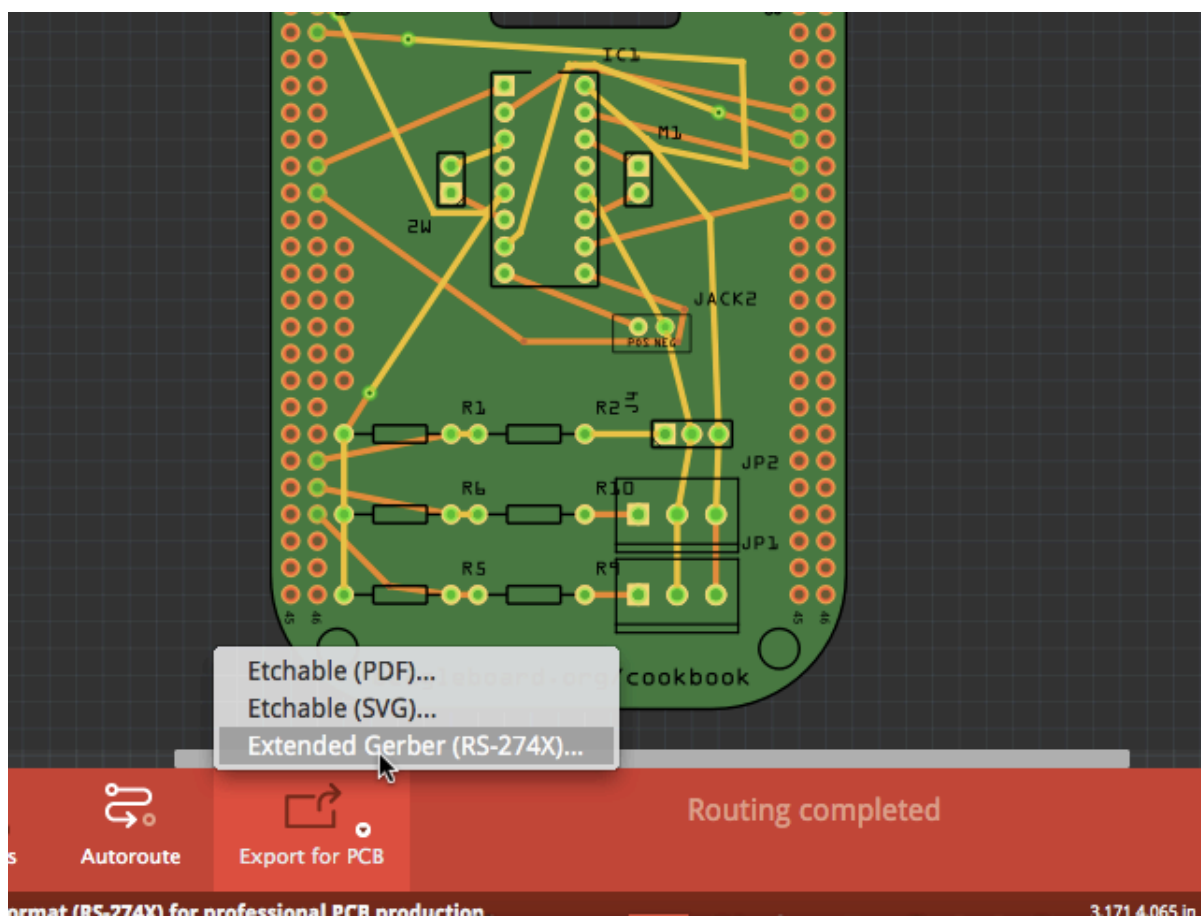


Fig. 9.23: Choosing “Extended Gerber” in Fritzing

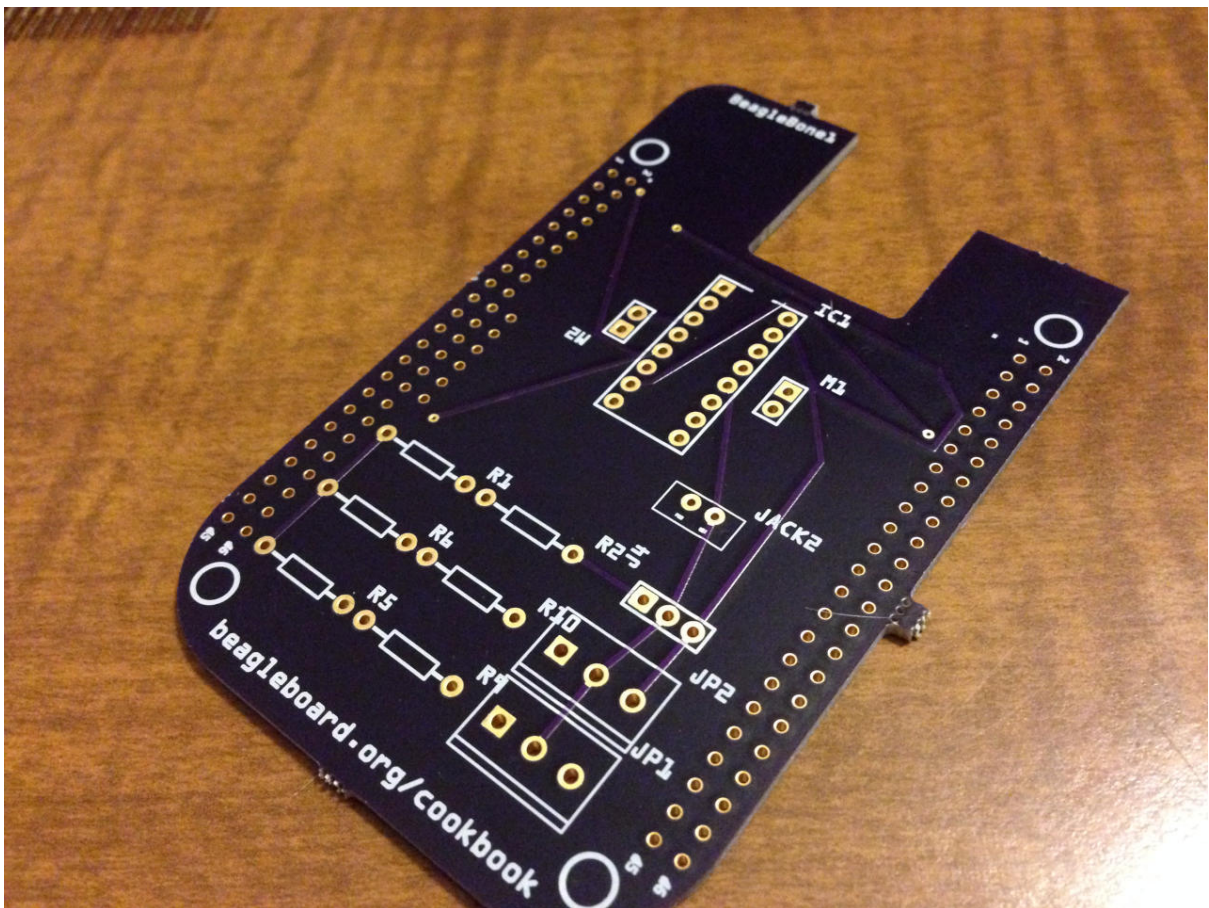


Fig. 9.24: QuickBot PCB

You'll also want to examine your board mechanically before soldering parts down. You don't want to waste components on a PCB that might need to be altered or replaced.

When you begin assembling your board, it is advisable to assemble it in functional subsections, if possible, to help narrow down any potential issues. [QuickBot motors under test](#) shows the motor portion wired up and running the test in [Testing the quickBot motors interface \(quickBot\\_motor\\_test.js\)](#).

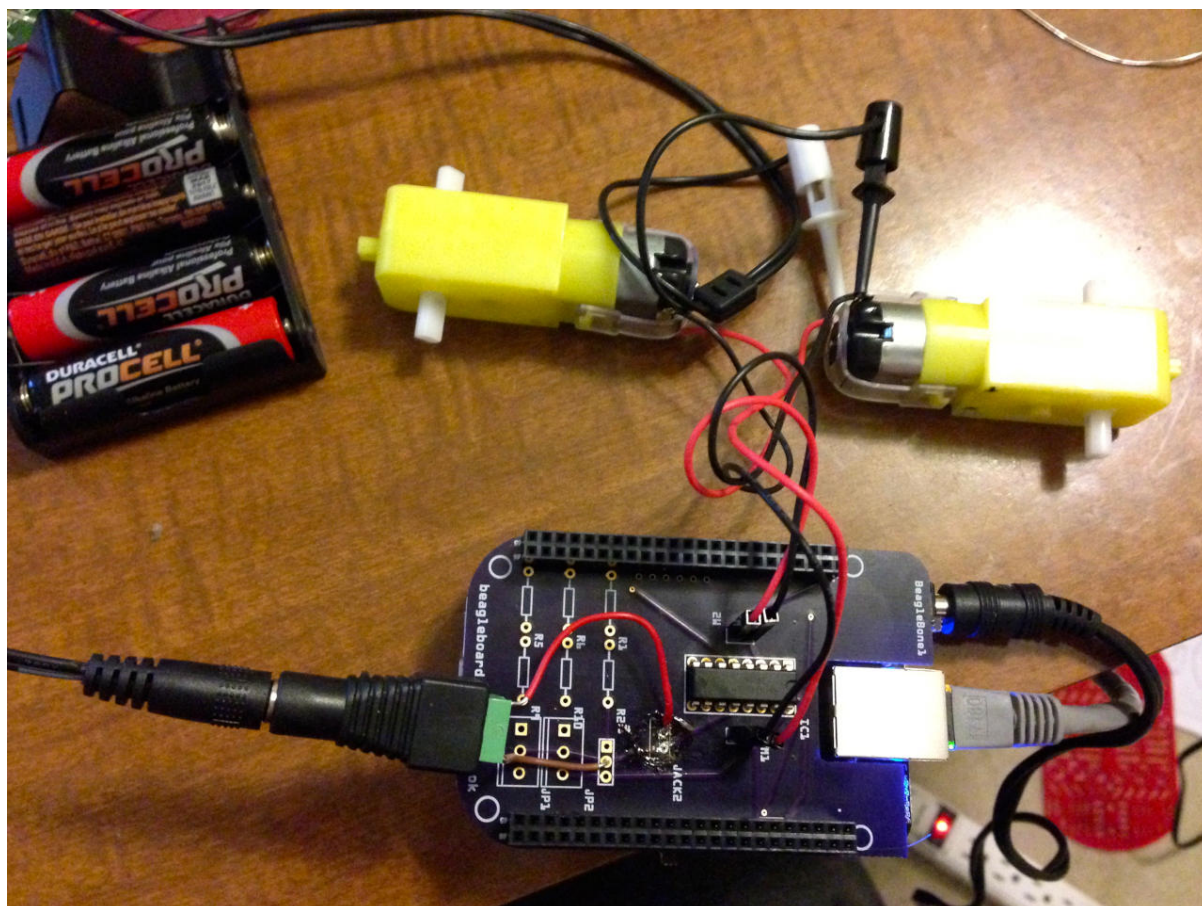


Fig. 9.25: QuickBot motors under test

Continue assembling and testing your board until you are happy. If you find issues, you might choose to cut traces and use point-to-point wiring to resolve your issues before placing an order for a new PCB. Better right the second time than the third!

## 9.14 Creating Contents for Your Cape Configuration EEPROM

### 9.14.1 Problem

Your cape is ready to go, and you want it to automatically initialize when the Bone boots up.

### 9.14.2 Solution

Complete capes have an I<sup>2</sup>C EEPROM on board that contains configuration information that is read at boot time. [Adventures in BeagleBone Cape EEPROMs](#) gives a helpful description of two methods for programming the EEPROM. [How to Roll your own BeagleBone Capes](#) is a good four-part series on creating a cape, including how to wire and program the EEPROM.

