BeagleBone Cookbook

Table of Contents

Basics

Introduction

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).

Picking Your Beagle

Problem

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

Solution

Current list of boards: https://git.beagleboard.org/explore/projects/topics/boards

Getting Started, Out of the Box

Problem

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

Solution

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.

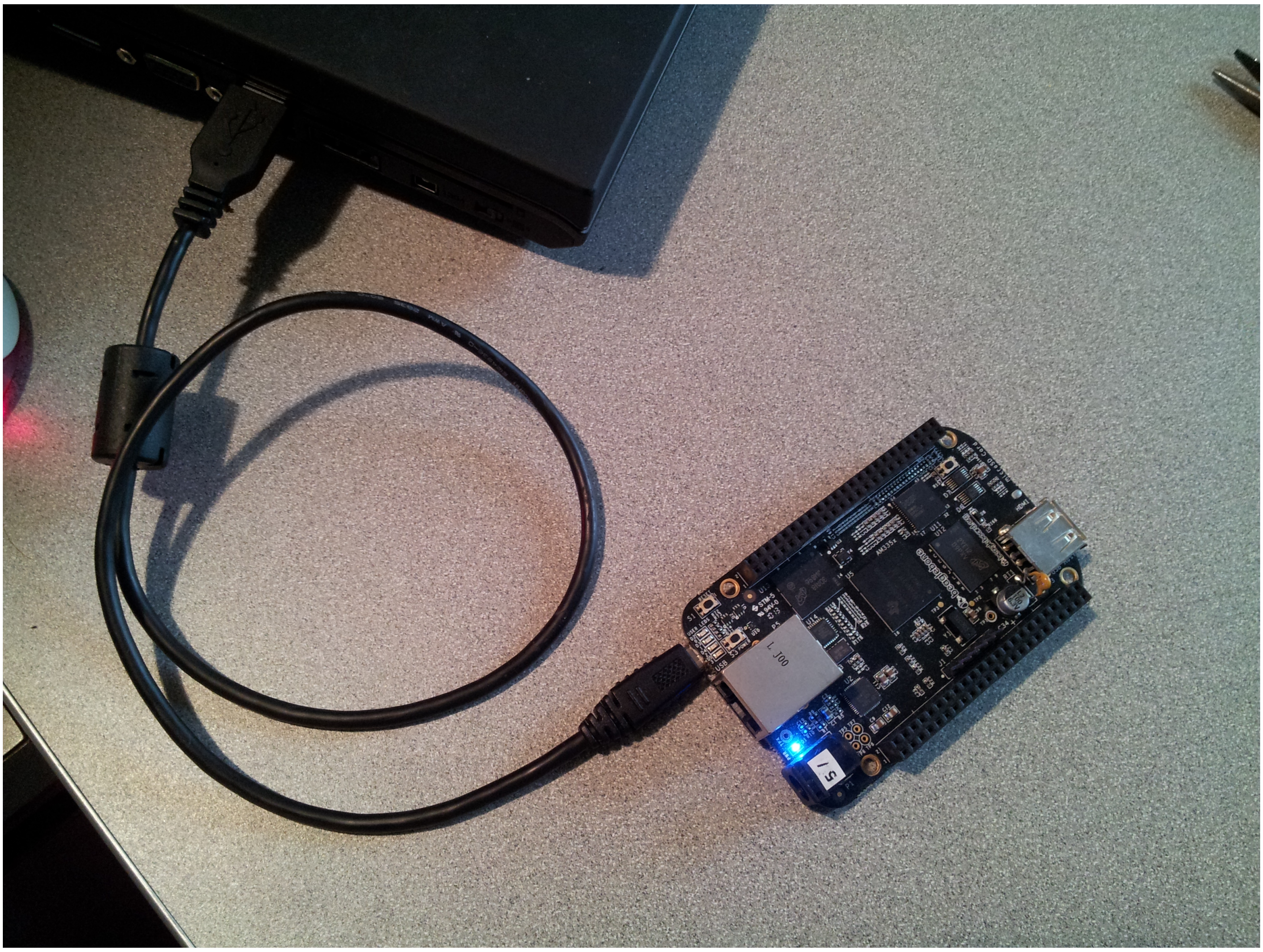


Figure 1. Plugging BeagleBone Black into a USB port

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.

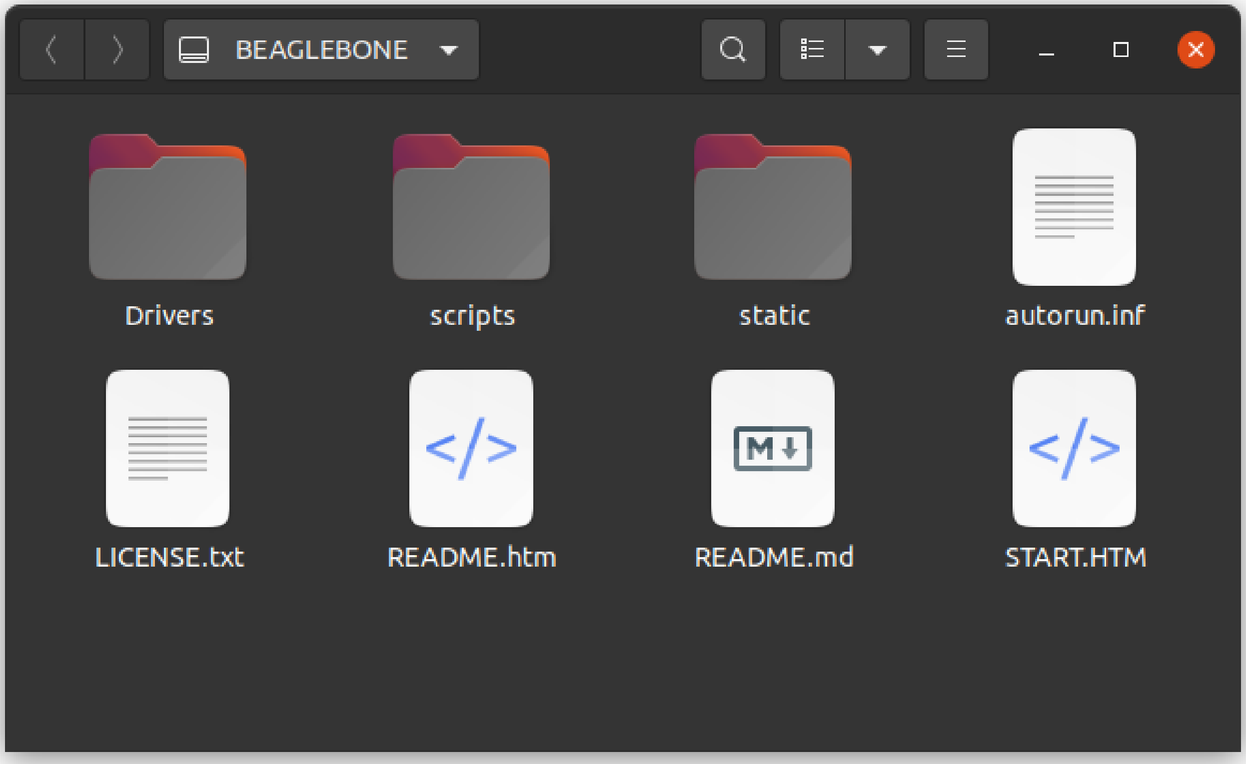


Figure 2. The Bone appears as a USB drive

Browse to http://192.168.7.2:3000 from your host computer (Visual Studio Code).

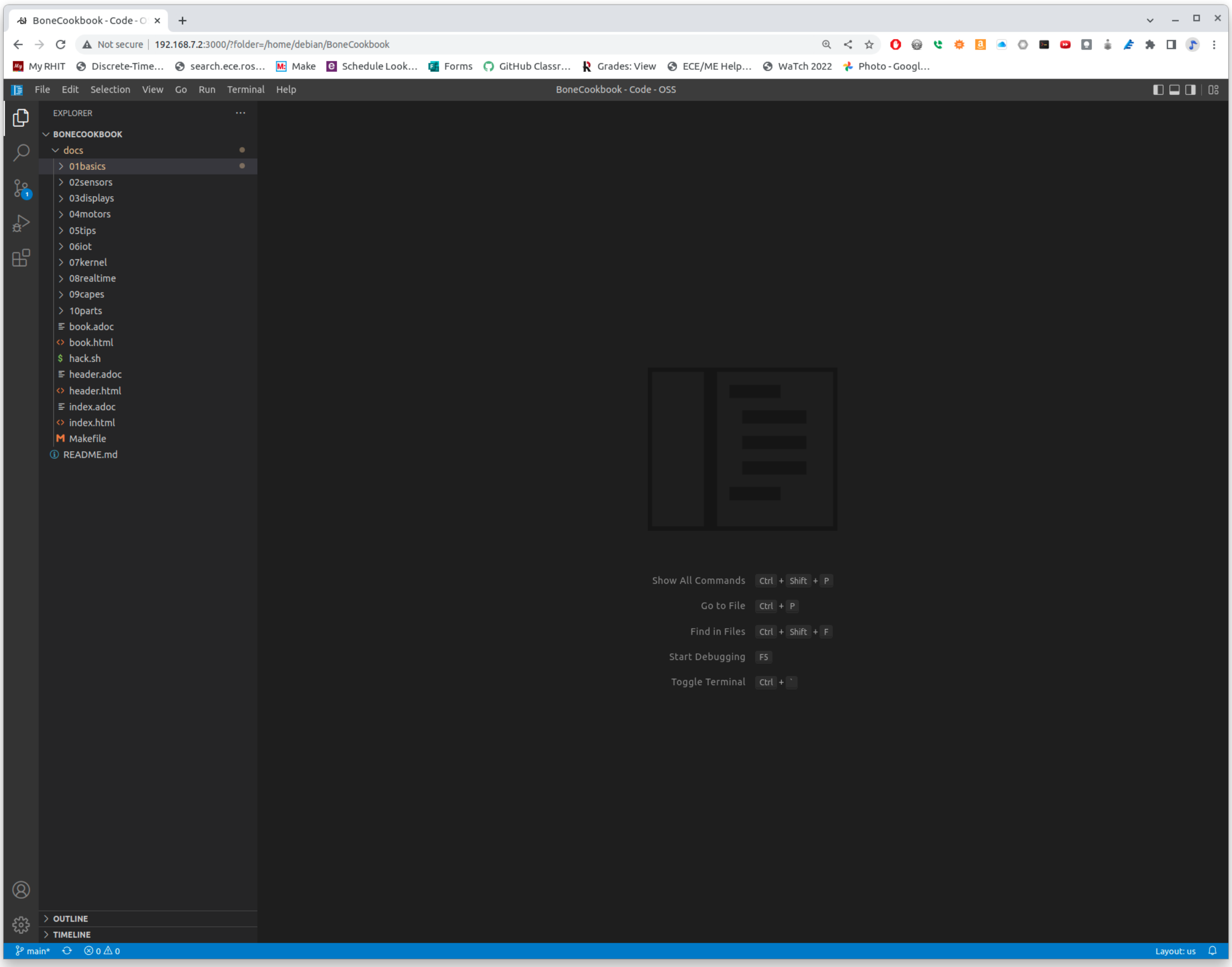


Figure 3. Visual Studio Code

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo halt</strong>
The system is going down for system halt NOW! (pts/0)
</pre>
```

This will ensure that the Bone shuts down correctly. If you just pull the power, it's possible that open files won't close properly and might become corrupt.

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.

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

Problem

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

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 2022-07-01) 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:

```
<pre data-type="programlisting">
bone$ <strong>cat /ID.txt</strong>
BeagleBoard.org Debian Bullseye IoT Image 2022-07-01
</pre>
```

I'm running the 2022-07-01 version.

Running the Python and JavaScript Examples

Problem

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

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.

Cloning the Cookbook Repository

Problem

You want to run the Cookbook examples.

Solution

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

```
<pre data-type="programlisting">
bone$ <strong>git clone git@github.com:MarkAYoder/BoneCookbook.git</strong>
bone$ <strong>cd BoneCookbook/docs</strong>
bone$ <strong>ls</strong>
</pre>
```

You can look around from the command line, or explore from Visual Sudio Code. If you ar using VSC, go to the *File* menu and select *Open Folder …* and selet BoneCookbook/docs. Then explore. You'll find there is a directory for each chapter and most chapters have a *code* directory for the sample scripts and a *figures* directory for the figures.

Wiring a Breadboard

Problem

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

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

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.

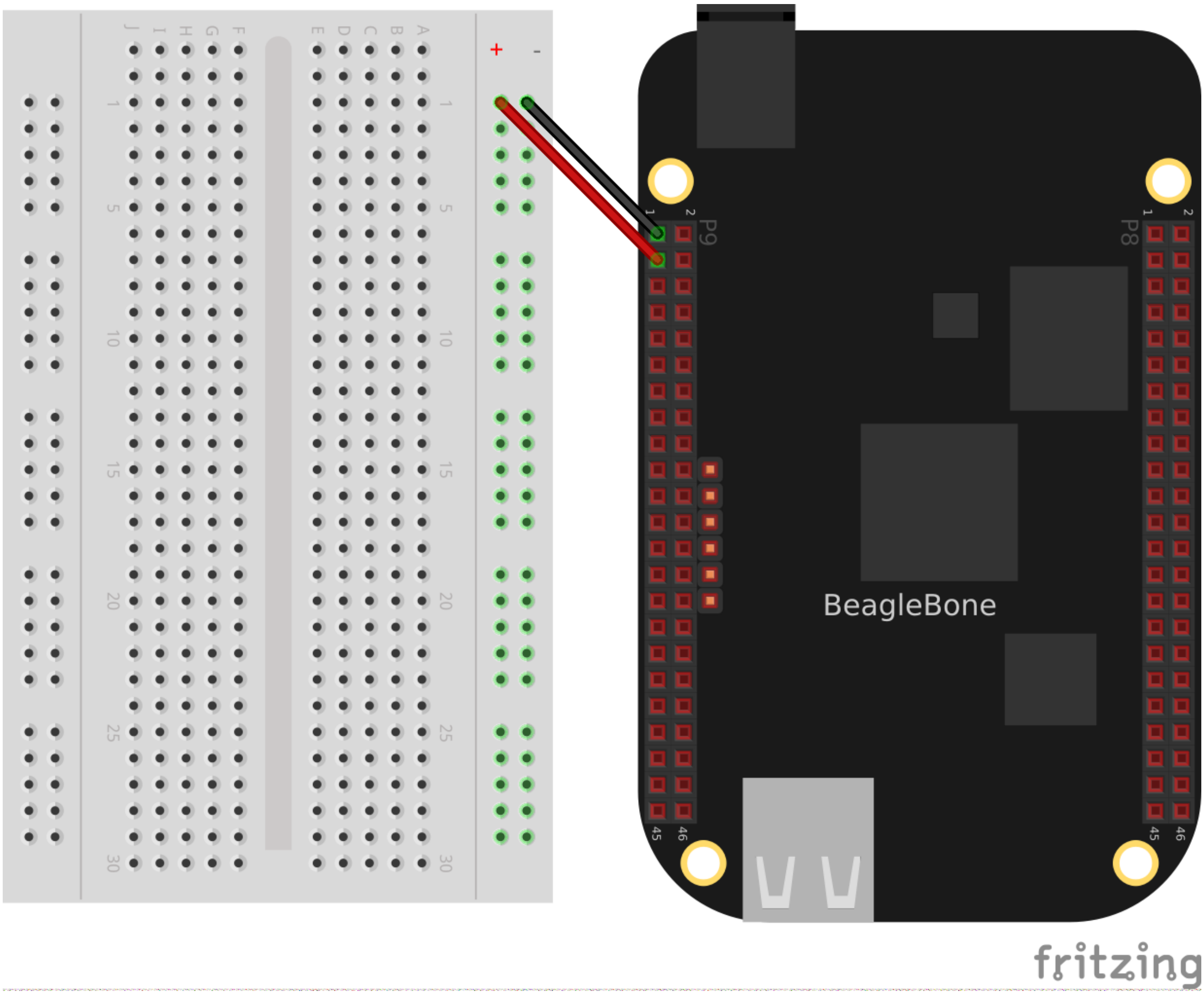


Figure 4. Breadboard wired to BeagleBone Black

Editing Code Using Visual Studio Code

Problem

You want to edit and debug files on the Bone.

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.

Running Python and JavaScript Applications from Visual Studio Code

Problem

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

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.

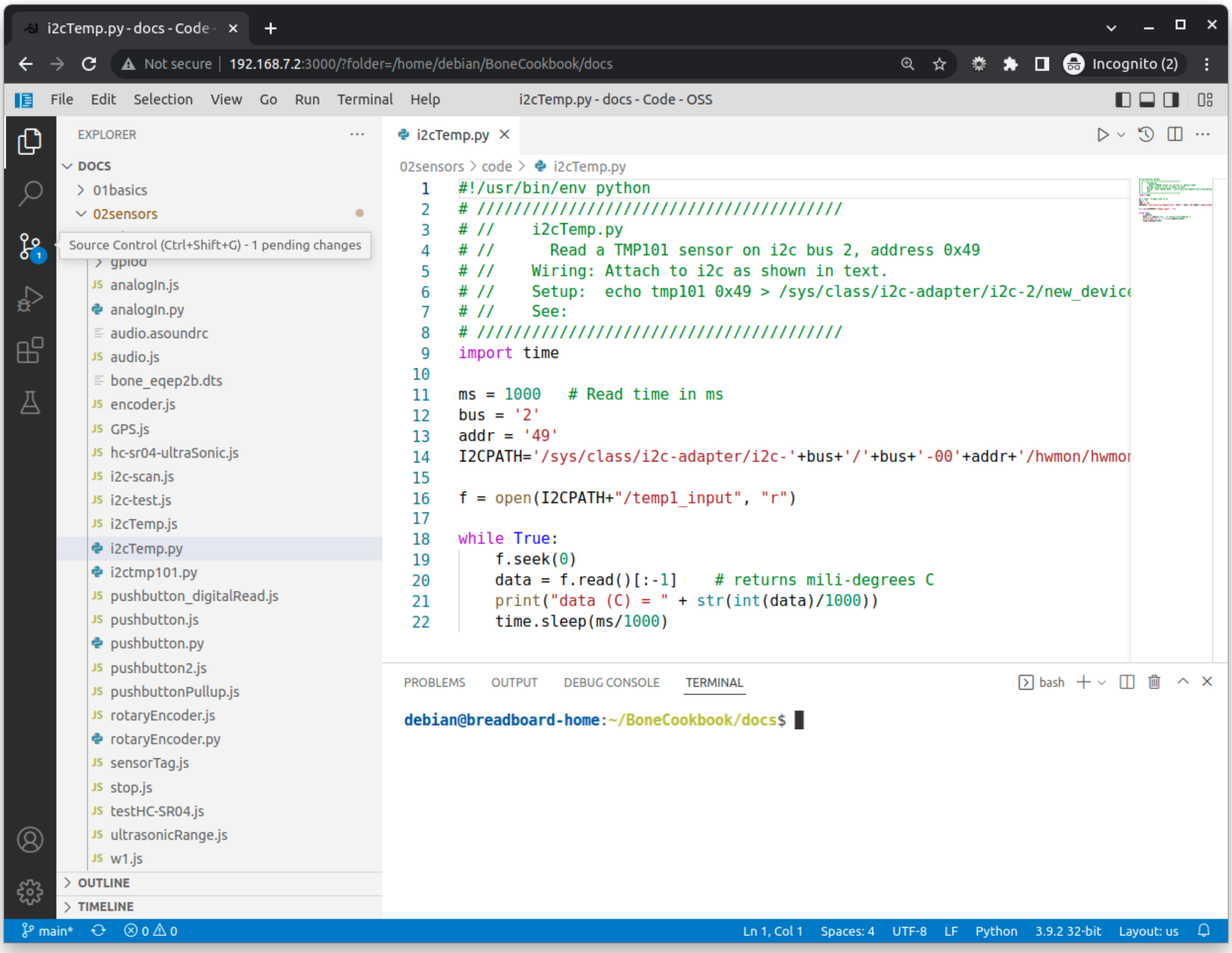


Figure 5. Visual Studio Code showing bash terminal

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

```
<pre data-type="programlisting">
bone$ <strong>cd ~/examples/BeagleBone/Black/</strong>
bone$ <strong>./seqLEDs.py</strong>
</pre>
```

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.

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.

Solution

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).

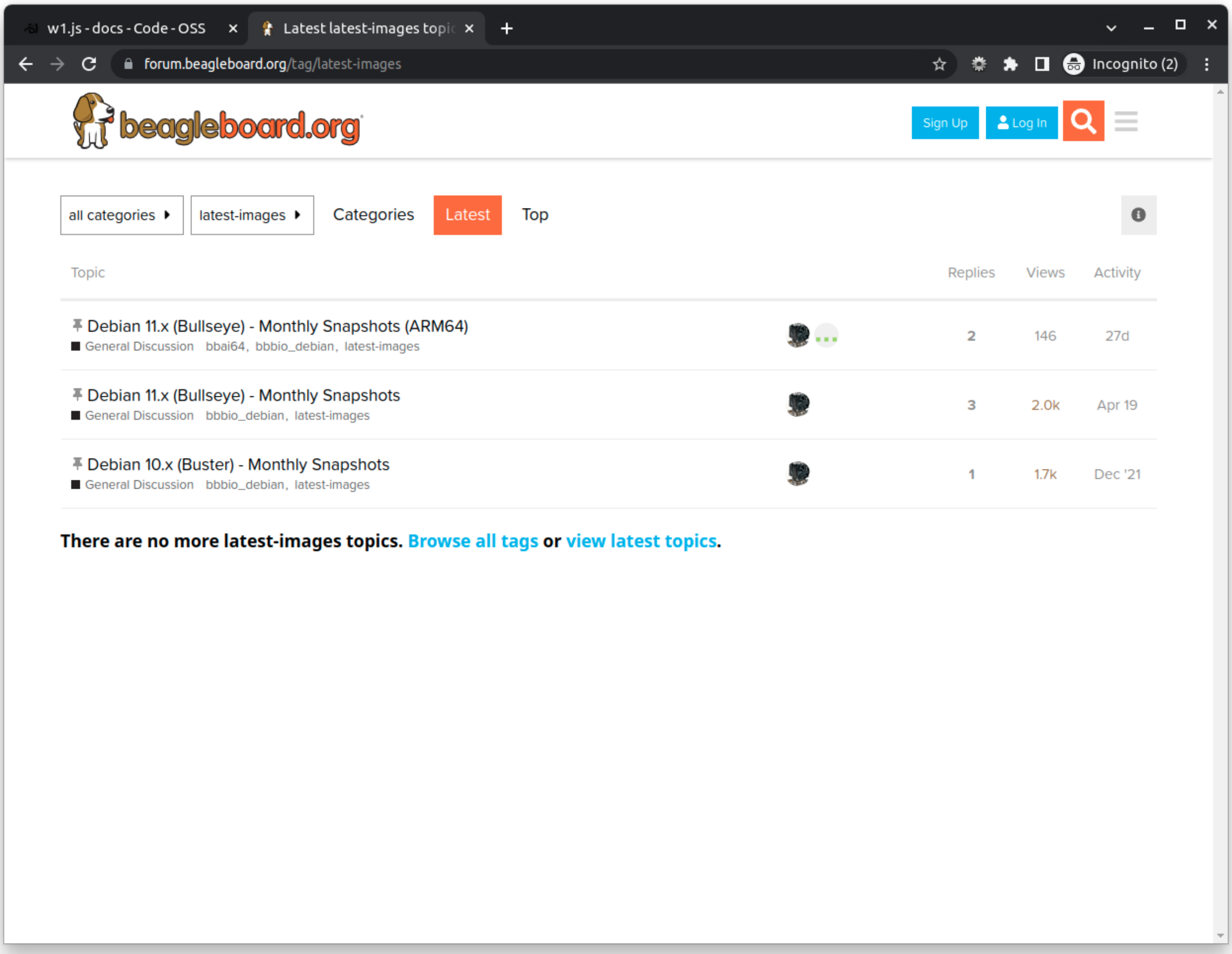


Figure 6. Latest Debian images

At the time of writing, we are using the *Bullseye* image. Click on it's 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.

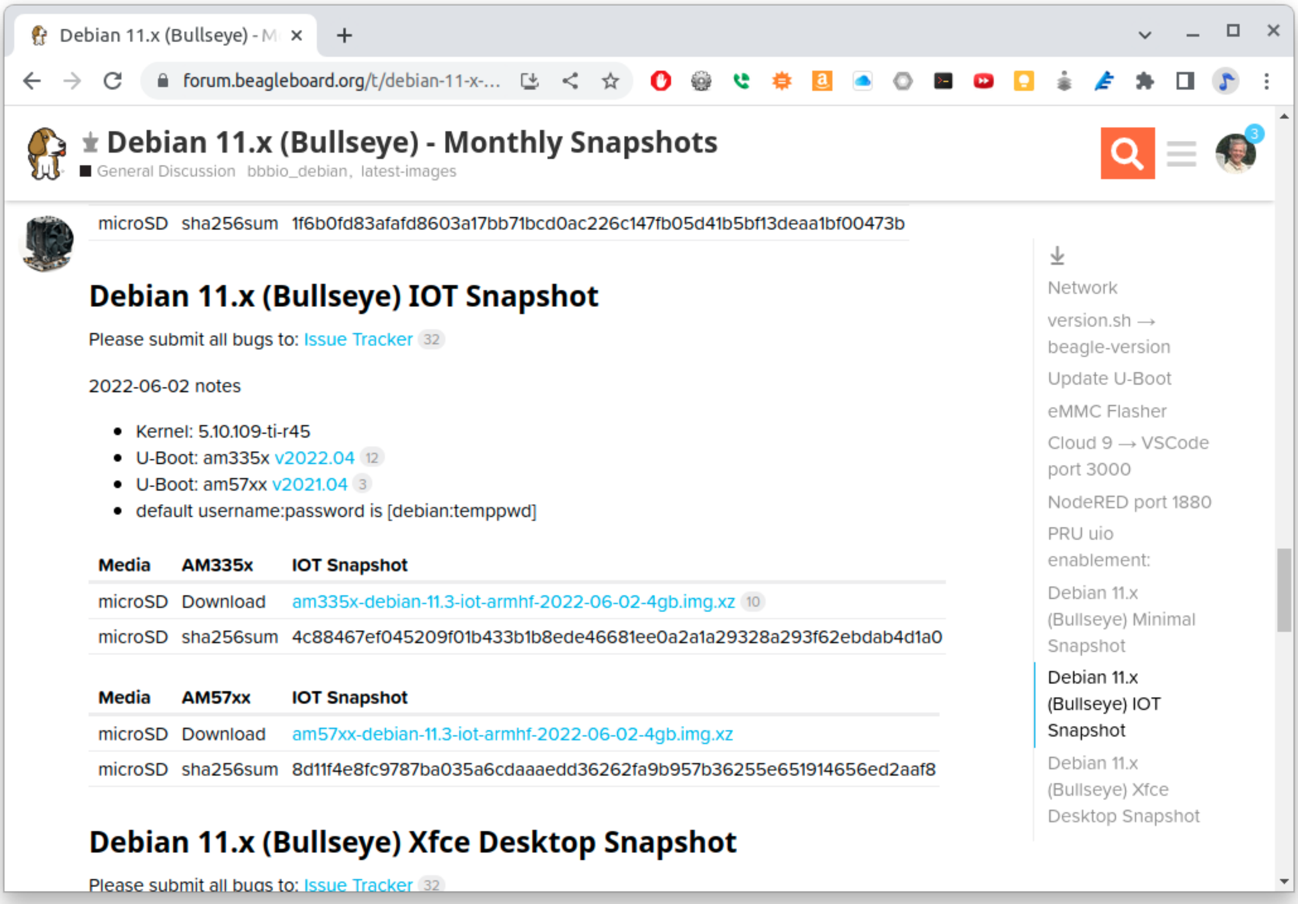


Figure 7. Latest Debian images

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 miscroSD 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.

NOTE

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

Running the Latest Version of the OS on Your Bone

Problem

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

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.

Updating the OS on Your Bone

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.

Solution

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

```
<pre data-type="programlisting">
bone$ <strong>sudo apt update</strong>
bone$ <strong>sudo apt upgrade</strong>
</pre>
```

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.

Discussion

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

Backing Up the Onboard Flash

Problem

You've modified the state of your Bone in a way that you'd like to preserve or share.

Solution

The eLinux 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:

1. Get a 4 GB or larger microSD card that is FAT formatted.
2. If you create a FAT-formatted microSD card, you must edit the partition and ensure that it is a bootable partition.
3. Download beagleboneblack-save-emmc.zip and uncompress and copy the contents onto your microSD card.
4. Eject the microSD card from your computer, insert it into the powered-off BeagleBone Black, and apply power to your board.
5. 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.
6. 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.
7. 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.

Updating the Onboard Flash

Problem

You want to copy the microSD card to the onboard flash.

Solution

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

1. Repeat the steps in Running the Latest Version of the OS on Your Bone to update the OS.
2. 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.
3. Boot from the microSD card.
4. Log on to the bone and edit /boot/uEnv.txt.
5. Uncomment out the last line cmdline=init=/usr/sbin/init-beagle-flasher.
6. Save the file and reboot.
7. The USR LEDs will flash back and forth for a few minutes.
8. When they stop flashing, remove the SD card and reboot.

9. 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.

Sensors

Introduction

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 www.github.com. Go here Cloning the Cookbook Repository for instructions.

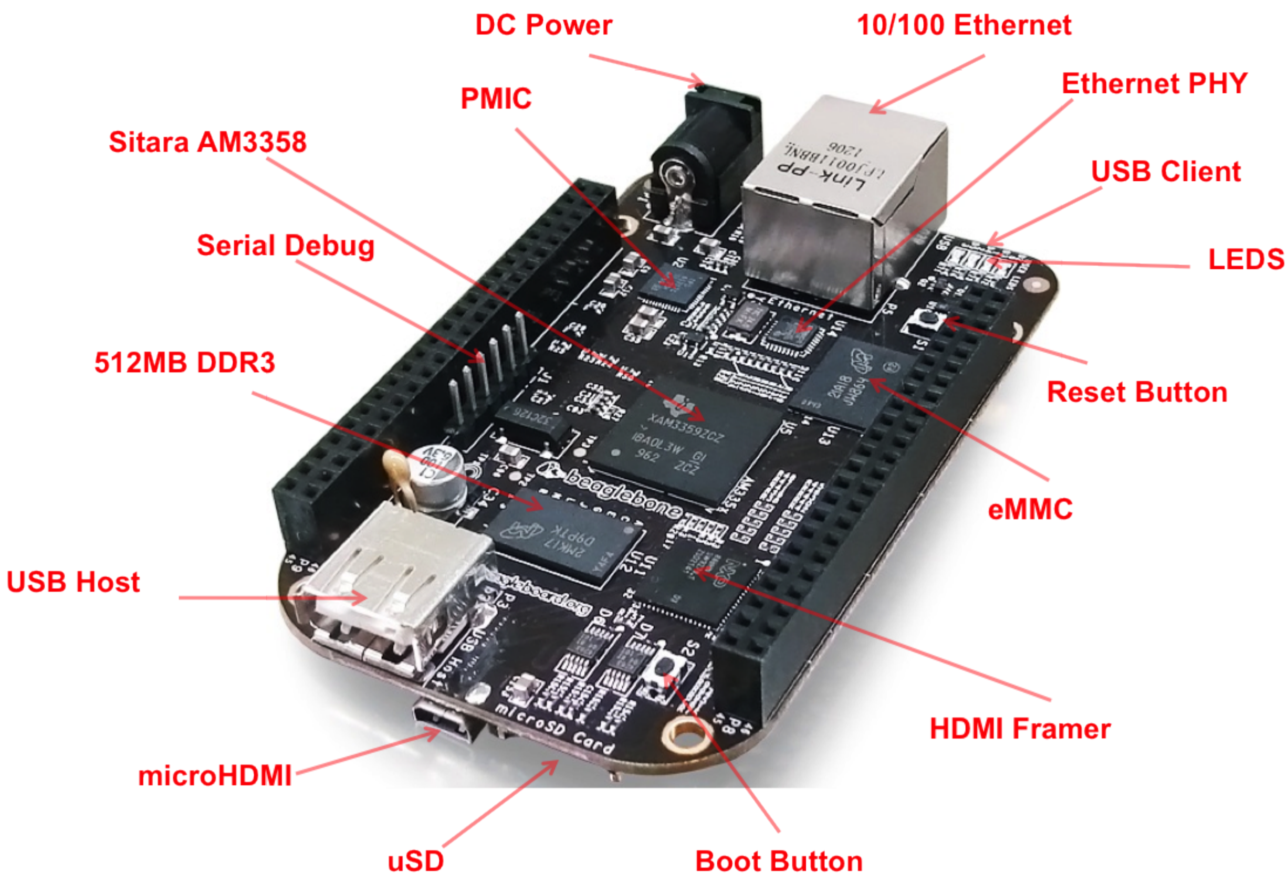


Figure 8. The USB 2.0 host port

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

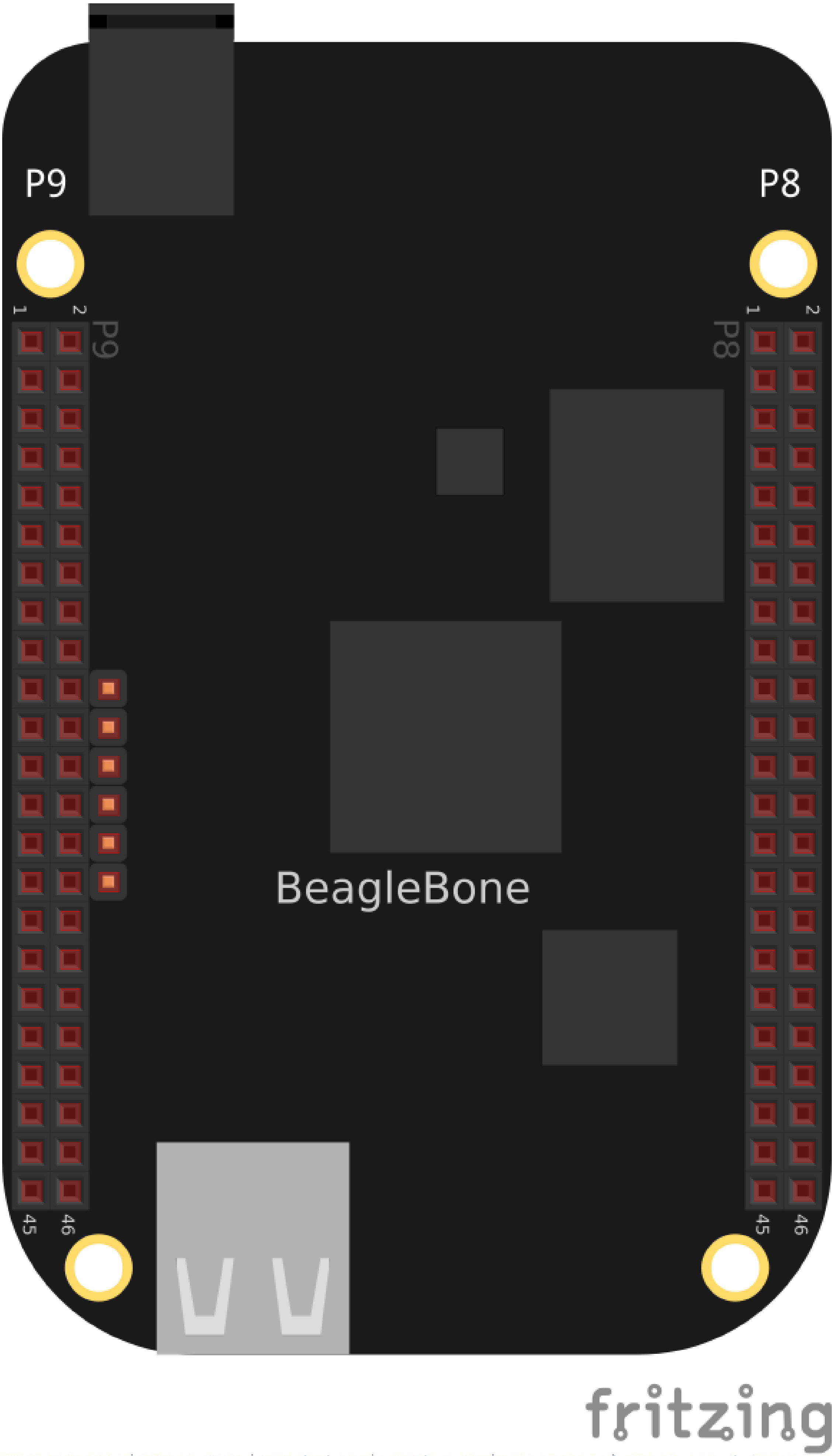


Figure 9. The P8 and P9 cape headers

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^2C 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).

Choosing a Method to Connect Your Sensor

Problem

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

Solution

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

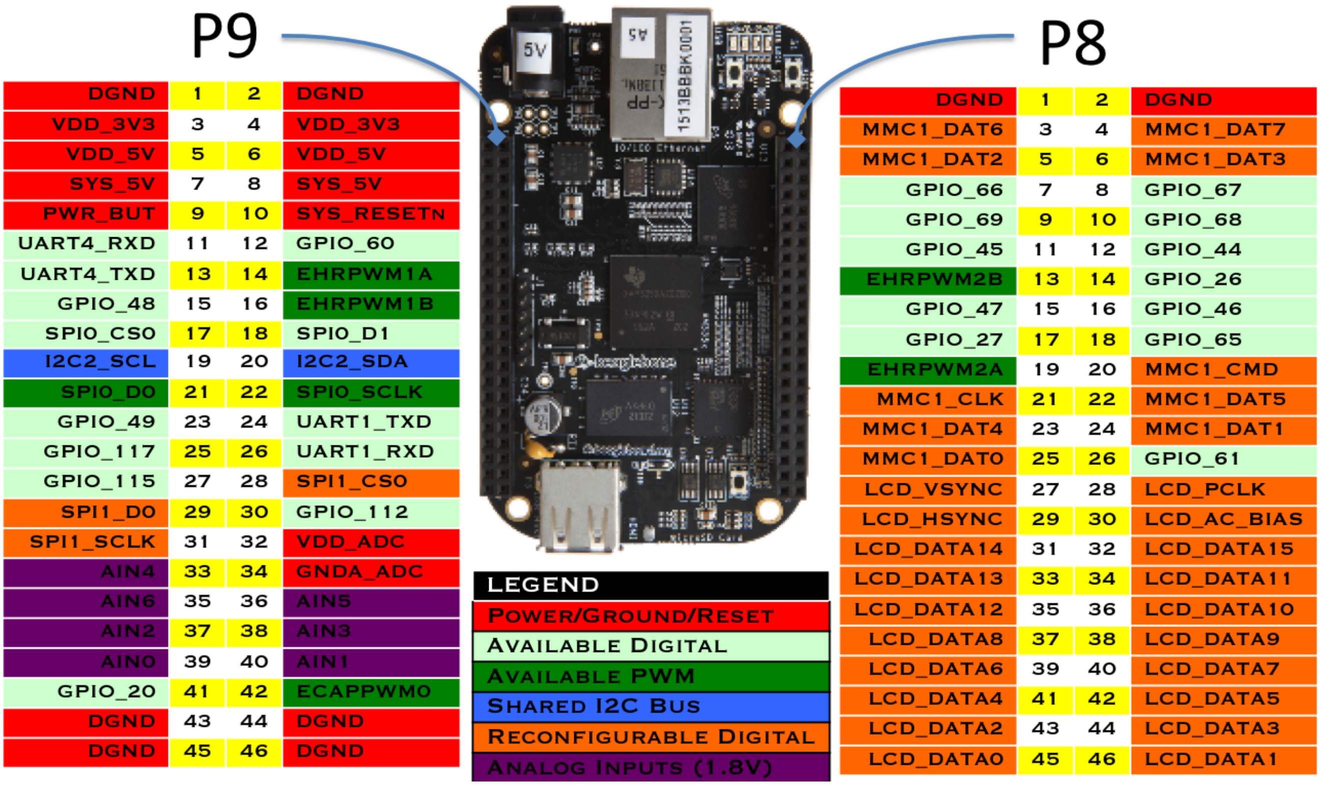


Figure 10. Some of the many sensor connection options on the Bone

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.

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

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.

Solution

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

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