**Note:** The current effort to document how to enable software for a cape is ongoing at <https://docs.beagleboard.org/latest/boards/capes>.

## 9.15 Putting Your Cape Design into Production

### 9.15.1 Problem

You want to share your cape with others. How do you scale up?

### 9.15.2 Solution

CircuitHub offers a great tool to get a quick quote on assembled PCBs. To make things simple, I downloaded the CircuitCo MiniDisplay Cape Eagle design materials and uploaded them to CircuitHub.

After the design is uploaded, you'll need to review the parts to verify that CircuitHub has or can order the right ones. Find the parts in the catalog by changing the text in the search box and clicking the magnifying glass. When you've found a suitable match, select it to confirm its use in your design, as shown in [CircuitHub part matching](#).

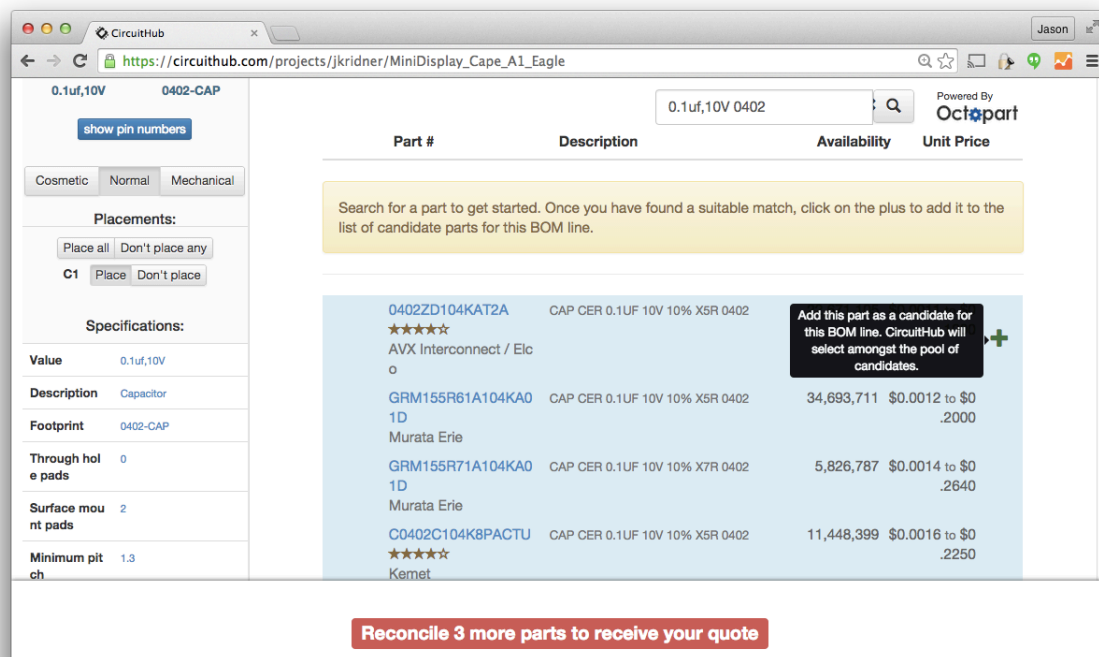


Fig. 9.26: CircuitHub part matching

When you've selected all of your parts, a quote tool appears at the bottom of the page, as shown in [CircuitHub quote generation](#).

Checking out the pricing on the MiniDisplay Cape (without including the LCD itself) in [CircuitHub price examples \(all prices USD\)](#), you can get a quick idea of how increased volume can dramatically impact the per-unit costs.



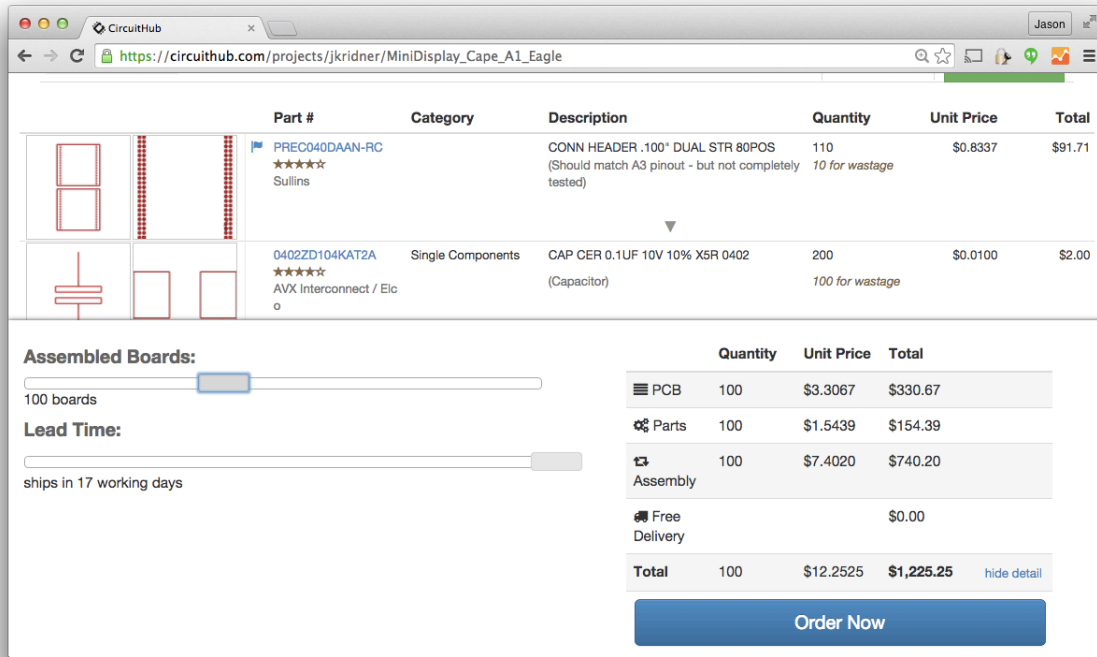


Fig. 9.27: CircuitHub quote generation

Table 9.1: CircuitHub price examples (all prices USD)

Quantity	1	10	100	1000	10,000
PCB	\$208.68	\$21.75	\$3.30	\$0.98	\$0.90
Parts	\$11.56	\$2.55	\$1.54	\$1.01	\$0.92
Assembly	\$249.84	\$30.69	\$7.40	\$2.79	\$2.32
Per unit	\$470.09	\$54.99	\$12.25	\$4.79	\$4.16
Total	\$470.09	\$550.00	\$1,225.25	\$4,796.00	\$41,665.79

Checking the [Crystalfontz web page](#) for the LCD, you can find the prices for the LCDs as well, as shown in [LCD pricing \(USD\)](#).

Table 9.2: LCD pricing (USD)

Quantity	1	10	100	1000	10,000
Per unit	\$12.12	\$7.30	\$3.86	\$2.84	\$2.84
Total	\$12.12	\$73.00	\$386.00	\$2,840.00	\$28,400.00

To enable more cape developers to launch their designs to the market, CircuitHub has launched a [group buy campaign site](#). You, as a cape developer, can choose how much markup you need to be paid for your work and launch the campaign to the public. Money is only collected if and when the desired target quantity is reached, so there's no risk that the boards will cost too much to be affordable. This is a great way to cost-effectively launch your boards to market!

There's no real substitute for getting to know your contract manufacturer, its capabilities, communication style, strengths, and weaknesses. Look around your town to see if anyone is doing this type of work and see if they'll give you a tour.

---

**Note:** Don't confuse CircuitHub and CircuitCo. CircuitCo is closed.

---

## Chapter 10

# Parts and Suppliers

The following tables list where you can find the parts used in this book. We have listed only one or two sources here, but you can often find a given part in many places.

Table 10.1: United States suppliers

Supplier	Website	Notes
Adafruit	<a href="http://www.adafruit.com">http://www.adafruit.com</a>	Good for modules and parts
Amazon	<a href="http://www.amazon.com/">http://www.amazon.com/</a>	Carries everything
Digikey	<a href="http://www.digikey.com/">http://www.digikey.com/</a>	Wide range of components
MakerShed	<a href="http://www.makershed.com/">http://www.makershed.com/</a>	Good for modules, kits, and tools
SeeedStudio	<a href="https://www.seeedstudio.com/SBC-Beaglebone-Original-c-2031.html?">https://www.seeedstudio.com/SBC-Beaglebone-Original-c-2031.html?</a>	Low-cost modules
SparkFun	<a href="http://www.sparkfun.com">http://www.sparkfun.com</a>	Good for modules and parts

Table 10.2: Other suppliers

Supplier	Website	Notes
Element14	<a href="http://element14.com/BeagleBone">http://element14.com/BeagleBone</a>	World-wide BeagleBoard.org-compliant clone of BeagleBone Black, carries many accessories

## 10.1 Prototyping Equipment

Many of the hardware projects in this book use jumper wires and a breadboard. We prefer the preformed wires that lie flat on the board. [Jumper wires](#) lists places with jumper wires, and [Breadboards](#) shows where you can get breadboards.

Table 10.3: Jumper wires

Supplier	Website
Amazon	<a href="http://www.amazon.com/Elenco-Piece-Pre-formed-Jumper-Wire/dp/B0002H7AIG">http://www.amazon.com/Elenco-Piece-Pre-formed-Jumper-Wire/dp/B0002H7AIG</a>
Digikey	<a href="http://www.digikey.com/product-detail/en/TW-E012-000/438-1049-ND/643115">http://www.digikey.com/product-detail/en/TW-E012-000/438-1049-ND/643115</a>
SparkFun	<a href="https://www.sparkfun.com/products/124">https://www.sparkfun.com/products/124</a>

Table 10.4: Breadboards

Supplier	Website
Amazon	<a href="http://www.amazon.com/s/ref=nb_sb_noss_1?url=search-alias%3Dtoys-and-games&amp;field-keywords=breadboards&amp;srefix=breadboards%2Ctoys-and-games">http://www.amazon.com/s/ref=nb_sb_noss_1?url=search-alias%3Dtoys-and-games&amp;field-keywords=breadboards&amp;srefix=breadboards%2Ctoys-and-games</a>
Digikey	<a href="https://www.digikey.com/en/products/filter/solderless-breadboards/638">https://www.digikey.com/en/products/filter/solderless-breadboards/638</a>
SparkFun	<a href="https://www.sparkfun.com/search/results?term=breadboard">https://www.sparkfun.com/search/results?term=breadboard</a>
CircuitCo	<a href="https://elinux.org/BeagleBoneBreadboard">https://elinux.org/BeagleBoneBreadboard</a> (no longer manufactured, but design available)

If you want something more permanent, try [Adafruit's Perma-Proto Breadboard](#), laid out like a breadboard.

## 10.2 Resistors

We use 220  $\Omega$ , 1k, 4.7k, 10k, 20k, and 22 k $\Omega$  resistors in this book. All are 0.25 W. The easiest way to get all these, and many more, is to order [SparkFun's Resistor Kit](#). It's a great way to be ready for future projects, because it has 500 resistors.

If you don't need an entire kit of resistors, you can order a la carte from a number of places. DigiKey has more than a quarter million [through-hole resistors](#) at good prices, but make sure you are ordering the right one.

You can find the 10 k $\Omega$  trimpot (or variable resistor) at [SparkFun 10k POT](#) or [Adafruit 10k POT](#).

Flex resistors (sometimes called *flex sensors* or *bend sensors*) are available at [SparkFun flex resistors](#) and [Adafruit flex resistors](#).