```
<pre data-type="programlisting">
bone$ <strong>cd</strong>
bone$ <strong>cd BoneCookbook/docs/02sensors/code</strong>
</pre>
```

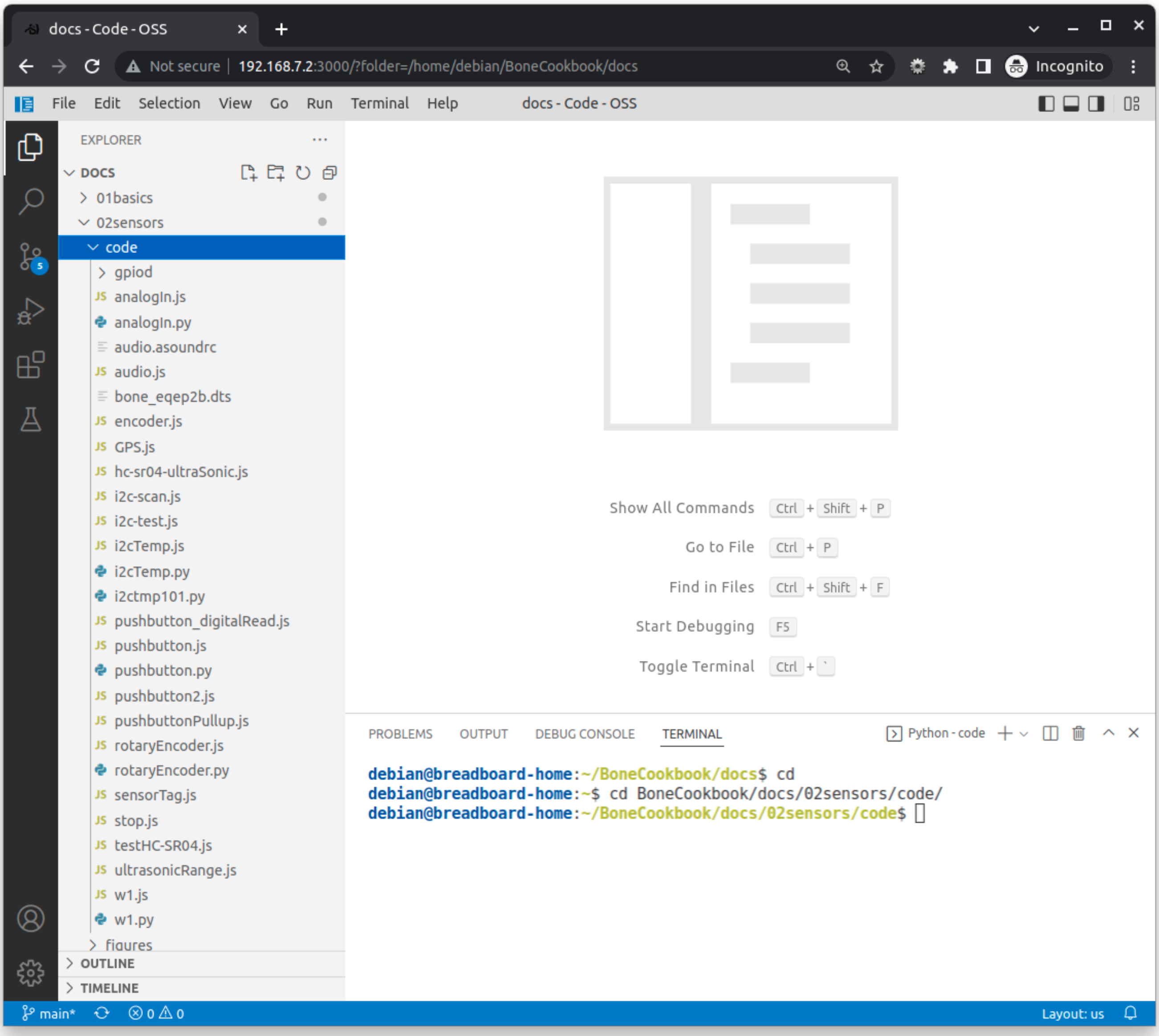


Figure 11. Entering commands in the VSC bash tab

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 www.github.com. Go here Cloning the Cookbook Repository for instructions.

1. Double-click the *pushbutton.py* file to open it.
2. 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.
3. In the bash tab, enter the following commands:

```
<pre data-type="programlisting">
root@beaglebone:~/boneSensors# <strong>./pushbutton.js</strong>
data= 0
data= 0
data= 1
data= 1
^C
</pre>
```

This process will work for any script in this book.

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

Problem

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

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.

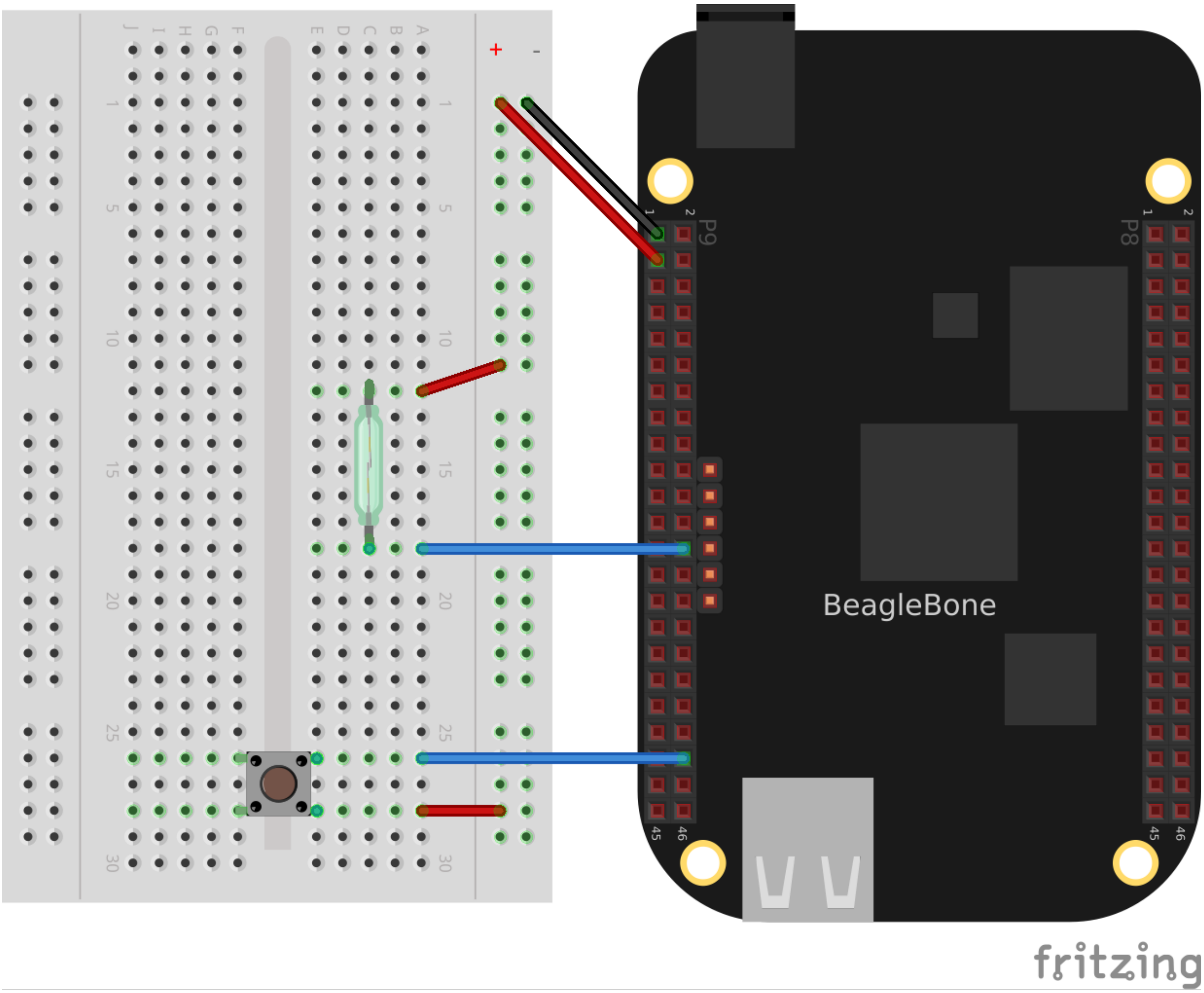


Figure 12. Diagram for wiring a pushbutton and magnetic reed switch input

The code in Monitoring a pushbutton (pushbutton.js) reads GPIO port P9_42, which is attached to the pushbutton.

Example 1. Monitoring a pushbutton (pushbutton.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    pushbutton.py
# //      Reads P9_42 and prints its value.
# //    Wiring: Connect a switch between P9_42 and 3.3V
# //    Setup:
# //    See:
# ////////////////////////////////////////
import time
import os

ms = 500    # Read time in ms
pin = '7' #  P9_42 is gpio 7
GPIOPATH="/sys/class/gpio"
```

```
# Make sure pin is exported
if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
    f = open(GPIOPATH+"/export", "w")
    f.write(pin)
    f.close()

# Make it an input pin
f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
f.write("in")
f.close()

f = open(GPIOPATH+"/gpio"+pin+"/value", "r")

while True:
    f.seek(0)
    data = f.read()[:-1]
    print("data = " + data)
    time.sleep(ms/1000)
```

Example 2. Monitoring a pushbutton (pushbutton.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  pushbutton.js
//      Reads P9_42 and prints its value.
//  Wiring: Connect a switch between P9_42 and 3.3V
//  Setup:
//  See:
////////////////////////////////////////
const fs = require("fs");

const ms = 500   // Read time in ms
const pin = '7'; // P9_42 is gpio 7
const GPIOPATH="/sys/class/gpio/";

// Make sure pin is exported
if(!fs.existsSync(GPIOPATH+"gpio"+pin)) {
    fs.writeFileSync(GPIOPATH+"export", pin);
}
// Make it an input pin
fs.writeFileSync(GPIOPATH+"gpio"+pin+"/direction", "in");

// Read every ms
setInterval(readPin, ms);

function readPin() {
    var data = fs.readFileSync(GPIOPATH+"gpio"+pin+"/value").slice(0, -1);
    console.log('data = ' + data);
```

```
  }
```

Put this code in a file called *pushbutton.js* 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:

```
<pre data-type="programlisting">
bone$ <strong>./pushbutton.js</strong>
data = 0
data = 0
data = 1
data = 1
^C
</pre>
```

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 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.

Mapping Header Numbers to gpio Numbers

Problem

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

Solution

The gpioinfo command displays information about all the P8 and P9 header pins. To see the info for just one pin, use grep.

```
<pre data-type="programlisting">
bone$ <strong>gpioinfo | grep -e chip -e P9.42</strong>
gpiochip0 - 32 lines:
    line   7: "P8_42A [ecappwm0]" "P9_42" input active-high [used]
gpiochip1 - 32 lines:
gpiochip2 - 32 lines:
gpiochip3 - 32 lines:

</pre>
```

This shows P9_42 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:

```
<pre data-type="programlisting">
```

```
bone$ <strong>gpioinfo | grep -e chip -e P9.26</strong>
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.

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

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.

Solution

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 the P9 header shows them on the lower part of the P9 header.

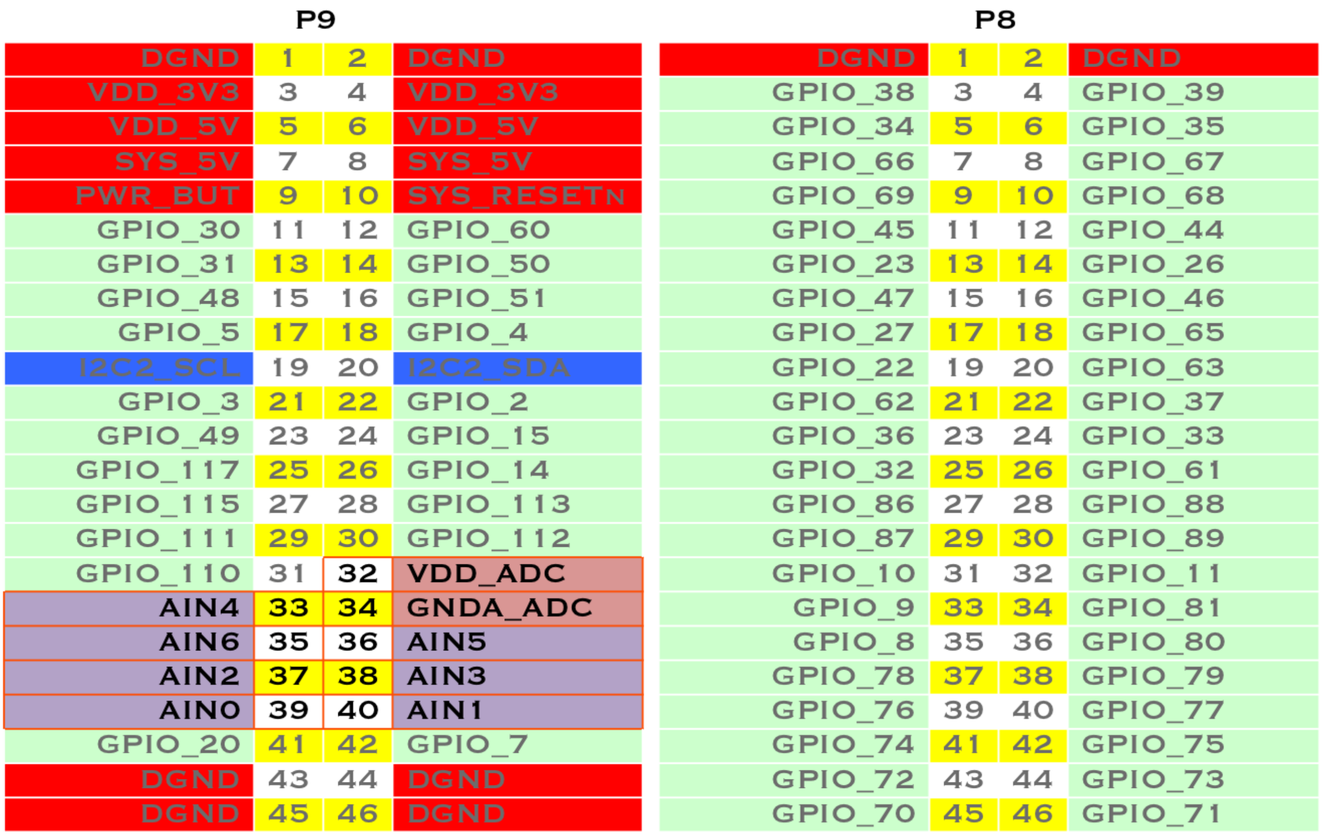


Figure 13. Seven analog inputs on the P9 header

To make this recipe, you will need:

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

A variable resistor with three terminals

Wiring a 10kΩ 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).

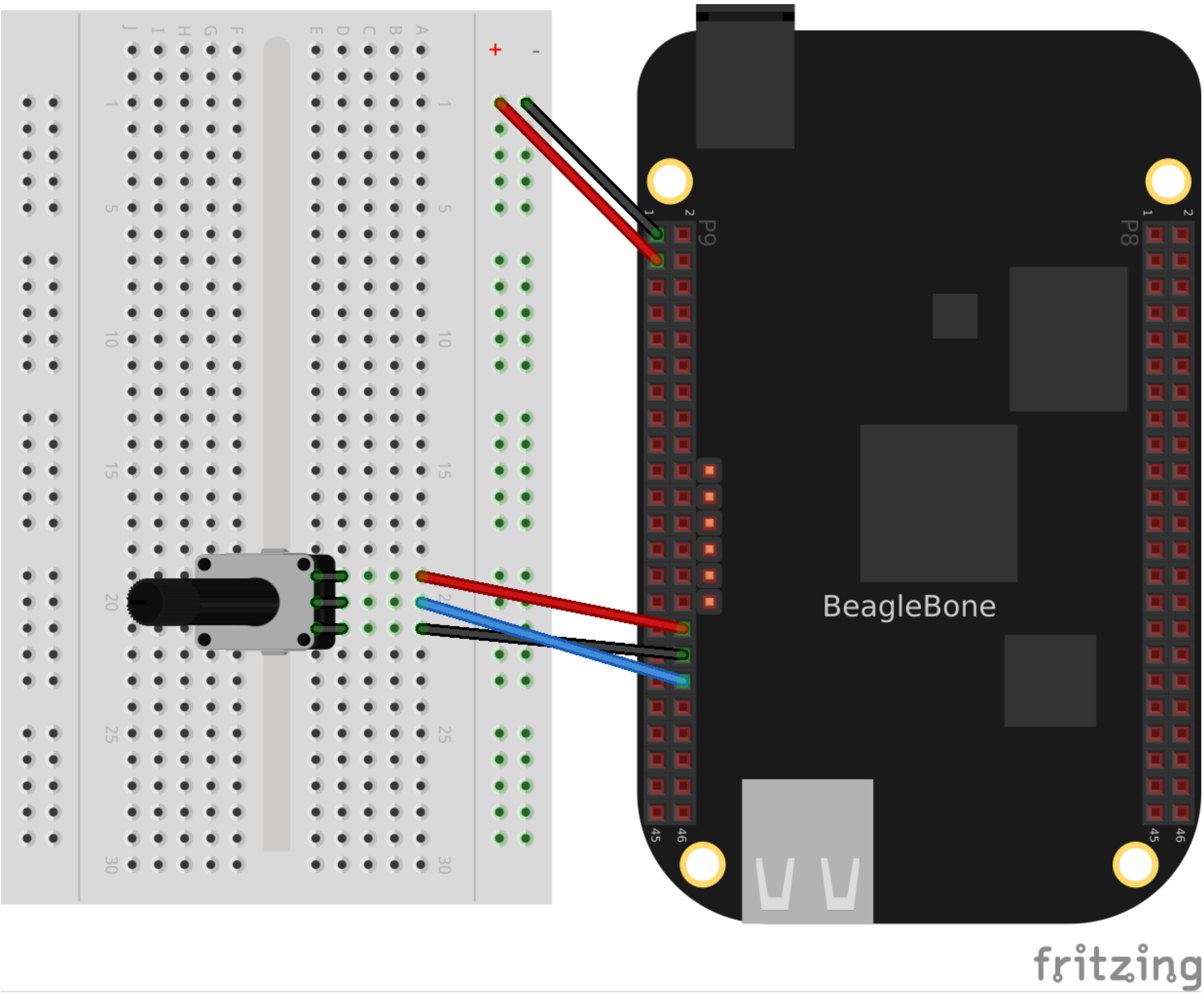


Figure 14. Wiring a 10kΩ variable resistor (trimpot) to an ADC port

Reading an analog voltage (analogIn.py) shows the Python 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.

Example 3. Reading an analog voltage (analogIn.py)

```
#!/usr/bin/env python3
#//////////////////////////////////////
#   analogin.py
#   Reads the analog value of the light sensor.
#//////////////////////////////////////
import time
import os

pin = "2"        # light sensor, A2, P9_37

IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw'

print('Hit ^C to stop')

f = open(IIOPATH, "r")

while True:
    f.seek(0)
    x = float(f.read())/4096
    print('{}: {:.1f}%, {:.3f} V'.format(pin, 100*x, 1.8*x), end = '\r')
    time.sleep(0.1)

# // Bone  | Pocket | AIN
# // ----- | ------ | ---
# // P9_39 | P1_19  | 0
# // P9_40 | P1_21  | 1
# // P9_37 | P1_23  | 2
# // P9_38 | P1_25  | 3
# // P9_33 | P1_27  | 4
# // P9_36 | P2_35  | 5
# // P9_35 | P1_02  | 6
```

Example 4. Reading an analog voltage (analogIn.js)

```
#!/usr/bin/env node
//////////////////////////////////////
//  analogin.js
//  Reads the analog value of the light sensor.
//////////////////////////////////////
const fs = require("fs");
const ms = 500;  // Time in milliseconds

const pin = "2";        // light sensor, A2, P9_37

const IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw';
```

```
console.log('Hit ^C to stop');

// Read every 500ms
setInterval(readPin, ms);

function readPin() {
    var data = fs.readFileSync(IIOPATH).slice(0, -1);
    console.log('data = ' + data);
 }
// Bone  | Pocket | AIN
// ----- | ------ | ---
// P9_39 | P1_19  | 0
// P9_40 | P1_21  | 1
// P9_37 | P1_23  | 2
// P9_38 | P1_25  | 3
// P9_33 | P1_27  | 4
// P9_36 | P2_35  | 5
// P9_35 | P1_02  | 6
```

NOTE The code in Reading an analog voltage (analogIn.js) 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 kΩ resistor to give a voltage to measure across the flex sensor.

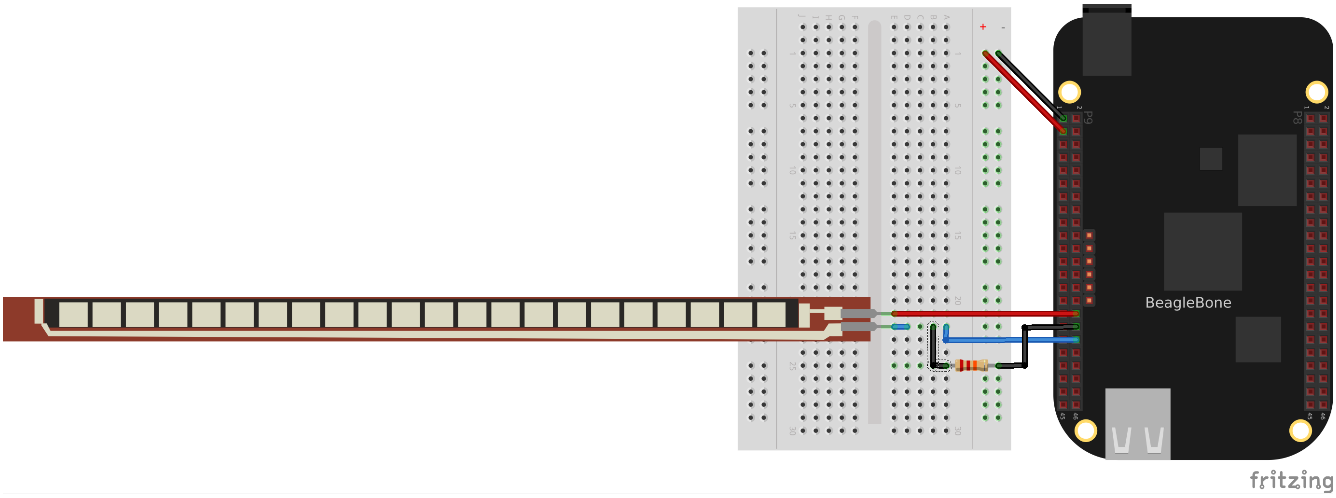


Figure 15. Reading a two-terminal flex resistor

The code in Reading an analog voltage (analogIn.py) and Reading an analog voltage (analogIn.js) also works for this setup.

Reading a Distance Sensor (Analog or Variable Voltage Sensor)

Problem

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

Solution

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 ~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.

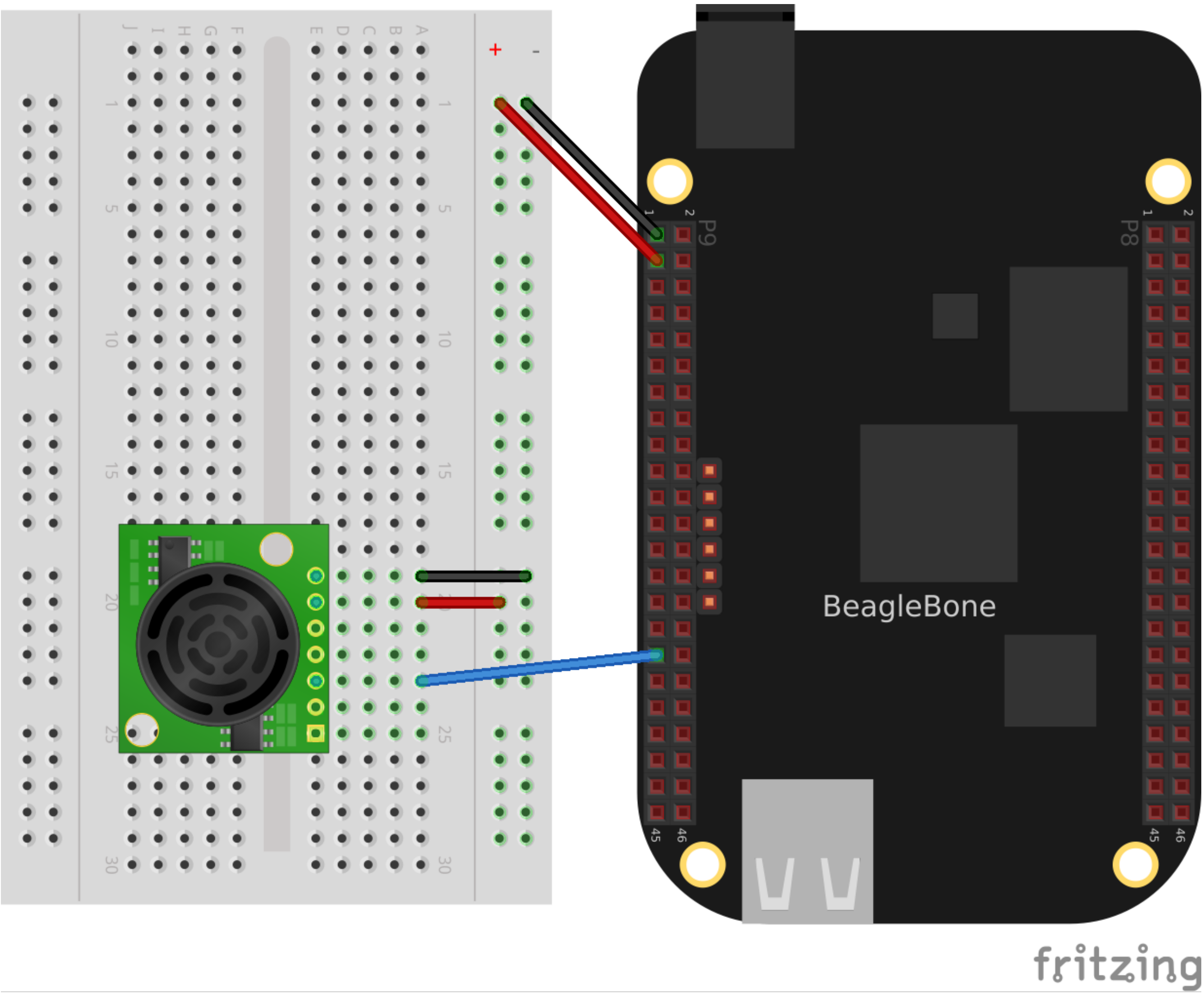


Figure 16. Wiring the LV-MaxSonar-EZ1 Sonar Range Finder to the P9_33 analog-in port

Reading an analog voltage (ultrasonicRange.js) shows the code that reads the sensor at a fixed interval.

Example 5. Reading an analog voltage (ultrasonicRange.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    ultrasonicRange.js
# //    Reads the analog value of the sensor.
# ////////////////////////////////////////
import time
ms = 250;  # Time in milliseconds

pin = "0"        # sensor, A0, P9_39

IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw'

print('Hit ^C to stop');

f = open(IIOPATH, "r")
```

```
while True:
    f.seek(0)
    data = f.read()[:-1]
    print('data= ' + data)
    time.sleep(ms/1000)

# // Bone  | Pocket | AIN
# // ----- | ------ | ---
# // P9_39 | P1_19  | 0
# // P9_40 | P1_21  | 1
# // P9_37 | P1_23  | 2
# // P9_38 | P1_25  | 3
# // P9_33 | P1_27  | 4
# // P9_36 | P2_35  | 5
# // P9_35 | P1_02  | 6
```

Example 6. Reading an analog voltage (ultrasonicRange.js)

```
#!/usr/bin/env node
//////////////////////////////////////
//  ultrasonicRange.js
//  Reads the analog value of the sensor.
//////////////////////////////////////
const fs = require("fs");
const ms = 250;  // Time in milliseconds

const pin = "0";        // sensor, A0, P9_39

const IIOPATH='/sys/bus/iio/devices/iio:device0/in_voltage'+pin+'_raw';

console.log('Hit ^C to stop');

// Read every ms
setInterval(readPin, ms);

function readPin() {
    var data = fs.readFileSync(IIOPATH);
    console.log('data= ' + data);
 }
// Bone  | Pocket | AIN
// ----- | ------ | ---
// P9_39 | P1_19  | 0
// P9_40 | P1_21  | 1
// P9_37 | P1_23  | 2
// P9_38 | P1_25  | 3
// P9_33 | P1_27  | 4
// P9_36 | P2_35  | 5
// P9_35 | P1_02  | 6
```

Accurately Reading the Position of a Motor or Dial

Problem

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

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.

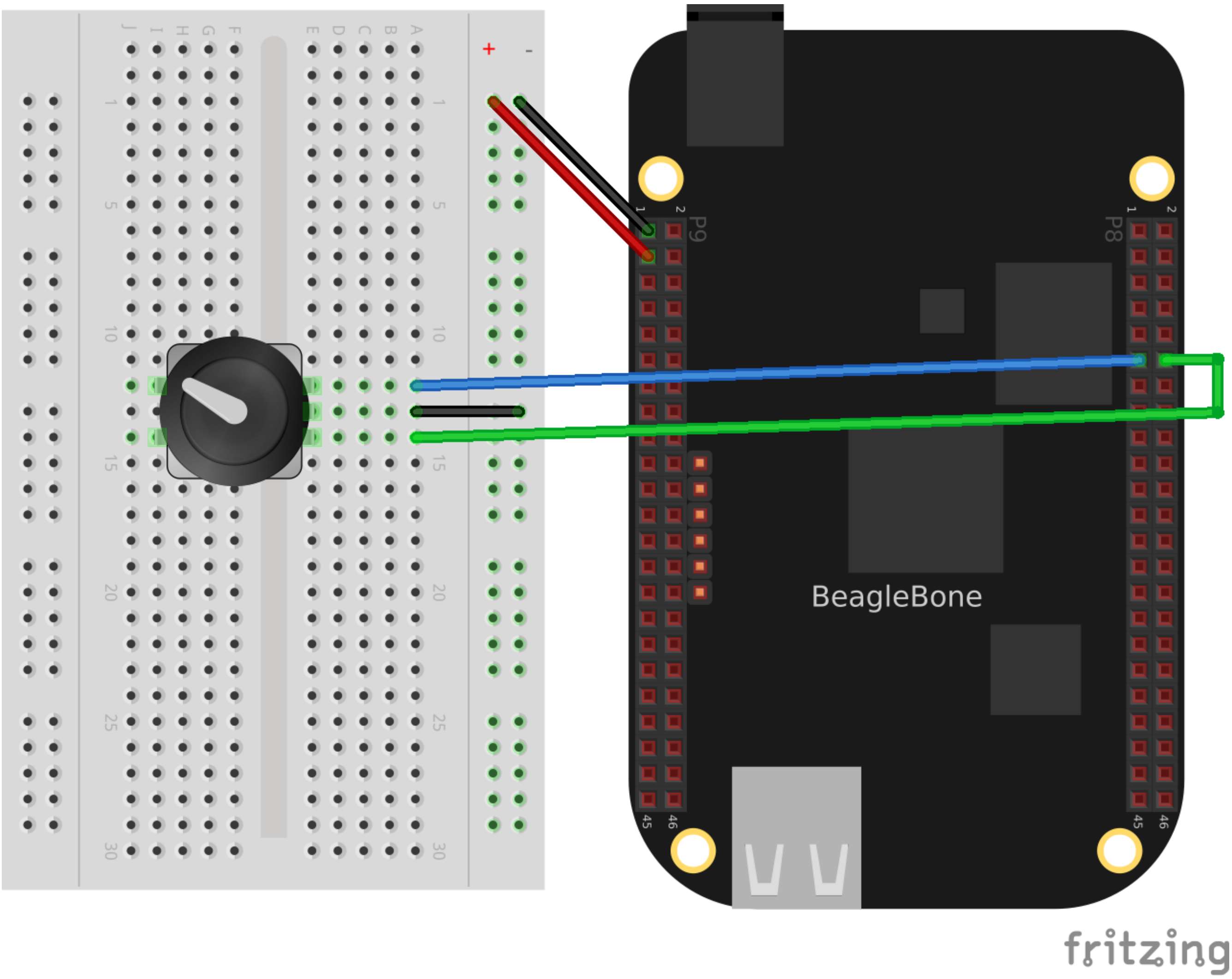


Figure 17. Wiring a rotary encoder using eQEP2

On the BeagleBone and PocketBeage 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

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:

```
<pre data-type="programlisting">
bone$ <strong>config-pin P8_11 qep</strong>
bone$ <strong>config-pin P8_12 qep</strong>
bone$ <strong>show-pins | grep qep</strong>
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)
</pre>
```

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 in Reading a rotary encoder (rotaryEncoder.js) to a file named *rotaryEncoder.js* and run it.

Example 7. Reading a rotary encoder (rotaryEncoder.py)

```
#!/usr/bin/env python
# // This uses the eQEP hardware to read a rotary encoder
# // bone$ config-pin P8_11 eqep
# // bone$ config-pin P8_12 eqep
import time

eQEP = '2'
COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0'

ms = 100    # Time between samples in ms
maxCount = '1000000'

# Set the eEQP maximum count
f = open(COUNTERPATH+'/ceiling', 'w')
```

```
f.write(maxCount)
f.close()

# Enable
f = open(COUNTERPATH+'/enable', 'w')
f.write('1')
f.close()

f = open(COUNTERPATH+'/count', 'r')

olddata = -1
while True:
    f.seek(0)
    data = f.read()[:-1]
    # Print only if data changes
    if data != olddata:
        olddata = data
        print("data = " + data)
    time.sleep(ms/1000)

# Black OR Pocket
# 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

# AI
# eQEP1:    P8.33 and P8.35
# eQEP2:    P8.11 and P8.12 or P9.19 and P9.41
# eQEP3:    P8.24 abd P8.25 or P9.27 and P9.42
```

Example 8. Reading a rotary encoder (rotaryEncoder.js)

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

const eQEP = "2";
const COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0';

const ms = 100;     // Time between samples in ms
const maxCount = '1000000';

// Set the eEQP maximum count
fs.writeFileSync(COUNTERPATH+'/ceiling', maxCount);

// Enable
fs.writeFileSync(COUNTERPATH+'/enable', '1');
```

```
setInterval(readEncoder, ms);    // Check state every ms

var olddata = -1;
function readEncoder() {
    var data = parseInt(fs.readFileSync(COUNTERPATH+'/count'));
    if(data != olddata) {
        // Print only if data changes
        console.log('data = ' + data);
        olddata = data;
    }
}

// Black OR Pocket
// 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

// AI
// eQEP1:   P8.33 and P8.35
// eQEP2:   P8.11 and P8.12 or P9.19 and P9.41
// eQEP3:   P8.24 abd P8.25 or P9.27 and P9.42
```

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.

See Also

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

Measuring a Temperature

Problem

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

Solution

The TMP101 sensor is a common digital temperature sensor that uses a standard I^{2}C-based serial protocol.

To make this recipe, you will need:

- Breadboard and jumper wires
- Two 4.7 kΩ resistors
- TMP101 temperature sensor

Wire the TMP101, as shown in Wiring an I^{2}C TMP101 temperature sensor.