## 10.3 Transistors and Diodes

The 2N3904 is a common NPN transistor that you can get almost anywhere. Even [Amazon NPN transistor](#) has it. [Adafruit NPN transistor](#) has a nice 10-pack. [SparkFun NPN transistor](#) lets you buy them one at a time. [DigiKey NPN transistor](#) will gladly sell you 100,000.

The 1N4001 is a popular 1A diode. Buy one at [SparkFun diode](#), 10 at [Adafruit diode](#), or 10,000 at [DigiKey diode](#).

## 10.4 Integrated Circuits

The PCA9306 is a small integrated circuit (IC) that converts voltage levels between 3.3 V and 5 V. You can get it cheaply in large quantities from [DigiKey PCA9306](#), but it's in a very small, hard-to-use, surface-mount package. Instead, you can get it from [SparkFun PCA9306 on a Breakout board](#), which plugs into a breadboard.

The L293D is an H-bridge IC with which you can control large loads (such as motors) in both directions. [SparkFun L293D](#), [Adafruit L293D](#), and [DigiKey L293D](#) all have it in a DIP package that easily plugs into a breadboard.

The ULN2003 is a 7 darlington NPN transistor IC array used to drive motors one way. You can get it from [DigiKey ULN2003](#). A possible substitution is ULN2803 available from [SparkFun ULN2003](#) and [Adafruit ULN2003](#).

The TMP102 is an I<sup>2</sup>C-based digital temperature sensor. You can buy them in bulk from [DigiKey TMP102](#), but it's too small for a breadboard. [SparkFun TMP102](#) sells it on a breakout board that works well with a breadboard.

The DS18B20 is a one-wire digital temperature sensor that looks like a three-terminal transistor. Both [SparkFun DS18B20](#) and [Adafruit DS18B20](#) carry it.

## 10.5 Opto-Electronics

LEDs are *light-emitting diodes*. LEDs come in a wide range of colors, brightnesses, and styles. You can get a basic red LED at [SparkFun red LED](#), [Adafruit red LED](#), and [DigiKey red LED](#).

Many places carry bicolor LED matrices, but be sure to get one with an I<sup>2</sup>C interface. [Adafruit LED matrix](#) is where I got mine.

## 10.6 Capes

There are a number of sources for capes for BeagleBone Black. [BeagleBoard.org capes page](#) keeps a current list.

## 10.7 Miscellaneous

Here are some things that don't fit in the other categories.

Table 10.5: Miscellaneous

3.3 V FTDI cable	SparkFun FTDI cable, Adafruit FTDI cable
USB WiFi adapter	Adafruit WiFi adapter
HDMI cable	SparkFun HDMI cable
Micro HDMI to HDMI cable	Adafruit HDMI to microHDMI cable
HDMI to DVI Cable	SparkFun HDMI to DVI cable
HDMI monitor	Amazon HDMI monitor
Powered USB hub	Amazon power USB hub, Adafruit power USB hub
Soldering iron	SparkFun soldering iron, Adafruit soldering iron
Oscilloscope	Adafruit oscilloscope
Multimeter	SparkFun multimeter, Adafruit multimeter
PowerSwitch Tail II	SparkFun PowerSwitch Tail II, Adafruit PowerSwitch Tail II
Servo motor	SparkFun servo motor, Adafruit servo motor
5 V power supply	SparkFun 5V power supply, Adafruit 5V power supply
3 V to 5 V motor	SparkFun 3V-5V motor, Adafruit 3V-5V motor
3 V to 5 V bipolar stepper motor	SparkFun 3V-5V bipolar stepper motor, Adafruit 3V-5V bipolar stepper motor
3 V to 5 V unipolar stepper motor	Adafruit 3V-5V unipolar stepper motor
Pushbutton switch	SparkFun pushbutton switch, Adafruit pushbutton switch
Magnetic reed switch	SparkFun magnetic reed switch
LV-MaxSonar-EZ1 Sonar Range Finder	SparkFun LV-MaxSonar-EZ1, Amazon LV-MaxSonar-EZ1
HC-SR04 Ultrasonic Range Sensor	Amazon HC-SR04
Rotary encoder	SparkFun rotary encoder, Adafruit rotary encoder
GPS receiver	SparkFun GPS, Adafruit GPS
BLE USB dongle	Adafruit BLE USB dongle
Syba SD-CM-UAUD USB Stereo Audio Adapter	Amazon USB audio adapter
Sabrent External Sound Box USB-SBCV	Amazon USB audio adapter (alt)
Vantec USB External 7.1 Channel Audio Adapter	Amazon USB audio adapter (alt2)



# Chapter 11

## Misc

Here are bits and pieces of ideas that are being developed.

### 11.1 BeagleConnect Freedom

Here are some notes on how to setup and use the Connect.

First get the flasher image from: <https://www.beagleboard.org/distros/beagleplay-home-assistant-webinar-demo-image>

Flash the eMMC (which also loads the cc1352 with the correct firmware)

Here's Jason's demo at the 2023 EOSS: [https://youtu.be/ZT9GEs3\\_ZYU?t=2195](https://youtu.be/ZT9GEs3_ZYU?t=2195)

```
debian@BeaglePlay:~$ sudo beagleconnect-start-gateway
[sudo] password for debian:
setting up wpanusb gateway for IEEE 802154 CHANNEL 1(906 Mhz)
RTNETLINK answers: File exists
RTNETLINK answers: Device or resource busy
PING 2001:db8::1(2001:db8::1) from fe80::212:4b00:29c4:cdee%lowpan0 lowpan0: 56 d
bytes
64 bytes from 2001:db8::1: icmp_seq=1 ttl=64 time=69.5 ms
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=66.2 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=37.5 ms
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=70.8 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=37.6 ms

--- 2001:db8::1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 37.528/56.326/70.823/15.411 ms
```

Fig. 11.1: beagleconnect-start-gateway

```
bone$ sudo beagleconnect-start-gateway
setting up wpanusb gateway for IEEE 802154 CHANNEL 1(906 Mhz)
RTNETLINK answers: File exists
RTNETLINK answers: Device or resource busy
PING 2001:db8::1(2001:db8::1) from fe80::212:4b00:29b9:9884%lowpan0 lowpan0:↵
↵56 data bytes
64 bytes from 2001:db8::1: icmp_seq=1 ttl=64 time=70.0 ms
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=66.6 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=37.6 ms
```

(continues on next page)

(continued from previous page)

```
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=37.6 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=37.6 ms

--- 2001:db8::1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 37.559/49.868/70.035/15.084 ms
```

### 11.1.1 Useful Links

<https://docs.micropython.org/en/latest/zephyr/quickref.html>

[https://docs.zephyrproject.org/latest/boards/arm/beagle\\_bcf/doc/index.html](https://docs.zephyrproject.org/latest/boards/arm/beagle_bcf/doc/index.html)

### 11.1.2 micropython Examples

Here is the output from running the examples from here: <https://docs.beagleboard.org/latest/boards/beagleconnect/freedom/demos-and-tutorials/using-micropython.html>

Plug the BeagleConnect Freedom into the USB on the Play.

```
bone:~$ sudo systemd-resolve --set-mdns=yes --interface=lowpan0
bone:~$ avahi-browse -r -t _zephyr._tcp
+ lowpan0 IPv6 zephyr                                _zephyr._tcp  -
  ↳ local
= lowpan0 IPv6 zephyr                                _zephyr._tcp  -
  ↳ local
hostname = [zephyr.local]
address = [2001:db8::1]
port = [12345]
txt = []
bone:~$ avahi-resolve -6 -n zephyr.local
zephyr.local    2001:db8::1
bone:~$ mcumgr conn add bcf0 type="udp" connstring="[2001:db8::1%lowpan0]:1337"
↳
Connection profile bcf0 successfully added
bone:~$ mcumgr -c bcf0 image list
Images:
image=0 slot=0
  version: hu.hu.hu
  bootable: true
  flags: active confirmed
  hash: 16a97391d2570eae80667cfd8c475cb051d4a4a600430b64cb52b59f5db4ce22
Split status: N/A (0)
bone:~$ mcumgr -c bcf0 shell exec "device list"
status=0

devices:
- GPIO_0 (READY)
- random@40028000 (READY)
- UART_1 (READY)
- UART_0 (READY)
- i2c@40002000 (READY)
- I2C_0S (READY)
requires: GPIO_0
requires: i2c@40002000
- flash-controller@40030000 (READY)
- spi@40000000 (READY)
requires: GPIO_0
- ieee802154g (READY)
```

(continues on next page)

(continued from previous page)

```

- gd25q16c@0 (READY)
requires: spi@40000000
- leds (READY)
- HDC2010-HUMIDITY (READY)
requires: I2C_0S
-
bone:~$ mcumgr -c bcf0 shell exec "net iface"
status=0

Hostname: zephyr

Interface 0x20002de4 (IEEE 802.15.4) [1]
=====
Link addr  : 3D:9A:B9:29:00:4B:12:00
MTU        : 125
Flags      : AUTO_START,IPv6
IPv6 unicast addresses (max 3):
           fe80::3f9a:b929:4b:1200 autoconf preferred infinite
           2001:db8::1 manual preferred infinite
IPv6 multicast addresses (max 4):
           ff02::1
           ff02::1:ff4b:1200
           ff02::1:ff00:1
bone:~$ tio /dev/ttyACM0

```

Press the RST button on the Connect.

```

I: gd25q16c@0: SFDP v 1.6 AP ff with 2 PH
I: PH0: ff00 rev 1.6: 16 DW @ 30
I: gd25q16c@0: 2 MiBy flash
I: PH1: ffc8 rev 1.0: 3 DW @ 90
*** Booting Zephyr OS build zephyr-v3.2.0-3470-g14e193081b1f ***
I: Starting bootloader
I: Primary image: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
I: Scratch: magic=unset, swap_type=0x1, copy_done=0x3, image_ok=0x3
I: Boot source: primary slot
I: Swap type: none
I: Bootloader chainload address offset: 0x20000
I: Jumping to the first image slot

[00:00:00.001,464] <inf> spi_nor: gd25q16c@0: SFDP v 1.6 AP ff with 2 PH
[00:00:00.001,464] <inf> spi_nor: PH0: ff00 rev 1.6: 16 DW @ 30
[00:00:00.001,983] <inf> spi_nor: gd25q16c@0: 2 MiBy flash
[00:00:00.002,014] <inf> spi_nor: PH1: ffc8 rev 1.0: 3 DW @ 90
uart:~$ build time: Feb 22 2023 08:09:25MicroPython v1.19.1 on 2023-02-22;
↳zephyr-beagleconnect_freedom with unknown-cpu
Type "help()" for more information.
>>>

```

## 11.2 Setting up shortcuts to make life easier

We'll be ssh'ing from the host to the bone often, here are some shortcuts I use so instead of typing `ssh debian@192.168.7.2` and a password every time. I can enter `ssh bone` and no password.

First edit `/etc/hosts` and add a couple of lines.

```
host$ sudo nano /etc/hosts
```



You may use whatever editor you want. I suggest `nano` since it's easy to figure out. Add the following to the end of `/etc/hosts` and quit the editor.

```
192.168.7.2 bone
```

Now you can connect with

```
host$ ssh debian@bone
```

Let's make it so you don't have to enter `debian`. On your host computer, put the following in `~/.ssh/config` (Note: `~` is a shortcut for your home directory.)

```
Host bone
User debian
UserKnownHostsFile /dev/null
StrictHostKeyChecking no
```

These say that whenever you login to `bone`, login as `debian`. Now you can enter.

```
host$ ssh bone
```

One last thing, let's make it so you don't have to add a password. Back to your host.

```
host$ ssh-keygen
```

Accept all the defaults and then

```
host$ ssh-copy-id bone
```

Now all you have to enter is

```
host$ ssh bone
```

and no password is required. If you, especially virtual machine users, get an error says "sign\_and\_send\_pubkey: signing failed: agent refused operation", you can solve this by entering

```
host$ ssh-add
```

which adds the private key identities to the authentication agent. Then you should be able to `ssh bone` without problems.