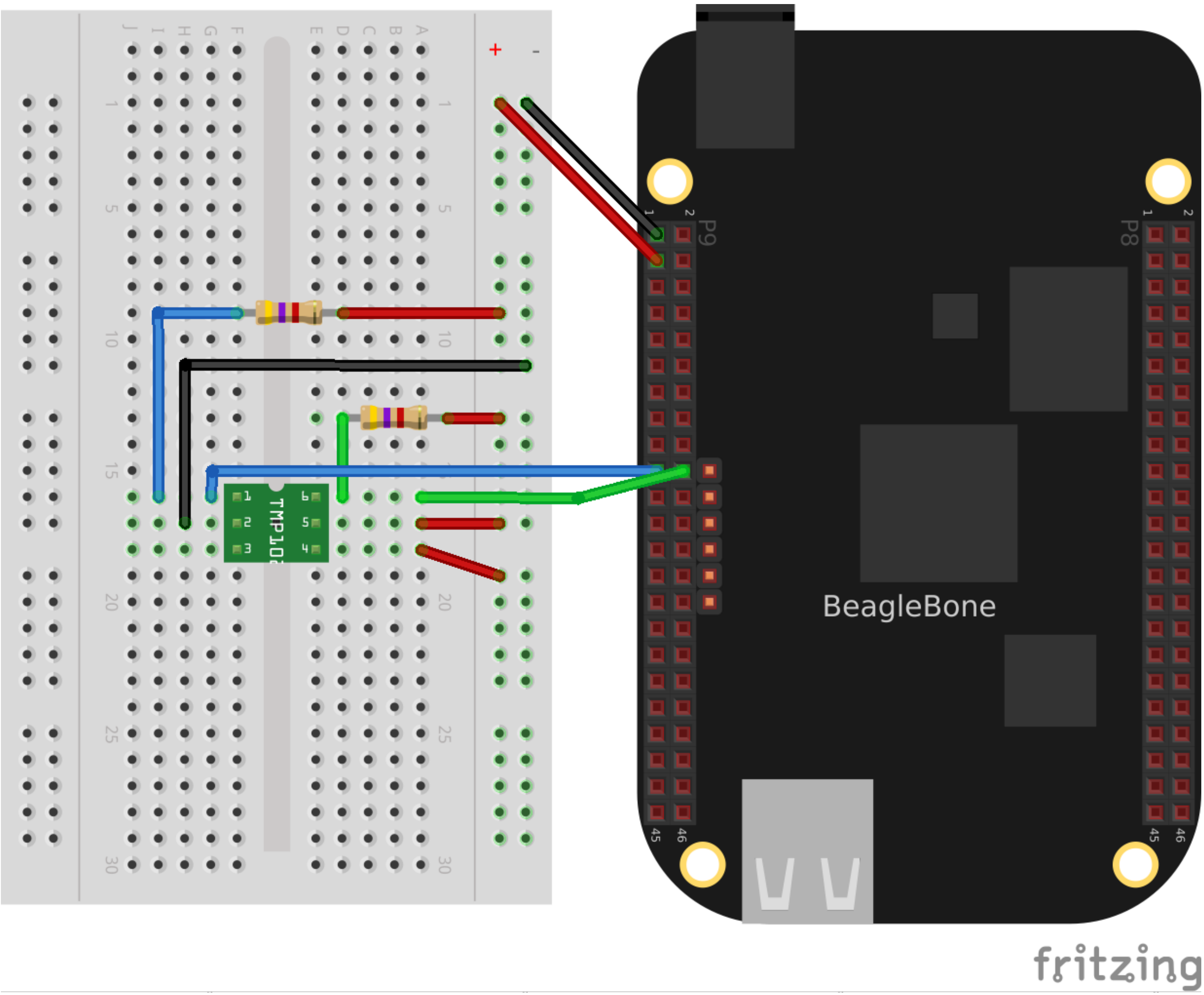


Figure 18. Wiring an I^{2}C TMP101 temperature sensor

There are two I^{2}C buses brought out to the headers. Table of I^{2}C outputs shows that you have wired your device to I^{2}C bus 2.

2 I2C ports

P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_50
GPIO_48	15	16	GPIO_51
I2C1_SCL	17	18	I2C1_SDA
I2C2_SCL	19	20	I2C2_SDA
I2C2_SCL	21	22	I2C2_SDA
GPIO_49	23	24	I2C1_SCL
GPIO_117	25	26	I2C1_SDA
GPIO_115	27	28	GPIO_113
GPIO_111	29	30	GPIO_112
GPIO_110	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	GPIO_7
DGND	43	44	DGND
DGND	45	46	DGND

P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

Figure 19. Table of I^{2}C outputs

Once the I^{2}C device is wired up, you can use a couple handy I^{2}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 I^{2}C tools, shows which I^{2}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 hexidecimal and degrees C. In this example, 0x18 = 24°C, which is 75.2°F. (Hmmm, the office is a bit warm today.) Try warming up the TMP101 with your finger and running i2cget again.

Example 9. I^{2}C tools

```
<pre data-type="programlisting">
bone$ <strong>i2cdetect -y -r 2</strong>
     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$ <strong>i2cget -y 2 0x49</strong>
0x18
</pre>
```

Reading the temperature via the kernel driver

The cleanest way to read the temperature from at TMP101 sensor is to use the kernel drive.

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

Example 10. I^2C TMP101 via Kernel

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/class/i2c-adapter/</strong>
bone$ <strong>ls</strong>
i2c-0  i2c-1  i2c-2         # Three i2c busses (bus 0 is internal)
bone$ <strong>cd i2c-2</strong> # Pick bus 2
bone$ <strong>ls -ls</strong>
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@100
000/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
</pre>
```

Assuming the TMP101 is at address 0x48:

```
<pre data-type="programlisting">
bone$ <strong>echo tmp101 0x49 > new_device</strong>
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>dmesg -H | tail -3</strong>
[ +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.

```
<pre data-type="programlisting">
bone$ <strong>ls</strong>
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.

```
<pre data-type="programlisting">
bone$ <strong>cd 2-0048/hwmon/hwmon0</strong>
bone$ <strong>ls -F</strong>
device@  name  power/  subsystem@  temp1_input  temp1_max
temp1_max_hyst  uevent  update_interval
bone$ <strong>cat temp1_input</strong>
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. Reading an I^2C device (i2cTemp.py) shows how to read the TMP101 from BoneScript.

Example 11. Reading an I^2C device (i2cTemp.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    i2cTemp.py
# //      Read a TMP101 sensor on i2c bus 2, address 0x49
# //    Wiring: Attach to i2c as shown in text.
# //    Setup:  echo tmp101 0x49 > /sys/class/i2c-adapter/i2c-2/new_device
# //    See:
# ////////////////////////////////////////
import time

ms = 1000   # Read time in ms
bus = '2'
addr = '49'
I2CPATH='/sys/class/i2c-adapter/i2c-'+bus+'/'+bus+'-00'+addr+'/hwmon/hwmon0';

f = open(I2CPATH+"/temp1_input", "r")

while True:
    f.seek(0)
    data = f.read()[:-1]    # returns mili-degrees C
    print("data (C) = " + str(int(data)/1000))
```

```
    time.sleep(ms/1000)
```

Example 12. Reading an I^2C device (i2cTemp.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  i2cTemp.js
//      Read at TMP101 sensor on i2c bus 2, address 0x49
//  Wiring: Attach to i2c as shown in text.
//  Setup:  echo tmp101 0x49 > /sys/class/i2c-adapter/i2c-2/new_device
//  See:
////////////////////////////////////////
const fs = require("fs");

const ms = 1000;   // Read time in ms
const bus = '2';
const addr = '49';
I2CPATH='/sys/class/i2c-adapter/i2c-'+bus+'/'+bus+'-00'+addr+'/hwmon/hwmon0';

// Read every ms
setInterval(readTMP, ms);

function readTMP() {
    var data = fs.readFileSync(I2CPATH+"/temp1_input").slice(0, -1);
    console.log('data (C) = ' + data/1000);
 }
```

Run the code by using the following command:

```
<pre data-type="programlisting">
bone$ <strong>./i2cTemp.js</strong>
data (C) = 25.625
data (C) = 27.312
data (C) = 28.187
data (C) = 28.375
^C
</pre>
```

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

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.

```
<pre data-type="programlisting">
bone$ <strong>pip install smbus</strong>
</pre>
```

Example 13. Reading an I^2C device (i2cTemp.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    i2ctmp101.py
# //      Read at TMP101 sensor on i2c bus 2, address 0x49
# //    Wiring: Attach to i2c as shown in text.
# //    Setup:  pip install smbus
# //    See:
# ////////////////////////////////////////
import smbus
import time

ms = 1000               # Read time in ms
bus = smbus.SMBus(2)    # Using i2c bus 2
addr = 0x49             # TMP101 is at address 0x49

while True:
    data = bus.read_byte_data(addr, 0)
    print("temp (C) = " + str(data))
    time.sleep(ms/1000)
```

This gets only 8 bits for the temperature. See the TMP101 datasheet for details on how to get up to 12 bits.

Reading Temperature via a Dallas 1-Wire Device

Problem

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

Solution

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 [1].

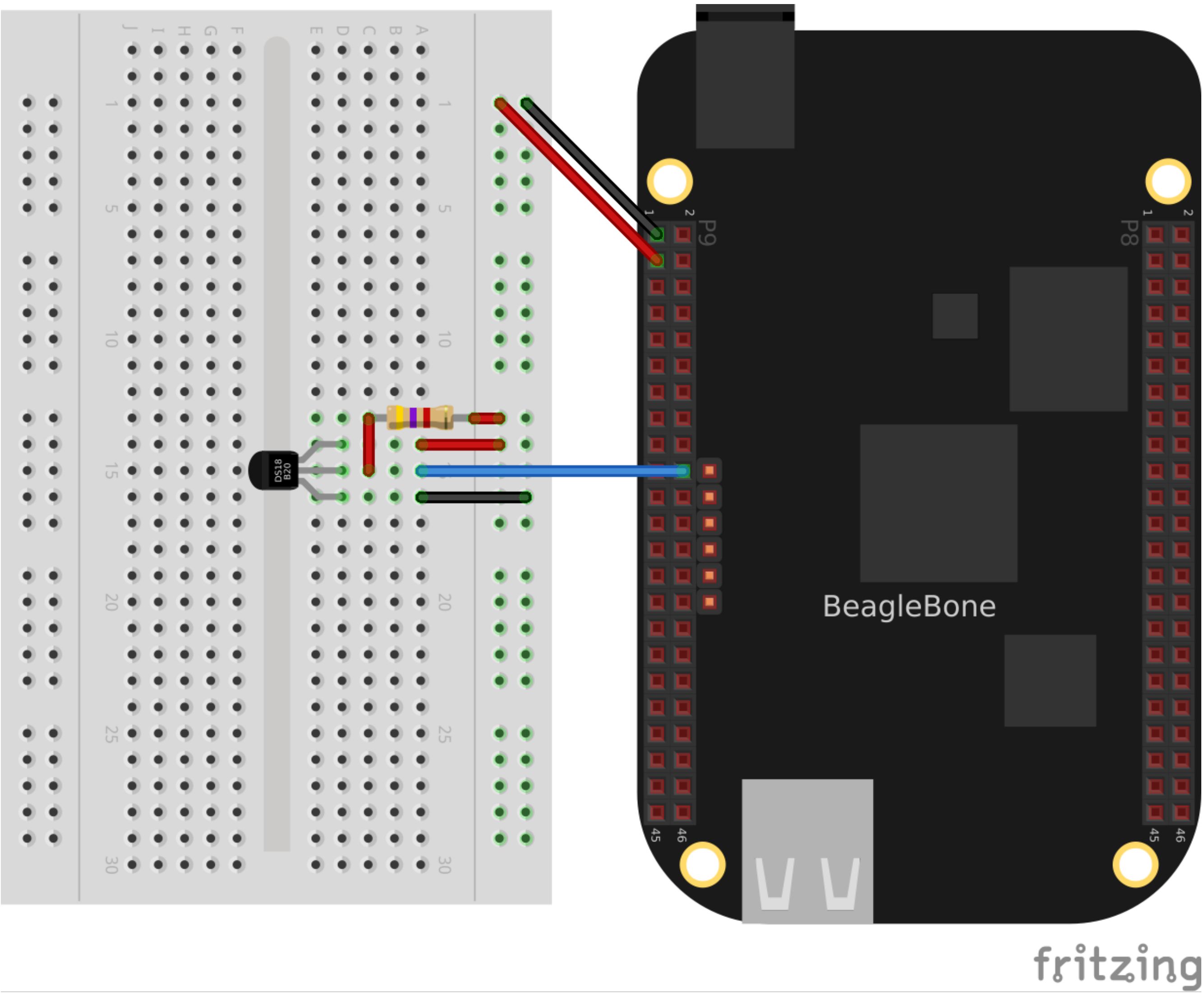


Figure 20. Wiring a Dallas 1-Wire temperature sensor [1]

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

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

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

Reboot the bone:

```
<pre data-type="programlisting">
bone$ <strong>reboot</strong>
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>ls /sys/bus/w1/devices/</strong>
28-00000114ef1b  28-00000128197d  w1_bus_master1
</pre>
```

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 in Reading a temperature with a DS18B20 (w1.js) 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.

Example 14. Reading a temperature with a DS18B20 (w1.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    w1.js
# //      Read a Dallas 1-wire device on P9_12
# //    Wiring: Attach gnd and 3.3V and data to P9_12
# //    Setup:  Edit /boot/uEnv.txt to include:
# //              uboot_overlay_addr4=BB-W1-P9.12-00A0.dtbo
# //    See:
# ////////////////////////////////////////
import time

ms = 500   # Read time in ms
#  Do ls /sys/bus/w1/devices and find the address of your device
addr = '28-00000d459c2c' # Must be changed for your device.
W1PATH ='/sys/bus/w1/devices/' + addr

f = open(W1PATH+'/temperature')

while True:
    f.seek(0)
    data = f.read()[:-1]
    print("temp (C) = " + str(int(data)/1000))
    time.sleep(ms/1000)
```

Example 15. Reading a temperature with a DS18B20 (w1.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  w1.js
//      Read a Dallas 1-wire device on P9_12
//  Wiring: Attach gnd and 3.3V and data to P9_12
//  Setup:  Edit /boot/uEnv.txt to include:
//              uboot_overlay_addr4=BB-W1-P9.12-00A0.dtbo
//  See:
////////////////////////////////////////
```

```
const fs = require("fs");

const ms = 500   // Read time in ms
// Do ls /sys/bus/w1/devices and find the address of your device
const addr = '28-00000d459c2c'; // Must be changed for your device.
const W1PATH ='/sys/bus/w1/devices/' + addr;

// Read every ms
setInterval(readW1, ms);

function readW1() {
    var data = fs.readFileSync(W1PATH+'/temperature').slice(0, -1);
    console.log('temp (C) = ' + data/1000);
 }
```

```
<pre data-type="programlisting">
bone$ <strong>./w1.js</strong>
temp (C) = 28.625
temp (C) = 29.625
temp (C) = 30.5
temp (C) = 31.0
^C
</pre>
```

Discussion

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

Playing and Recording Audio

Problem

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

Solution

One approach is to buy an audio cape ([app_capes]), but another, possibly cheaper approach is to buy a USB audio adapter, such as the one shown in A USB audio dongle. Some adapters that I've tested are provided in [app_misc].

Figure 21. A USB audio dongle

Drivers for the Advanced Linux Sound Architecture (ALSA) are already installed on the Bone. 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.

Example 16. Listing the ALSA audio output and input devices on the Bone

```
<pre data-type="programlisting">
bone$ <strong>aplay -l</strong>
**** 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$ <strong>arecord -l</strong>
**** 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
</pre>
```

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.

Example 17. Change the default audio out by putting this in ~/.asoundrc (audio.asoundrc)

```
pcm.!default {
  type plug
  slave {
    pcm "hw:1,0"
  }
}
ctl.!default {
  type hw
  card 1
}
```

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

```
<pre data-type="programlisting">
bone$ <strong>aplay test.wav</strong>
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>sudo apt update</strong>
bone$ <strong>sudo apt install mplayer</strong>
bone$ <strong>mplayer test.mp3</strong>
</pre>
```

Discussion

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

Displays and Other Outputs

Introduction

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 www.github.com. Go here Cloning the Cookbook Repository for instructions.

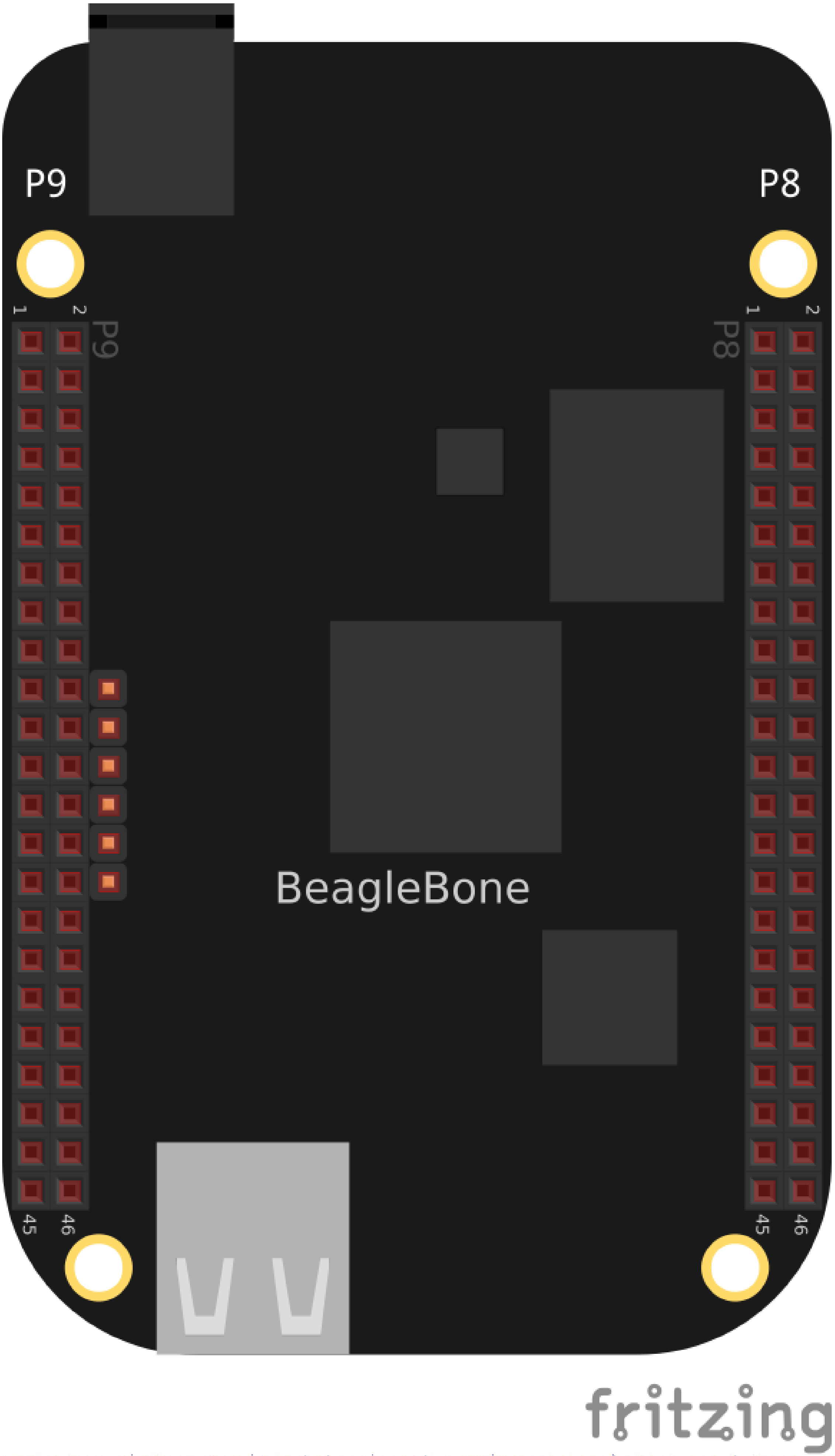


Figure 22. The P8 and P9 GPIO headers

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).

Toggling an Onboard LED

Problem

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

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.

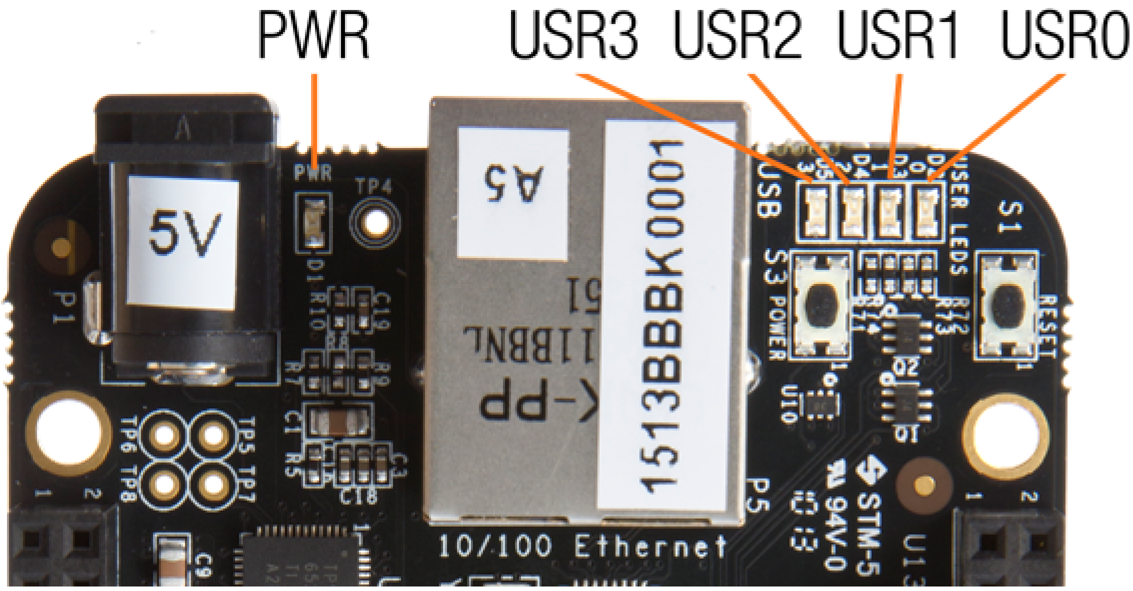


Figure 23. The four USER LEDs

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).

Example 18. Using an internal LED (internLED.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
#   internalLED.py
#   Blinks A USR LED.
#   Wiring:
#   Setup:
#   See:
# ////////////////////////////////////////
```

```
import time

ms = 250      # Blink time in ms
LED = 'usr0'; # LED to blink
LEDPATH = '/sys/class/leds/beaglebone:green:'+LED+'/brightness'

state = '1'    # Initial state

f = open(LEDPATH, "w")

while True:
    f.seek(0)
    f.write(state)
    if (state == '1'):
        state = '0'
    else:
        state = '1'
    time.sleep(ms/1000)
```

Example 19. Using an internal LED (internLED.js)

```
#!/usr/bin/env node
// ////////////////////////////////////////
//  internalLED.js
//  Blinks the USR LEDs.
//  Wiring:
//  Setup:
//  See:
// ////////////////////////////////////////
const fs = require('fs');
const ms = 250;     // Blink time in ms
const LED = 'usr0'; // LED to blink
const LEDPATH = '/sys/class/leds/beaglebone:green:'+LED+'/brightness';

var state = '1';    // Initial state

setInterval(flash, ms);    // Change state every ms

function flash() {
    fs.writeFileSync(LEDPATH, state)
    if(state === '1') {
        state = '0';
    } else {
        state = '1';
    }
}
```

In the bash command window, enter the following commands:

```
<pre data-type="programlisting">
bone$ <strong>cd ~/BoneCookbook/docs/03displays/code</strong>
bone$ <strong>./internLED.js</strong>
</pre>
```

The USER0 LED should now be flashing.

Toggling an External LED

Problem

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

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 Ω to 470 Ω 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 Ω or 470 Ω 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.

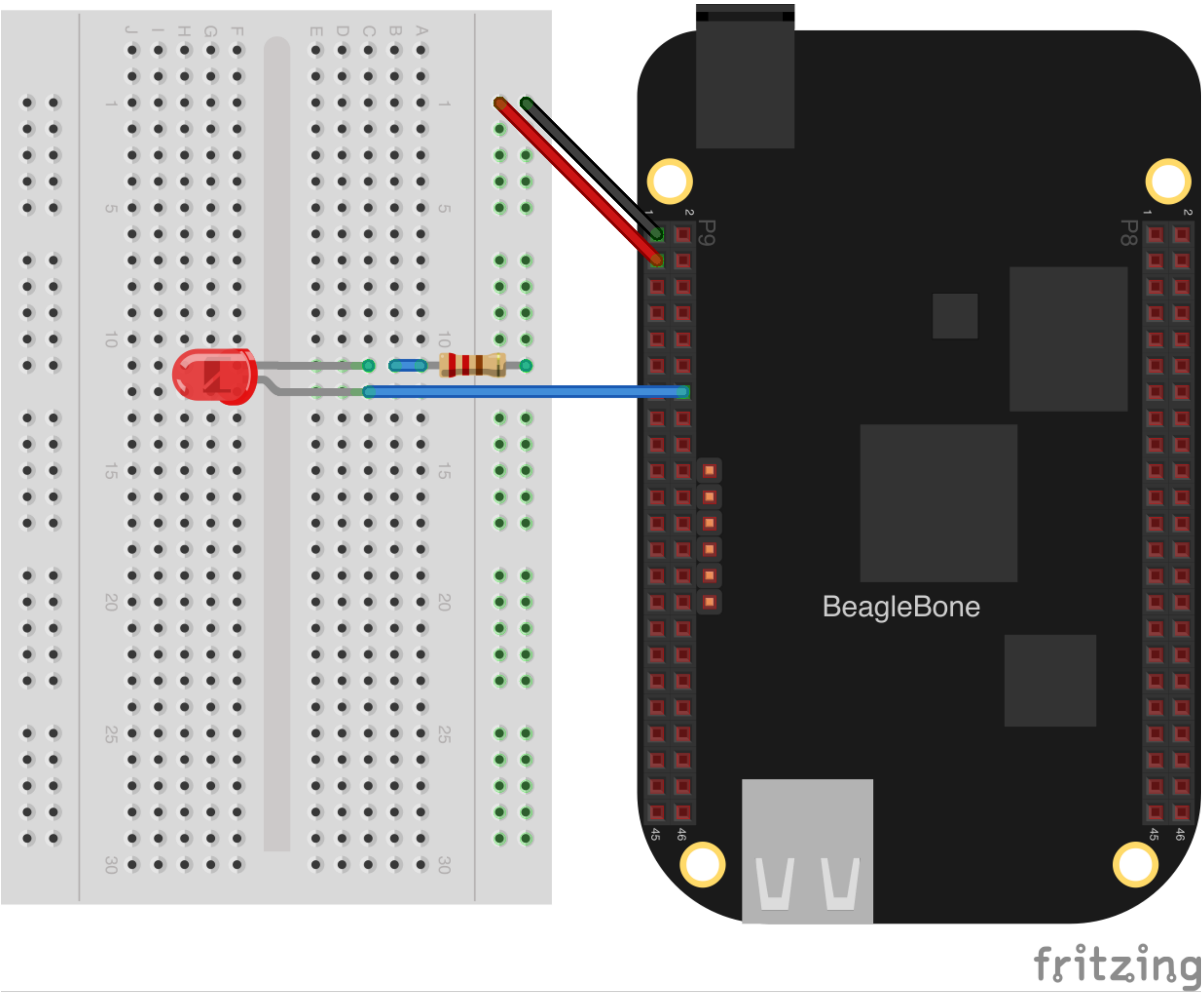


Figure 24. Diagram for using an external LED

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).

Example 20. Code for using an external LED (externLED.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    externalLED.py
# //    Blinks an external LED wired to P9_14.
# //    Wiring: P9_14 connects to the plus lead of an LED.  The negative lead of
the
#           LED goes to a 220 Ohm resistor.  The other lead of the resistor goes
#           to ground.
# //    Setup:
# //    See:
# ////////////////////////////////////////
import time
import os

ms = 250        # Time to blink in ms
```

```
# Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1, line 18 maps to
50
pin = '50'

GPIOPATH='/sys/class/gpio/'
# Make sure pin is exported
if (not os.path.exists(GPIOPATH+"gpio"+pin)):
    f = open(GPIOPATH+"export", "w")
    f.write(pin)
    f.close()

# Make it an output pin
f = open(GPIOPATH+"gpio"+pin+"/direction", "w")
f.write("out")
f.close()

f = open(GPIOPATH+"gpio"+pin+"/value", "w")
# Blink
while True:
    f.seek(0)
    f.write("1")
    time.sleep(ms/1000)

    f.seek(0)
    f.write("0")
    time.sleep(ms/1000)
f.close()
```

Example 21. Code for using an external LED (externLED.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  externalLED.js
//  Blinks the P9_14 pin
//  Wiring:
//  Setup:
//  See:
////////////////////////////////////////
const fs = require("fs");

// Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1, line 18 maps to
50
pin="50";

GPIOPATH="/sys/class/gpio/";
// Make sure pin is exported
if(!fs.existsSync(GPIOPATH+"gpio"+pin)) {
    fs.writeFileSync(GPIOPATH+"export", pin);
}
```

```
// Make it an output pin
fs.writeFileSync(GPIOPATH+"gpio"+pin+"/direction", "out");

// Blink every 500ms
setInterval(toggle, 500);

state="1";
function toggle() {
    fs.writeFileSync(GPIOPATH+"gpio"+pin+"/value", state);
    if(state == "0") {
        state = "1";
    } else {
        state = "0";
    }
}
```

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

Discussion

This method of controlling and external device uses **sysfs**, a virtual file system that exposes drivers to userspace. /sys/class is a comment interace.

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/class</strong>
bone$ <strong>ls</strong>
ata_device  devcoredump    extcon          hidraw       leds      phy           rtc
tpm        usb_role
ata_link    devfreq        firmware        hwmon        mbox      power_supply
scsi_device  tpmrm      vc
ata_port    devfreq-event  gb_authenticate  i2c-adapter  mdio_bus  pps
scsi_disk    tty        video4linux
backlight   devlink        gb_fw_mgmt      i2c-dev      mem       ptp
scsi_host    typec      vtconsole
bdi         dma            gb_loopback     ieee802154   misc      pwm
spi_master   typec_mux  wakeup
block       drm            gb_raw          input        mmc_host  regulator
spi_slave    ubi        watchdog
bluetooth   drm_dp_aux_dev  <strong>gpio</strong>           iommu        mtd
remoteproc    spidev      udc
bsg         dvb            graphics        lcd          net       rfkill        thermal
uio
</pre>
```

There are many interfaces. We're interested in the **gpio** interface.

```
<pre data-type="programlisting">
bone$ <strong>cd gpio</strong>
bone$ <strong>ls</strong>
export   gpio113  gpio13  gpio23  gpio32  gpio38  gpio47  gpio60  gpio67  gpio72  gpio78
```

```
gpio87      gpiochip64
gpio10   gpio114  gpio14  gpio26  gpio33  gpio39  gpio48  gpio61  gpio68  gpio73  gpio79
gpio88      gpiochip96
gpio11   gpio115  gpio15  gpio27  gpio34  gpio4   gpio49  gpio62  gpio69  gpio74  gpio8
gpio89      unexport
gpio110  gpio116  gpio2   gpio3   gpio35  gpio44  gpio5   gpio63  gpio7   gpio75  gpio80
gpio9
gpio111  gpio117  gpio20  gpio30  gpio36  gpio45  gpio50  gpio65  gpio70  gpio76  gpio81
gpiochip0
gpio112  gpio12   gpio22  gpio31  gpio37  gpio46  gpio51  gpio66  gpio71  gpio77  gpio86
gpiochip32
</pre>
```

In the code in Toggling an External LED we saw **P9_14** is pin **50**. That same 50 maps to the **gpio50** directory shown above. If **gpio50** doesn't show up, export it. A close ook at the Python and JavaScript code shows it looks for gpio50 and exports it if it isn't there.

```
<pre data-type="programlisting">
bone$ <strong>echo 50 > export</strong>
bone$ <strong>cd gpio50</strong>
bone$ <strong>ls</strong>
active_low  device  direction  edge  label  power  subsystem  uevent  value
bone$ <strong>cat direction</strong>
out
bone$ <strong>cat value</strong>
1
</pre>
```

Here we see it's configured to be an output and it's currently turned on. Turn it off with:

```
<pre data-type="programlisting">
bone$ <strong>echo 0 > value</strong>
</pre>
```

The LED should now be off.

Toggling an External LED via gpiod

Problem

You hear sysfs is dead and want to know what to use instead.

Solution

sysfs is deprecated, but most believe it will be around for a long time. The repository has many examples of gpiod in docs/03displays/code/gpiod. Here's an example:

Example 22. Code for using an external LED (toggle1.py)

```
#!/usr/bin/env python3
```

```
# ////////////////////////////////////////
#   toggle1.py
#  Toggles the P9_14 pin as fast as it can. P9_14 is line 18 on chip 1.
#   Wiring: Attach an oscilloscope to P9_14 to see the squarewave or
#          uncomment the sleep and attach an LED.
#   Setup:  sudo apt update; pip install gpiod
#   See:
https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/tree/bindings/python/exa
mples
# ////////////////////////////////////////

import gpiod
import time

LED_CHIP = 'gpiochip1'
LED_LINE_OFFSET = [18]  # P9_14

chip = gpiod.Chip(LED_CHIP)

lines = chip.get_lines(LED_LINE_OFFSET)
lines.request(consumer='blink', type=gpiod.LINE_REQ_DIR_OUT)

while True:
    lines.set_values([0])
    time.sleep(0.1)
    lines.set_values([1])
    time.sleep(0.1)
```

Toggling a High-Voltage External Device

Problem

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

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.

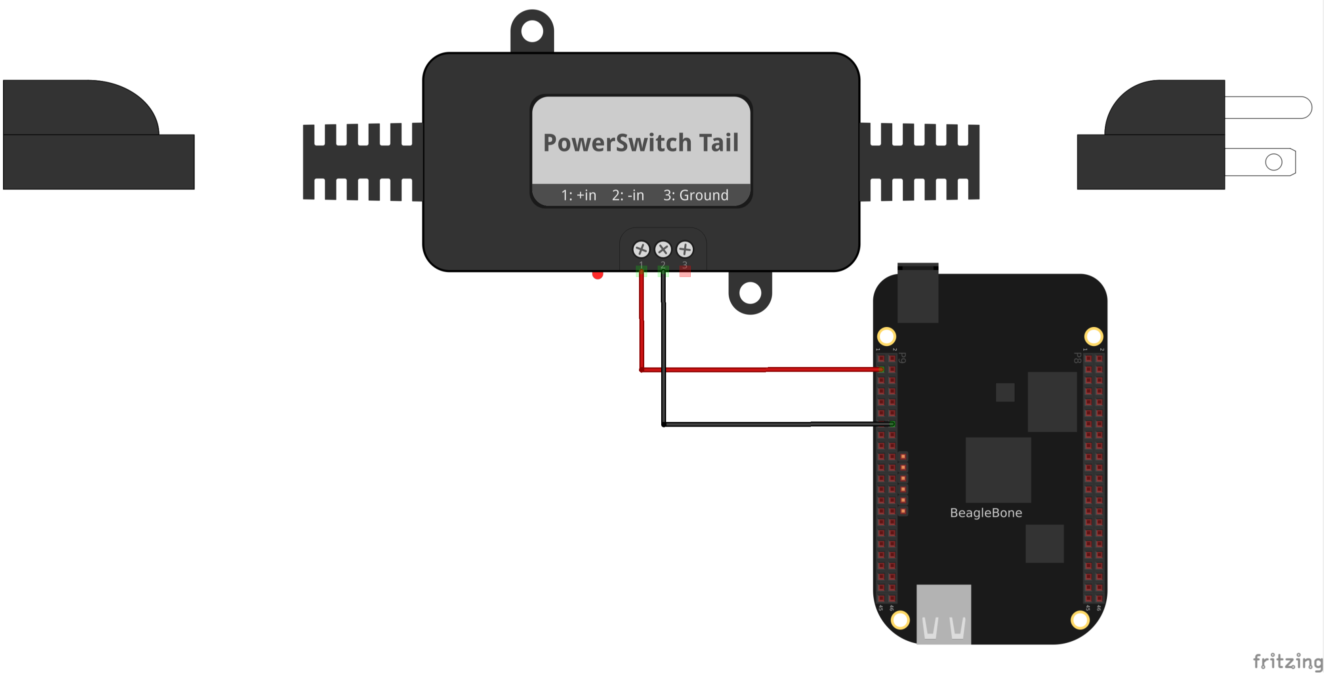


Figure 25. Diagram for wiring PowerSwitch Tail II

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)).

Fading an External LED

Problem

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

Solution

```
Use the Bone's pulse width modulation (PWM) hardware to fade an LED. We'll use the
same circuit as before (<<displays_externLED_fig>>). Find the code in
<<py_fadeLED_code>> Next configure the pins.  We are using P9_14 so run:
```

```
<pre>
bone$ <strong>config-pin P9_14 pwm
</pre>
```

Then run it as before.

Example 23. Code for using an external LED (fadeLED.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    fadeLED.py
# //    Blinks the P9_14 pin
# //    Wiring:
# //    Setup:  config-pin P9_14 pwm
```

```
# //    See:
# ////////////////////////////////////////
import time
ms = 20;   # Fade time in ms

pwmPeriod = 1000000    # Period in ns
pwm     = '1'  # pwm to use
channel = 'a'  # channel to use
PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel
step = 0.02    # Step size
min = 0.02     # dimmest value
max = 1        # brightest value
brightness = min # Current brightness

f = open(PWMPATH+'/period', 'w')
f.write(str(pwmPeriod))
f.close()

f = open(PWMPATH+'/enable', 'w')
f.write('1')
f.close()

f = open(PWMPATH+'/duty_cycle', 'w')
while True:
    f.seek(0)
    f.write(str(round(pwmPeriod*brightness)))
    brightness  += step
    if(brightness >= max or brightness <= min):
        step = -1 * step
    time.sleep(ms/1000)

# | 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
```

Example 24. Code for using an external LED (fadeLED.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  fadeLED.js
//  Blinks the P9_14 pin
//  Wiring:
//  Setup:  config-pin P9_14 pwm
//  See:
////////////////////////////////////////
```

```
const fs = require("fs");
const ms = '20';    // Fade time in ms

const pwmPeriod = '1000000';    // Period in ns
const pwm     = '1';  // pwm to use
const channel = 'a';  // channel to use
const PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel;
var   step = 0.02;  // Step size
const min = 0.02,     // dimmest value
    max = 1;        // brightest value
var brightness = min; // Current brightness;


// Set the period in ns
fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
fs.writeFileSync(PWMPATH+'/duty_cycle', pwmPeriod/2);
fs.writeFileSync(PWMPATH+'/enable', '1');

setInterval(fade, ms);      // Step every  ms

function fade() {
     fs.writeFileSync(PWMPATH+'/duty_cycle',
        parseInt(pwmPeriod*brightness));
    brightness += step;
    if(brightness >= max || brightness <= min) {
        step = -1 * step;
    }
}

// | 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
```

Discussion

The Bone has several outputs that can be used as pwm's as shown in Table of PWM outputs. There are three EHRPWM's (enhanced high resolution pulse width modulation) which each has a pair of pwm channels. Each pair must have the same pulse 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
EHRPWM0B	21	22	EHRPWM0A	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
EHRPWM0B	29	30	GPIO_112	GPIO_87	29	30	GPIO_89
EHRPWM0A	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	ECAPPWM0	GPIO_74	41	42	GPIO_75
DGND	43	44	DGND	GPIO_72	43	44	GPIO_73
DGND	45	46	DGND	EHRPWM2A	45	46	EHRPWM2B

Figure 26. Table of PWM outputs

The pwm’s are accessed through /dev/bone/pwm

```
<pre>
bone$ <strong>cd /dev/bone/pwm</strong>
bone$ <strong>ls</strong>
0  1  2
</pre>
```

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

```
<pre>
bone$ <strong>cd 1</strong>
bone$ <strong>ls</strong>
a  b
bone$ <strong>cd a</strong>
bone$ <strong>ls</strong>
capture  duty_cycle  enable  period  polarity  power  uevent
</pre>
```

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.