### 11.3 Setting up a root login

By default the image we are running doesn't allow a root login. You can always `sudo` from `debian`, but sometimes it's nice to login as `root`. Here's how to setup `root` so you can login from your host without a password.

```
host$ ssh bone
bone$ sudo -i
root@bone# nano /etc/ssh/sshd_config
```

Search for the line

```
#PermitRootLogin prohibit-password
```

and change it to

```
PermitRootLogin yes
```

(The `#` symbol indicates a comment and must be removed in order for the setting to take effect.)

Save the file and quit the editor. Restart `ssh` so it will reread the file.

```
root@bone# systemctl restart sshd
```

And assign a password to root.

```
root@bone# passwd
```

Now open another window on your host computer and enter:

```
host$ ssh-copy-id root@bone
```

and enter the root password. Test it with:

```
host$ ssh root@bone
```

You should be connected without a password. Now go back to the Bone and turn off the root password access.

```
root@bone# nano /etc/ssh/sshd_config
```

Restore the line:

```
#PermitRootLogin prohibit-password
```

and restart sshd.

```
root@bone# systemctl restart sshd
root@bone# exit
bone$ exit
```

You should now be able to go back to your host computer and login as root on the bone without a password.

```
host$ ssh root@bone
```

You have access to your bone without passwords only from you host computer. Try it from another computer and see what happens

## 11.4 Wireshark

[Wireshark](#) is a network protocol analyzer that can be run on the Beagle or the host computer to see what's happening on the network.

### 11.4.1 Running Wireshark on the Beagle

If you have X11 installed on the Beagle and you are running Linux on your host you can run Wireshark on the Beagle and have it display on the host.

---

**Tip:** A quick way to see if you have X windows installed is to ssh to your Beagle. At the prompt enter `xfce` then enter `<TAB><TAB>`. If you see a list of completions, you have X installed.

---

1. First ssh to the Beagle using the `-X` flag.

```
host$ ssh -X debian@10.0.5.10

bone$ sudo apt update
bone$ sudo apt install wireshark
bone$ sudo usermod -a -G wireshark debian
bone$ exit

host$ ssh -X debian@10.0.5.10
host$ wireshark
```

The `-X` flag sets the `DISPLAY` variable on the Beagle so it knows where to display the Beagle's graphical data on the host. We then install `wireshark` and add `debian` to the `wireshark` group. We then log out and log back in again to be sure we are in the `wireshark` group. Finally we start `wireshark`.

You should see something like [Wireshark start screen](#).

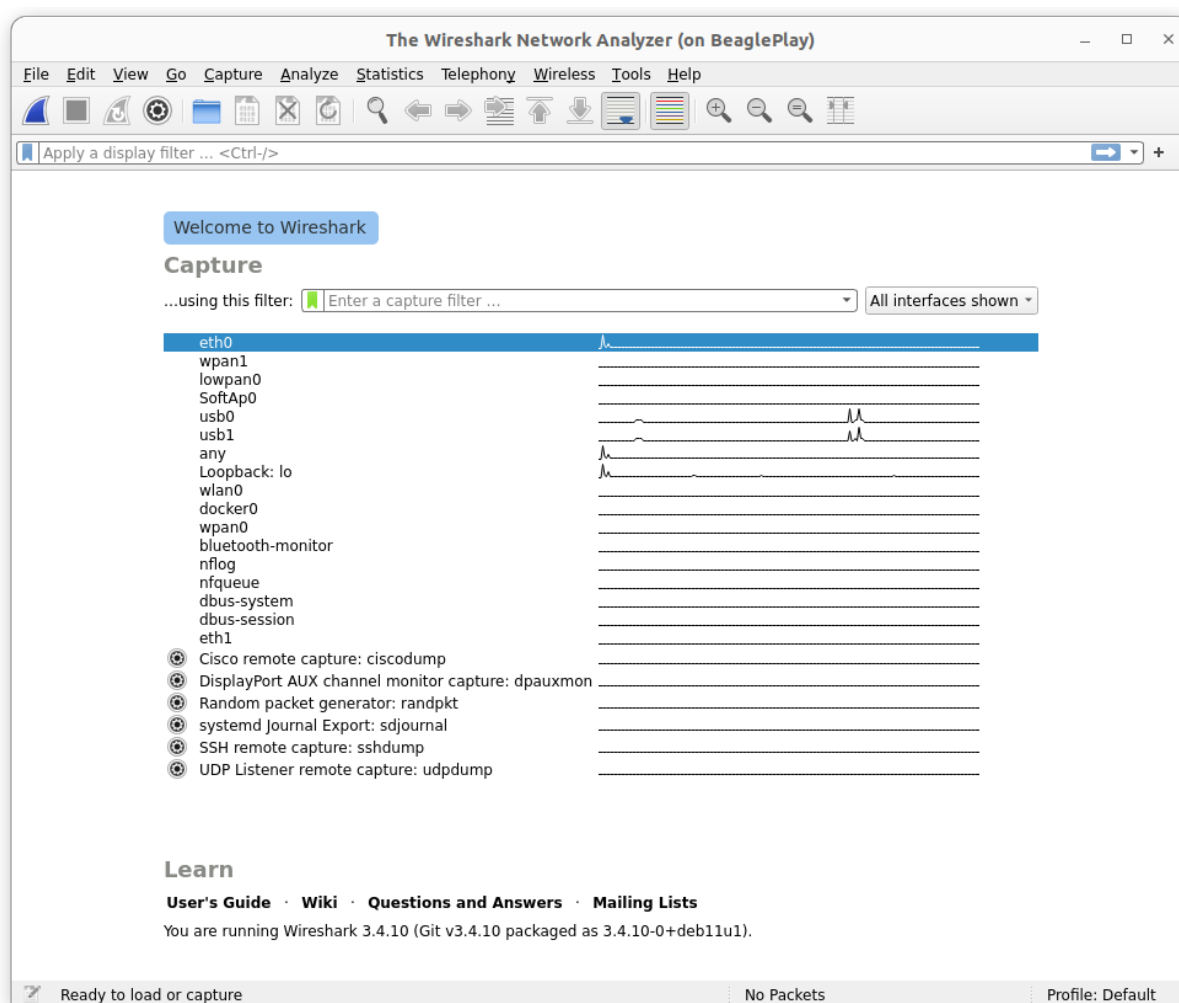


Fig. 11.2: Wireshark start screen

### 11.4.2 Running Wireshark on the host

If you don't have X11 installed on the Beagle, you can run `wireshark` on your host computer and capture the packets on the Beagle. These instructions come from: <https://serverfault.com/questions/362529/how-can-i-sniff-the-traffic-of-remote-machine-with-wireshark>

First login to the Beagle and install `tcpdump`. Use your Beagle's IP address.

```
host$ ssh 192.168.7.2
bone$ sudo apt update
bone$ sudo apt install tcpdump
bone$ exit
```

Next, create a named pipe and have `wireshark` read from it.

```
host$ mkfifo /tmp/remote
host$ wireshark -k -i /tmp/remote
```

Then, run tcpdump over ssh on your remote machine and redirect the packets to the named pipe:

```
host$ ssh root@192.168.7.2 "tcpdump -s 0 -U -n -w - -i any not port 22" > /
↳tmp/remote
```

**Tip:** For this to work you will need to follow in instructions in [Setting up a root login](#).

### 11.4.3 Sharking the wpan radio

Now that you have Wireshark set up, you can view traffic from the Play's wpan radio. First, set up the network by running:

```
bone:~$ beagleconnect-start-gateway
```

Go to Wireshark and in the field that says *Apply a display filter...* enter, `wpan || 6lowpan || ipv6`. This will display three types of packets. Be sure to hit Enter.

Now generate some traffic:

```
bone:~$ ping6 -I lowpan0 2001:db8::1 -c 5 -p ca11ab1ebeeF
```

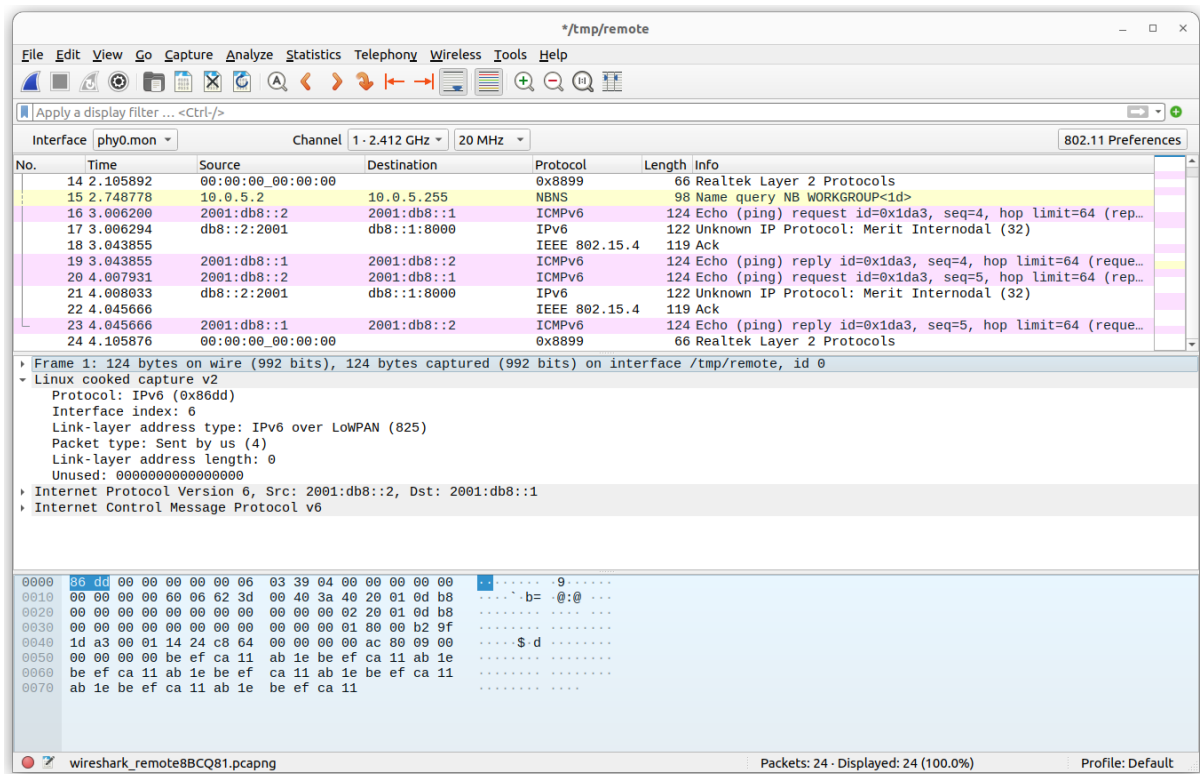


Fig. 11.3: Wireshark ping6 -I lowpan0 2001:db8::1 -c 5 -p ca11ab1ebeeF

You can see the pattern `ca11ab1ebeeF` appears in the packets.

## 11.5 Find what UU is in i2cdetect

### 11.5.1 Problem

You run `i2cdetect` and want to know what the **UU**'s are.

```
bone:~$ i2cdetect -y -r 2
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  UU  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  40  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  UU  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  UU  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

### 11.5.2 Solution

Running `man i2cdetect` shows that:

“UU”. Probing was skipped, because this address is currently in use by a driver. This strongly suggests that there is a chip at this address.

You can quickly see what the drivers are by looking at `/sys/bus/i2c/devices`

```
bone:~$ cd /sys/bus/i2c/devices
bone:~$ ls
2-0030  2-0050  2-0068  4-004c  i2c-1  i2c-2  i2c-3  i2c-4  i2c-5
```

Here on the BeagleY-AI we see there are 5 i2c buses (`i2c-1`, `i2c-2`, `i2c-3`, `i2c-4` and `i2c-5`). There are three devices on bus 2 (`2-0030`, `2-0050` and `2-0068`) and one device on bus 4 (`4-004c`). The first digit is the bus number and the last digits are the address on the bus in hex. You can see what these devices are by running:

```
bone:~$ cat */name
tps65219
24c32
ds1340
it66122
OMAP I2C adapter
OMAP I2C adapter
OMAP I2C adapter
OMAP I2C adapter
OMAP I2C adapter
```

You can Google the names to see what they are. For example, the `24c32` is a 32K EEPROM by Microchip.

## 11.6 Converting a tmp117 to a tmp114

### 11.6.1 Problem

You have a tmp114 temperature sensor and you need a driver for it.

### 11.6.2 Solution

Find a similar driver and convert it to the tmp114.

Let's first see if there is a driver for it already. Run the following on the bone using the tab key in place of `<tab>`.

```
bone$ modinfo tmp<tab><tab>
tmp006 tmp007 tmp102 tmp103 tmp108 tmp401 tmp421 tmp513
bone$ modinfo tmp
```

Here you see a list of modules that match *tmp*, unfortunately *tmp114* is not there. Let's see if there are any matches in */lib/modules*.

```
bone$ find /lib/modules/ -iname "*tmp*"
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/iio/temperature/tmp006.ko.
→xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/iio/temperature/tmp007.ko.
→xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp103.ko.xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp421.ko.xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp108.ko.xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp513.ko.xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp401.ko.xz
/lib/modules/5.10.168-ti-arm64-r104/kernel/drivers/hwmon/tmp102.ko.xz
```

Looks like the same list, but here we can see what type of driver it is, either *hwmon* or *iio*. *hwmon* is an older [hardware monitor](#). *iio* is the newer, and preferred, [Industrial IO driver](#). Googling *tmp006* and *tmp007* shows that they are Infrared Thermopile Sensors, not the same as the *tmp114*. (Google it). Let's keep looking for a more compatible device.

Browse over to <http://kernel.org> to see if there are *tmp114* drivers in the newer versions of the kernel. The first line in the table is **mainline**. Click on the **browse** link on the right.

Here you will see the top level of the Linux source tree for the *mainline* version of the kernel.

Click on **drivers** and then **iio**. Finally, since *tmp114* is a temperature sensor, click on **temperature**.

Here you see all the source code for the *iio* temperature drivers for the mainline version of the kernel. We've seen *tmp006* and *tmp007* as before, *tmp117* is new. Maybe it will work. Click on **tmp117.c** to see the code. Looks like it also works for the *tmp116* too.

Let's try converting it to work with the *tmp114*.

A quick way to copy the code to the bone is to right-click on the **plain** link and select *Copy link address*. Then, on the bone enter **wget** and paste the link. Mine looks like the following, yours will be similar.

```
bone$ wget https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.
→git/plain/drivers/iio/temperature/tmp117.c?h=v6.4-rc7
bone$ mv 'tmp117.c?h=v6.4-rc7' tmp117.c
bone$ cp tmp117.c tmp114.c
```

The **mv** command moves the downloaded file to a usable name and the **cp** copies to a new file with the new name.

### Compiling the module

Next we need to compile the driver. To do this we need to load the corresponding header files for the version of the kernel that's being run.

```
bone$ uname -r
5.10.168-ti-arm64-r105
```

Here you see which version I'm running, yours will be similar. Now load the headers.

```
bone$ sudo apt install linux-headers-`uname -r`
```

Next create a *Makefile*. Put the following in a file called *Makefile*.

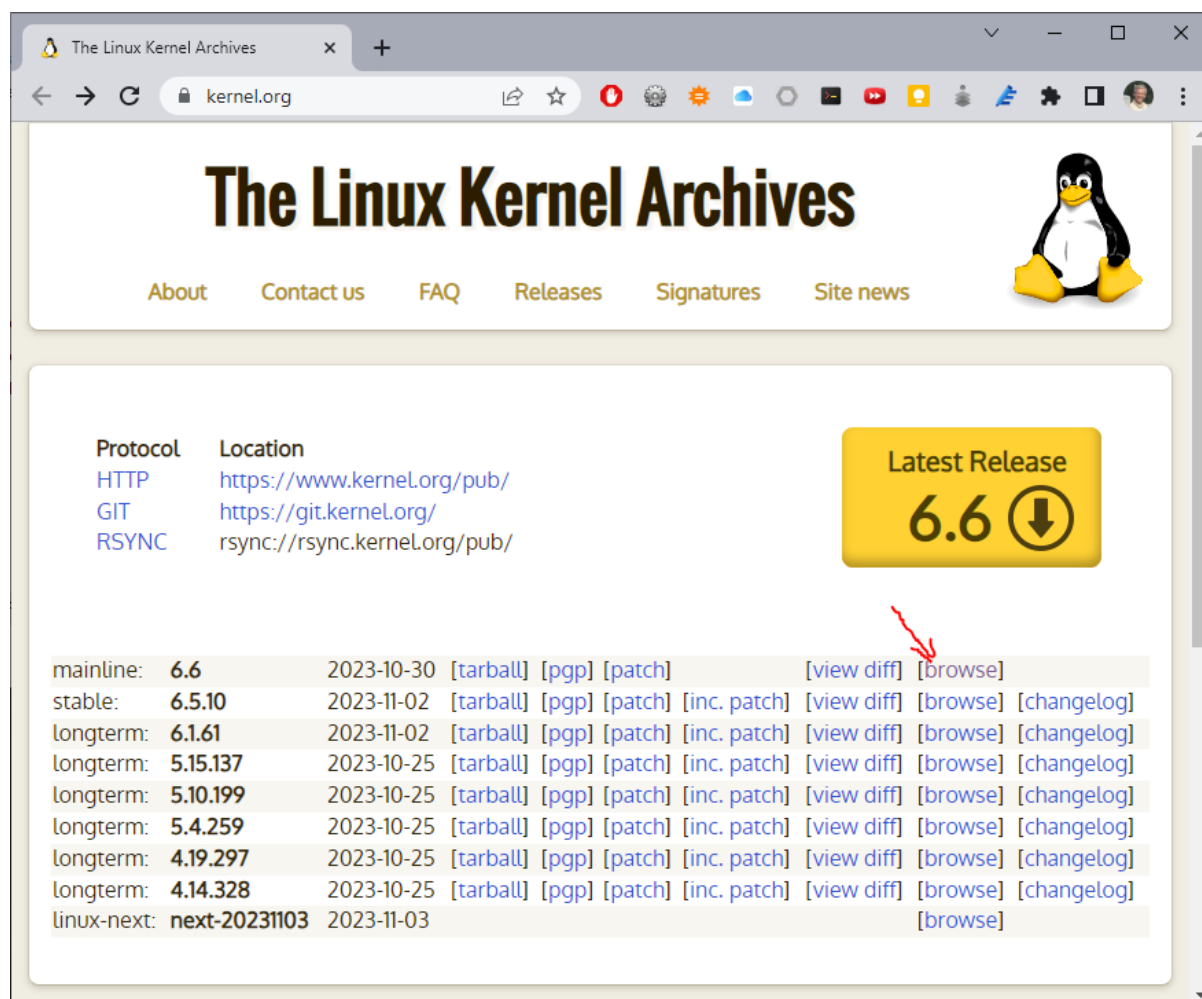


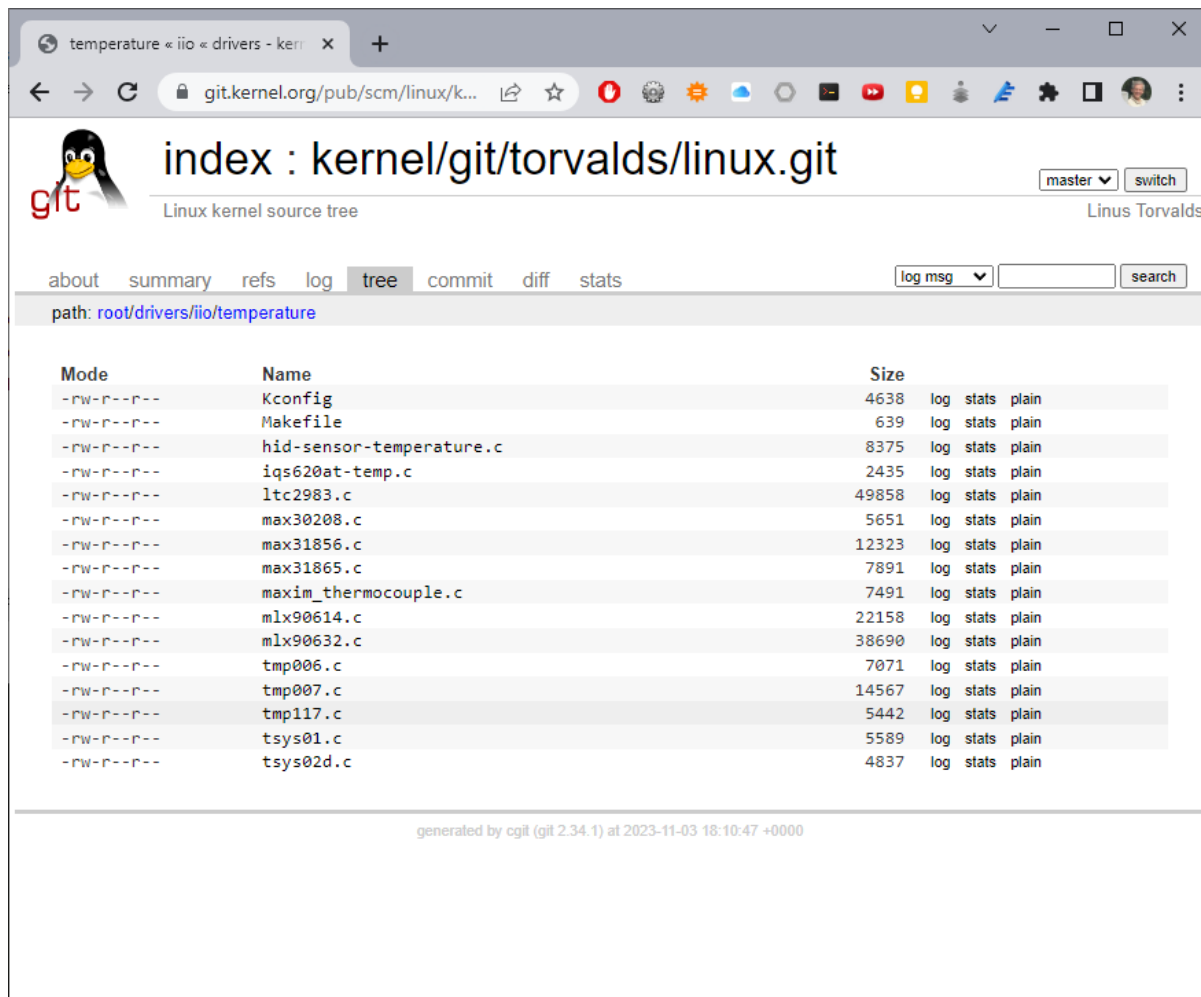
Fig. 11.4: The Linux Kernel Archives, kernel.org

The screenshot shows the GitHub interface for the Linux kernel source tree. The page title is "index : kernel/git/torvalds/linux.git" and it is identified as the "Linux kernel source tree" by "Linus Torvalds". The "tree" tab is selected, displaying a list of files and directories. The "drivers" directory is highlighted, showing its size and file type.

Mode	Name	Size	log	stats	plain
-rw-r--r--	.clang-format	20561	log	stats	plain
-rw-r--r--	.cocciconfig	59	log	stats	plain
-rw-r--r--	.get_maintainer.ignore	151	log	stats	plain
-rw-r--r--	.gitattributes	105	log	stats	plain
-rw-r--r--	.gitignore	2087	log	stats	plain
-rw-r--r--	.mailmap	36608	log	stats	plain
-rw-r--r--	.rustfmt.toml	369	log	stats	plain
-rw-r--r--	COPYING	496	log	stats	plain
-rw-r--r--	CREDITS	102435	log	stats	plain
d-----	Documentation	3049	log	stats	plain
-rw-r--r--	Kbuild	2573	log	stats	plain
-rw-r--r--	Kconfig	555	log	stats	plain
d-----	LICENSES	141	log	stats	plain
-rw-r--r--	MAINTAINERS	726660	log	stats	plain
-rw-r--r--	Makefile	67432	log	stats	plain
-rw-r--r--	README	727	log	stats	plain
d-----	arch	778	log	stats	plain
d-----	block	3112	log	stats	plain
d-----	certs	522	log	stats	plain
d-----	crypto	5954	log	stats	plain
d-----	drivers	4582	log	stats	plain
d-----	fs	5339	log	stats	plain
d-----	include	929	log	stats	plain
d-----	init	592	log	stats	plain
d-----	io_uring	1986	log	stats	plain
d-----	ipc	467	log	stats	plain
d-----	kernel	5152	log	stats	plain

Fig. 11.5: The Linux Kernel Archives, drivers





The screenshot shows a web browser displaying the Linux kernel source tree for the tmp117 driver. The browser address bar shows the URL `git.kernel.org/pub/scm/linux/k...`. The page title is "index : kernel/git/torvalds/linux.git" and the subtitle is "Linux kernel source tree". The page is for the "master" branch, and the user is "Linus Torvalds". The navigation menu includes "about", "summary", "refs", "log", "tree", "commit", "diff", and "stats". The current path is `root/drivers/iio/temperature`. The file list shows the following files and their sizes:

Mode	Name	Size	log	stats	plain
-rw-r--r--	Kconfig	4638	log	stats	plain
-rw-r--r--	Makefile	639	log	stats	plain
-rw-r--r--	hid-sensor-temperature.c	8375	log	stats	plain
-rw-r--r--	iqs620at-temp.c	2435	log	stats	plain
-rw-r--r--	ltc2983.c	49858	log	stats	plain
-rw-r--r--	max30208.c	5651	log	stats	plain
-rw-r--r--	max31856.c	12323	log	stats	plain
-rw-r--r--	max31865.c	7891	log	stats	plain
-rw-r--r--	maxim_thermocouple.c	7491	log	stats	plain
-rw-r--r--	mlx90614.c	22158	log	stats	plain
-rw-r--r--	mlx90632.c	38690	log	stats	plain
-rw-r--r--	tmp006.c	7071	log	stats	plain
-rw-r--r--	tmp007.c	14567	log	stats	plain
-rw-r--r--	tmp117.c	5442	log	stats	plain
-rw-r--r--	tsys01.c	5589	log	stats	plain
-rw-r--r--	tsys02d.c	4837	log	stats	plain

generated by cgit (git 2.34.1) at 2023-11-03 18:10:47 +0000

Fig. 11.6: The Linux Kernel Archives, tmp117 driver

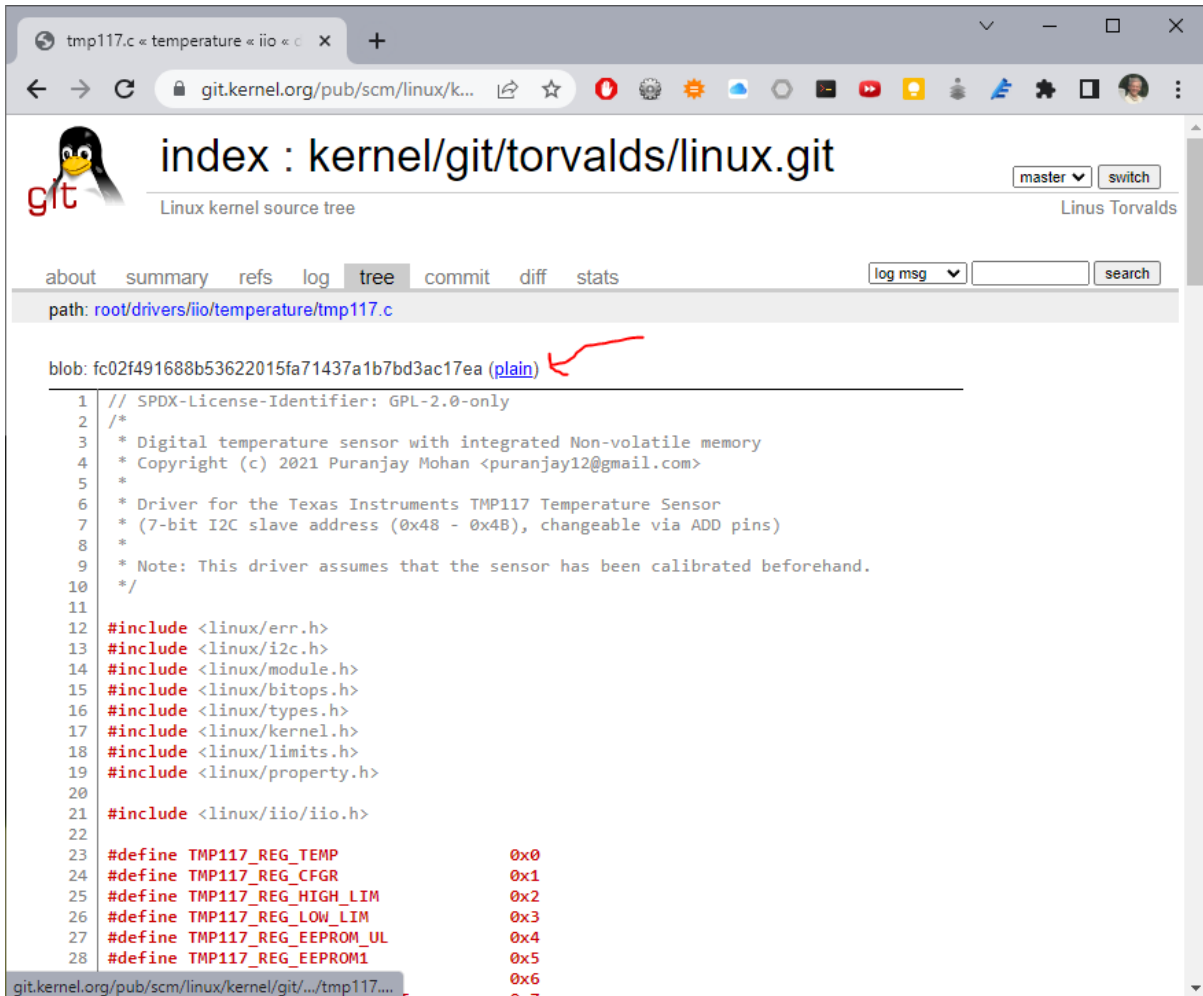


Fig. 11.7: The Linux Kernel Archives, plain button

Listing 11.1: Makefile for compiling module (Makefile)