```
<pre>
bone$ <strong>echo 1000000000 > period</strong>
bone$ <strong>echo  500000000 > duty_cycle</strong>
bone$ <strong>echo 1 > enable</strong>
</pre>
```

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 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

Writing to an LED Matrix

Problem

You have an I^{2}C-based LED matrix to interface.

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^{2}C Backpack.

To make this recipe, you will need:

- Breadboard and jumper wires
- Two 4.7 k□ resistors
- I^{2}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^{2}C LED matrix.

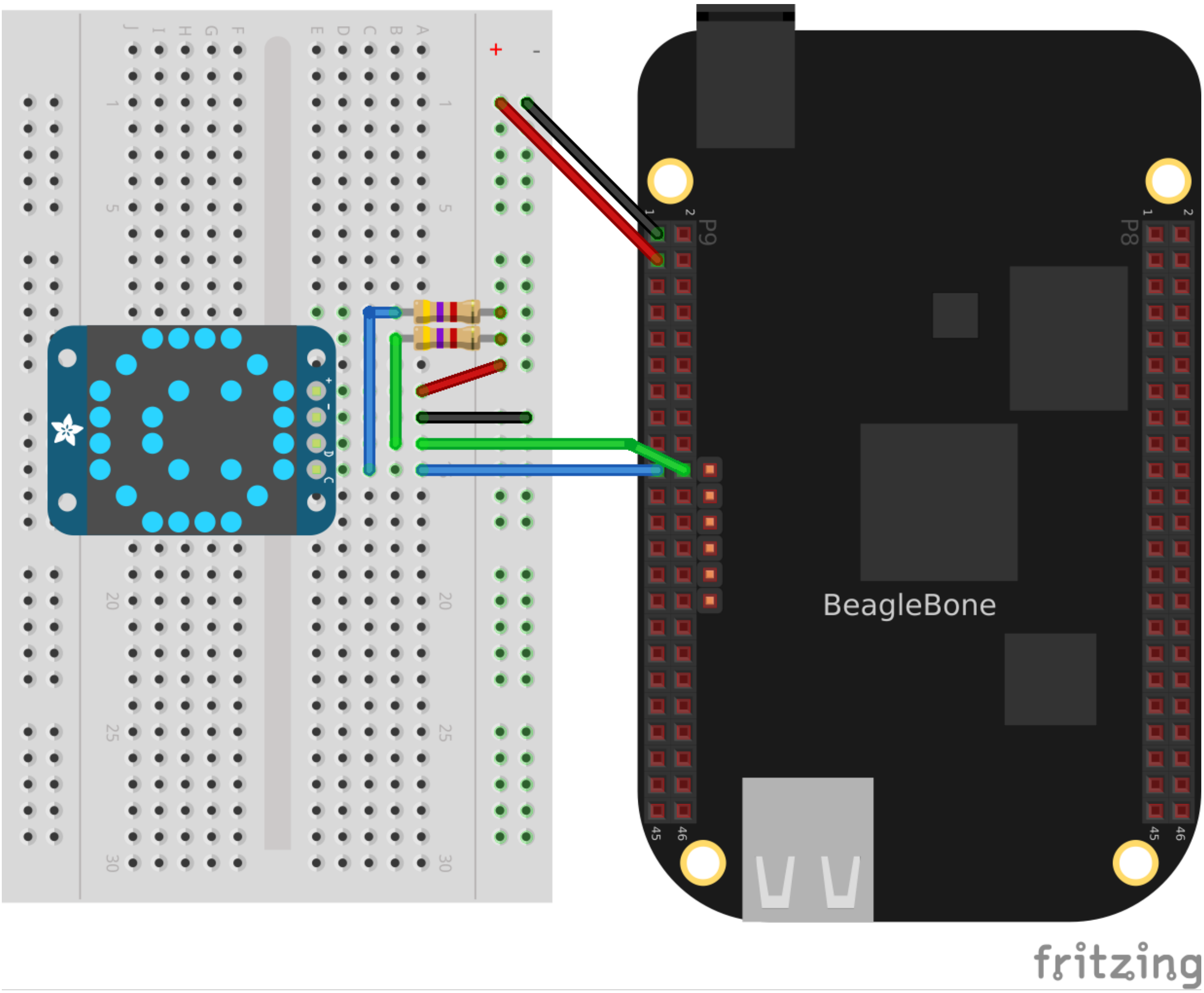


Figure 27. Wiring an I^{2}C LED matrix

Measuring a Temperature shows how to use i2cdetect to discover the address of an I^{2}C device.

Run the i2cdetect -y -r 2 command to discover the address of the display on I^{2}C bus 2, as shown in Using I^{2}C command-line tools to discover the address of the display.

Example 25. Using I^{2}C command-line tools to discover the address of the display

```
<pre data-type="programlisting">
bone$ <strong>i2cdetect -y -r 2</strong>
     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: 70 -- -- -- -- -- -- --
</pre>
```

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 0.70.

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

```
<pre data-type="programlisting">
bone$ <strong>pip install smbus</strong>  # (Do this only once.)
bone$ <strong>./matrixLEDi2c.py</strong>
</pre>
```

Example 26. LED matrix display (matrixLEDi2c.py)

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

bus = smbus.SMBus(2)  # Use i2c bus 2       ①
matrix = 0x70         # Use address 0x70    ②
ms = 1;               # Delay between images in ms

# The first byte is GREEN, the second is RED.   ③
smile = [0x00, 0x3c, 0x00, 0x42, 0x28, 0x89, 0x04, 0x85,
    0x04, 0x85, 0x28, 0x89, 0x00, 0x42, 0x00, 0x3c
]
frown = [0x3c, 0x00, 0x42, 0x00, 0x85, 0x20, 0x89, 0x00,
    0x89, 0x00, 0x85, 0x20, 0x42, 0x00, 0x3c, 0x00
]
neutral = [0x3c, 0x3c, 0x42, 0x42, 0xa9, 0xa9, 0x89, 0x89,
    0x89, 0x89, 0xa9, 0xa9, 0x42, 0x42, 0x3c, 0x3c
]

bus.write_byte_data(matrix, 0x21, 0)   # Start oscillator (p10) ④
bus.write_byte_data(matrix, 0x81, 0)   # Disp on, blink off (p11)
bus.write_byte_data(matrix, 0xe7, 0)   # Full brightness (page 15)

bus.write_i2c_block_data(matrix, 0, frown)      ⑤
for fade in range(0xef, 0xe0, -1):              ⑥
    bus.write_byte_data(matrix, fade, 0)
    time.sleep(ms/10)

bus.write_i2c_block_data(matrix, 0, neutral)
for fade in range(0xe0, 0xef, 1):
```

```
    bus.write_byte_data(matrix, fade, 0)
    time.sleep(ms/10)

bus.write_i2c_block_data(matrix, 0, smile)
```

① This line states which bus to use.

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 disply fade out and back in again.

Driving a 5 V Device

Problem

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

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^2C 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.

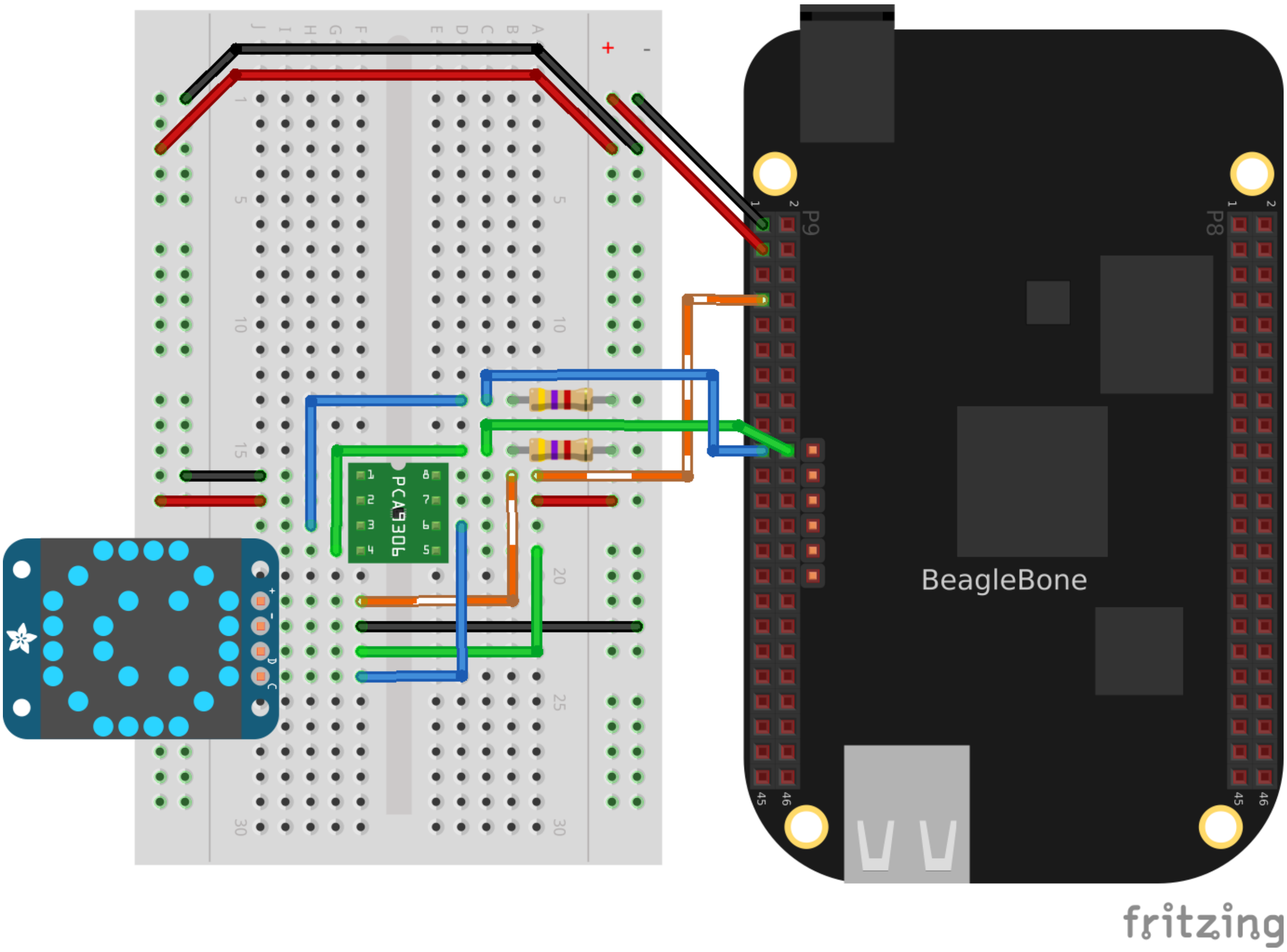


Figure 28. Wiring a PCA9306 level translator to an LED matrix

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

Writing to a NeoPixel LED String Using the PRUs

Problem

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

Solution

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

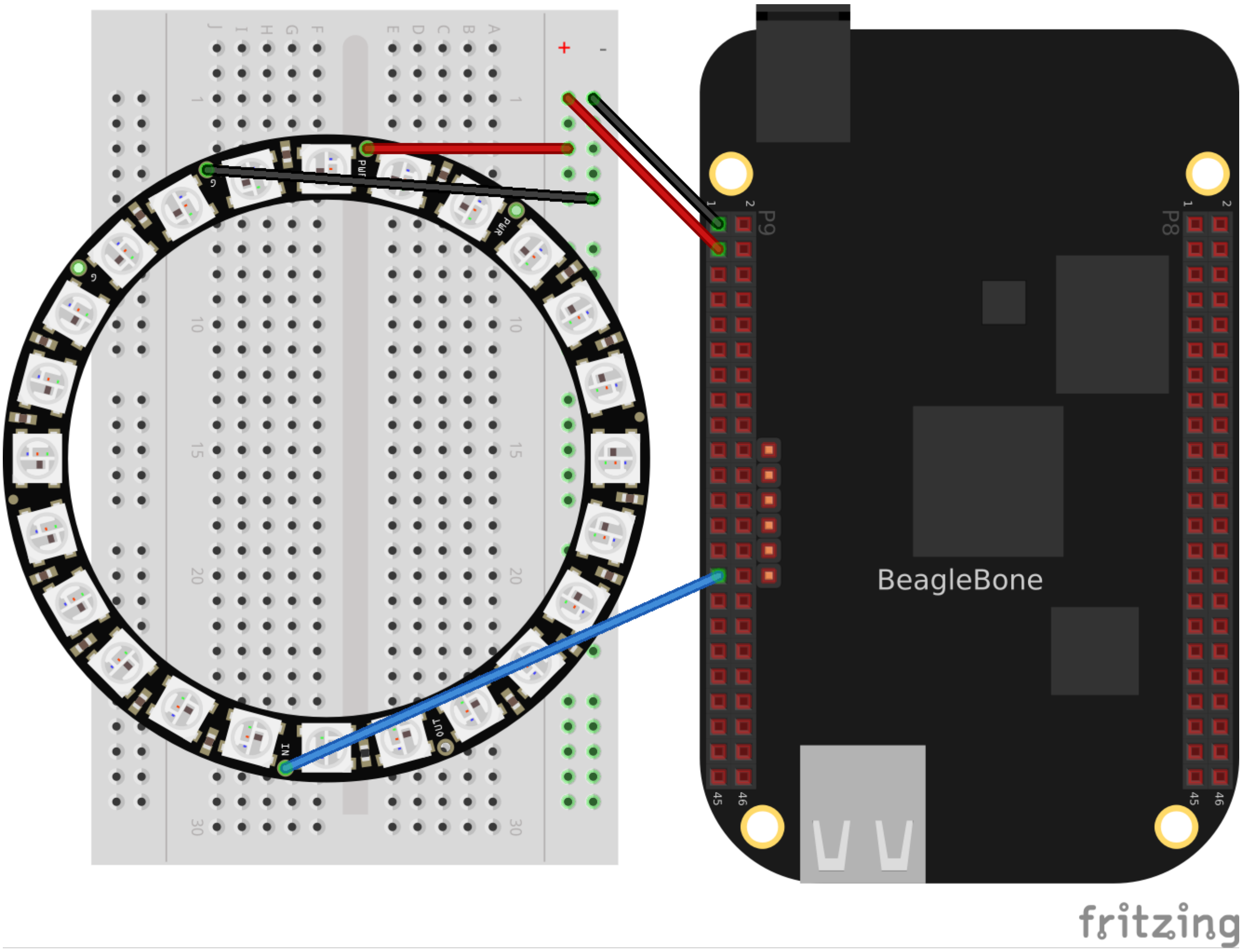


Figure 29. Wiring an Adafruit NeoPixel LED matrix to P9_29

Motors

Introduction

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°.

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 www.github.com. Go here Cloning the Cookbook Repository for instructions.

Controlling a Servo Motor

Problem

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

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:

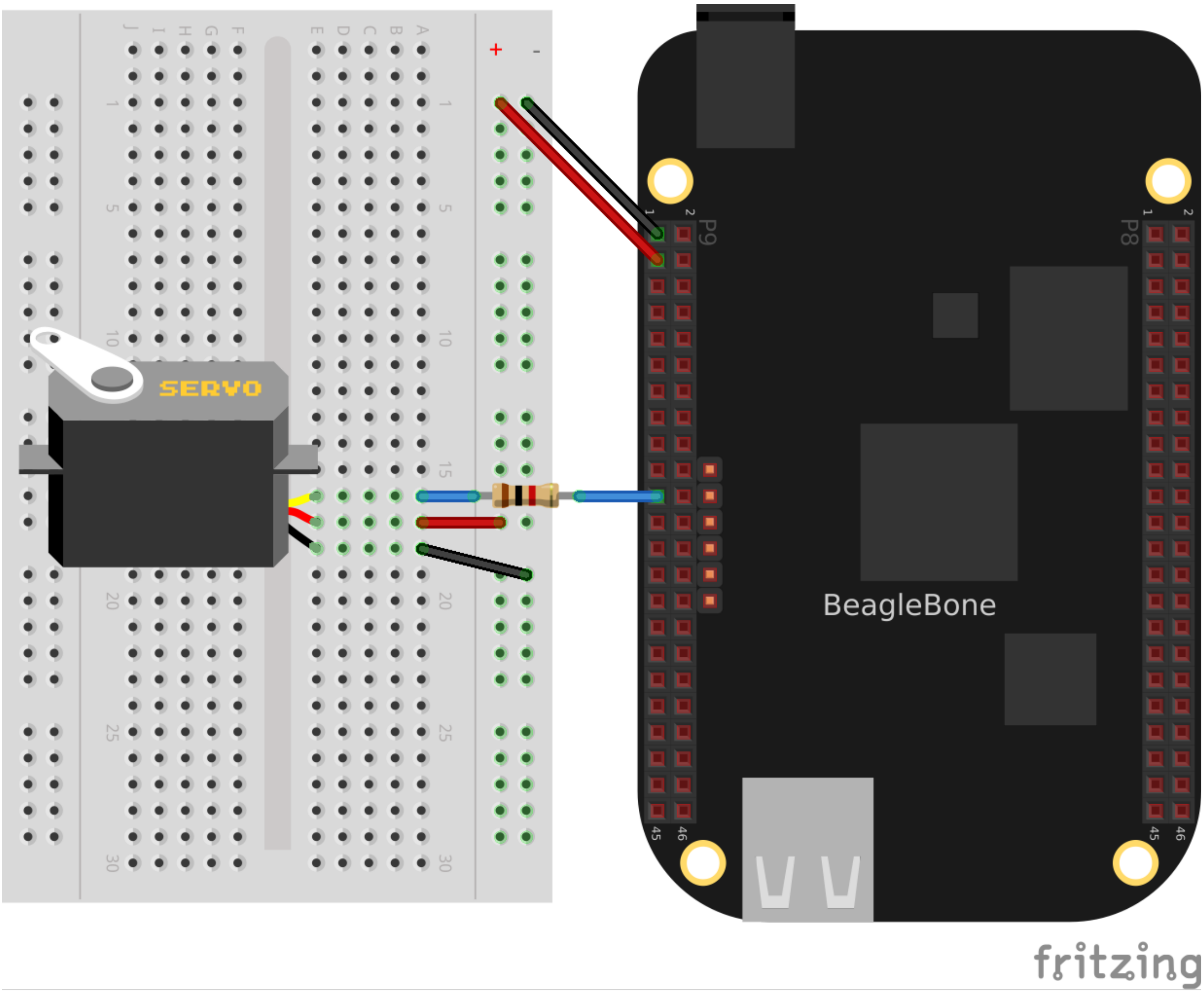
- Servo motor
- Breadboard and jumper wires
- 1 kΩ resistor (optional)
- 5 V power supply (optional)

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.

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). You need to configure the pin for PWM.

```
<pre data-type="programlisting">
bone$ <strong>cd ~/BoneCookbook/docs/04motors/code</strong>
bone$ <strong>config-pin P9_16 pwm</strong>
bone$ <strong>./servoMotor.py</strong>
</pre>
```

Example 27. Code for driving a servo motor (servoMotor.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    servoMotor.py
# //    Drive a simple servo motor back and forth on P9_16 pin
# //    Wiring:
# //    Setup:  config-pin P9_16 pwm
# //    See:
# ////////////////////////////////////////
import time
import signal
import sys
```

```
pwmPeriod = '20000000'    # Period in ns, (20 ms)
pwm =     '1' # pwm to use
channel = 'b' # channel to use
PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel
low  = 0.8 # Smallest angle (in ms)
hi   = 2.4 # Largest angle (in ms)
ms   = 250 # How often to change position, in ms
pos  = 1.5 # Current position, about middle ms)
step = 0.1 # Step size to next position

def signal_handler(sig, frame):
    print('Got SIGINT, turning motor off')
    f = open(PWMPATH+'/enable', 'w')
    f.write('0')
    f.close()
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)
print('Hit ^C to stop')

f = open(PWMPATH+'/period', 'w')
f.write(pwmPeriod)
f.close()
f = open(PWMPATH+'/enable', 'w')
f.write('1')
f.close()

f = open(PWMPATH+'/duty_cycle', 'w')
while True:
    pos += step    # Take a step
    if(pos > hi or pos < low):
        step *= -1
    duty_cycle = str(round(pos*1000000))    # Convert ms to ns
    # print('pos = ' + str(pos) + ' duty_cycle = ' + duty_cycle)
    f.seek(0)
    f.write(duty_cycle)
    time.sleep(ms/1000)

# | 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
```

Example 28. Code for driving a servo motor (servoMotor.js)

```
#!/usr/bin/env node
```

```
////////////////////////////////////////
//  servoMotor.js
//  Drive a simple servo motor back and forth on P9_16 pin
//  Wiring:
//  Setup:  config-pin P9_16 pwm
//  See:
////////////////////////////////////////
const fs = require("fs");

const pwmPeriod = '20000000';    // Period in ns, (20 ms)
const pwm     = '1';  // pwm to use
const channel = 'b';  // channel to use
const PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel;
const low  = 0.8, // Smallest angle (in ms)
      hi   = 2.4, // Largest angle (in ms)
      ms  = 250;  // How often to change position, in ms
var   pos = 1.5,  // Current position, about middle ms)
      step = 0.1; // Step size to next position

console.log('Hit ^C to stop');
fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
fs.writeFileSync(PWMPATH+'/enable', '1');

var timer = setInterval(sweep, ms);

// Sweep from low to hi position and back again
function sweep() {
    pos += step;    // Take a step
    if(pos > hi || pos < low) {
        step *= -1;
    }
    var dutyCycle = parseInt(pos*1000000);    // Convert ms to ns
    // console.log('pos = ' + pos + ' duty cycle = ' + dutyCycle);
    fs.writeFileSync(PWMPATH+'/duty_cycle', dutyCycle);
}

process.on('SIGINT', function() {
    console.log('Got SIGINT, turning motor off');
    clearInterval(timer);             // Stop the timer
    fs.writeFileSync(PWMPATH+'/enable', '0');
});

// | 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
```

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.

Discussion

Fading an External LED gies details on how to use a PWM.

Controlling a Servo with an Rotary Encoder

Problem

You have a rotary encoder from Reading a rotary encoder (rotaryEncoder.js) that you want to control a servo motor.

Solution

Combine the code from Reading a rotary encoder (rotaryEncoder.js) and Controlling a Servo Motor.

```
<pre data-type="programlisting">
bone$ <strong>config-pin P9_16 pwm</strong>
bone$ <strong>config-pin P8_11 eqep</strong>
bone$ <strong>config-pin P8_12 eqep</strong>
bone$ <strong>./servoEncoder.py</strong>
</pre>
```

Example 29. Code for driving a servo motor with a rotary encorder(servoEncoder.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    servoEncoder.py
# //    Drive a simple servo motor using rotary encoder viq eQEP
# //    Wiring: Servo on P9_16, rotary encoder on P8_11 and P8_12
# //    Setup:  config-pin P9_16 pwm
# //            config-pin P8_11 eqep
# //            config-pin P8_12 eqep
# //    See:
# ////////////////////////////////////////
import time
import signal
import sys

# Set up encoder
eQEP = '2'
COUNTERPATH = '/dev/bone/counter/counter'+eQEP+'/count0'
maxCount = '180'

ms = 100    # Time between samples in ms

# Set the eEQP maximum count
```

```
fQEP = open(COUNTERPATH+'/ceiling', 'w')
fQEP.write(maxCount)
fQEP.close()

# Enable
fQEP = open(COUNTERPATH+'/enable', 'w')
fQEP.write('1')
fQEP.close()

fQEP = open(COUNTERPATH+'/count', 'r')

# Set up servo
pwmPeriod = '20000000'    # Period in ns, (20 ms)
pwm     = '1'  # pwm to use
channel = 'b'  # channel to use
PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel
low  = 0.6 # Smallest angle (in ms)
hi   = 2.5 # Largest angle (in ms)
ms   = 250 # How often to change position, in ms
pos  = 1.5 # Current position, about middle ms)
step = 0.1 # Step size to next position

def signal_handler(sig, frame):
    print('Got SIGINT, turning motor off')
    f = open(PWMPATH+'/enable', 'w')
    f.write('0')
    f.close()
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)

f = open(PWMPATH+'/period', 'w')
f.write(pwmPeriod)
f.close()
f = open(PWMPATH+'/duty_cycle', 'w')
f.write(str(round(int(pwmPeriod)/2)))
f.close()
f = open(PWMPATH+'/enable', 'w')
f.write('1')
f.close()

print('Hit ^C to stop')

olddata = -1
while True:
    fQEP.seek(0)
    data = fQEP.read()[:-1]
    # Print only if data changes
    if data != olddata:
        olddata = data
        # print("data = " + data)
        # # map 0-180  to low-hi
```

```
        duty_cycle = -1*int(data)*(hi-low)/180.0 + hi
        duty_cycle = str(int(duty_cycle*1000000))   # Convert from ms to ns
        # print('duty_cycle = ' + duty_cycle)
        f = open(PWMPATH+'/duty_cycle', 'w')
        f.write(duty_cycle)
        f.close()
    time.sleep(ms/1000)

# Black OR Pocket
# 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

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

# | 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
```

Controlling the Speed of a DC Motor

Problem

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

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 ma) 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:

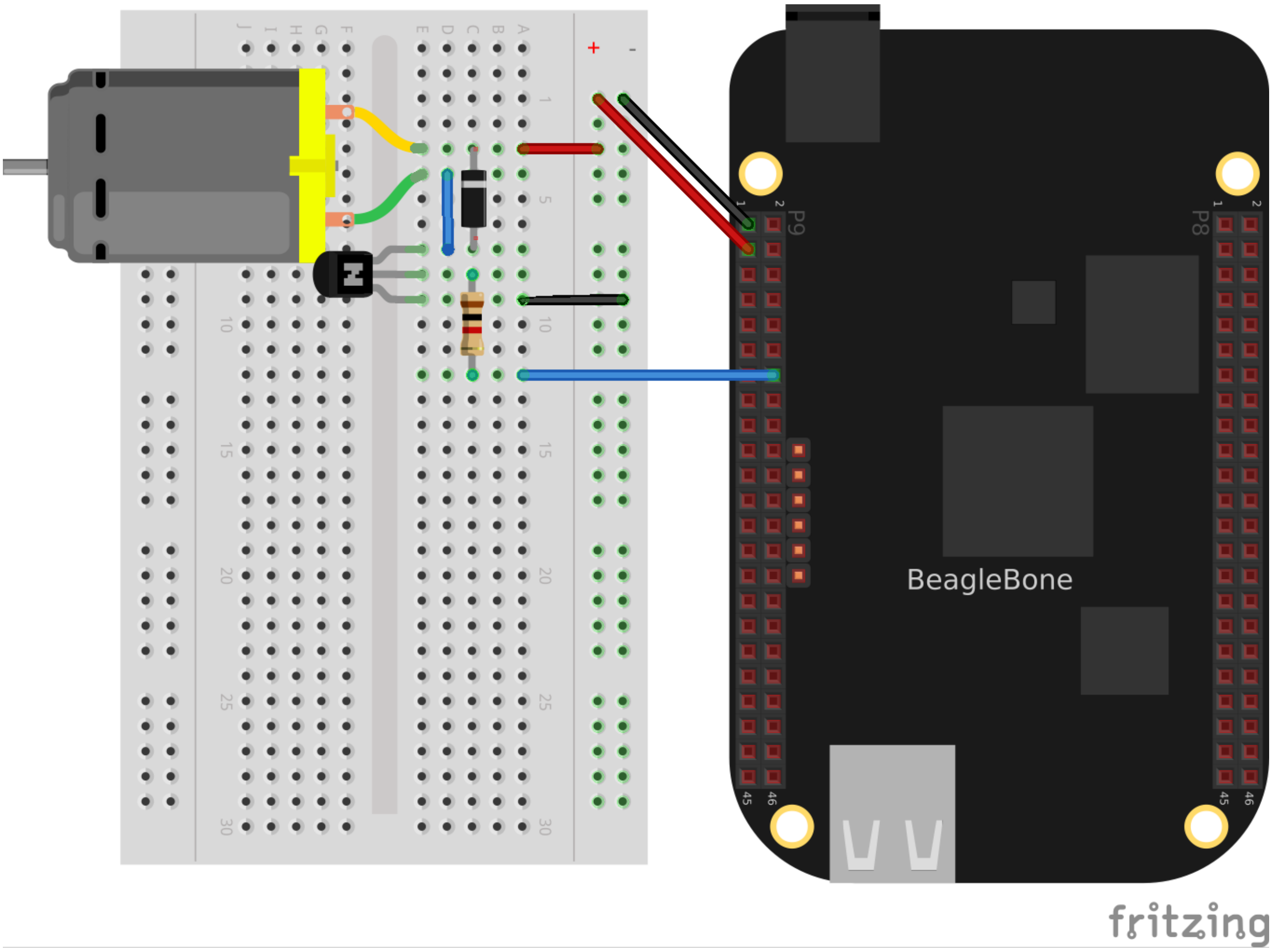
- 3 V to 5 V DC motor
- Breadboard and jumper wires

- 1 kΩ 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.

Wiring a DC motor to spin one direction



Use the code in Driving a DC motor in one direction (dcMotor.js) (*dcMotor.js*) to run the motor.

Example 30. Driving a DC motor in one direction (dcMotor.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    dcMotor.js
# //    This is an example of driving a DC motor
# //    Wiring:
# //    Setup:  config-pin P9_16 pwm
# //    See:
# ////////////////////////////////////////
import time
```

```
import signal
import sys

def signal_handler(sig, frame):
    print('Got SIGINT, turning motor off')
    f = open(PWMPATH+'/enable', 'w')
    f.write('0')
    f.close()
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)

pwmPeriod = '1000000'    # Period in ns
pwm     = '1'  # pwm to use
channel = 'b'  # channel to use
PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel

low = 0.05     # Slowest speed (duty cycle)
hi  = 1        # Fastest (always on)
ms = 100       # How often to change speed, in ms
speed = 0.5    # Current speed
step = 0.05    # Change in speed

f = open(PWMPATH+'/duty_cycle', 'w')
f.write('0')
f.close()
f = open(PWMPATH+'/period', 'w')
f.write(pwmPeriod)
f.close()
f = open(PWMPATH+'/enable', 'w')
f.write('1')
f.close()

f = open(PWMPATH+'/duty_cycle', 'w')
while True:
    speed += step
    if(speed > hi or speed < low):
        step *= -1
    duty_cycle = str(round(speed*1000000))    # Convert ms to ns
    f.seek(0)
    f.write(duty_cycle)
    time.sleep(ms/1000)
```

Example 31. Driving a DC motor in one direction (dcMotor.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  dcMotor.js
//  This is an example of driving a DC motor
//  Wiring:
```

```
//  Setup:  config-pin P9_16 pwm
//  See:
////////////////////////////////////////
const fs = require("fs");

const pwmPeriod = '1000000';    // Period in ns
const pwm     = '1';  // pwm to use
const channel = 'b';  // channel to use
const PWMPATH='/dev/bone/pwm/'+pwm+'/'+channel;

const low = 0.05,     // Slowest speed (duty cycle)
      hi  = 1,        // Fastest (always on)
      ms = 100;       // How often to change speed, in ms
var   speed = 0.5,    // Current speed;
      step = 0.05;    // Change in speed

// fs.writeFileSync(PWMPATH+'/export', pwm);   // Export the pwm channel
// Set the period in ns, first 0 duty_cycle,
fs.writeFileSync(PWMPATH+'/duty_cycle', '0');
fs.writeFileSync(PWMPATH+'/period', pwmPeriod);
fs.writeFileSync(PWMPATH+'/duty_cycle', pwmPeriod/2);
fs.writeFileSync(PWMPATH+'/enable', '1');

timer = setInterval(sweep, ms);

function sweep() {
    speed += step;
    if(speed > hi || speed < low) {
        step *= -1;
    }
    fs.writeFileSync(PWMPATH+'/duty_cycle', parseInt(pwmPeriod*speed));
    // console.log('speed = ' + speed);
}

process.on('SIGINT', function() {
    console.log('Got SIGINT, turning motor off');
    clearInterval(timer);       // Stop the timer
    fs.writeFileSync(PWMPATH+'/enable', '0');
});
```

See Also

How do you change the direction of the motor? See Controlling the Speed and Direction of a DC Motor.

Controlling the Speed and Direction of a DC Motor

Problem

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

Solution

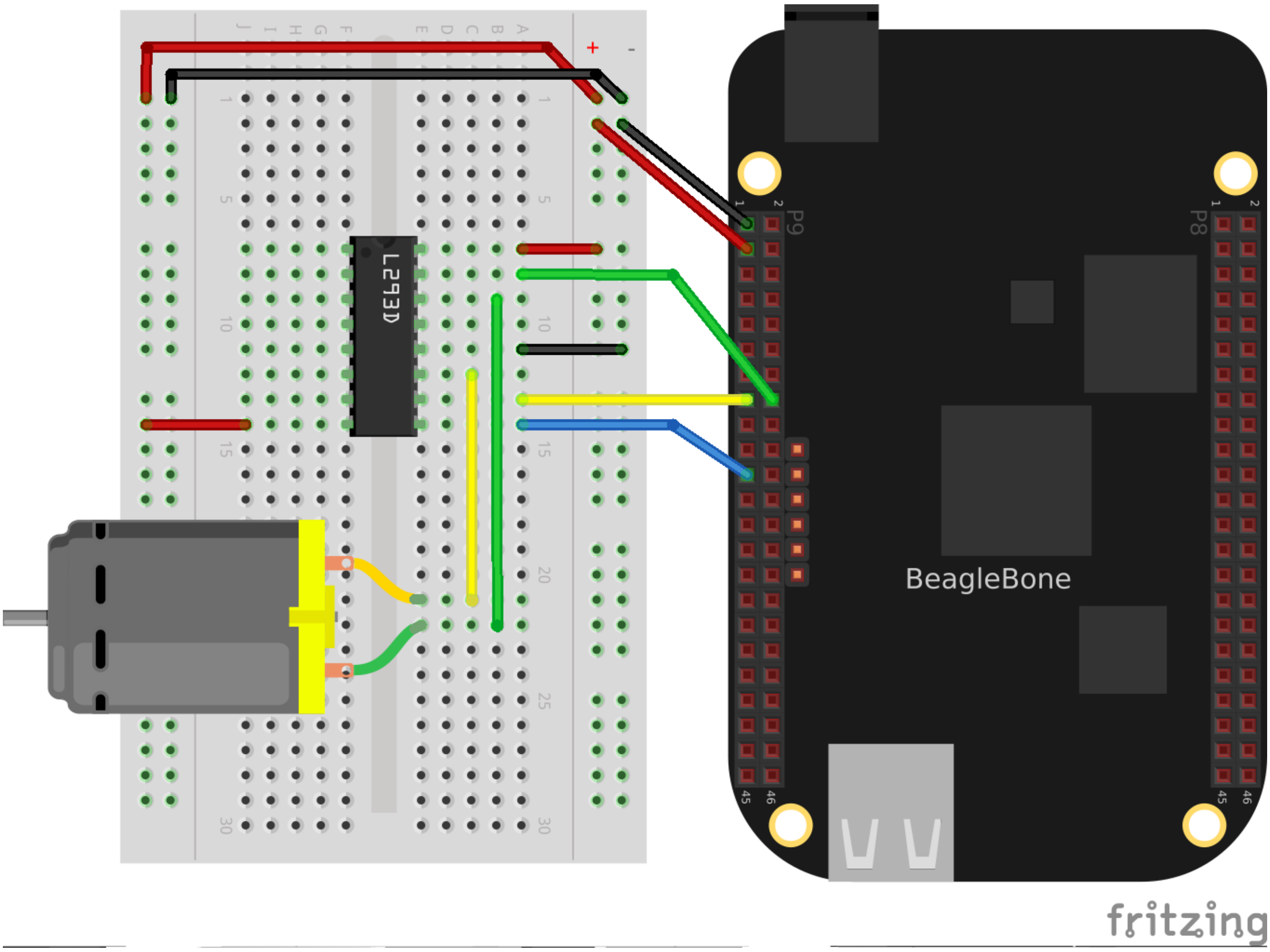
Use an H-bridge to switch the terminals on the motor so that it will run both backward and forward. We'll use the *L293D*: a common, single-chip H-bridge.

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.

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.

Example 32. Code for driving a DC motor with an H-bridge (h-bridgeMotor.js)

```
#!/usr/bin/env node

// This example uses an H-bridge to drive a DC motor in two directions

var b = require('bonescript');

var enable = 'P9_21';    // Pin to use for PWM speed control
    in1    = 'P9_15',
    in2    = 'P9_16',
    step = 0.05,    // Change in speed
    min  = 0.05,    // Min duty cycle
    max  = 1.0,     // Max duty cycle
    ms   = 100,     // Update time, in ms
    speed = min;    // Current speed;

b.pinMode(enable, b.ANALOG_OUTPUT, 6, 0, 0, doInterval);
b.pinMode(in1, b.OUTPUT);
b.pinMode(in2, b.OUTPUT);

function doInterval(x) {
    if(x.err) {
        console.log('x.err = ' + x.err);
        return;
    }
    timer = setInterval(sweep, ms);
}

clockwise();        // Start by going clockwise

function sweep() {
    speed += step;
    if(speed > max || speed < min) {
        step *= -1;
        step>0 ? clockwise() : counterClockwise();
    }
    b.analogWrite(enable, speed);
    console.log('speed = ' + speed);
}

function clockwise() {
    b.digitalWrite(in1, b.HIGH);
    b.digitalWrite(in2, b.LOW);
}

function counterClockwise() {
    b.digitalWrite(in1, b.LOW);
    b.digitalWrite(in2, b.HIGH);
}
```

```
process.on('SIGINT', function() {
    console.log('Got SIGINT, turning motor off');
    clearInterval(timer);         // Stop the timer
    b.analogWrite(enable, 0);     // Turn motor off
});
```

Driving a Bipolar Stepper Motor

Problem

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

Solution

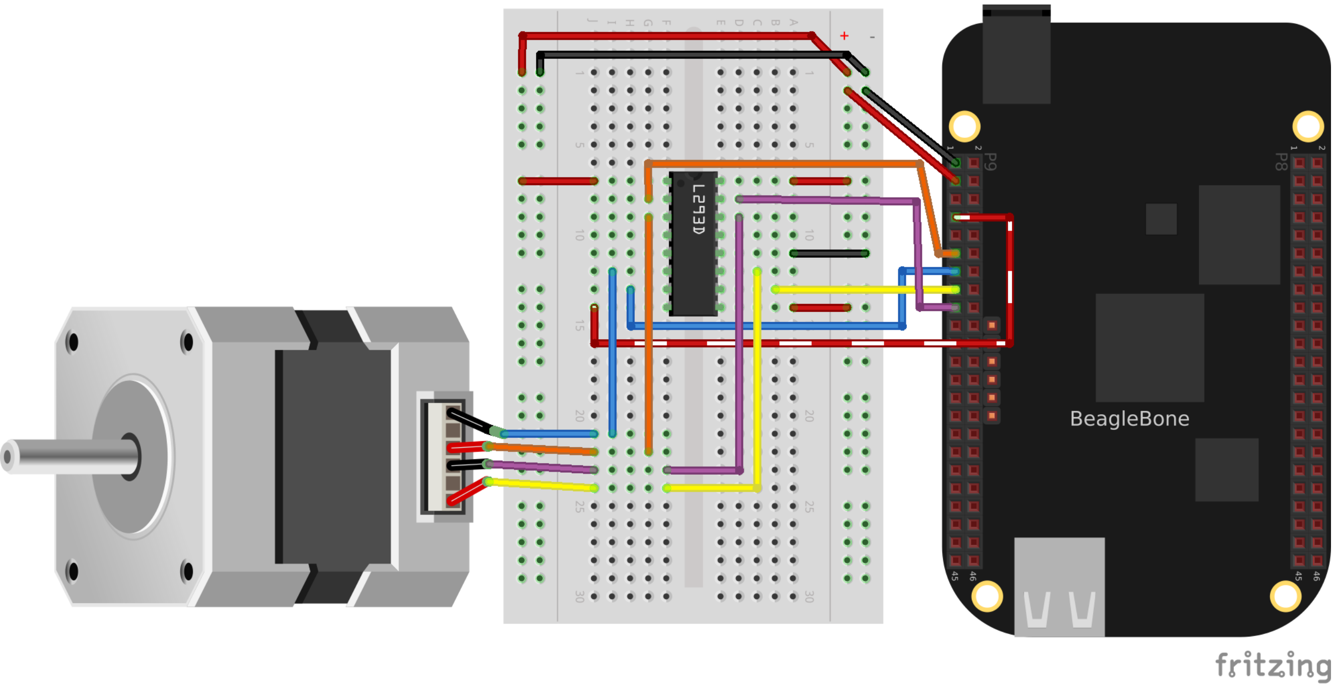
Use an L293D H-bridge. The bipolar stepper motor requires us to reverse the coils, so we need to use an H-bridge.

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.

Bipolar stepper motor wiring



Use the code in [motors_stepperMotor_code] to drive the motor.

Example 33. Driving a bipolar stepper motor (bipolarStepperMotor.py)

```
#!/usr/bin/env python
import time
import os
import signal
import sys

# Motor is attached here
# controller = ["P9_11", "P9_13", "P9_15", "P9_17"];
# controller = ["30", "31", "48", "5"]
# controller = ["P9_14", "P9_16", "P9_18", "P9_22"];
controller = ["50", "51", "4", "2"]
states = [[1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]]
statesHiTorque = [[1,1,0,0], [0,1,1,0], [0,0,1,1], [1,0,0,1]]
statesHalfStep = [[1,0,0,0], [1,1,0,0], [0,1,0,0], [0,1,1,0],
                      [0,0,1,0], [0,0,1,1], [0,0,0,1], [1,0,0,1]]

curState = 0    # Current state
ms = 100        # Time between steps, in ms
maxStep = 22    # Number of steps to turn before turning around
minStep = 0     # minimum step to turn back around on

CW  =  1       # Clockwise
CCW = -1
pos =  0       # current position and direction
direction = CW
GPIOPATH="/sys/class/gpio"

def signal_handler(sig, frame):
    print('Got SIGINT, turning motor off')
    for i in range(len(controller)) :
        f = open(GPIOPATH+"/gpio"+controller[i]+"/value", "w")
        f.write('0')
        f.close()
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)
print('Hit ^C to stop')

def move():
    global pos
    global direction
    global minStep
    global maxStep
    pos += direction
    print("pos: " + str(pos))
    # Switch directions if at end.
    if (pos >= maxStep or pos <= minStep) :
        direction *= -1
    rotate(direction)
```

```
# This is the general rotate
def rotate(direction) :
    global curState
    global states
    # print("rotate(%d)", direction);
    # Rotate the state acording to the direction of rotation
    curState +=  direction
    if(curState >= len(states)) :
        curState = 0;
    elif(curState<0) :
        curState = len(states)-1
    updateState(states[curState])

# Write the current input state to the controller
def updateState(state) :
    global controller
    print(state)
    for i in range(len(controller)) :
        f = open(GPIOPATH+"/gpio"+controller[i]+"/value", "w")
        f.write(str(state[i]))
        f.close()

# Initialize motor control pins to be OUTPUTs
for i in range(len(controller)) :
    # Make sure pin is exported
    if (not os.path.exists(GPIOPATH+"/gpio"+controller[i])):
        f = open(GPIOPATH+"/export", "w")
        f.write(pin)
        f.close()
    # Make it an output pin
    f = open(GPIOPATH+"/gpio"+controller[i]+"/direction", "w")
    f.write("out")
    f.close()

# Put the motor into a known state
updateState(states[0])
rotate(direction)

# Rotate
while True:
    move()
    time.sleep(ms/1000)
```

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

Driving a Unipolar Stepper Motor

Problem

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

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

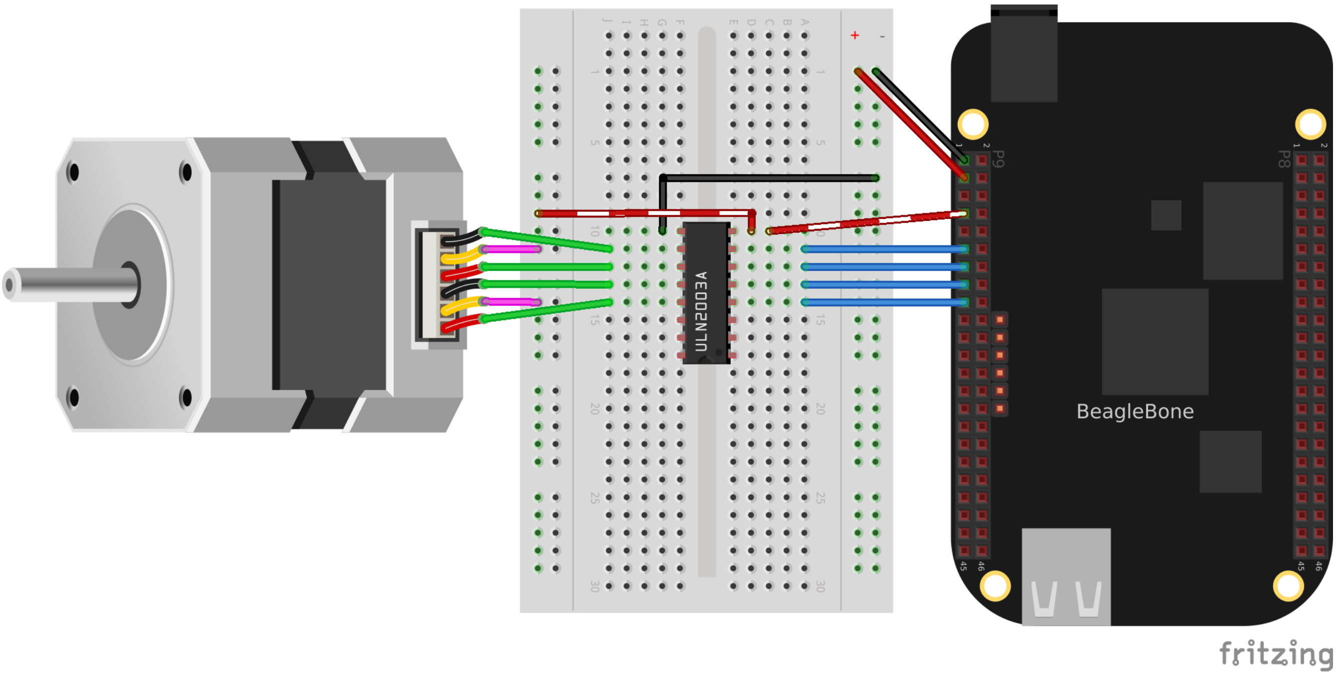
Wire, as shown in Unipolar stepper motor wiring.

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.

Unipolar stepper motor wiring



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

Example 34. Changes to bipolar code to drive a unipolar stepper motor (unipolarStepperMotor.py.diff)

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

Example 35. Changes to bipolar code to drive a unipolar stepper motor (unipolarStepperMotor.js.diff)

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

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.

Beyond the Basics

Introduction

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 ([tips_pick_os]) and are comfortable with using Linux. We continue to assume that you are logged in as debian on your Bone.

Running Your Bone Standalone

Problem

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

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 ([app_misc]) and use your Bone with a DVI-D display.

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

Figure 30. 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.

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.

TIP	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.

Discussion

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.

Getting to the Command Shell via SSH

Problem

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

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):

```
<pre data-type="programlisting">
host$ <strong>ssh debian@192.168.7.2</strong>
Warning: Permanently added 'bone,192.168.7.2' (ECDSA) to the list of known hosts.
Last login: Mon Dec 22 07:53:06 2014 from yoder-linux.local
bone$
</pre>
```

debian has the default password tempped It’s best to change the password:

```
<pre data-type="programlisting">
bone$ <strong>passwd</strong>
Changing password for debian.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
</pre>
```

Getting to the Command Shell via the Virtual Serial Port

Problem

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

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.)

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

```
<pre data-type="programlisting">
host$ <strong>ls -ls /dev/ttyACM0</strong>
0 crw-rw---- 1 root dialout 166, 0 Jun 19 11:47 /dev/ttyACM0
</pre>
```

/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:

```
<pre data-type="programlisting">
host$ <strong>groups</strong>
yoder adm tty uucp <strong>dialout</strong> cdrom sudo dip plugdev lpadmin sambashare
</pre>
```

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

```
<pre data-type="programlisting">
host$ <strong>sudo adduser $USER dialout</strong>
</pre>
```

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:

```
<pre data-type="programlisting">
host$ <strong>sudo apt install screen</strong>
host$ <strong>screen /dev/ttyACM0 115200</strong>
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:
</pre>
```

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.

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

Problem

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

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.

Figure 31. FTDI cable

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

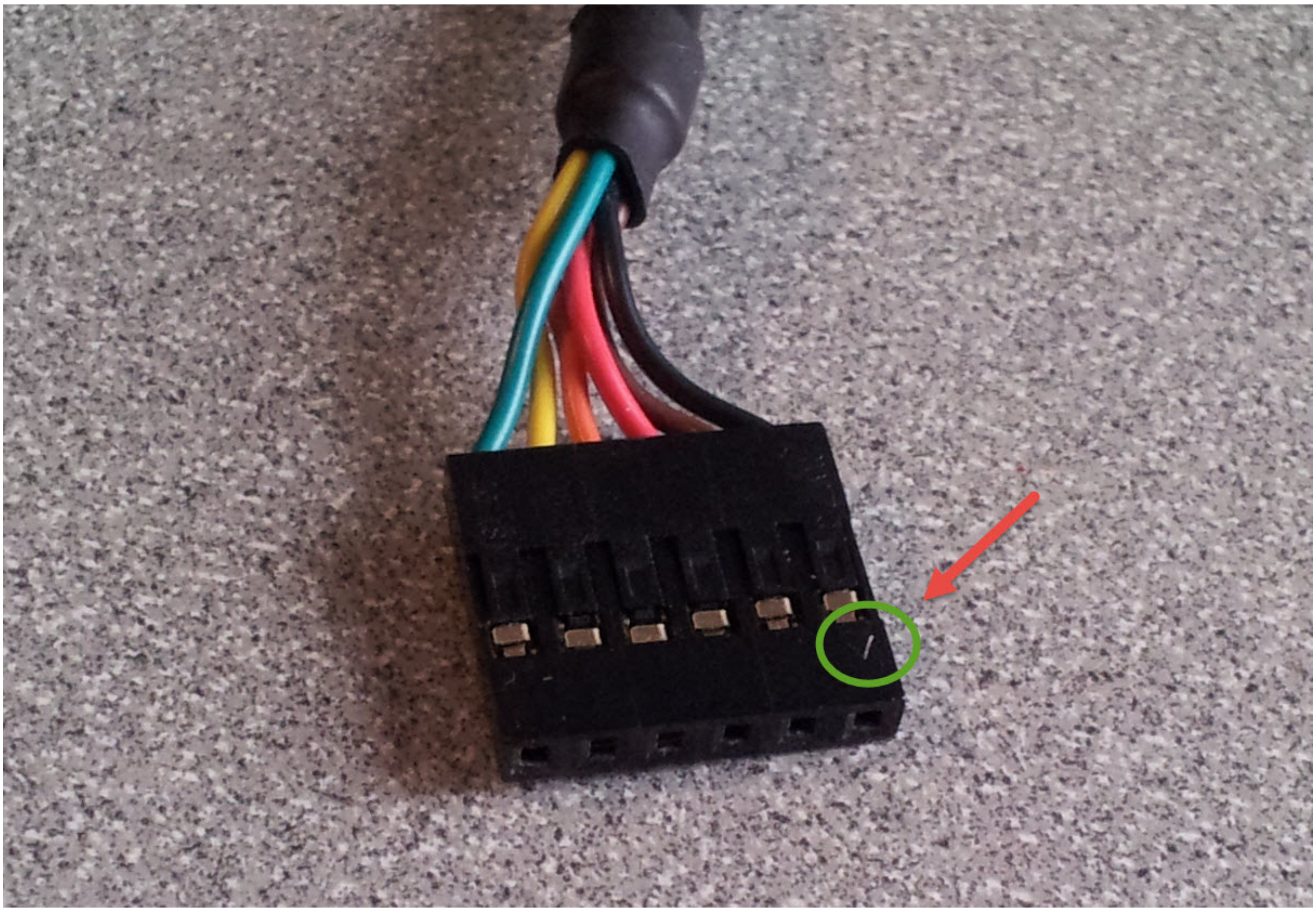


Figure 32. FTDI connector

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.

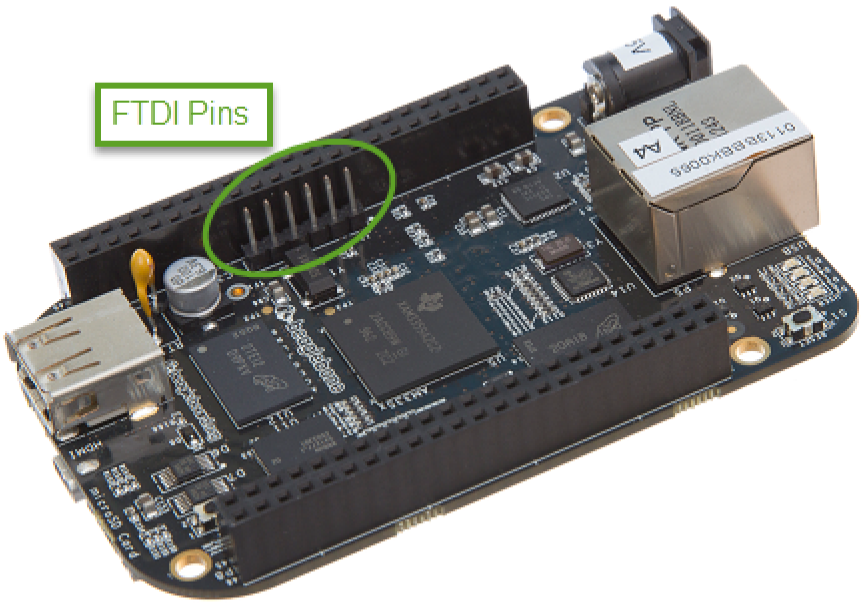


Figure 33. FTDI pins for the FTDI connector

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.

Now, run the following commands on your host computer:

```
<pre data-type="programlisting">
host$ <strong>ls -ls /dev/ttyUSB0</strong>
0 crw-rw---- 1 root dialout 188, 0 Jun 19 12:43 /dev/ttyUSB0
host$ <strong>sudo adduser $USER dialout</strong>
host$ <strong>screen /dev/ttyUSB0 115200</strong>
Debian GNU/Linux 7 beaglebone ttyO0

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:
</pre>
```

NOTE	Your screen might initially be blank. Press Enter a couple times to see the login prompt.

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

Problem

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

Solution

Log in to your Bone and enter the following command:

```
<pre data-type="programlisting">
bone$ <strong>cat /etc/dogtag</strong>
BeagleBoard.org Debian Bullseye IoT Image 2022-07-01
</pre>
```

Discussion

Verifying You Have the Latest Version of the OS on Your Bone shows how to open the *ID.txt* file to see the OS version. The */etc/dogtag* file has the same contents and is easier to find if you already have a command prompt. See Running the Latest Version of the OS on Your Bone if you need to update your OS.

Controlling the Bone Remotely with a VNC

Problem

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

Solution

Run the installed Virtual Network Computing (VNC) server:

```
<pre data-type="programlisting">
bone$ <strong>tightvncserver</strong>

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

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

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).

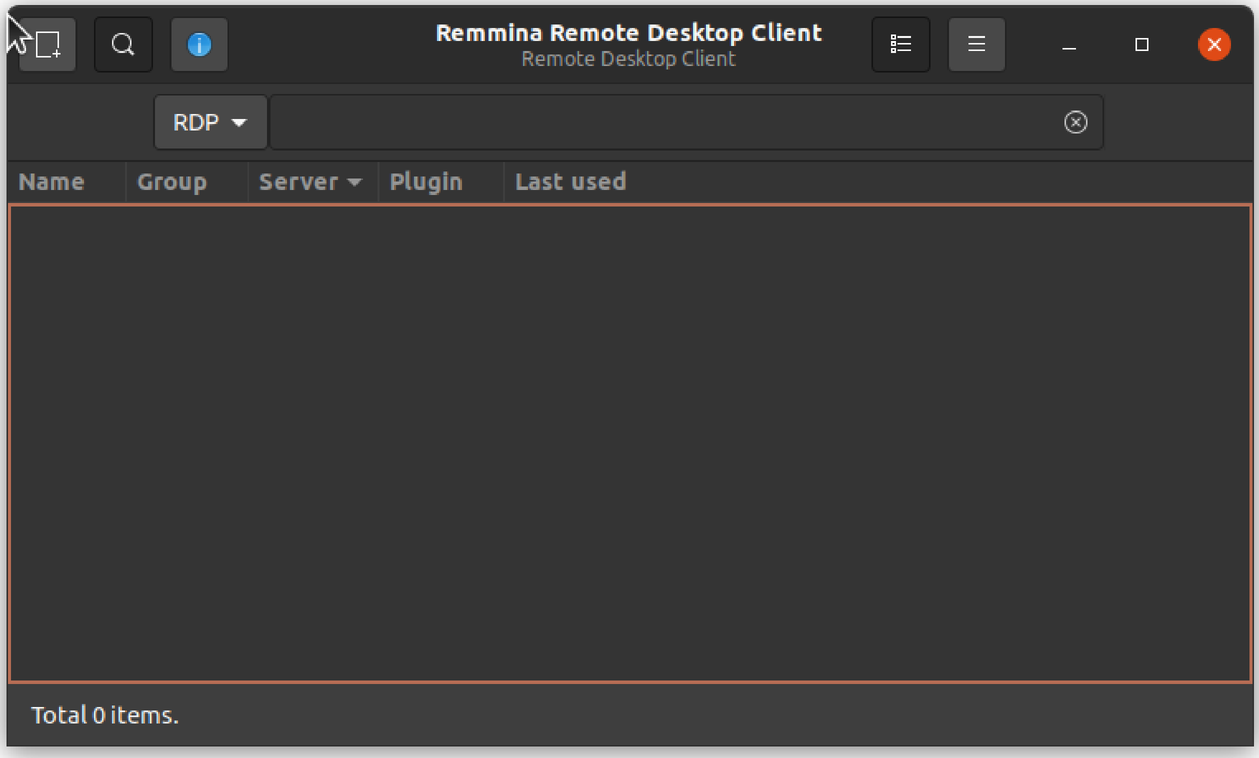


Figure 34. Creating a new remote desktop file in Remmina Remote Desktop Client

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.

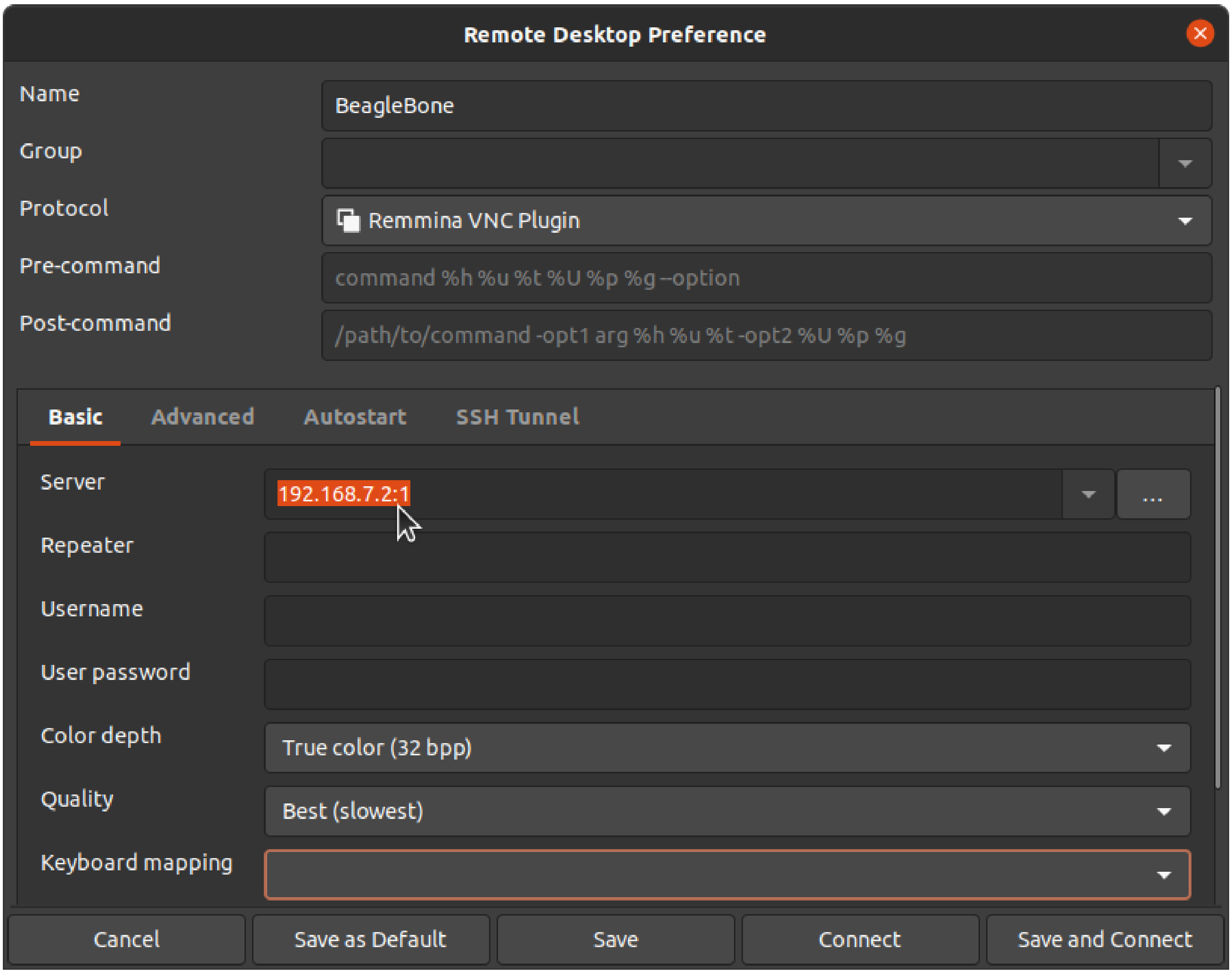


Figure 35. Configuring the Remmina Remote Desktop Client

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

Figure 36. 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.

```
<pre data-type="programlisting">
bone$ <strong>bone$ sudo apt install bbb.io-xfce4-desktop</strong>
bone$ <strong>cp /etc/bbb.io/templates/fbdev.xorg.conf
/etc/X11/xorg.conf</strong>
bone$ <strong>startxfce4</strong>
/usr/bin/startxfce4: Starting X server
/usr/bin/startxfce4: 122: exec: xinit: not found
<pre>
```

Learning Typical GNU/Linux Commands

Problem

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

Solution

Common Linux commands lists many common Linux commands.

Table 2. 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
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

Command	Action
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

Editing a Text File from the GNU/Linux Command Shell

Problem

You want to run an editor to change a file.

Solution

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

```
<pre data-type="programlisting">
bone$ <strong>nano file</strong>
</pre>
```

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.

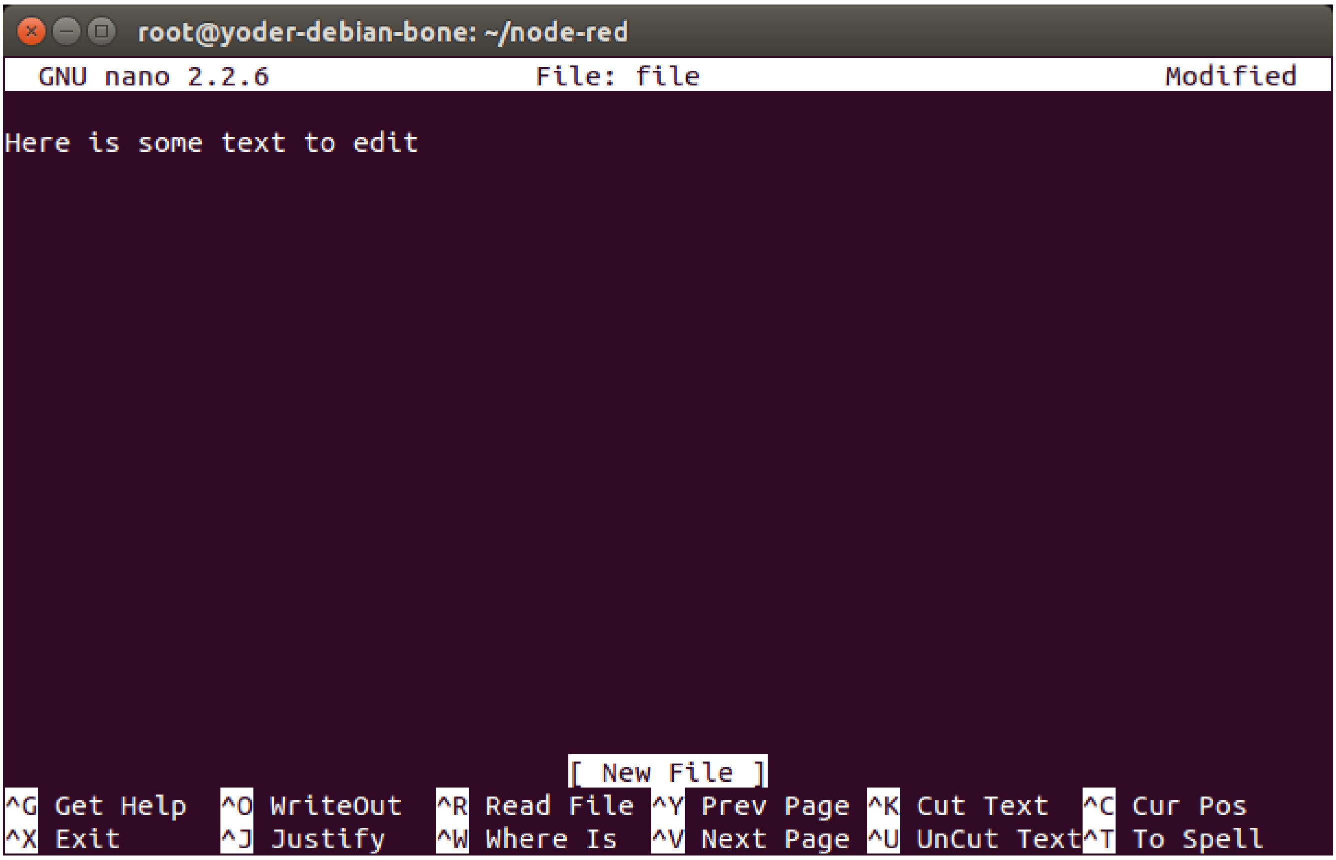


Figure 37. Editing a file with nano

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

Discussion

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.

Establishing an Ethernet-Based Internet Connection

Problem

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

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 flash.

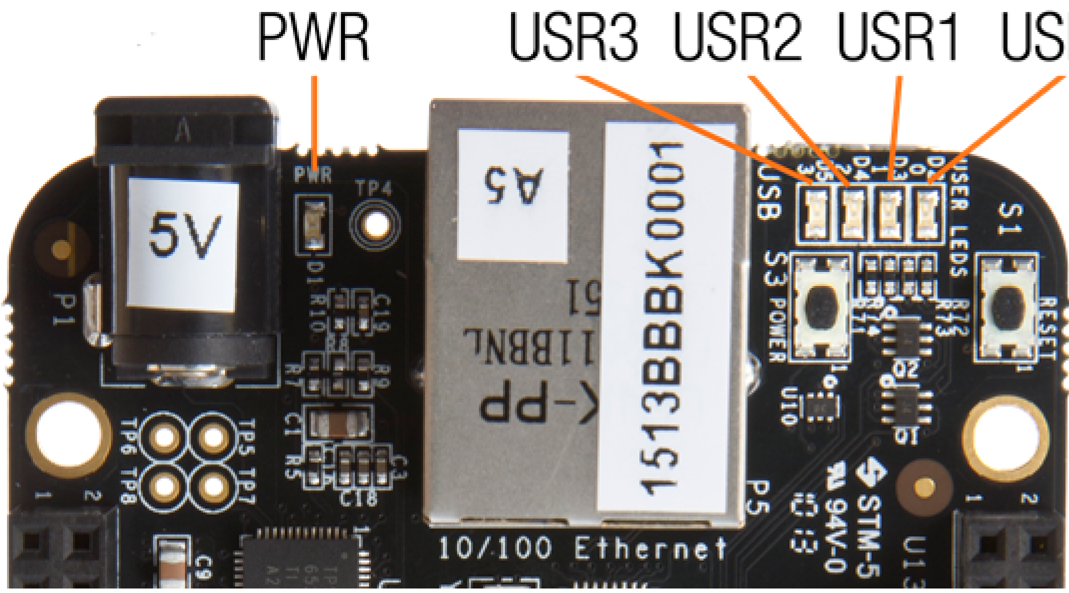


Figure 38. The RJ45 port on the Bone

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:

```
<pre data-type="programlisting">
bone$ <strong>ip a</strong>
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
       valid_lft forever preferred_lft forever
4: usb1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state 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 noop state DOWN group default qlen 10
    link/can
6: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
</pre>
```

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 [tips_passwords] and Setting Up a Firewall. On many home networks, you will be behind a firewall and won't be as visible.

Establishing a WiFi-Based Internet Connection

Problem

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

Solution

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 middle one.

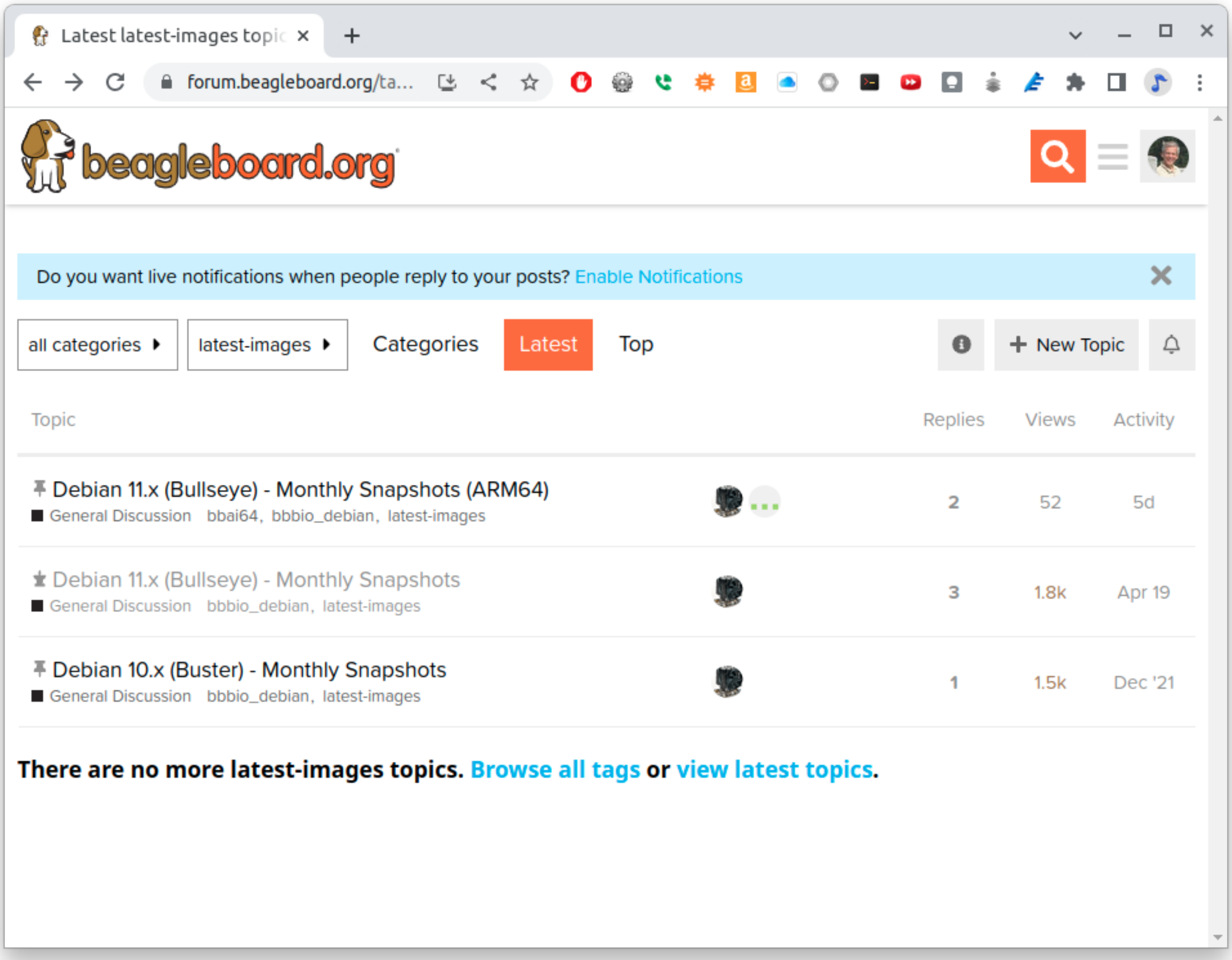


Figure 39. Latested Beagle Images

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

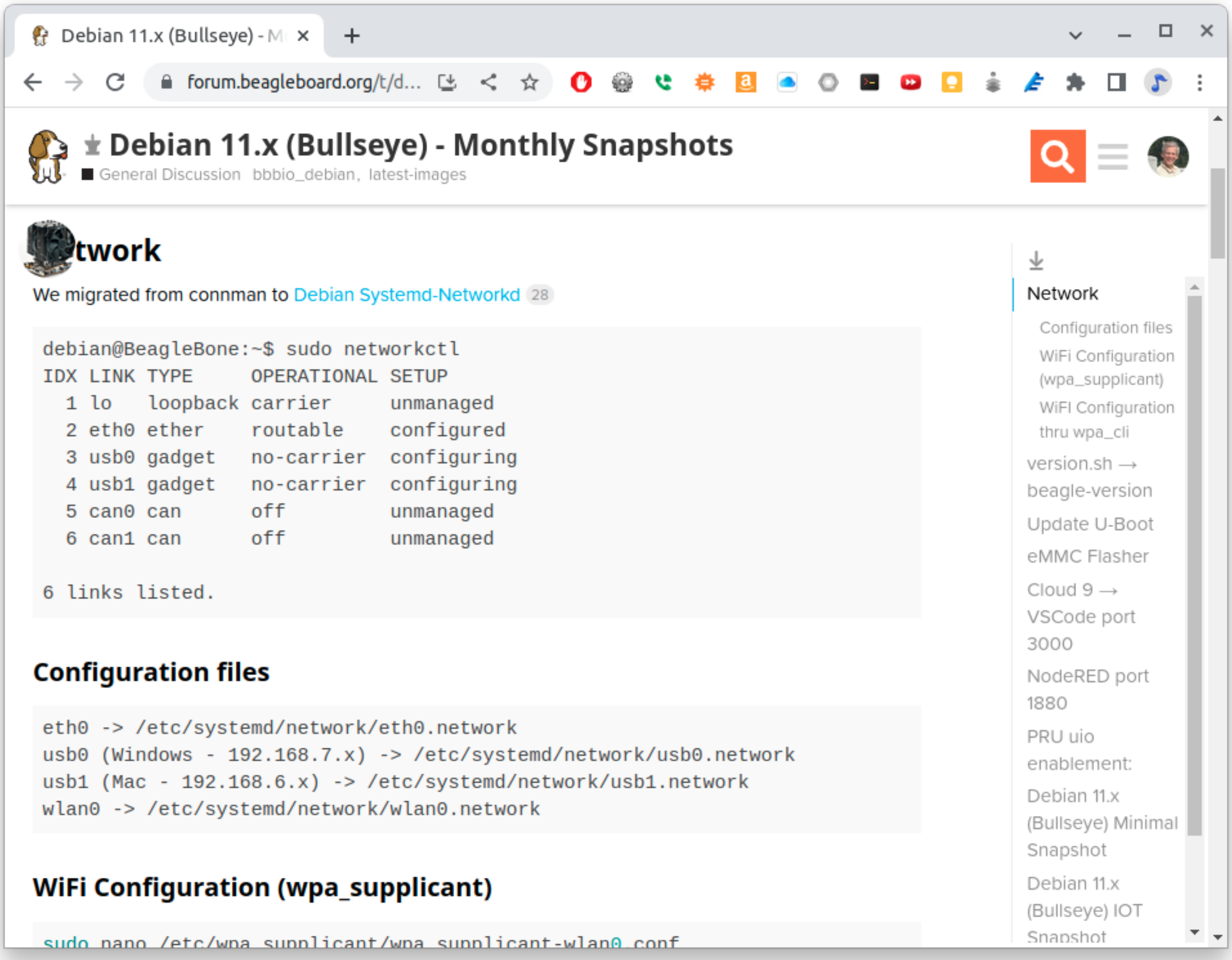


Figure 40. Instructions for setting up your network.

To make this recipe, you will need:

- USB Wifi adapter
- 5 V external power supply

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:

```
<pre data-type="programlisting">
bone$ <strong>lsusb</strong>
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
</pre>
```

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.

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

```
<pre data-type="programlisting">
bone$ <strong>networkctl</strong>
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.
</pre>
```

If no name appears, try ip a:

```
<pre data-type="programlisting">
bone$ <strong>ip a</strong>
...
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
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>sudo nano /etc/wpa_supplicant/wpa_supplicant-wlan0.conf</strong>
```

```
</pre>
```

In the file you'll see:

```
<pre data-type="programlisting">
ctrl_interface=DIR=/run/wpa_supplicant GROUP=netdev
update_config=1
#country=US

network={
    ssid="Your SSID"
    psk="Your Password"
}
</pre>
```

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

```
<pre>
bone$ <strong>sudo systemctl restart systemd-networkd</strong>
bone$ <strong> ip a</strong>
bone$ <strong>ping -c2 google.com</strong>
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
</pre>
```

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

Sharing the Host's Internet Connection over USB

Problem

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

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.

Example 36. Code for IP Masquerading (ipMasquerade.sh)

```
#!/bin/bash
# These are the commands to run on the host to set up IP
#  masquerading so the Bone can access the Internet through
#  the USB connection.
# This configures the host, run ./setDNS.sh to configure the Bone.
# Inspired by http://thoughtshubham.blogspot.com/2010/03/
#  internet-over-usb-otg-on-beagleboard.html

if [ $# -eq 0 ] ; then
echo "Usage: $0 interface (such as eth0 or wlan0)"
exit 1
fi

interface=$1
hostAddr=192.168.7.1
beagleAddr=192.168.7.2
ip_forward=/proc/sys/net/ipv4/ip_forward

if [ `cat $ip_forward` == 0 ]
  then
    echo "You need to set IP forwarding. Edit /etc/sysctl.conf using:"
    echo "$ sudo nano /etc/sysctl.conf"
    echo "and uncomment the line   \"net.ipv4.ip_forward=1\""
    echo "to enable forwarding of packets. Then run the following:"
    echo "$ sudo sysctl -p"
    exit 1
  else
    echo "IP forwarding is set on host."
fi
# Set up IP masquerading on the host so the bone can reach the outside world
sudo iptables -t nat -A POSTROUTING -s $beagleAddr -o $interface -j MASQUERADE
```

Then, on your host, run the following commands:

```
<pre data-type="programlisting">
host$ <strong>chmod +x ipMasquerade.sh</strong>
host$ <strong>./ipMasquerade.sh eth0</strong>
</pre>
```

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.

Example 37. Code for setting the DNS on the Bone (setDNS.sh)

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

hostAddr=192.168.7.1
beagleAddr=${1:-192.168.7.2}

# Save the /etc/resolv.conf on the Beagle in case we mess things up.
ssh root@$beagleAddr "mv -n /etc/resolv.conf /etc/resolv.conf.orig"
# Create our own resolv.conf
cat - << EOF > /tmp/resolv.conf
# This is installed by ./setDNS.sh on the host

EOF

TMP=/tmp/nmcli
# Look up the nameserver of the host and add it to our resolv.conf
# From: http://askubuntu.com/questions/197036/how-to-know-what-dns-am-i-using-in-
ubuntu-12-04
# Use nmcli dev list for older version nmcli
# Use nmcli dev show for newer version nmcli
nmcli dev show > $TMP
if [ $? -ne 0 ]; then   # $? is the return code, if not 0 something bad happened.
    echo "nmcli failed, trying older 'list' instead of 'show'"
    nmcli dev list > $TMP
    if [ $? -ne 0 ]; then
        echo "nmcli failed again, giving up..."
        exit 1
    fi
fi

grep IP4.DNS $TMP | sed 's/IP4.DNS\[.\]:/nameserver/' >> /tmp/resolv.conf

scp /tmp/resolv.conf root@$beagleAddr:/etc

# Tell the beagle to use the host as the gateway.
ssh root@$beagleAddr "/sbin/route add default gw $hostAddr" || true
```

Then, on your host, run the following commands:

```
<pre data-type="programlisting">
host$ <strong>chmod +x setDNS.sh</strong>
host$ <strong>./setDNS.sh</strong>
host$ <strong>ssh -X root@192.168.7.2</strong>
bone$ <strong>ping -c2 google.com</strong>
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
</pre>
```

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:

```
<pre data-type="programlisting">
host$ <strong>ip a</strong>
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: enp0s25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    link/ether 54:ee:75:19:8d:53 brd ff:ff:ff:ff:ff:ff
    inet 10.0.5.117/24 brd 10.0.5.255 scope global dynamic noprefixroute enp0s25
       valid_lft 82690sec preferred_lft 82690sec
    inet6 fe80::e6d0:8d85:4488:6b5f/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: wlp3s0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
...
</pre>
```

It’s the number following inet, which in my case is 10.0.5.117.