```

1 obj-m += tmp114.o
2
3 KDIR ?= /lib/modules/$(shell uname -r)/build
4 PWD := $(CURDIR)
5
6 all:
7     make -C $(KDIR) M=$(PWD) modules
8
9 clean:
10    make -C $(KDIR) M=$(PWD) cleanobj-m += tmp114.o
11
12 KDIR ?= /lib/modules/$(shell uname -r)/build
13 PWD := $(CURDIR)
14
15 all:
16    make -C $(KDIR) M=$(PWD) modules
17
18 clean:
19    make -C $(KDIR) M=$(PWD) clean

```

## Makefile

Now you are ready to compile:

```

bone$ make
make -C /lib/modules/5.10.168-ti-arm64-r105/build M=/home/debian/play modules
make[1]: Entering directory '/usr/src/linux-headers-5.10.168-ti-arm64-r105'
CC [M] /home/debian/play/tmp114.o
/home/debian/play/tmp114.c: In function 'tmp117_identify':
/home/debian/play/tmp114.c:150:7: error: implicit declaration of function
↳ 'i2c_client_get_device_id'; did you mean 'i2c_get_device_id'? [-
↳ Werror=implicit-function-declaration]
150 |   id = i2c_client_get_device_id(client);
    |         ^~~~~~
    |         i2c_get_device_id
/home/debian/play/tmp114.c:150:5: warning: assignment to 'const struct i2c_
↳ device_id *' from 'int' makes pointer from integer without a cast [-Wint-
↳ conversion]
150 |   id = i2c_client_get_device_id(client);
    |         ^
cc1: some warnings being treated as errors
make[2]: *** [scripts/Makefile.build:286: /home/debian/play/tmp114.o] Error 1
make[1]: *** [Makefile:1822: /home/debian/play] Error 2
make[1]: Leaving directory '/usr/src/linux-headers-5.10.168-ti-arm64-r105'
make: *** [Makefile:7: all] Error 2

```

Well, the good news is, it is compiling, that means it found the correct headers. But now the work begins converting to the tmp114.

## Converting to the tmp114

You are mostly on your own for this part, but here are some suggestions:

- First get it to compile without errors. In this case, the function at line 150 isn't defined. Try commenting it out and recompiling.
- Once it's compiling without errors, try running it. First open another window and login to beagle. Then run:

```
bone$ dmesg -Hw
```

This will display the kernel messages. The **-H** put them in *human* readable form, and the **-w** waits for more messages.

- Next, “insert” it in the running kernel:

```
bone$ sudo insmod tmp114.ko
```

If all worked you shouldn't see any messages, either after the command or in the dmesg window. If you want to insert the module again, you will have to remove it first. Remove with:

```
bone$ sudo rmmod tmp114
```

Now we need to tell the kernel we have an I<sup>2</sup>C device and which bus and which address.

### Finding your I<sup>2</sup>C device

Each I<sup>2</sup>C device appears at a certain address on a given bus. My device is on bus 3, so I run:

```
bone$ i2cdetect -y -r 3
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  4d  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

This shows there is a device at address **0x4d**. If you don't know your bus number, just try a few until you find it.

The temperature is in register 0 for my device and it's 16 bits (one word), it is read with:

```
bone$ i2cget -y 3 0x4d 0 w
0xb510
```

The tmp114 swaps the two bytes, so the real temperature is **0x10b5**, or so. You need to look up the data sheet to learn how to convert it.

### Registers and IDs

Each I<sup>2</sup>C device has a number of internal registers that interact with the device. The tmp114 uses different register numbers than the tmp117, so you need to change these values. To do this, Google for the data sheets for each and look them up. I found them at: <https://www.ti.com/lit/gpn/tmp114> and <https://www.ti.com/lit/gpn/tmp117>.