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:

```
<pre data-type="programlisting">
```

```
host$ <strong>sudo iptables -t nat -A PREROUTING -p tcp -s 0/0 \
     -d 10.0.5.117 --dport 1080 -j DNAT --to 192.168.7.2:80</strong>
</pre>
```

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.)

Setting Up a Firewall

Problem

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

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install ufw</strong>
bone$ <strong>sudo ufw status</strong>
Status: inactive
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>sudo ufw default deny incoming</strong>
bone$ <strong>sudo ufw default allow outgoing</strong>
</pre>
```

Don’t enable yet, make sure ssh still has access

```
<pre data-type="programlisting">
bone$ <strong>sudo ufw allow 22</strong>
</pre>
```

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

```
<pre data-type="programlisting">
host$ <strong>sudo apt update</strong>
host$ <strong>sudo apt install nmap</strong>
host$ <strong>nmap 192.168.7.2</strong>
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
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>sudo ufw enable</strong>
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is active and enabled on system startup


host$ <strong>nmap 192.168.7.2</strong>
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
</pre>
```

Only port 22 (ssh) is accessable now.

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

```
<pre data-type="programlisting">
bone$ <strong>sudo ufw disable</strong>
Firewall stopped and disabled on system startup
</pre>
```

See the How-To Geek article for more examples.

Installing Additional Packages from the Debian Package Feed

Problem

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

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt update</strong>
bone$ <strong>sudo apt install "name of software"</strong>
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>apt-cache pkgnames | sort > /tmp/list</strong>
bone$ <strong>wc /tmp/list</strong>
  67303   67303 1348342 /tmp/list
bone$ <strong>less /tmp/list</strong>
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install dict</strong>
</pre>
```

Now you can run dict.

Removing Packages Installed with apt

Problem

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

Solution

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

```
<pre data-type="programlisting">
```

```
bone$ <strong>sudo apt remove dict</strong>
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
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]? <strong>y</strong>
</pre>
```

Copying Files Between the Onboard Flash and the MicroSD Card

Problem

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

Solution

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

```
<pre data-type="programlisting">
bone$ <strong>df -h</strong>
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$ <strong>ls /dev/mmcblk*</strong>
/dev/mmcblk0    /dev/mmcblk0p2  /dev/mmcblk1boot0  /dev/mmcblk1p1
/dev/mmcblk0p1  /dev/mmcblk1    /dev/mmcblk1boot1
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>cd /mnt</strong>
bone$ <strong>sudo mkdir onboard</strong>
bone$ <strong>ls onboard</strong>
bone$ <strong>sudo mount /dev/mmcblk1p1 onboard/</strong>
bone$ <strong>ls onboard</strong>
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
</pre>
```

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.

Discussion

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo umount /mnt/onboard</strong>
</pre>
```

Freeing Space on the Onboard Flash or MicroSD Card

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.

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt remove bb-node-red-installer</strong> (171M)
bone$ <strong>sudo apt autoremove</strong>
bone$ <strong>sudo -rf /usr/share/doc</strong>               (116M)
bone$ <strong>sudo -rf /usr/share/man</strong>               (19M)
</pre>
```

Discovering big files

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

```
<pre data-type="programlisting">
bone$ <strong>sudo du -shx /*</strong>
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
</pre>
```

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*:

```
<pre data-type="programlisting">
bone$ <strong>sudo du -sh /usr/*</strong>
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
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install ncdu</strong>
bone$ <strong>ncdu /</strong>
</pre>
```

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.

Using C to Interact with the Physical World

Problem

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

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.

Example 38. Use C to blink an LED (blinkLED.c)

```
////////////////////////////////////////
//  blinkLED.c
//  Blinks the P9_14 pin
//  Wiring:
//  Setup:
//  See:
////////////////////////////////////////
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define MAXSTR 100
// Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1, line 18 maps to
50
int main() {
  FILE *fp;
  char pin[] = "50";
  char GPIOPATH[] = "/sys/class/gpio";
  char path[MAXSTR] = "";

  // Make sure pin is exported
  snprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", pin);
  if (!access(path, F_OK) == 0) {
    snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
    fp = fopen(path, "w");
    fprintf(fp, "%s", pin);
    fclose(fp);
  }

  // Make it an output pin
  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", pin, "/direction");
  fp = fopen(path, "w");
```

```
  fprintf(fp, "out");
  fclose(fp);

  // Blink every .25 sec
  int state = 0;
  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", pin, "/value");
  fp = fopen(path, "w");
  while (1) {
    fseek(fp, 0, SEEK_SET);
    if (state) {
      fprintf(fp, "1");
    } else {
      fprintf(fp, "0");
    }
    state = ~state;
    usleep(250000);   // sleep time in microseconds
  }
}
```

Here, as with JavaScript and Python, the gpio pins are refered 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.

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

Figure 41. Mapping from header pin to internal GPIO number

Compile and run the code:

<pre data-type="programlisting">

```
bone$ <strong>gcc -o blinkLED blinkLED.c </strong>
bone$ <strong>./blinkLED</strong>
^C
</pre>
```

Hit ^C to stop the blinking.

Internet of Things

Introduction

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.

Accessing Your Host Computer’s Files on the Bone

Problem

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

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install sshfs</strong>
</pre>
```

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):

```
<pre data-type="programlisting">
bone$ <strong>mkdir host</strong>
bone$ <strong>sshfs username@$192.168.7.1:. host</strong>
bone$ <strong>cd host</strong>
bone$ <strong>ls</strong>
</pre>
```

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:

```
<pre data-type="programlisting">
host$ <strong>sudo apt install sshfs</strong>
</pre>
```

and then access:

```
<pre data-type="programlisting">
host$ <strong>mkdir /mnt/bone</strong>
host$ <strong>sshfs debian@$192.168.7.2:/ /mnt/bone</strong>
host$ <strong>cd /mnt/bone</strong>
host$ <strong>ls</strong>
</pre>
```

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.

Discussion

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:

```
<pre data-type="programlisting">
host$ <strong>umount /mnt/bone</strong>
bone$ <strong>umount home</strong>
</pre>
```

Serving Web Pages from the Bone

Problem

You want to use BeagleBone Black as a web server.

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*.

Example 39. A sample web page (test.html)

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>
```

You will see the web page shown in test.html as served by nginx.

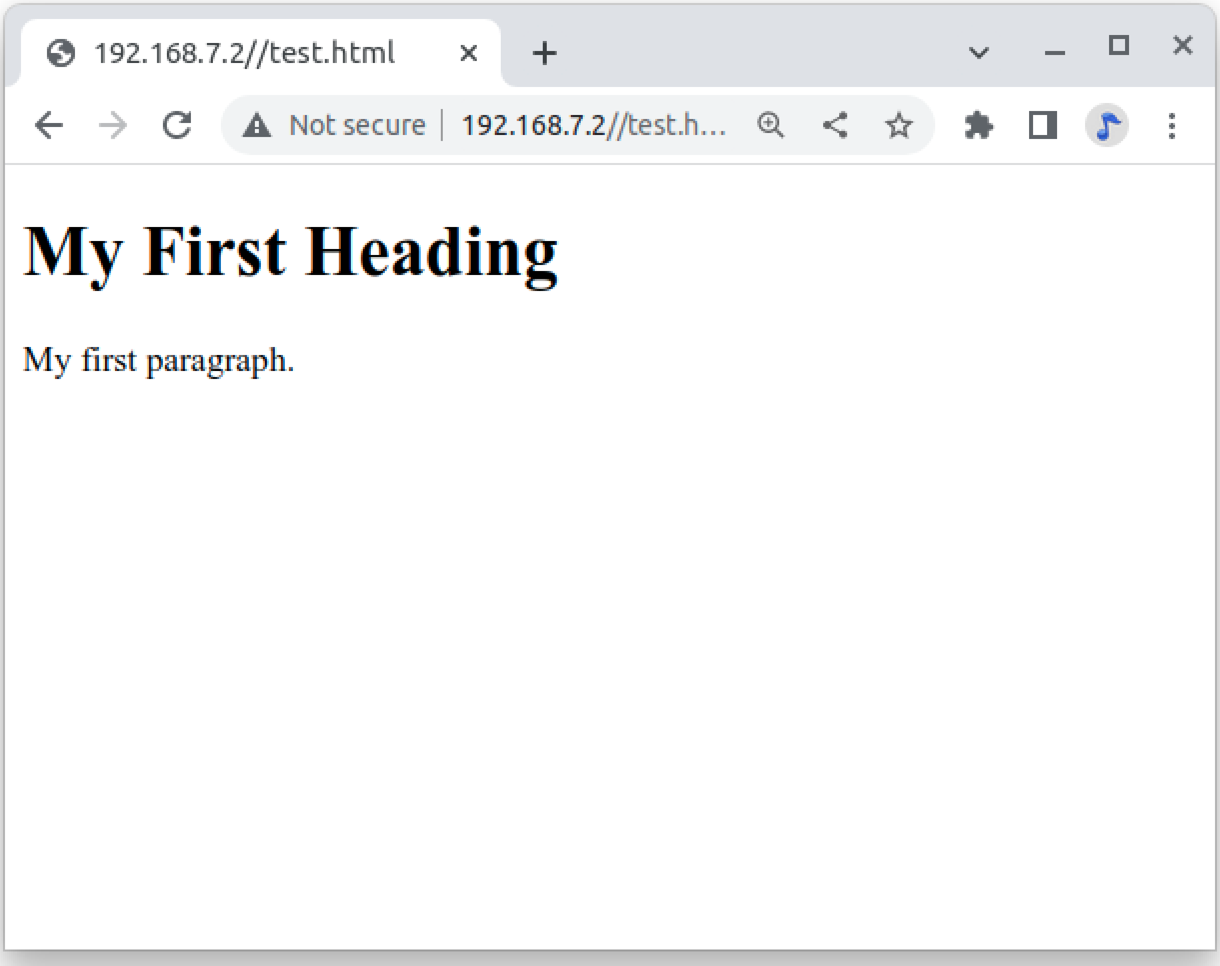


Figure 42. test.html as served by nginx

Interacting with the Bone via a Web Browser

Problem

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

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt update</strong>
bone$ <strong>sudo apt install python3-flask</strong>
</pre>
```

All the code in is the Cookbook repo:

```
<pre data-type="programlisting">
bone$ <strong>git clone https://github.com/MarkAYoder/BoneCookbook</strong>
bone$ <strong>cd BoneCookbook/doc/06iod/code/flash</strong>
</pre>
```

First Flask - hello, world

Our first example is **helloWorld.py**

Example 40. Python code for flask hello world (helloWorld.py)

```
#!/usr/bin/env python
# From: https://towardsdatascience.com/python-webserver-with-flask-and-raspberry-
pi-398423cc6f5d

from flask import Flask     ①
app = Flask(__name__)       ②
@app.route('/')             ③
def index():
    return 'hello, world'
if __name__ == '__main__':
    app.run(debug=True, port=8080, host='0.0.0.0')④
```

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 and you should see.

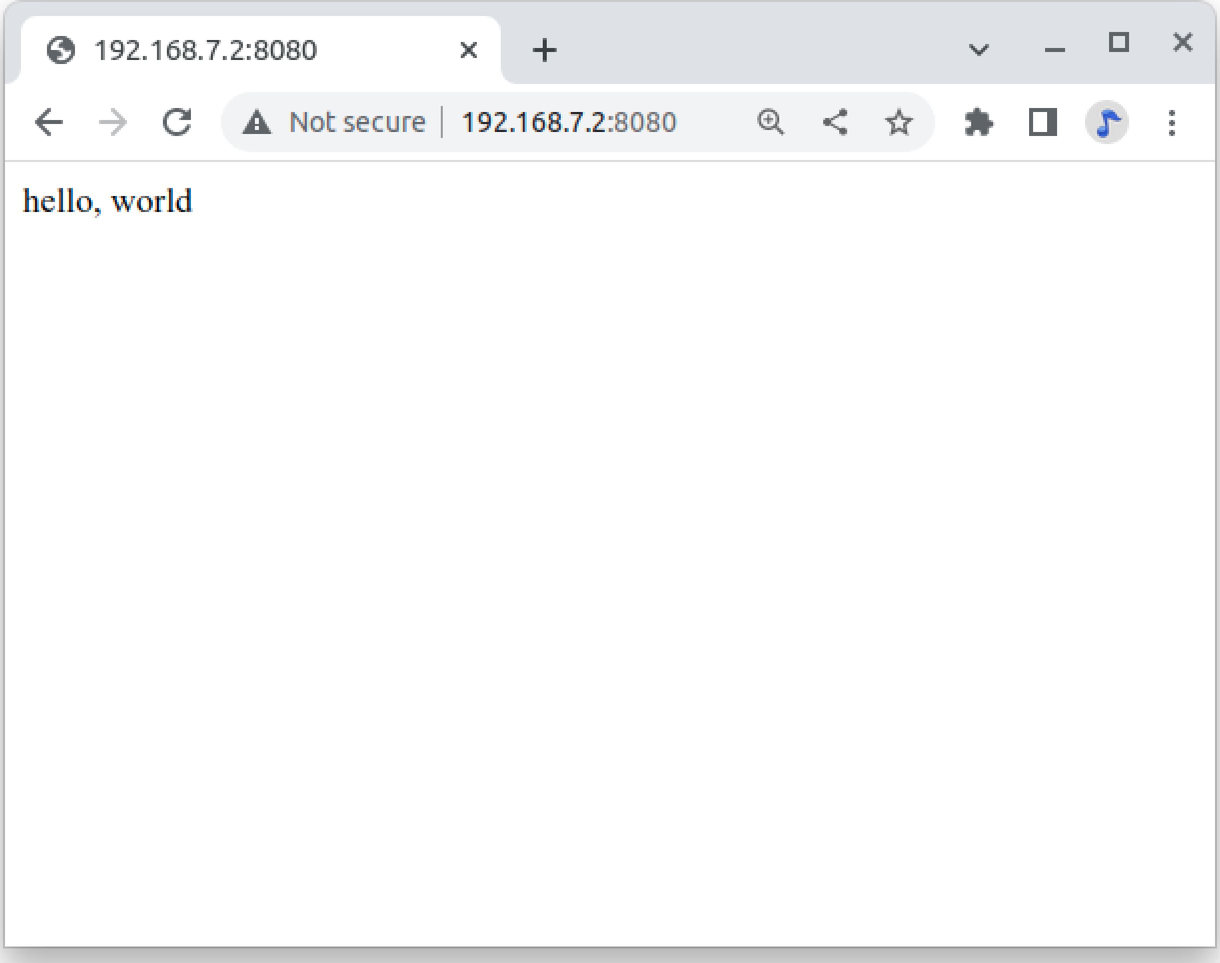


Figure 43. Test page served by our custom flask server

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**:

Example 41. html code for flask hello world (index1.html)

```
<!DOCTYPE html>
   <head>
      <title>{{ title }}</title>
   </head>
   <body>
      <h1>Hello, World!</h1>
      <h2>The date and time on the server is: {{ time }}</h2>
   </body>
</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:

Example 42. Python code for flask index1.html (app1.py)

```
#!/usr/bin/env python
# From: https://towardsdatascience.com/python-webserver-with-flask-and-raspberry-
pi-398423cc6f5d

'''
Code created by Matt Richardson
for details, visit:  http://mattrichardson.com/Raspberry-Pi-Flask/inde...
'''
from flask import Flask, render_template
import datetime
app = Flask(__name__)
@app.route("/")
def hello():
   now = datetime.datetime.now()
   timeString = now.strftime("%Y-%m-%d %H:%M")
   templateData = {
      'title' : 'HELLO!',
      'time': timeString
      }
   return render_template('index1.html', **templateData)
if __name__ == "__main__":
   app.run(host='0.0.0.0', port=8080, debug=True)
```

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 index.html template to the web browser using the variables in the templateData dictionary.

Execute the Python script:

```
<pre data-type="programlisting">
bone$ <strong>.\app1.py</strong>
</pre>
```

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

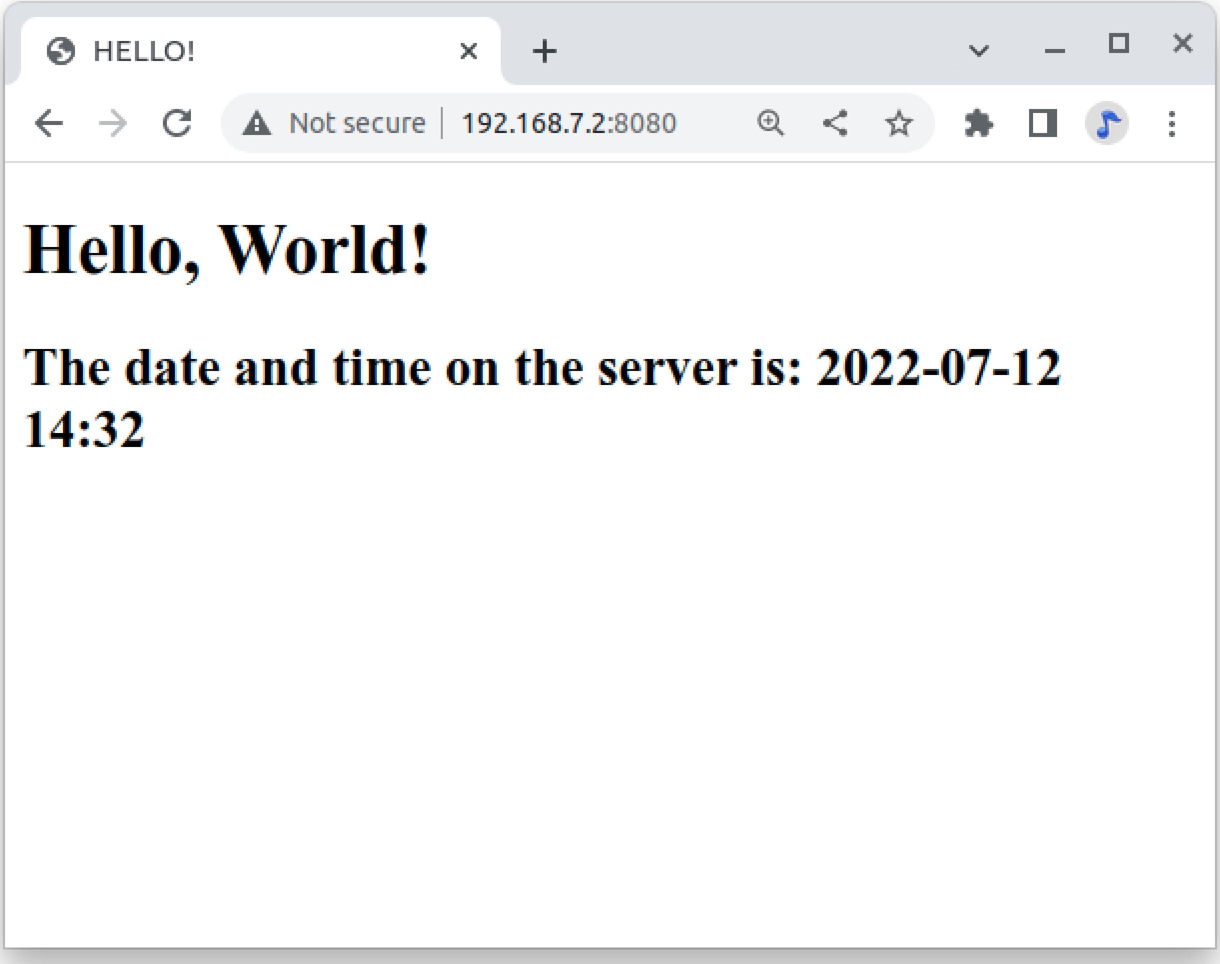


Figure 44. Test page served by app1.py

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 second.

Displaying GPIO Status in a Web Browser - reading a button

Problem

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

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**.

Example 43. A simple Flask-based web server to read a GPIO (app2.py)

```
#!/usr/bin/env python
# From: https://towardsdatascience.com/python-webserver-with-flask-and-raspberry-
pi-398423cc6f5d
import os
from flask import Flask, render_template
app = Flask(__name__)

# gpioinfo | grep -e chip -e P9_11
pin = '30' #  P9_11 is gpio 30
GPIOPATH="/sys/class/gpio"
buttonSts = 0

# Make sure pin is exported
if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
    f = open(GPIOPATH+"/export", "w")
    f.write(pin)
    f.close()

# Make it an input pin
f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
f.write("in")
f.close()

@app.route("/")
def index():
    # Read Button Status
    f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
    buttonSts = f.read()[:-1]
    f.close()

    # buttonSts = GPIO.input(button)
    templateData = {
      'title' : 'GPIO input Status!',
      'button'  : buttonSts,
      }
    return render_template('index2.html', **templateData)
if __name__ == "__main__":
   app.run(host='0.0.0.0', port=8080, debug=True)
```

Look that 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:

Example 44. A simple Flask-based web server to read a GPIO (index2.html)

```
<!DOCTYPE html>
   <head>
      <title>{{ title }}</title>
      <link rel="stylesheet" href='../static/style.css'/>
   </head>
   <body>
      <h1>{{ title }}</h1>
      <h2>Button pressed:  {{ button }}</h1>
   </body>
</html>
```

Now, run the following command:

```
<pre data-type="programlisting">
bone$ <strong>./app2.py</strong>
</pre>
```

Point your browser to *http://192.168.7.2:8080*, and the page will look like [networking_GPIOserver_fig].

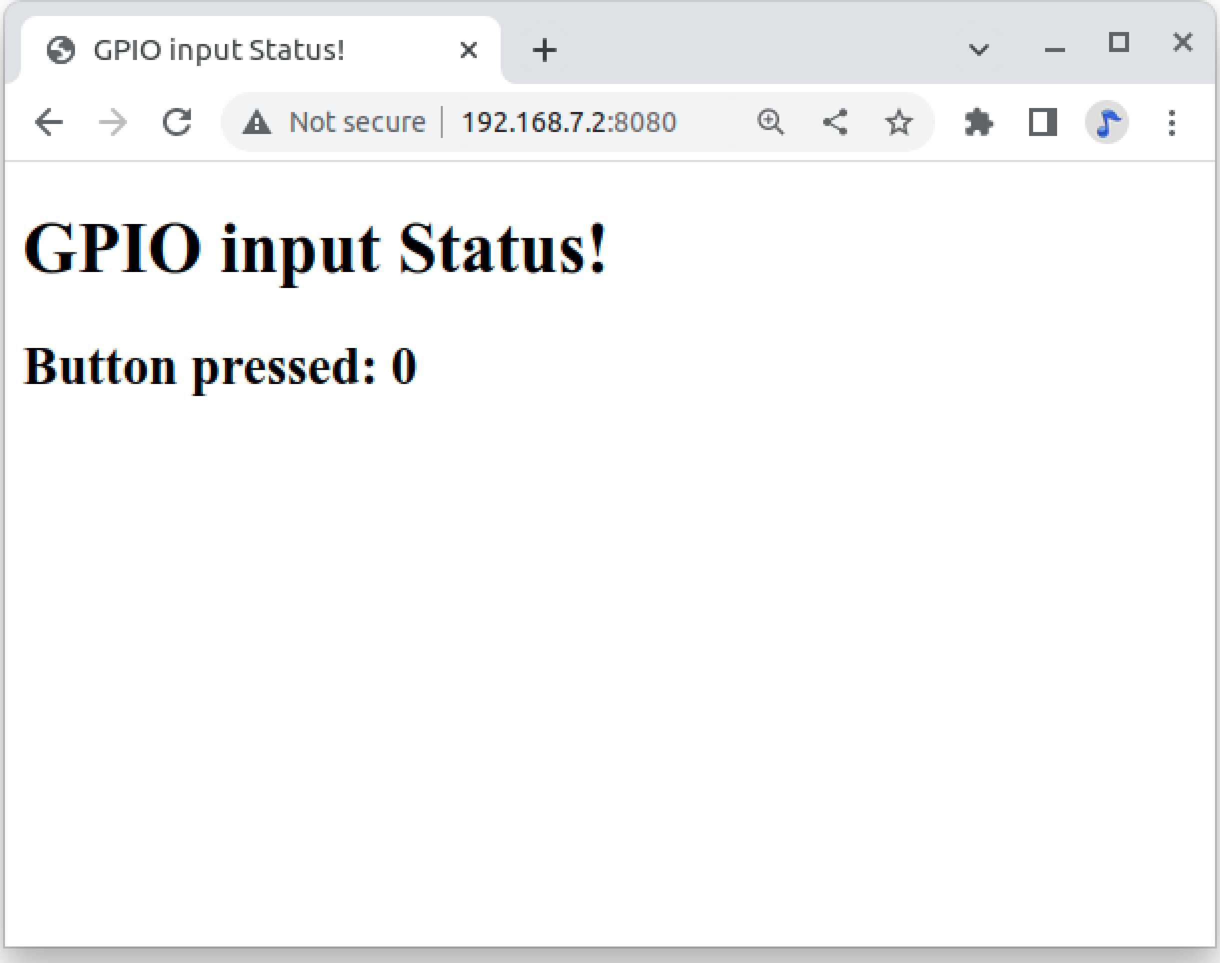


Figure 45. Status of a GPIO pin on a web page

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

Discussion

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.

Controlling GPIOs

Problem

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

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.

The python script Let's create a new Python script and named it **app3.py**.

Example 45. A simple Flask-based web server to read a GPIO (app3.py)

```
#!/usr/bin/env python
# From: https://towardsdatascience.com/python-webserver-with-flask-and-raspberry-
pi-398423cc6f5d
# import Adafruit_BBIO.GPIO as GPIO
import os
from flask import Flask, render_template, request
app = Flask(__name__)
#define LED GPIO
ledRed = "P9_14"
pin = '50' #  P9_14 is gpio 50
GPIOPATH="/sys/class/gpio"

#initialize GPIO status variable
ledRedSts = 0
# Make sure pin is exported
if (not os.path.exists(GPIOPATH+"/gpio"+pin)):
    f = open(GPIOPATH+"/export", "w")
    f.write(pin)
    f.close()
# Define led pin as output
f = open(GPIOPATH+"/gpio"+pin+"/direction", "w")
f.write("out")
f.close()
# turn led OFF
f = open(GPIOPATH+"/gpio"+pin+"/value", "w")
f.write("0")
f.close()

@app.route("/")
def index():
    # Read Sensors Status
    f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
    ledRedSts = f.read()
    f.close()
    templateData = {
              'title' : 'GPIO output Status!',
              'ledRed'  : ledRedSts,
        }
    return render_template('index3.html', **templateData)

@app.route("/<deviceName>/<action>")
def action(deviceName, action):
    if deviceName == 'ledRed':
        actuator = ledRed
    f = open(GPIOPATH+"/gpio"+pin+"/value", "w")
    if action == "on":
        f.write("1")
    if action == "off":
```

```
        f.write("0")
    f.close()

    f = open(GPIOPATH+"/gpio"+pin+"/value", "r")
    ledRedSts = f.read()
    f.close()

    templateData = {
              'ledRed'  : ledRedSts,
    }
    return render_template('index3.html', **templateData)
if __name__ == "__main__":
   app.run(host='0.0.0.0', port=8080, debug=True)
```

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
- Update the variable library
- 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:

Example 46. A simple Flask-based web server to write a GPIO (index3.html)

```
<!DOCTYPE html>
   <head>
      <title>GPIO Control</title>
      <link rel="stylesheet" href='../static/style.css'/>
   </head>
   <body>
        <h2>Actuators</h2>
        <h3> Status </h3>
             RED LED ==>  {{ ledRed  }}
        <br>
        <h3> Commands </h3>
            RED LED Ctrl ==>
            <a href="/ledRed/on" class="button">TURN ON</a>
            <a href="/ledRed/off"class="button">TURN OFF</a>
   </body>
</html>
```

```
<pre data-type="programlisting">
bone$ <strong>./app3.py</strong>
</pre>
```

Point your browser as before and you will see:

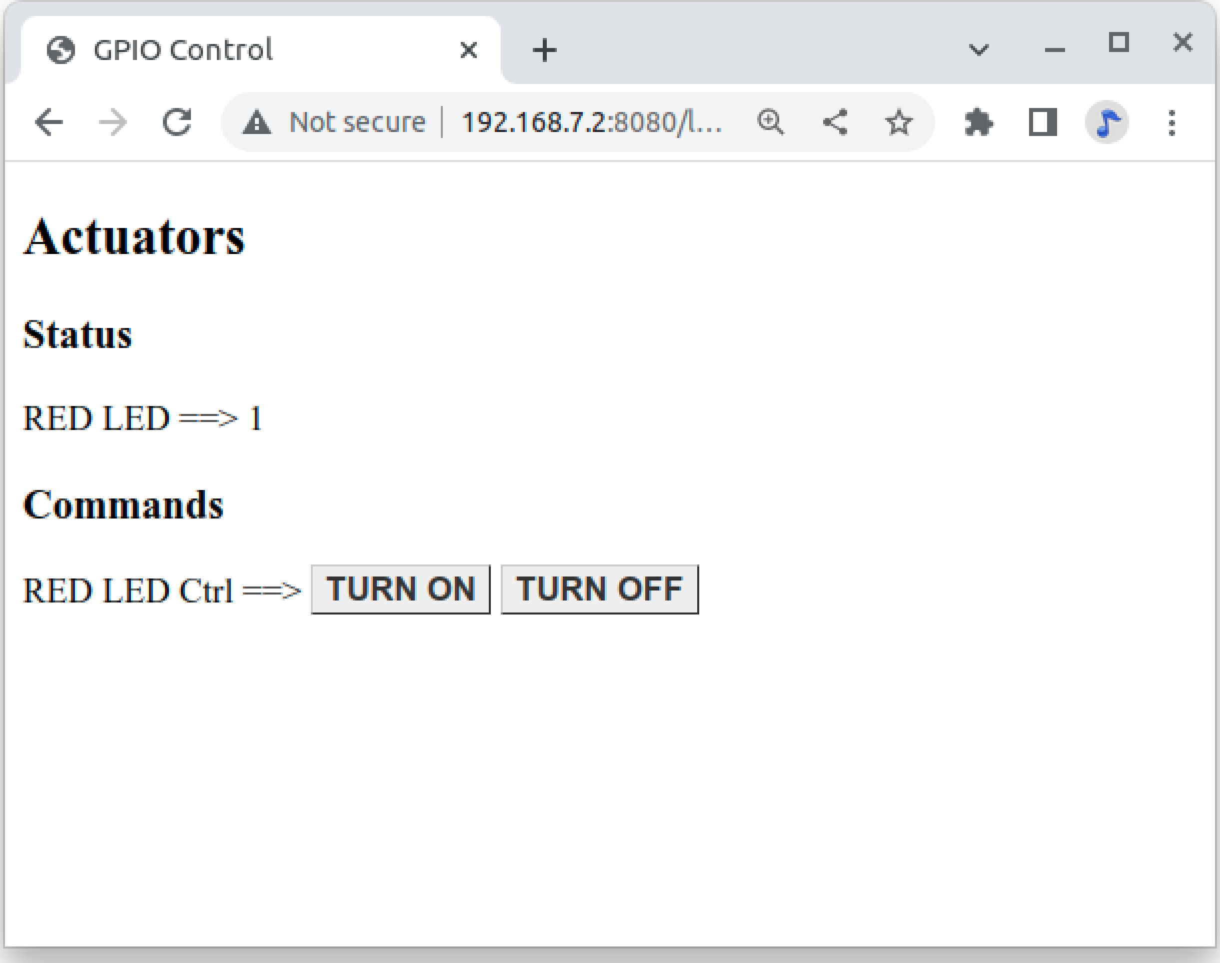


Figure 46. Status of a GPIO pin on a web page

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.

Plotting Data

Problem

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

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)

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

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/bus/iio/devices/iio:device0</strong>
bone$ <strong>ls -F</strong>
```

```
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
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>ls scan_elements</strong>
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
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>echo 1 > scan_elements/in_voltage1_en</strong>
</pre>
```

Next set the buffer size.

```
<pre data-type="programlisting">
bone$ <strong>ls buffer</strong>
data_available  enable  length  watermark
</pre>
```

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

```
<pre data-type="programlisting">
bone$ <strong>echo 512 > buffer/length</strong>
</pre>
```

Then start it running.

```
<pre data-type="programlisting">
bone$ <strong>echo 1 > buffer/enable</strong>
</pre>
```

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

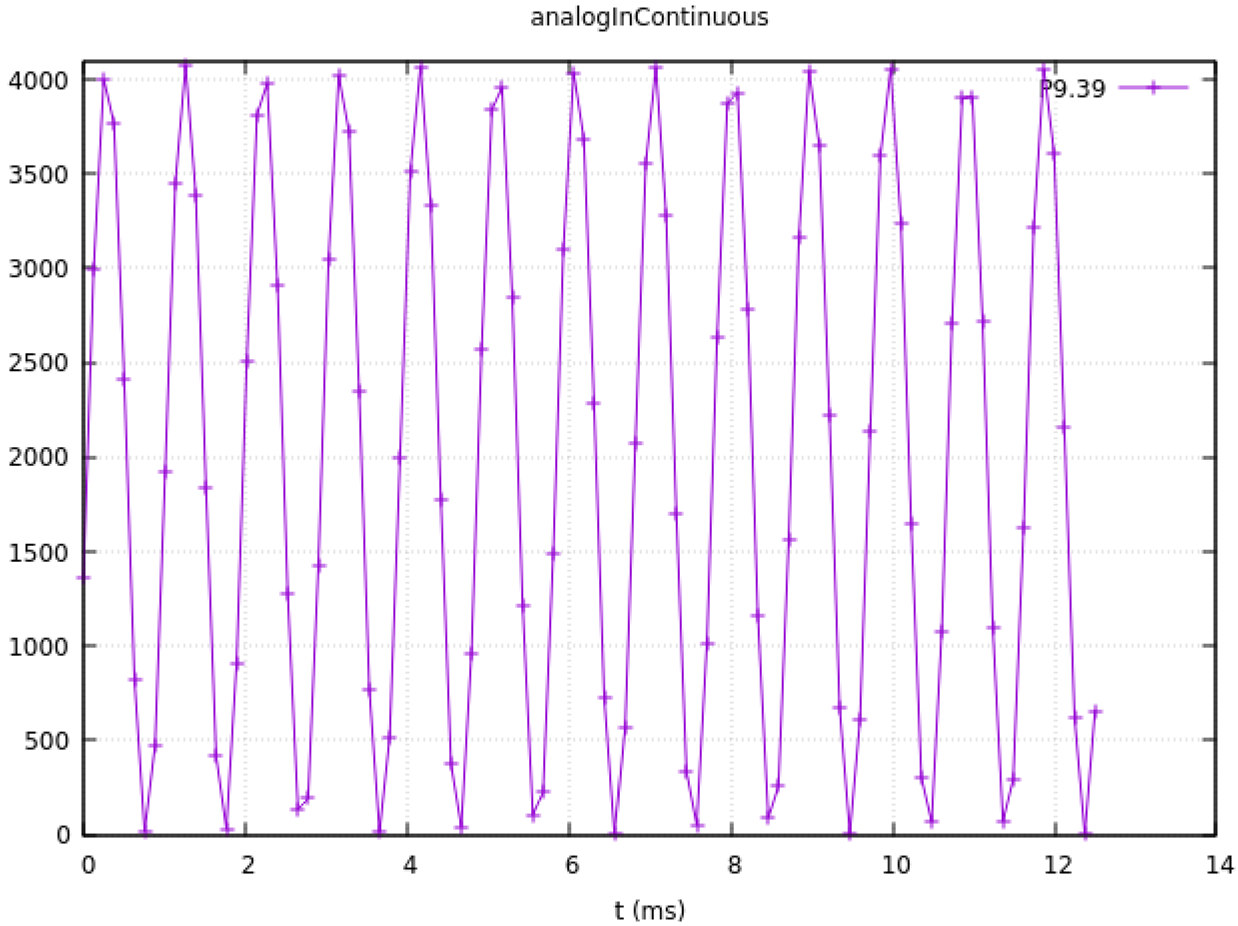


Figure 47. 1KHz sine wave sampled at 8KHz

An example Python program that does the above and the reads and plot the buffer is here: analogInContinuous.py

Example 47. Code to read and plot a continuous analog input(analogInContinuous.py)

```
#!/usr/bin/python
#//////////////////////////////////////
#   analogInContinuous.py
#   Read analog data via IIO continous mode and plots it.
#//////////////////////////////////////
# From: https://stackoverflow.com/questions/20295646/python-ascii-plots-in-
terminal
# https://github.com/dkogan/gnuplotlib
# https://github.com/dkogan/gnuplotlib/blob/master/guide/guide.org
# sudo apt install gnuplot  (10 minute to install)
# sudo apt install libatlas-base-dev
# pip3 install gnuplotlib
# This uses X11, so when connecting to the bone from the host use:  ssh -X bone

# See https://elinux.org/index.php?title=EBC_Exercise_10a_Analog_In#Analog_in_-
_Continuous.2C_Change_the_sample_rate
# for instructions on changing the sampling rate.  Can go up to 200KHz.

import numpy      as np
```

```
import gnuplotlib as gp
import time
# import struct

IIOPATH='/sys/bus/iio/devices/iio:device0'
IIODEV='/dev/iio:device0'
LEN = 100
SAMPLERATE=8000
AIN='2'

# Setup IIO for Continous reading
# Enable AIN
try:
    file1 = open(IIOPATH+'/scan_elements/in_voltage'+AIN+'_en', 'w')
    file1.write('1')
    file1.close()
except:     # carry on if it's already enabled
    pass
# Set buffer length
file1 = open(IIOPATH+'/buffer/length', 'w')
file1.write(str(2*LEN))     # I think LEN is in 16-bit values, but here we pass
bytes
file1.close()
# Enable continous
file1 = open(IIOPATH+'/buffer/enable', 'w')
file1.write('1')
file1.close()

x = np.linspace(0, 1000*LEN/SAMPLERATE, LEN)
# Do a dummy plot to give time of the fonts to load.
gp.plot(x, x)
print("Waiting for fonts to load")
time.sleep(10)

print('Hit ^C to stop')

fd = open(IIODEV, "r")

try:
    while True:
        y = np.fromfile(fd, dtype='uint16', count=LEN)*1.8/4096
        # print(y)
        gp.plot(x, y,
            xlabel = 't (ms)',
            ylabel = 'volts',
            _yrange = [0, 2],
            title  = 'analogInContinuous',
            legend = np.array( ("P9.39", ), ),
            # ascii=1,
            # terminal="xterm",
            # legend = np.array( ("P9.40", "P9.38"), ),
```

```
            # _with  = 'lines'
            )

except KeyboardInterrupt:
    print("Turning off input.")
    # Disable continous
    file1 = open(IIOPATH+'/buffer/enable', 'w')
    file1.write('0')
    file1.close()

    file1 = open(IIOPATH+'/scan_elements/in_voltage'+AIN+'_en', 'w')
    file1.write('0')
    file1.close()

# // Bone  | Pocket | AIN
# // ----- | ------ | ---
# // P9_39 | P1_19  | 0
# // P9_40 | P1_21  | 1
# // P9_37 | P1_23  | 2
# // P9_38 | P1_25  | 3
# // P9_33 | P1_27  | 4
# // P9_36 | P2_35  | 5
# // P9_35 | P1_02  | 6
```

Be sure to read the instillation 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:

```
<pre data-type="programlisting">
host$ <strong>ssh -X bone</strong>

bone$ <strong>cd <Cookbook repo>/doc/06iot/code</strong>
bone$ <strong>./analogInContinuous.py</strong>
Hit ^C to stop
</pre>
```

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.

```
<pre>
bone$ <strong>echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable</strong>
</pre>
```

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.

```
<pre>
bone$ <strong>cd /opt/source/bb.org-overlays</strong>
bone$ <strong>make</strong>
</pre>
```

This will take a while the first time as it compiles all the device trees.

```
<pre>
bone$ <strong>vi src/arm/src/arm/BB-ADC-00A0.dts</strong>
</pre>
```

Around line 57 you'll see

```
<pre>
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-opendelay = <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>;

</pre>
```

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 got as fast a possible.

```
<pre>
ti,chan-step-avg = <1 1 1 1 1 1 1 1>;
ti,chan-step-opendelay = <0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00>;
</pre>
```

Now compile it.

```
<pre>
bone$ <strong>make</strong>
  DTC     src/arm/BB-ADC-00A0.dtbo
gcc -o config-pin ./tools/pmunts_muntsos/config-pin.c
</pre>
```

make knows to only recompile the file you just edited. Now install and reboot.

```
<pre>
bone$ <strong>sudo make install</strong>
...
'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$ <strong>reboot</strong>
</pre>
```

A number of files get installed, including the ADC file. Now try rerunning.

```
<pre>
bone$ <strong>cd <Cookbook repo>/docs/06iot/code></strong>
bone$ <strong>./analogInContinuous.py</strong>
Hit ^C to stop
</pre>
```

Here’s the output of a 10KHz triangle wave.

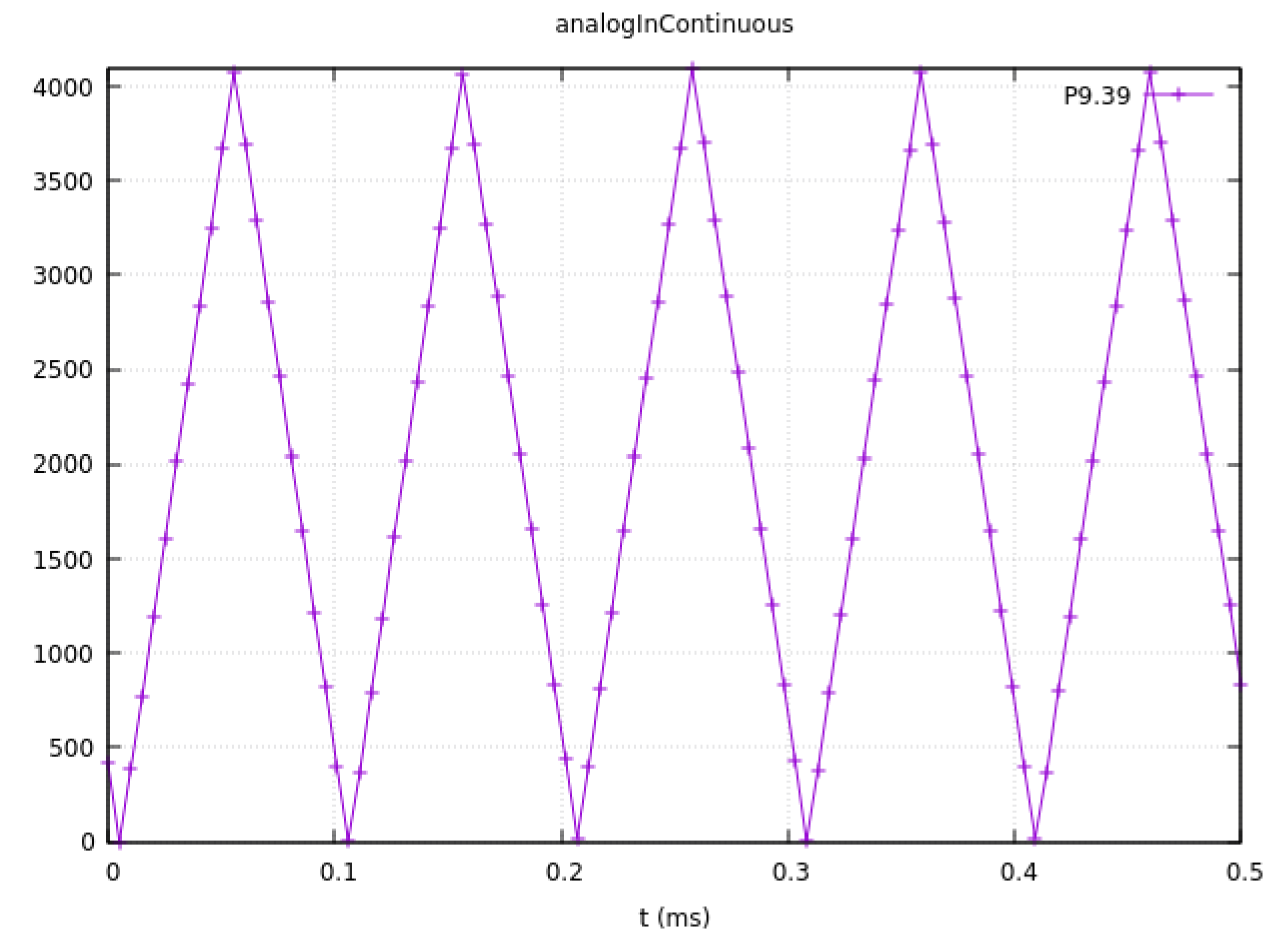


Figure 48. 10KHz triangle wave sampled at 200KHz

It’s still a good idea to disable the buffer when done.

```
<pre>
bone$ </strong>echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable</strong>
</pre>
```

Sending an Email

Problem

You want to send an email via Gmail from the Bone.

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
- 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 .

Example 48. Sending email using nodemailer (emailTest.py)

```
#!/usr/bin/env python
# From: https://realpython.com/python-send-email/
import smtplib, ssl

port = 587  # For starttls
smtp_server  = "smtp.gmail.com"
sender_email = "from_account@gmail.com"
receiver_email = "to_account@gmail.com"
# Go to: https://myaccount.google.com/security
# Select App password
# Generate your own 16 char password, copy here
# Delete password when done
password = "cftqhcejjdjfdwjh"
message = """\
Subject: Testing email

This message is sent from Python.

"""
context = ssl.create_default_context()
with smtplib.SMTP(smtp_server, port) as server:
    server.starttls(context=context)
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)
```

Then run the script to send the email:

```
<pre data-type="programlisting">
```

```
bone$ <strong>chmod +x emailTest.py</strong>
bone$ <strong>.\emailTest.py</strong>
</pre>
```

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.

Discussion

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.

Displaying the Current Weather Conditions

Problem

You want to display the current weather conditions.

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 and get your API key.
- Store your key in the bash variable APPID.

```
<pre>
bash$ <strong>export APPID="Your key"</strong>
</pre>
```

- Then add the code in Code for getting current weather conditions (*weather.py*) to a file named *weather.js*.
- Run the pyhon script.

Example 49. Code for getting current weather conditions (weather.py)

```
#!/usr/bin/env python3
# Displays current weather and forcast
import os
import sys
from datetime import datetime
import requests     # For getting weather
```

```
# http://api.openweathermap.org/data/2.5/onecall
params = {
    'appid': os.environ['APPID'],
    # 'city': 'brazil,indiana',
    'exclude': "minutely,hourly",
    'lat':  '39.52',
    'lon': '-87.12',
    'units': 'imperial'
    }
urlWeather = "http://api.openweathermap.org/data/2.5/onecall"

print("Getting weather")

try:
    r = requests.get(urlWeather, params=params)
    if(r.status_code==200):
        # print("headers: ", r.headers)
        # print("text: ", r.text)
        # print("json: ", r.json())
        weather = r.json()
        print("Temp: ", weather['current']['temp'])         ①
        print("Humid:", weather['current']['humidity'])
        print("Low:  ", weather['daily'][1]['temp']['min'])
        print("High: ", weather['daily'][0]['temp']['max'])
        day = weather['daily'][0]['sunrise']-weather['timezone_offset']
        print("sunrise: " + datetime.utcfromtimestamp(day).strftime('%Y-%m-%d
%H:%M:%S'))
        # print("Day: " + datetime.utcfromtimestamp(day).strftime('%a'))
        # print("weather: ", weather['daily'][1])           ②
        # print("weather: ", weather)                       ③
        # print("icon: ", weather['current']['weather'][0]['icon'])
        # print()

    else:
        print("status_code: ", r.status_code)
except IOError:
    print("File not found: " + tmp101)
    print("Have you run setup.sh?")
except:
    print("Unexpected error:", sys.exc_info())
```

① Prints current conditions.

② Prints the forecast for the next day.

③ Prints everything returned by the weather site.

Run this by using the following commands:

<pre data-type="programlisting">

```
bone$ <strong>./weather.js</strong>
Getting weather
Temp:  85.1
Humid: 50
Low:   62.02
High:  85.1
sunrise: 2022-07-14 14:32:46
</pre>
```

Discussion

The weather API returns lots of information. Use Python to extract the information you want.

Sending and Receiving Tweets

Problem

You want to send and receive tweets (Twitter posts) with your Bone.

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.

Creating a Project and App

- Follow the directions here to create a project and 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.

```
<pre data-type="programlisting">
export API_KEY='XXX'
export API_SECRET_KEY='XXX'
export BEARER_TOKEN='XXX'
export TOKEN='4XXX'
export TOKEN_SECRET='XXX'
</pre>
```

- Next, source the file so the values will appear in your bash session.

```
<pre>
bash$ <strong>source twitterKeys.sh</strong>
</pre>
```

You'll need to do this every time you open a new bash window.

Creating a tweet

Run *twitter_create_tweet*.py_ to see your timeline.

Example 50. Create a Tweet (twitter_create_tweet.py)

```
#!/usr/bin/env python
# From: https://github.com/twitterdev/Twitter-API-v2-sample-code/blob/main/Manage-
Tweets/create_tweet.py
from requests_oauthlib import OAuth1Session
import os
import json

# In your terminal please set your environment variables by running the following
lines of code.
# export 'API_KEY'='<your_consumer_key>'
# export 'API_SECRET_KEY'='<your_consumer_secret>'

consumer_key = os.environ.get("API_KEY")
consumer_secret = os.environ.get("API_SECRET_KEY")

# Be sure to add replace the text of the with the text you wish to Tweet. You can
also add parameters to post polls, quote Tweets, Tweet with reply settings, and
Tweet to Super Followers in addition to other features.
payload = {"text": "Hello world!"}

# Get request token
request_token_url =
"https://api.twitter.com/oauth/request_token?oauth_callback=oob&x_auth_access_type
=write"
oauth = OAuth1Session(consumer_key, client_secret=consumer_secret)

try:
    fetch_response = oauth.fetch_request_token(request_token_url)
except ValueError:
    print(
        "There may have been an issue with the consumer_key or consumer_secret you
entered."
    )

resource_owner_key = fetch_response.get("oauth_token")
resource_owner_secret = fetch_response.get("oauth_token_secret")
print("Got OAuth token: %s" % resource_owner_key)

# Get authorization
base_authorization_url = "https://api.twitter.com/oauth/authorize"
authorization_url = oauth.authorization_url(base_authorization_url)
print("Please go here and authorize: %s" % authorization_url)
```

```
verifier = input("Paste the PIN here: ")

# Get the access token
access_token_url = "https://api.twitter.com/oauth/access_token"
oauth = OAuth1Session(
    consumer_key,
    client_secret=consumer_secret,
    resource_owner_key=resource_owner_key,
    resource_owner_secret=resource_owner_secret,
    verifier=verifier,
)
oauth_tokens = oauth.fetch_access_token(access_token_url)

access_token = oauth_tokens["oauth_token"]
access_token_secret = oauth_tokens["oauth_token_secret"]

# Make the request
oauth = OAuth1Session(
    consumer_key,
    client_secret=consumer_secret,
    resource_owner_key=access_token,
    resource_owner_secret=access_token_secret,
)

# Making the request
response = oauth.post(
    "https://api.twitter.com/2/tweets",
    json=payload,
)

if response.status_code != 201:
    raise Exception(
        "Request returned an error: {} {}".format(response.status_code,
response.text)
    )

print("Response code: {}".format(response.status_code))

# Saving the response as JSON
json_response = response.json()
print(json.dumps(json_response, indent=4, sort_keys=True))
```

Run the code and you'll have to authorize.

```
<pre>
bash$ <strong>./twitter_create_tweet.py</strong>
Got OAuth token: tWBldQAAAAAAWBJgAAABggJt7qg
Please go here and authorize:
https://api.twitter.com/oauth/authorize?oauth_token=tWBldQAAAAAAWBJgAAABggJt7qg
Paste the PIN here: <strong>4859044</strong>
```

```
Response code: 201
{
    "data": {
        "id": "1547963178700533760",
        "text": "Hello world!"
    }
}
</pre>
```

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.

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.

Example 51. Code to delete a tweet (twitter_delete_tweet.py)

```
#!/usr/bin/env python
# From: https://github.com/twitterdev/Twitter-API-v2-sample-code/blob/main/Manage-
Tweets/delete_tweet.py
from requests_oauthlib import OAuth1Session
import os
import json

# In your terminal please set your environment variables by running the following
lines of code.
# export 'API_KEY'='<your_consumer_key>'
# export 'API_SECRET_KEY'='<your_consumer_secret>'

consumer_key = os.environ.get("API_KEY")
consumer_secret = os.environ.get("API_SECRET_KEY")

# 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
id = "1547963178700533760"

# Get request token
request_token_url =
"https://api.twitter.com/oauth/request_token?oauth_callback=oob&x_auth_access_type
=write"
oauth = OAuth1Session(consumer_key, client_secret=consumer_secret)

try:
    fetch_response = oauth.fetch_request_token(request_token_url)
except ValueError:
    print(
        "There may have been an issue with the consumer_key or consumer_secret you
entered."
```

```
    )

resource_owner_key = fetch_response.get("oauth_token")
resource_owner_secret = fetch_response.get("oauth_token_secret")
print("Got OAuth token: %s" % resource_owner_key)

# Get authorization
base_authorization_url = "https://api.twitter.com/oauth/authorize"
authorization_url = oauth.authorization_url(base_authorization_url)
print("Please go here and authorize: %s" % authorization_url)
verifier = input("Paste the PIN here: ")

# Get the access token
access_token_url = "https://api.twitter.com/oauth/access_token"
oauth = OAuth1Session(
    consumer_key,
    client_secret=consumer_secret,
    resource_owner_key=resource_owner_key,
    resource_owner_secret=resource_owner_secret,
    verifier=verifier,
)
oauth_tokens = oauth.fetch_access_token(access_token_url)

access_token = oauth_tokens["oauth_token"]
access_token_secret = oauth_tokens["oauth_token_secret"]

# Make the request
oauth = OAuth1Session(
    consumer_key,
    client_secret=consumer_secret,
    resource_owner_key=access_token,
    resource_owner_secret=access_token_secret,
)

# Making the request
response = oauth.delete("https://api.twitter.com/2/tweets/{}".format(id))

if response.status_code != 200:
    raise Exception(
        "Request returned an error: {} {}".format(response.status_code,
response.text)
    )

print("Response code: {}".format(response.status_code))

# Saving the response as JSON
json_response = response.json()
print(json_response)
```

To see many other examples, go to iStrategyLabs' node-twitter GitHub page.

Discussion

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?

Wiring the IoT with Node-RED

Problem

You want your BeagleBone to interact with the Internet, but you want to program it graphically.

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.

Installing Node-RED

Node-RED is already installed and 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 interface.

> **NOTE**
>
> If NODE-RED doesn't appear, run the following in a terminal widow.
>
> ```
> <pre data-type="programlisting">
> bone$ sudo systemctl start nodered.service
> </pre>
> ```
>
> Wait a bit, then refresh the browser window running NODE-RED.

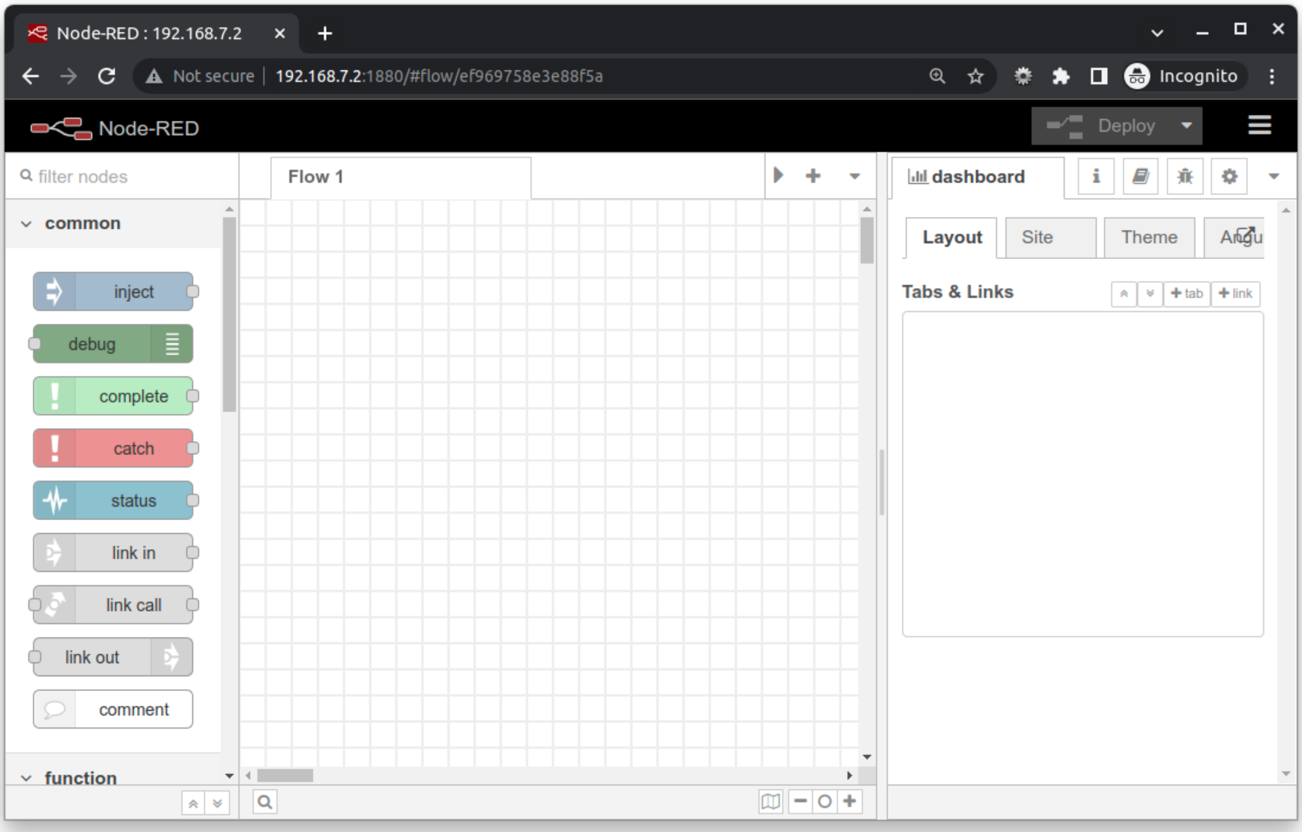


Figure 49. The Node-RED interface

Building a Node-RED Flow - Blink an LED

The example in this recipe builds a Node-RED flow that turn on and off an LED in response to button pushes. First drag the **inject** node to the canvas and double-click on it.

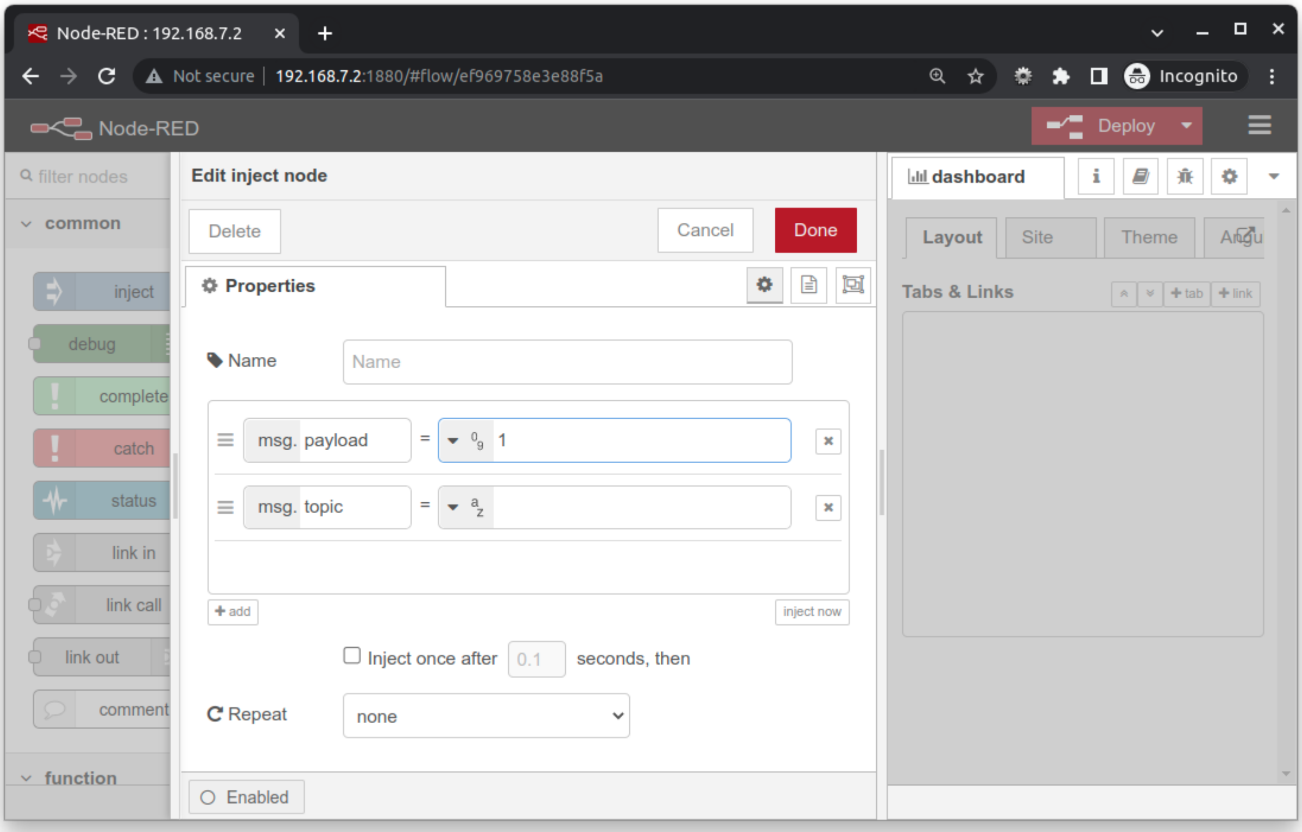


Figure 50. Configuring inject

Set the **msg.payload** to **number** and enter **1**. Click **Done**. Make a second **inject** node, but make it **0**.

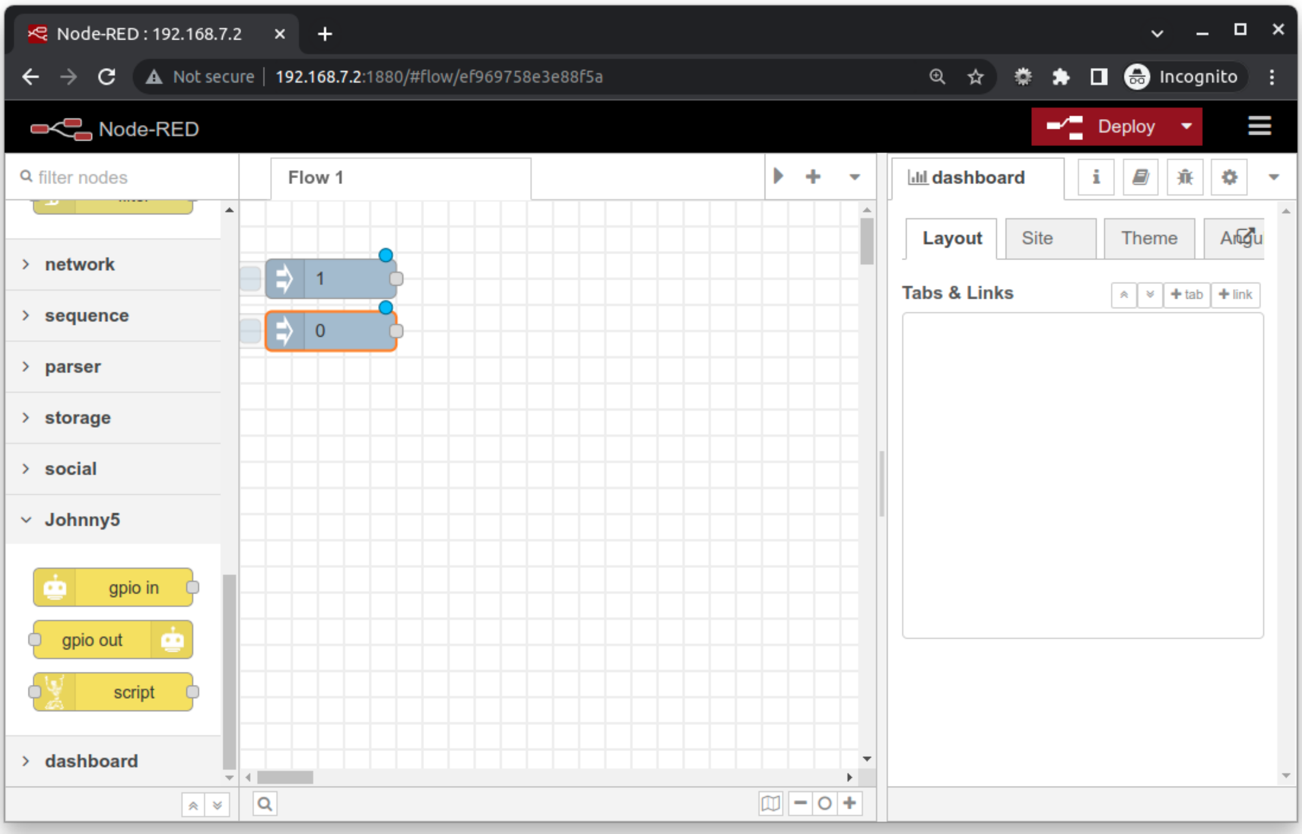


Figure 51. Placing two injects

Next scroll down on the left list of nodes to **Johnny5** and drag the **gpio out** node to the canvas and double-click to open it.

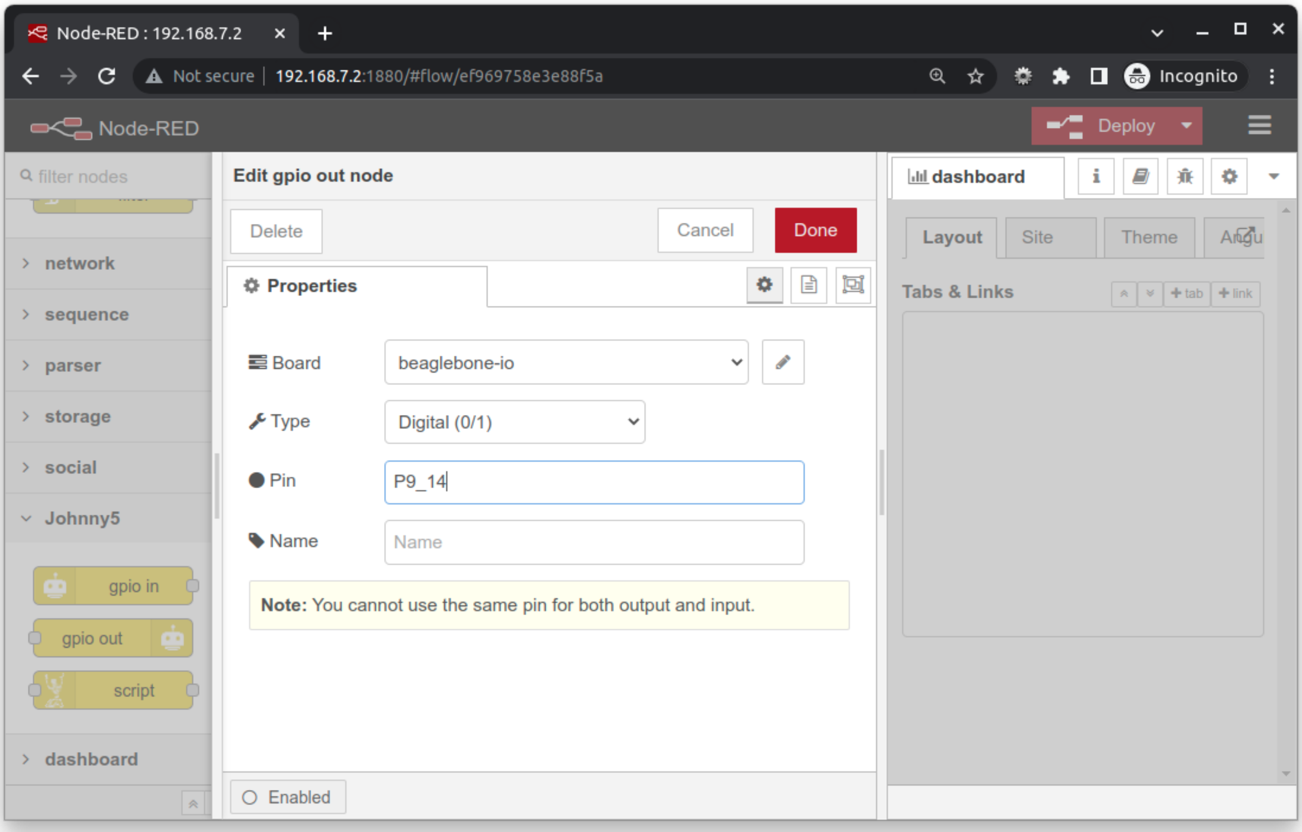


Figure 52. Configuring gpio

Set the properties as shown, using the Pin that you've wired up. Click **Done** and wire as shown.

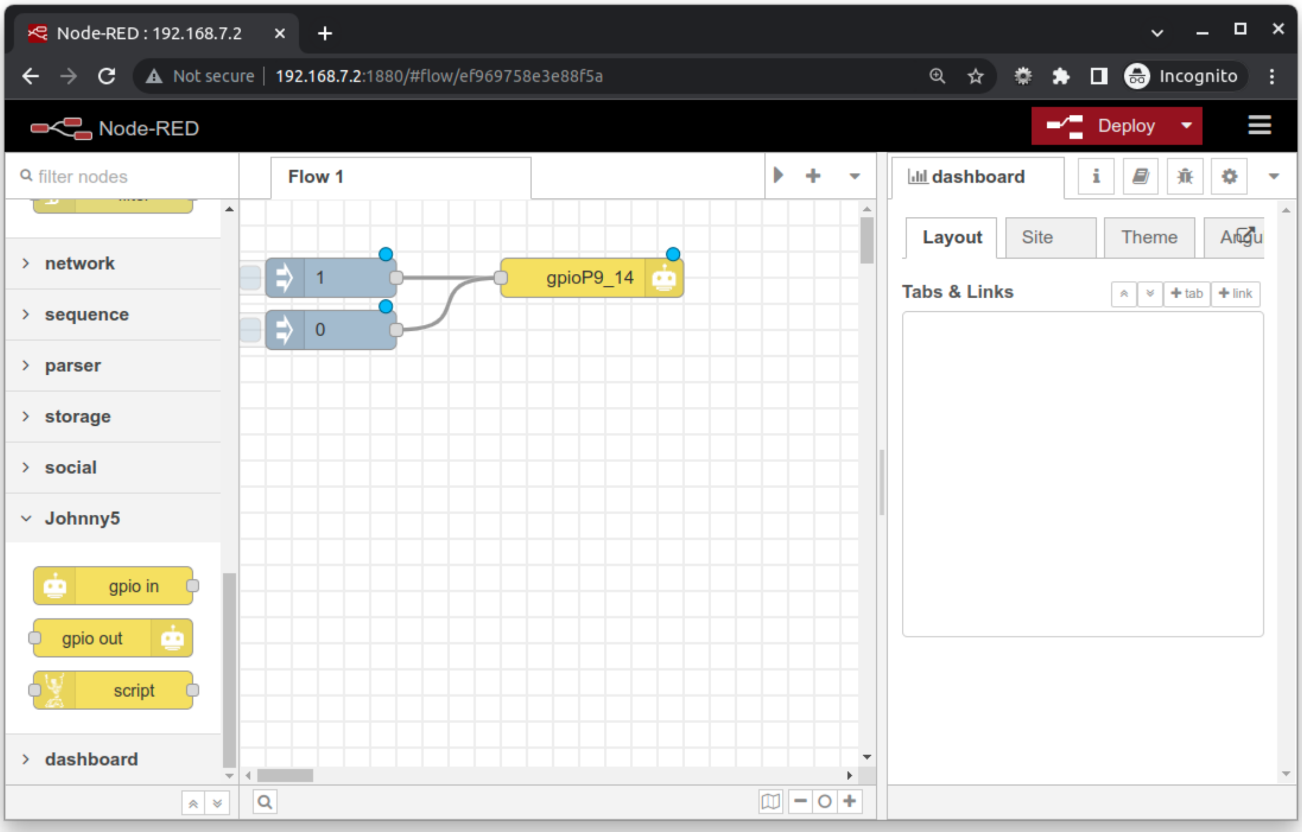


Figure 53. Wiring the nodes

Once wired, click the big red **Deploy** button. Click the **1** inject button and the LED will light up. Click the **0** to turn it off.

Congratulations, your first NODE-RED flow is working.

Installing more nodes

NODE-RED has many other flows and nodes that can be installed. You can see them https://flows.nodered.org/. Here we'll install **twitter**. First open **Manage palette**.

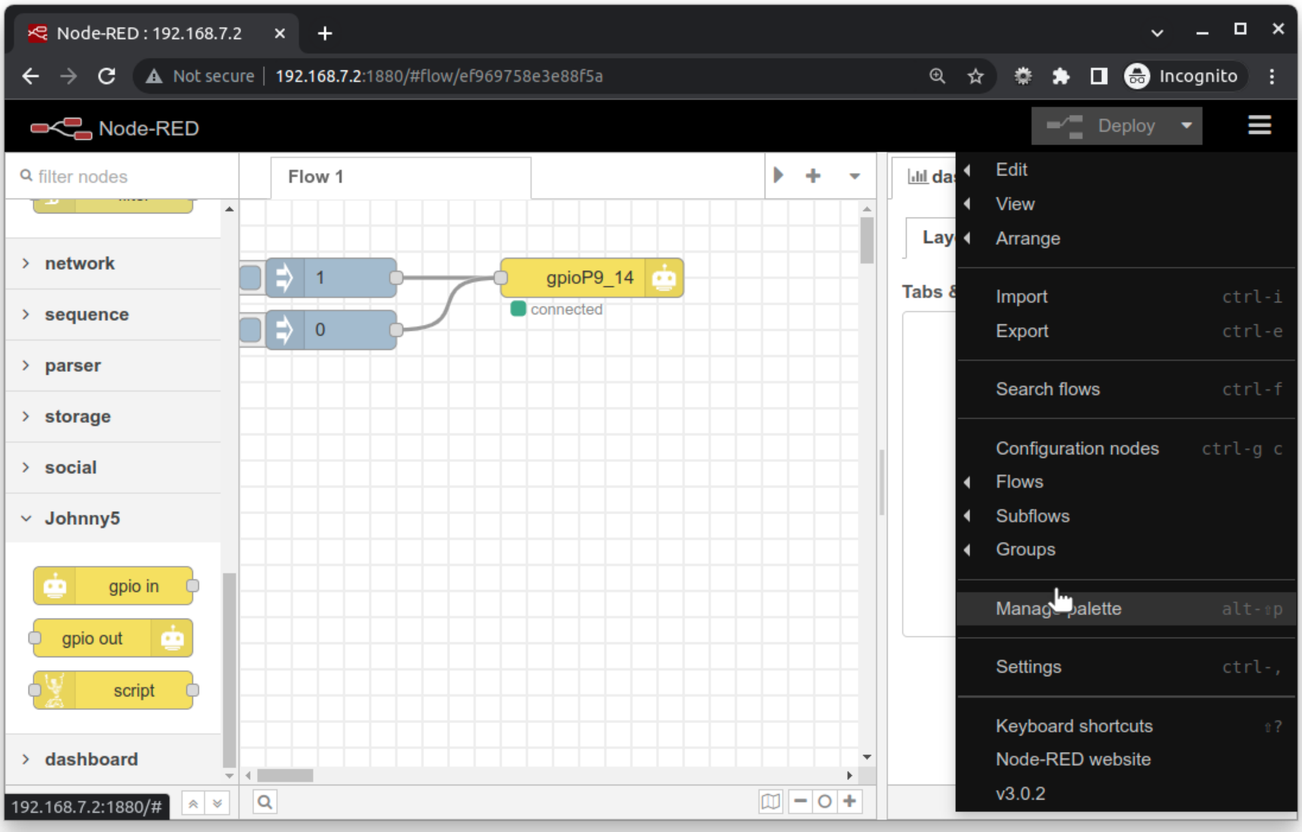


Figure 54. Opening Manage palette

NOTE

If **Manage palette** doesn't appear. Run the following in a terminal widow.

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install npm</strong>
bone$ <strong>sudo systemctl restart nodered.service</strong>
</pre>
```

Wait a bit, then refresh the browser window running NODE-RED.