### Creating a new device

Once you've converted the module for the tmp114 and inserted it, you can now create a new device.

```
bone$ cd /sys/class/i2c-adapter/i2c-3
bone$ sudo chgrp gpio *
bone$ sudo chmod g+w *
bone$ ls -ls
total 0
0 --w--w---- 1 root gpio 4096 Jun 22 18:24 delete_device
```

(continues on next page)

(continued from previous page)

```

0 lrwxrwxrwx 1 root root    0 Jan  1  1970 device -> ../../20030000.i2c
0 drwxrwxr-x 3 root gpio    0 Jun 22 18:20 i2c-dev
0 -r--rw-r-- 1 root gpio 4096 Jun 22 18:20 name
0 --w--w---- 1 root gpio 4096 Jun 22 18:20 new_device
0 lrwxrwxrwx 1 root root    0 Jan  1  1970 of_node -> ../../../../../../
->firmware/devicetree/base/bus@f0000/i2c@20030000
0 drwxrwxr-x 2 root gpio    0 Jun 22 18:20 power
0 lrwxrwxrwx 1 root root    0 Jan  1  1970 subsystem -> ../../../../../../bus/
->i2c
0 -rw-rw-r-- 1 root gpio 4096 Jun 22 18:20 uevent

```

The first line changes to the directory to where we can create the new device. The final **3** in the path is for bus **3**, your mileage may vary. We then change the group to **gpio** and give it write permission. You only need to do this once.

Now make a new device.

```
bone$ echo tmp114 0x4d > new_device
```

Look in the dmesg window and you should see:

```

[Jun22 19:24] tmp114 3-004d: tmp114_identify id (0x1114)
[ +0.000027] tmp114 3-004d: tmp114_probe id (0x1114)
[ +0.000502] i2c i2c-3: new_device: Instantiated device tmp114 at 0x4d

```

It's been found! Let's see what it knows about it.

```

bone$ iio_info
Library version: 0.24 (git tag: v0.24)
...
    iio:device1: tmp114
        1 channels found:
            temp: (input)
            2 channel-specific attributes found:
                attr 0: raw value: 4257
                attr 1: scale value: 7.812500
        No trigger on this device

```

I've left out some of the lines, at the bottom you see the tmp114, and two values (**raw** and **scale**) that were read from it. Let's read them ourselves. Do an `ls` and you'll see a new directory, **3-004d**. This is address 0x4d on bus 3, just what we wanted.

```

bone$ cd 3-004d/iio:device1
bone$ ls
dev in_temp_raw in_temp_scale name power subsystem uevent
bone$ cat in_temp_raw
4275

```

You'll have to look in the datasheet to learn how to convert the temperature.

If you try to run `i2cget` again, you'll get an error:

```

bone$ i2cget -y 3 0x4d 0 w
Error: Could not set address to 0x4d: Device or resource busy

```

This is because the module is using it. Delete the device and you'll have access again.

```

bone$ echo 0x4d > /sys/class/i2c-adapter/i2c-3/delete_device
bone$ i2cget -y 3 0x4d 0 w
0x8e10

```

You should also see a message in `dmesg`.

## 11.7 Documenting with Sphinx

### 11.7.1 Problem

You want to add or update the Beagle documentation.

### 11.7.2 Solution

BeagleBoard.org uses the [Sphinx Python Documentation Generator](#) and the `rst` markup language.

Here's what you need to do to fork the repository and render a local copy of the documentation. Browse to <https://docs.beagleboard.org/latest/> and click on the **Edit on GitLab** button on the upper-right of the page. Clone the repository.

```
bash$ git clone git@git.beagleboard.org:docs/docs.beagleboard.io.git
bash$ cd docs.beagleboard.io
```

Then run the following to load the **code** submodule

```
bash$ git submodule update --init
```

Set up the environment for Sphinx.

```
bash$ python -m venv .venv
bash$ source .venv/bin/activate
bash$ pip install -r ./requirements.txt
bash$ make livehtml
```

This starts a local web server that you can point your browser to to see the formatted text.

Now, sync changes with upstream:

```
bone$ git remote add upstream https://git.beagleboard.org/docs/docs.
->beagleboard.io.git
bone$ git fetch upstream
bone$ git pull upstream main
```

### Downloading Sphinx

Run the following to download Sphinx. Note: This will take a while, it loads some 6G bytes.

```
bone$ sudo apt update
bone$ sudo apt upgrade
bone$ sudo apt install -y \
    make git wget \
    doxygen graphviz librsvg2-bin \
    texlive-latex-base texlive-latex-extra latexmk texlive-fonts-recommended
->\
    python3 python3-pip \
    python3-sphinx python3-sphinx-rtd-theme python3-sphinxcontrib.
->svg2pdfconverter \
    python3-pil \
    imagemagick-6.q16 librsvg2-bin webp \
    texlive-full texlive-latex-extra texlive-fonts-extra \
    fonts-freefont-otf fonts-dejavu fonts-dejavu-extra fonts-freefont-ttf
bone$ python3 -m pip install --upgrade pip
bone$ pip install -U sphinx_design
bone$ pip install -U sphinxcontrib-images
bone$ pip install -U sphinx-serve
```

These instructions came from [lorforlinux](#) on the Beagleboard Slack channel.

Now go to the cloned `docs.beagleboard.io` repository folder and do the following. To clean build directory:

```
bone$ cd docs.beagleboard.io
bone$ make clean
```

To generate HTML output of docs:

```
bone$ make html
```

To generate PDF output of docs:

```
bone$ make latexpdf
```

To preview docs on your local machine:

```
bone$ sphinx-serve
```

Then point your browser to `localhost:8081`.

---

**Tip:** You can keep the `sphinx-serve` running until you clean the build directory using `make clean`. Warnings will be hidden after first run of `make html` or `make latexpdf`, to see all the warnings again just run `make clean` before building HTML or PDF

---

### Creating A New Book

- Create a new book folder here: <https://git.beagleboard.org/docs/docs.beagleboard.io/-/tree/main/books>
- Create rst files for all the chapters in there respective folders so that you can easily manage media for that chapter as shown here: <https://git.beagleboard.org/docs/docs.beagleboard.io/-/tree/main/books/pru-cookbook>
- Create an `index.rst` file in the book folder and add a table of content (toc) for all the chapters. For example see this file: <https://git.beagleboard.org/docs/docs.beagleboard.io/-/raw/main/books/pru-cookbook/index.rst>
- Add the `bookname/index.rst` reference in the main index file as well: <https://git.beagleboard.org/docs/docs.beagleboard.io/-/raw/main/books/index.rst>
- At last you have to update the two files below to render the book in HTML and PDF version of the docs respectively: <https://git.beagleboard.org/docs/docs.beagleboard.io/-/raw/main/index.rst> <https://git.beagleboard.org/docs/docs.beagleboard.io/-/raw/main/index-tex.rst>

## 11.8 Running Sparkfun's qwiic Python Examples

Many of the Sparkfun qwiic devices have Python examples showing how to use them. Unfortunately the examples assume I<sup>2</sup>C bus 1 is used, but the qwiic bus on the Play is bus 5. Here is a quick hack to get the Sparkfun Python examples to use bus 5. I'll show it for the Joystick, but it should work for the others as well.

First, browse to Sparkfun's qwiic Joystick page, <https://www.sparkfun.com/products/15168> and click on the **DOCUMENTS** tab and then on **Python Package**. Follow the pip installation instructions (`sudo pip install sparkfun-qwiic-joystick`)

Next, uninstall the current qwiic I<sup>2</sup>C package.

```
bone$ sudo pip uninstall sparkfun-qwiic-i2c
```

Then clone the Qwiic I<sup>2</sup>C repo:

```
bone$ git clone git@github.com:sparkfun/Qwiic_I2C_Py.git
bone$ cd Qwiic_I2C_Py/qwiic_i2c
```

Edit `linux_i2c.py` and go to around line 62 and change it to:

```
iBus = 5
```

Next, cd up a level to the `Qwiic_I2C_Py` directory and reinstall

```
bone$ cd ..
bone$ sudo python setup.py install
```

Finally, run one of the Joystick examples. If it isn't using bus 5, try reinstalling `setup.py` again.

### 11.8.1 Qwiic Alphanumeric display

Here's the repo I used for this display. [https://github.com/thess/qwiic\\_alphanumeric\\_py](https://github.com/thess/qwiic_alphanumeric_py)

## 11.9 Controlling LEDs by Using SYSFS Entries

### 11.9.1 Problem

You want to control the onboard LEDs from the command line.

### 11.9.2 Solution

On Linux, *everything is a file* that is, you can access all the inputs and outputs, the LEDs, and so on by opening the right file and reading or writing to it. For example, try the following:

```
bone$ cd /sys/class/leds/
bone$ ls
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
```

What you are seeing are four directories, one for each onboard LED. Now try this:

```
bone$ cd beaglebone\:green\:usr0
bone$ ls
brightness device max_brightness power subsystem trigger uevent
bone$ cat trigger
none nand-disk mmc0 mmc1 timer oneshot [heartbeat]
backlight gpio cpu0 default-on transient
```

The first command changes into the directory for LED `usr0`, which is the LED closest to the edge of the board. The `[heartbeat]` indicates that the default trigger (behavior) for the LED is to blink in the heartbeat pattern. Look at your LED. Is it blinking in a heartbeat pattern?

Then try the following:

```
bone$ echo none > trigger
bone$ cat trigger
[none] nand-disk mmc0 mmc1 timer oneshot heartbeat
backlight gpio cpu0 default-on transient
```

This instructs the LED to use `none` for a trigger. Look again. It should be no longer blinking.

Now, try turning it on and off:



```
bone$ echo 1 > brightness
bone$ echo 0 > brightness
```

The LED should be turning on and off with the commands.

## 11.10 Controlling GPIOs by Using SYSFS Entries

### 11.10.1 Problem

You want to control a GPIO pin from the command line.

### 11.10.2 Solution

*Controlling LEDs by Using SYSFS Entries* introduces the `sysfs`. This recipe shows how to read and write a GPIO pin.

## 11.11 Reading a GPIO Pin via sysfs

Suppose that you want to read the state of the `P9_42` GPIO pin. (*Reading the Status of a Pushbutton or Magnetic Switch (Passive On/Off Sensor)* shows how to wire a switch to `P9_42`.) First, you need to map the `P9` header location to GPIO number using *Mapping P9\_42 header position to GPIO 7*, which shows that `P9_42` maps to GPIO 7.

P9				P8			
DGND	1	2	DGND	DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3	GPIO_38	3	4	GPIO_39
VDD_5V	5	6	VDD_5V	GPIO_34	5	6	GPIO_35
SYS_5V	7	8	SYS_5V	GPIO_66	7	8	GPIO_67
PWR_BUT	9	10	SYS_RESETN	GPIO_69	9	10	GPIO_68
GPIO_30	11	12	GPIO_60	GPIO_45	11	12	GPIO_44
GPIO_31	13	14	GPIO_50	GPIO_23	13	14	GPIO_26
GPIO_48	15	16	GPIO_51	GPIO_47	15	16	GPIO_46
GPIO_5	17	18	GPIO_4	GPIO_27	17	18	GPIO_65
I2C2_SCL	19	20	I2C2_SDA	GPIO_22	19	20	GPIO_63
GPIO_3	21	22	GPIO_2	GPIO_62	21	22	GPIO_37
GPIO_49	23	24	GPIO_15	GPIO_36	23	24	GPIO_33
GPIO_117	25	26	GPIO_14	GPIO_32	25	26	GPIO_61
GPIO_115	27	28	GPIO_113	GPIO_86	27	28	GPIO_88
GPIO_111	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
GPIO_110	31	32	VDD_ADC	GPIO_10	31	32	GPIO_11
AIN4	33	34	GNDA_ADC	GPIO_9	33	34	GPIO_81
AIN6	35	36	AIN5	GPIO_8	35	36	GPIO_80
AIN2	37	38	AIN3	GPIO_78	37	38	GPIO_79
AIN0	39	40	AIN1	GPIO_76	39	40	GPIO_77
GPIO_20	41	42	GPIO_7	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	GPIO_70	45	46	GPIO_71

Fig. 11.8: Mapping P9\_42 header position to GPIO 7

Next, change to the GPIO `sysfs` directory:

```
bone$ cd /sys/class/gpio/
bone$ ls
export gpiochip0 gpiochip32 gpiochip64 gpiochip96 unexport
```

The `ls` command shows all the GPIO pins that have been exported. In this case, none have, so you see only the four GPIO controllers. Export using the `export` command:

```
bone$ echo 7 > export
bone$ ls
export gpio7 gpiochip0 gpiochip32 gpiochip64 gpiochip96 unexport
```

Now you can see the `gpio7` directory. Change into the `gpio7` directory and look around:

```
bone$ cd gpio7
bone$ ls
active_low direction edge power subsystem uevent value
bone$ cat direction
in
bone$ cat value
0
```

Notice that the pin is already configured to be an input pin. (If it wasn't already configured that way, use `echo in > direction` to configure it.) You can also see that its current value is `0`—that is, it isn't pressed. Try pressing and holding it and running again:

```
bone$ cat value
1
```

The `1` informs you that the switch is pressed. When you are done with GPIO 7, you can always `unexport` it:

```
bone$ cd ..
bone$ echo 7 > unexport
bone$ ls
export gpiochip0 gpiochip32 gpiochip64 gpiochip96 unexport
```

## 11.12 Writing a GPIO Pin via sysfs

Now, suppose that you want to control an external LED. [Toggling an External LED](#) shows how to wire an LED to `P9_14`. [Mapping P9\\_42 header position to GPIO 7](#) shows `P9_14` is GPIO 50. Following the approach in [Controlling GPIOs by Using SYSFS Entries](#), enable GPIO 50 and make it an output:

```
bone$ cd /sys/class/gpio/
bone$ echo 50 > export
bone$ ls
gpio50 gpiochip0 gpiochip32 gpiochip64 gpiochip96
bone$ cd gpio50
bone$ ls
active_low direction edge power subsystem uevent value
bone$ cat direction
in
```

By default, `P9_14` is set as an input. Switch it to an output and turn it on:

```
bone$ echo out > direction
bone$ echo 1 > value
bone$ echo 0 > value
```

The LED turns on when a `1` is written to `value` and turns off when a `0` is written.

## 11.13 The Play's Boot Sequence

The BeagleBoard Play is based on the Texas Instrument's AM625 Sitara processor which supports many boot modes.

---

**Note:** bootlin (<https://bootlin.com/>) has many great Linux training materials for free on their site. Their embedded Linux workshop (<https://bootlin.com/training/embedded-linux/>) gives a detailed presentation of the Play's boot sequence (<https://bootlin.com/doc/training/embedded-linux-beagleplay/embedded-linux-beagleplay-labs.pdf>, starting at page 9). Check it out for details on building the boot sequence from scratch.

---

Here we'll take a high-level look at booting from both the user's view and the developer's view.

### 11.13.1 Booting for the User

The most common way for the Play to boot is the power up the board, if the micro SD card is present, it will boot from it, if it isn't present it will boot from the built in eMMC.

You can override the boot sequence by using the **USR** button (located near the micro SD cage). If the **USR** button is pressed the Play will boot from the micro SD card.

---

**Note:** If the eMMC fails to boot, it will attempt to boot from the UART. If the SD card fails to boot, it will try booting via the USB.

---

### 11.13.2 Booting for the Developer

---

**Tip:** These diagrams might help: <https://github.com/u-boot/u-boot/blob/6e8fa0611f19824e200fe4725f18bce7e2000071/doc/board/ti/k3.rst>

---

If you are developing firmware for the Play you may need to have access to the processor early in the booting sequence. Much can happen before the Linux kernel starts its boot process. Here are some notes on what the BeagleBoard Play does when it boots up. Many of the booting details come from Chapter 5 (Initialization) of the AM62x Technical Reference Manual (TRM) (<https://www.ti.com/product/AM625>, <https://www.ti.com/lit/pdf/spruiv7>). The following figure, taken from page 2456, shows the Initialization Process.



**Figure 5-1. Initialization Process**

Fig. 11.9: Initialization Process

We are interested in what happens in the **ROM code**. Page 2457, of the TRM, shows the different ROM Code Boot Modes.

These are selected at boot time based on the state of the BOOTMODE pins. The table on page 2465 shows the BOOTMODE pins.

Page 14 of of the Play's schematic ([https://git.beagleboard.org/beagleplay/beagleplay/-/blob/main/BeaglePlay\\_sch.pdf](https://git.beagleboard.org/beagleplay/beagleplay/-/blob/main/BeaglePlay_sch.pdf)) shows how the BOOTMODE pins are set during boot.

Table 5-1. ROM Code Boot Modes

Boot Mode	Boot Media/Host	SoC Peripheral	Can be a Backup Mode? <sup>(1)</sup>	Notes
No-boot/Dev-boot	No media or host	None	N	No boot or development boot – debug modes
OSPI	OSPI flash	FSS0_OSPI0	N	On OSPI port
QSPI	QSPI flash	FSS0_OSPI0	N	On OSPI port
SPI	SPI flash	FSS0_OSPI0	Y	On OSPI port
Ethernet	External host	CPSW0	Y	In BOOTP mode. RGMII or RMII PHY
I2C	I <sup>2</sup> C EEPROM	I2C0	Y	I2C target boot is not supported
UART	External host	UART0	Y	XMODEM protocol
MMCSD	eMMC flash or SD card	MMCSD0 (8bit) or MMCSD1 (4bit)	Y	Boot from User Data Area (UDA) in raw or file system mode
eMMC	eMMC flash	MMCSD0 (8bit)	N	Boot from boot partition
USB - target	USB boot from external host	USB0	Y	USB device mode boot using DFU (device firmware upgrade), Boot is running on USB2.0 speeds.
USB - host	USB mass storage	USB0	Y	USB2.0 host mode, boot from FAT32 filesystem
Serial Flash	Serial Flash	FSS0_OSPI0	N	On OSPI port
xSPI	xSPI flash	FSS0_OSPI0	N	On OSPI port
GPMC	NOR flash, NAND flash	GPMC0	N	CSn0 connected, 8 bit NAND flash only, 16-bit non-mux NOR flash only

(1) The peripheral can be selected also as a backup boot mode. A backup mode is tried if primary boot mode fails.

Fig. 11.10: ROM Code Boot Modes

Table 5-2. BOOTMODE Pin Mapping

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	Reserved	Backup Boot Mode Config	Backup Boot Mode			Primary Boot Mode Config			Primary Boot Mode			PLL Config			

Table 5-3 describes the BOOTMODE pins that need to be set according to the system clock provided to the device.

Fig. 11.11: BOOTMODE Pin Mapping

## Bootstrap

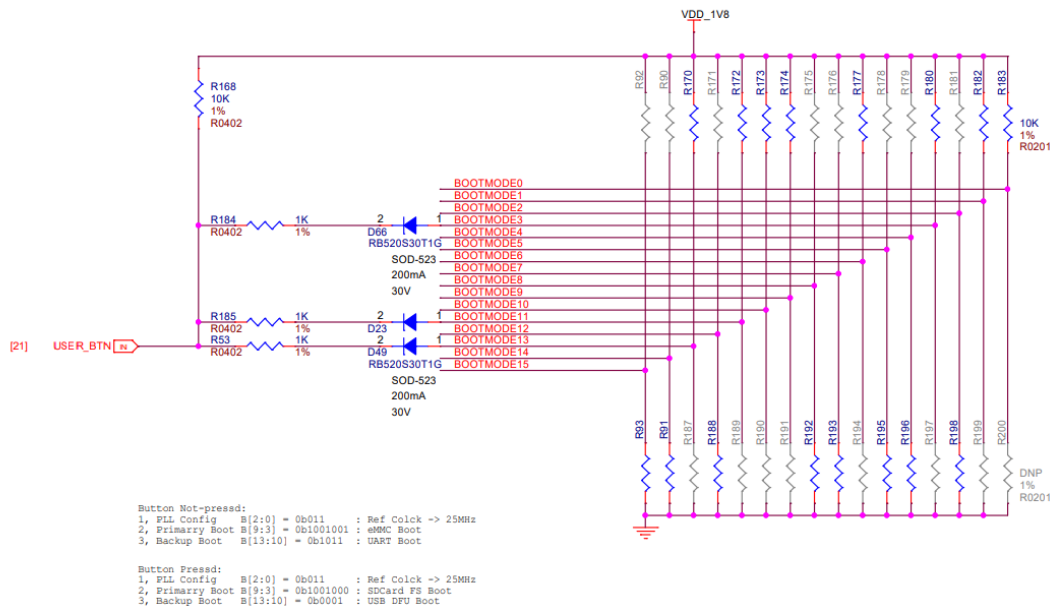


Fig. 11.12: Bootstrap

Therefore the following modes are selected if **Button Not-pressed**

1,	PLL Config	B[2:0] = 0b011	: Ref Clcok -> 25MHz
2,	Primary Boot	B[9:3] = 0b1001001	: eMMC Boot
3,	Backup Boot	B[13:10] = 0b1011	: UART Boot

That is, you boot off the eMMC and if that fails you boot off the UART.

If **Button is Pressed**

1,	PLL Config	B[2:0] = 0b011	: Ref Clcok -> 25MHz
2,	Primary Boot	B[9:3] = 0b1001000	: SD Card FS Boot
3,	Backup Boot	B[13:10] = 0b0001	: USB DFU Boot

Here you are booting off the SD card (in filesystem mode), or the USB if that fails.

### 11.13.3 Boot Flow

There are many steps that occur after the BOOTMODE is selected and before the Linux Kernel boots. **Boot Flow** shows those steps for the **R5** processor and the arm (**A53**) processor. The key parts are **tiboot3.bin** and **tispl.bin** running on the R5 and **u-boot.img** running on the A53. These binary files are found on the Play in `/boot/firmware`.

**Note:** The files on the SD card and the eMMC are in ext4 format. The files used for booting must be in vfat format. Therefore `/boot/firmware` is mounted in vfat as seen in `/etc/fstab`.

```
# /etc/fstab: static file system information.
#
/dev/mmcblk0p2 / ext4 noatime,errors=remount-ro 0 1
/dev/mmcblk0p1 /boot/firmware vfat defaults 0 0
debugfs /sys/kernel/debug debugfs mode=755,uid=root,gid=gpio,defaults 0 0
→0
```

### 11.13.4 Source Code

The source code and examples of how to compile the source is found in: <https://git.beagleboard.org/beagleboard/repos-arm64/-/blob/main/bb-u-boot-beagleplay/suite/bookworm/debian/rules#L29>

## 11.14 Home Assistant

1. **Get an image here:**

<https://www.beagleboard.org/distros/beagleplay-home-assistant-webinar-demo-image> I chose the boot from SD image.

2. Boot the Play from the SD card

3. Log into the Play

4. **Find the Play's IP address by running**

```
bone$ ip a show eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state_
↪UP group default qlen 1000
    link/ether 34:08:e1:85:1b:a6 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.24/24 brd 10.0.5.255 scope global dynamic_
↪noprofixroute eth0
```

The address is after `inet`, in my case it's 10.0.5.24.

5. Wait 5 or 10 minutes and then open a browser at 10.0.5.24:8123 using your IP address.

---

**Tip:** If you get a "This site can't be reached" error message, try running `journalctl -f` to see the log messages.

---

1. **Open another browser and follow the instructions at:**

<https://www.home-assistant.io/getting-started/onboarding>

### 11.14.1 mqtt

Here are Jason's addons. <https://git.beagleboard.org/jkridner/home-assistant-addons>