Once **Manage palette** is open, click on the **Install** tab and search for **twitter**. Click the **Install** button on the first response. Wait a bit.

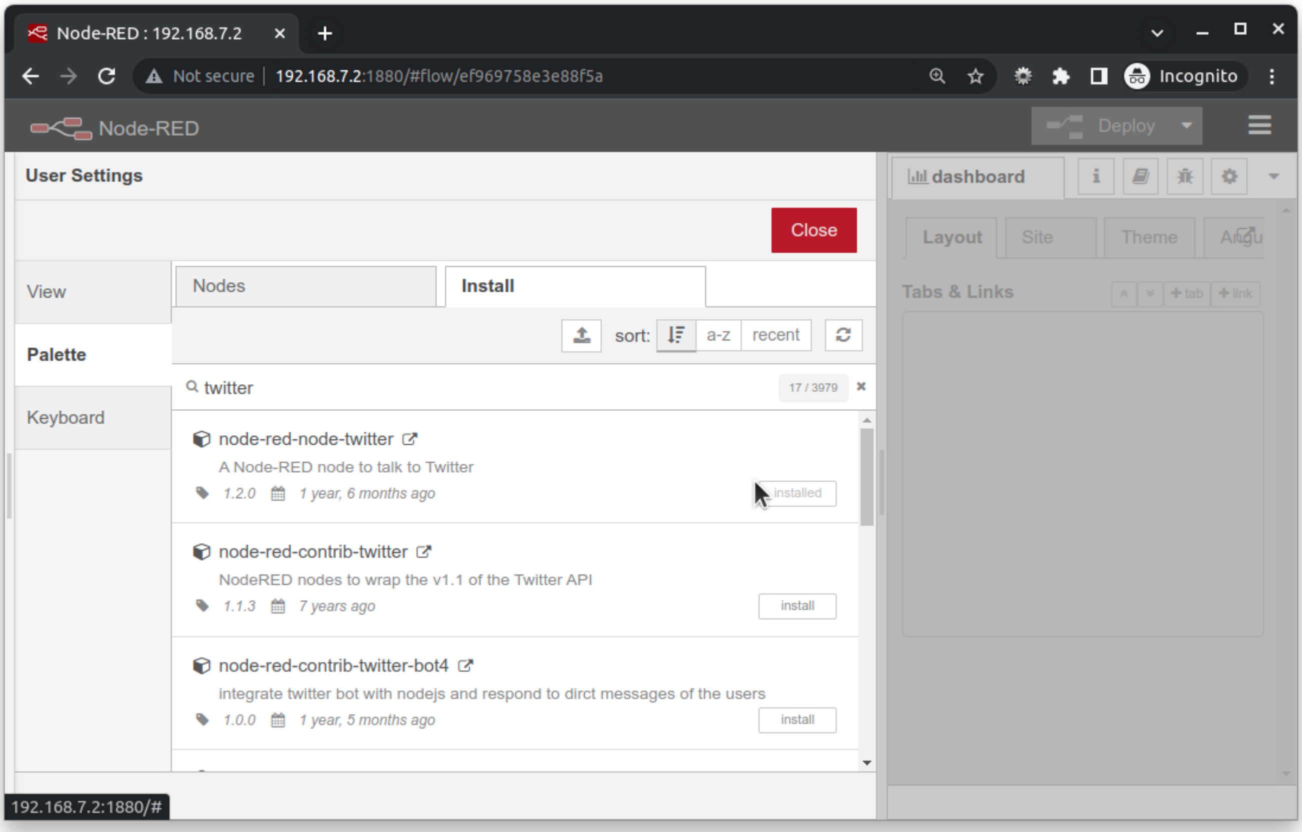


Figure 55. Installing Twitter

Click **Close** once it's installed.

Building a Node-RED Flow - Twitter

The example in this recipe builds a Node-RED flow that will blink an LED whenever a certain hashtag is tweeted. But first, you need to set up the Node-RED flow with the twitter node:

1. In Node-RED, scroll down until you see the **social** nodes on the left side of the page.
2. Drag the **twitter in** node to the canvas, as shown in Node-RED twitter node.

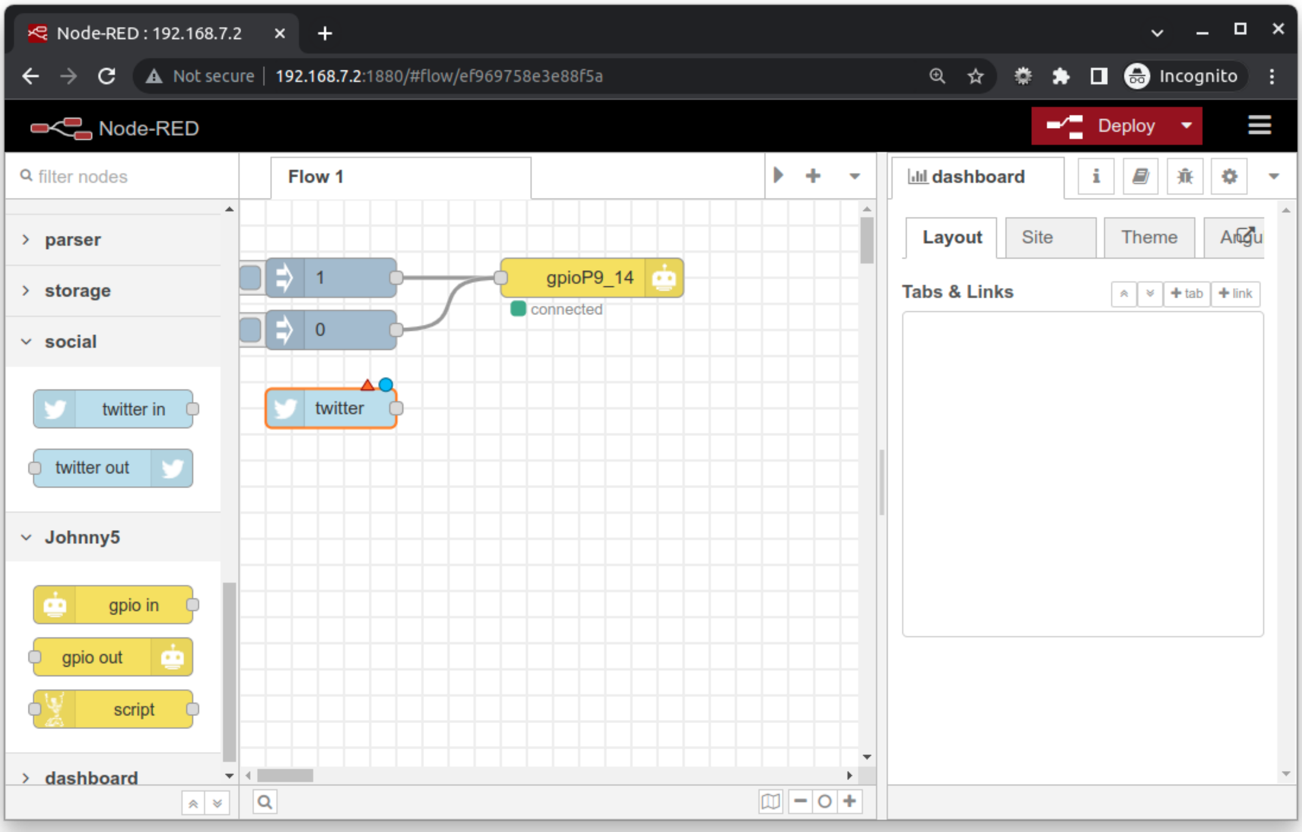


Figure 56. Node-RED twitter node

3. Authorize Twitter by double-clicking the twitter node. You'll see the screen shown in Node-RED Twitter authorization, step 1.

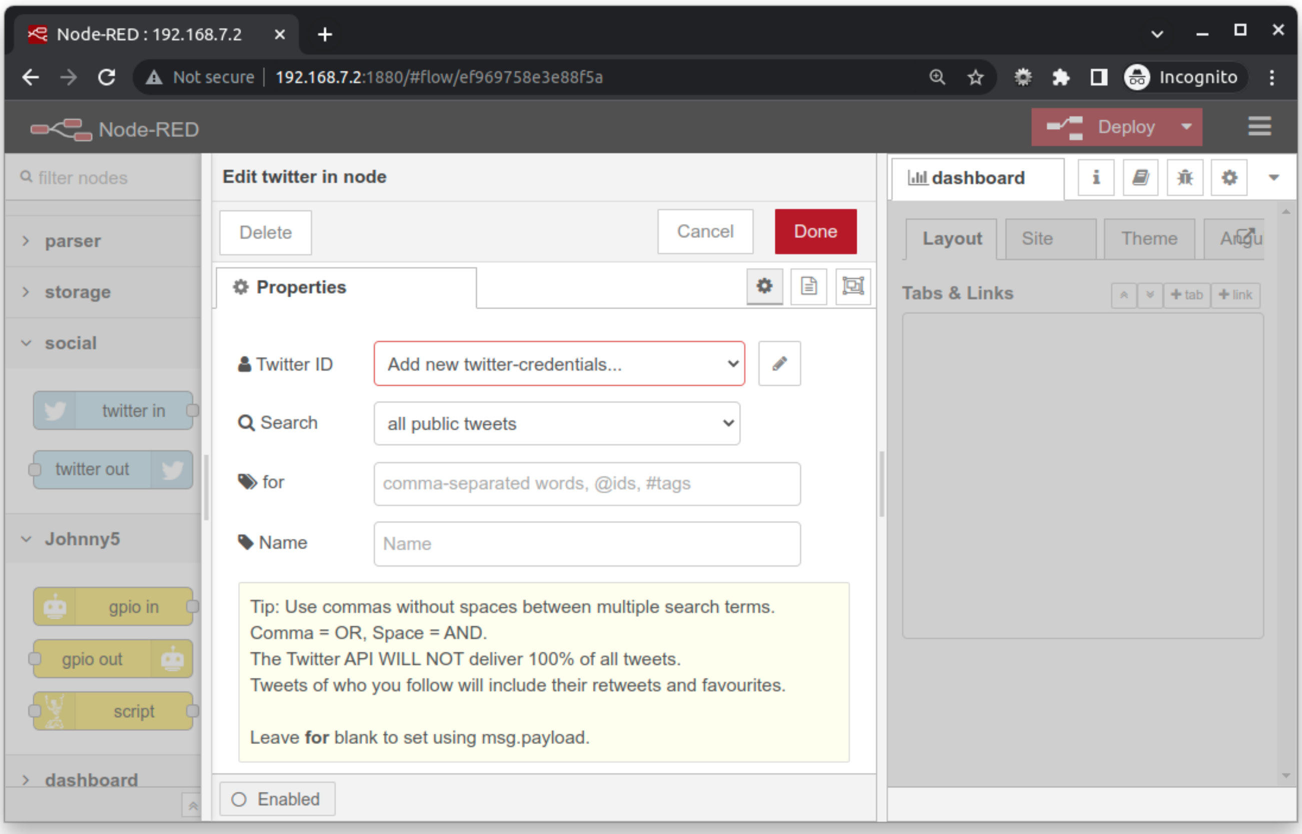


Figure 57. Node-RED Twitter authorization, step 1

4. Click the pencil button to bring up the dialog box shown in Node-RED twitter authorization, step 2.

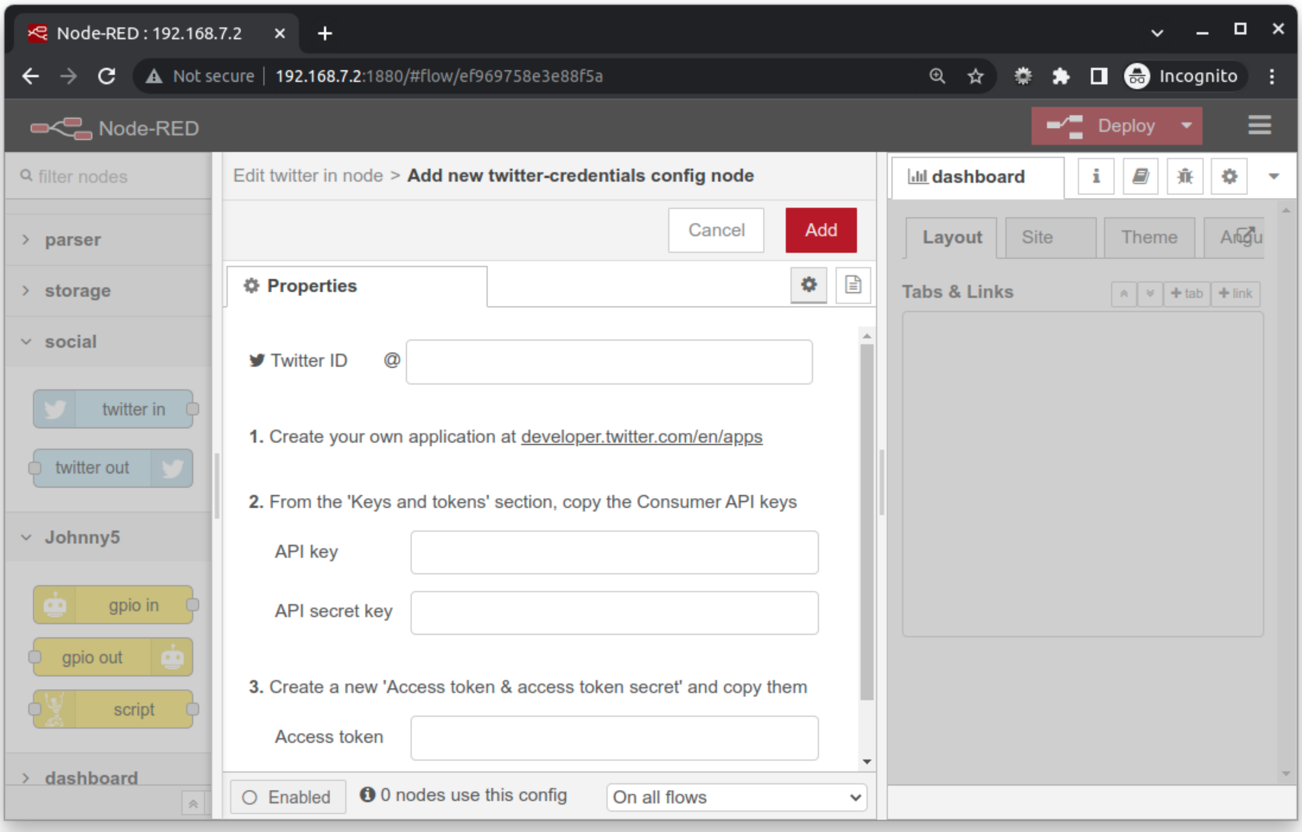


Figure 58. Node-RED twitter authorization, step 2

5. Click on the link, http://developer.twitter.com/en/apps shown in Node-RED twitter authorization, step 2, and follow the steps to set up your twitter app. Copy your **API key** etc. from there.

7. 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 **Done** button.

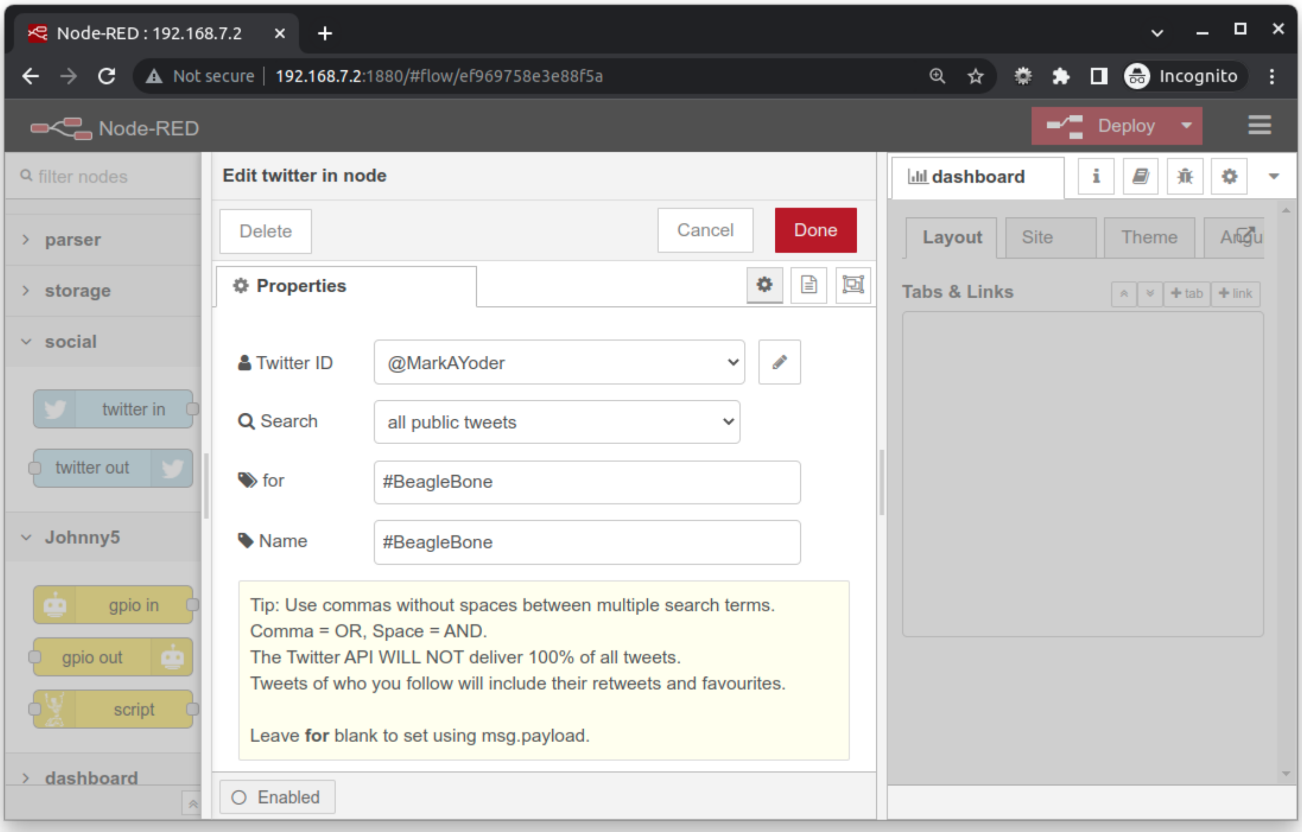


Figure 59. Node-RED adding the #BeagleBone hashtag

8. Go back to the left panel, scroll up to the top, and then drag the **debug** node to the canvas.
9. Connect the two nodes by clicking and dragging (Node-RED Twitter adding debug node and connecting).

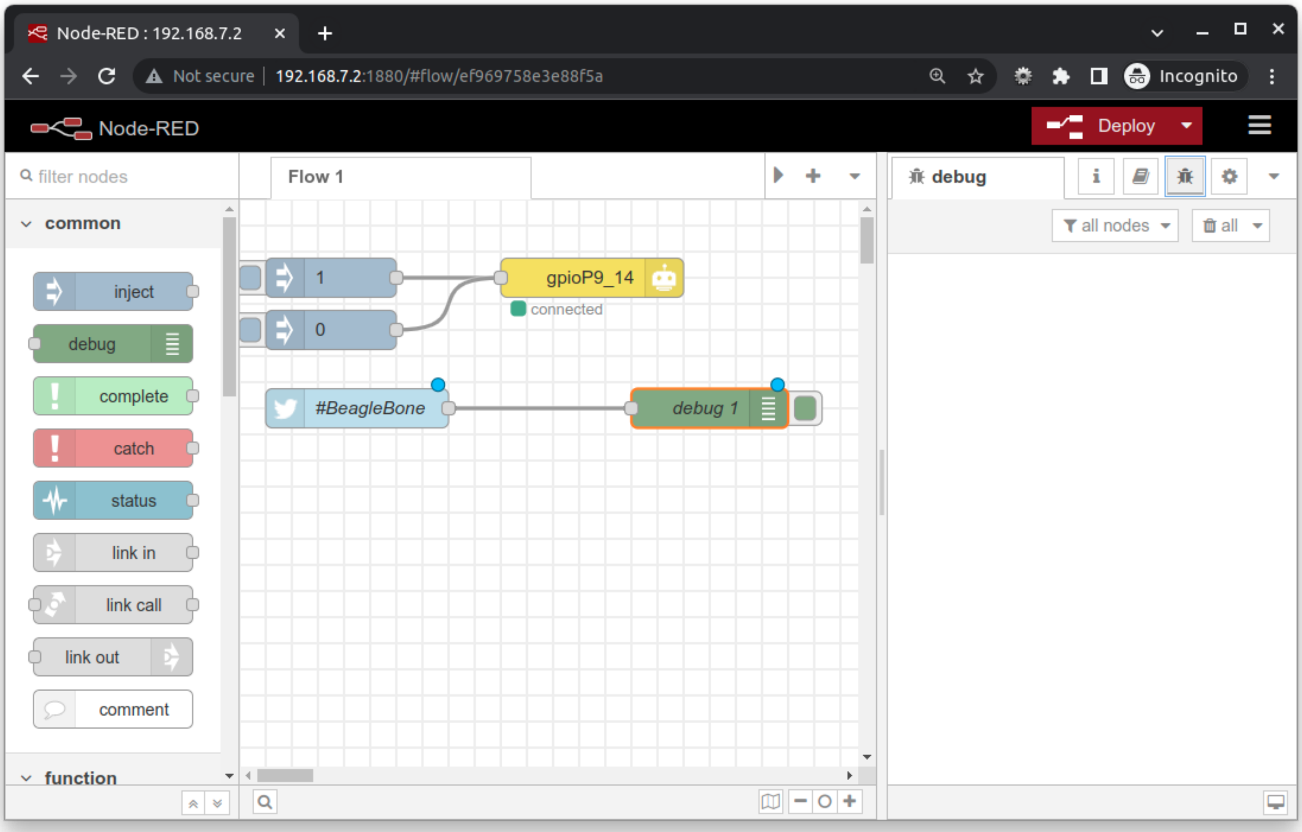


Figure 60. Node-RED Twitter adding debug node and connecting

10. In the right panel, in the upper-right corner, click the "debug" tab.
11. Finally, click the **Deploy** button.

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.

Adding an LED Toggle

Now, we're ready to add an LED to response to tweets:

1. Wire up an LED as shown in Toggling an External LED. Mine is wired to P9_14.
2. Scroll down the left panel and drag the **trigger** node to the canvas and wire it (Node-RED adding trigger node).

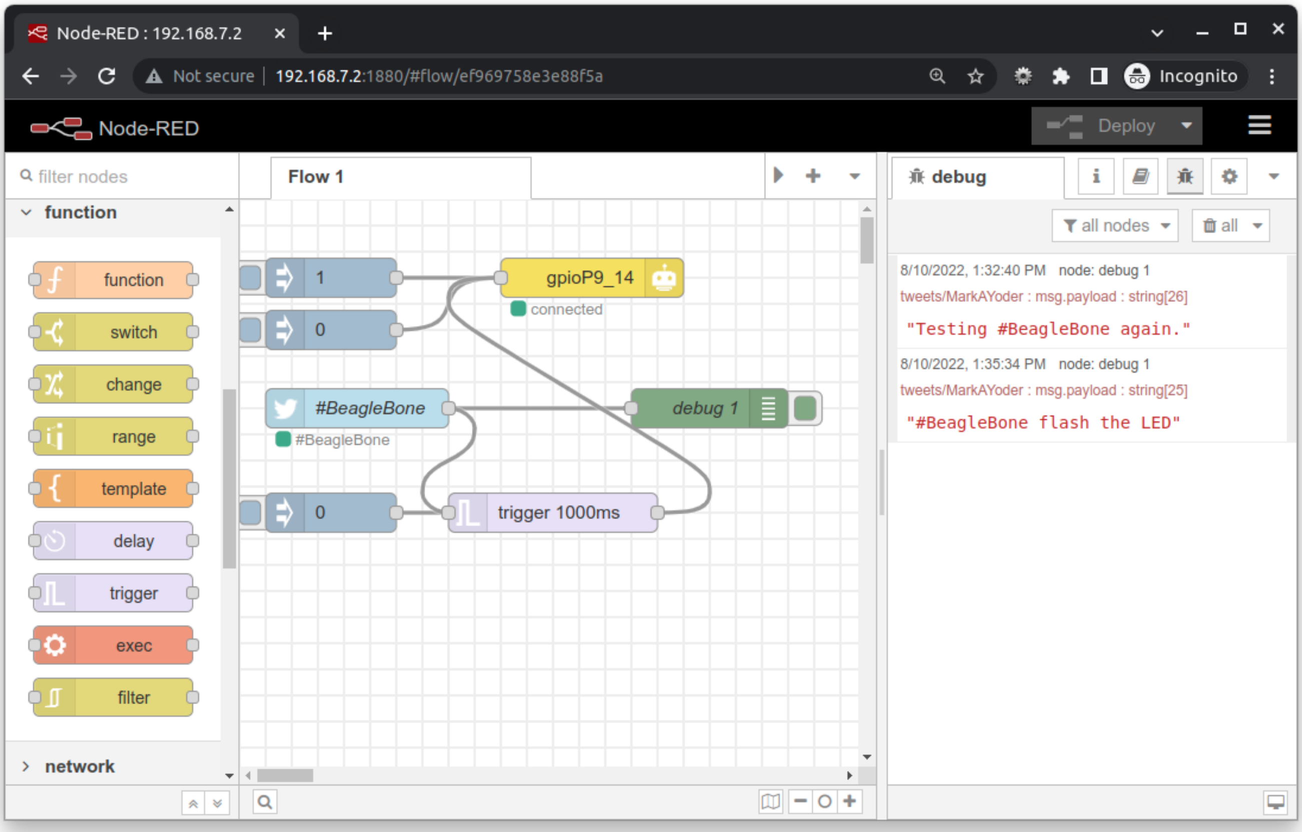


Figure 61. Node-RED adding trigger node

3. Double-click the trigger node and set the time to **1000 ms**.

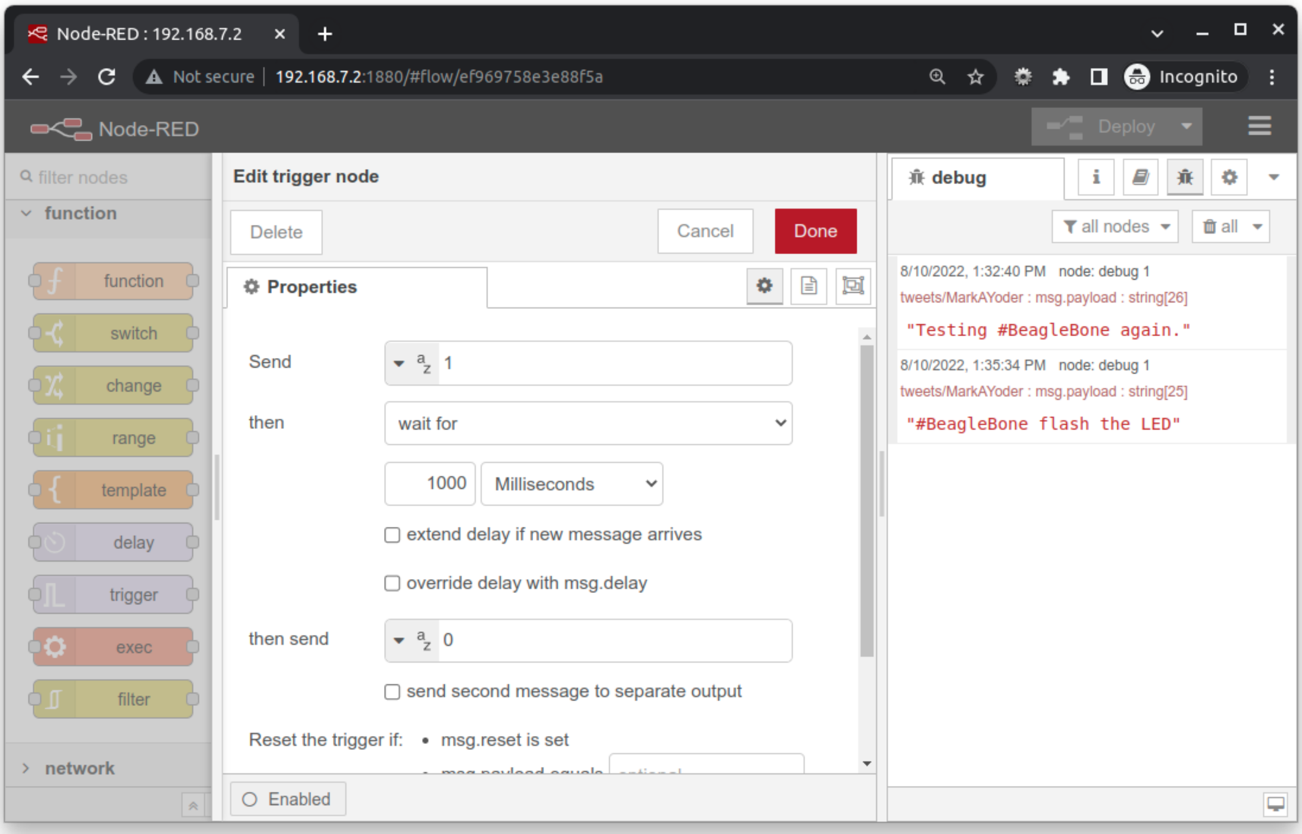


Figure 62. Node-RED adding trigger configuration

4. Click **Done** and then **Deploy**.

Test again. The LED will turn one for 1 second 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.

The Kernel

Introduction

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.

NOTE	We assume here that you are logged on to your Bone as debian and have superuser privileges. You also need to be logged in to your Linux host computer as a nonsuperuser.

Updating the Kernel

Problem

You have an out-of-date kernel and want to want to make it current.

Solution

Use the following command to determine which kernel you are running:

```
<pre data-type="programlisting">
bone$ <strong>uname -a</strong>
Linux beaglebone <strong>3.8.13-bone67</strong> #1 SMP Wed Sep 24 21:30:03 UTC 2014 armv7l
 GNU/Linux
</pre>
```

The 3.8.13-bone67 string is the kernel version (which is really old).

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). Next follow Finding the Latest Version of the OS for Your Bone and Running the Latest Version of the OS on Your Bone and Updating the OS on Your Bone to find the current image with the current kernel, install it and update it.

Building and Installing Kernel Modules

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.

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, download the cookbook repository and:

```
<pre data-type="programlisting">
bone$ <strong>cd ~/BoneCookbook/docs/07kernel/code</strong>
</pre>
```

And look at *hello.c*.

Example 52. Simple Kernel Module (hello.c)

```
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */

static int __init hello_start(void)
{
    printk(KERN_INFO "Loading hello module...\n");
    printk(KERN_INFO "Hello, World!\n");
    return 0;
}

static void __exit hello_end(void)
{
    printk(KERN_INFO "Goodbye Boris\n");
}

module_init(hello_start);
module_exit(hello_end);

MODULE_AUTHOR("Boris Houndleroy");
MODULE_DESCRIPTION("Hello World Example");
MODULE_LICENSE("GPL");
```

When compiling on the Bone, all you need to do is load the Kernel Headers for the version of the kernel you're running:

```
<pre data-type="programlisting">
bone$ <strong>sudo apt install linux-headers-`uname -r`</strong>
</pre>
```

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.

Next look at *Makefile*.

Example 53. Simple Kernel Module (Makefile)

```
obj-m := hello.o
KDIR  := /lib/modules/$(shell uname -r)/build

all:
<TAB>make -C $(KDIR) M=$$PWD

clean:
<TAB>rm hello.mod.c hello.o modules.order hello.mod.o Module.symvers
```

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:

```
<pre data-type="programlisting">
bone$ <strong>make</strong>
make -C /lib/modules/5.10.120-ti-r48/build M=$PWD
make[1]: Entering directory '/usr/src/linux-headers-5.10.120-ti-r48'
  CC [M]  /home/debian/BoneCookbook/docs/07kernel/code/hello.o
  MODPOST /home/debian/BoneCookbook/docs/07kernel/code/Module.symvers
  CC [M]  /home/debian/BoneCookbook/docs/07kernel/code/hello.mod.o
  LD [M]  /home/debian/BoneCookbook/docs/07kernel/code/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.10.120-ti-r48'
bone$ <strong>ls</strong>
Makefile        hello.c   hello.mod.c  hello.o
Module.symvers  hello.ko  hello.mod.o  modules.order
</pre>
```

Notice that several files have been created. *hello.ko* is the one you want. Try a couple of commands with it:

```
<pre data-type="programlisting">
bone$ <strong>modinfo hello.ko</strong>
filename:       /home/debian/BoneCookbook/docs/07kernel/code/hello.ko
license:        GPL
description:    Hello World Example
author:         Boris Houndleroy
```

```
depends:
name:           hello
vermagic:       5.10.120-ti-r48 SMP preempt mod_unload modversions ARMv7 p2v8
bone$ <strong>sudo insmod hello.ko</strong>
bone$ <strong>dmesg -H | tail -4</strong>
[  +0.000024] Bluetooth: BNEP filters: protocol multicast
[  +0.000034] Bluetooth: BNEP socket layer initialized
[Aug 8 13:49] Loading hello module...
[  +0.000022] Hello, World!

</pre>
```

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!

Controlling LEDs by Using SYSFS Entries

Problem

You want to control the onboard LEDs from the command line.

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:

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/class/leds/</strong>
bone$ <strong>ls</strong>
beaglebone:green:usr0  beaglebone:green:usr2
beaglebone:green:usr1  beaglebone:green:usr3
</pre>
```

What you are seeing are four directories, one for each onboard LED. Now try this:

```
<pre data-type="programlisting">
bone$ <strong>cd beaglebone\:green\:usr0</strong>
bone$ <strong>ls</strong>
brightness  device  max_brightness  power  subsystem  trigger  uevent
bone$ <strong>cat trigger</strong>
none nand-disk mmc0 mmc1 timer oneshot [heartbeat]
     backlight gpio cpu0 default-on transient
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>echo none > trigger</strong>
bone$ <strong>cat trigger</strong>
[none] nand-disk mmc0 mmc1 timer oneshot heartbeat
       backlight gpio cpu0 default-on transient
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>echo 1 > brightness</strong>
bone$ <strong>echo 0 > brightness</strong>
</pre>
```

The LED should be turning on and off with the commands.

Controlling GPIOs by Using SYSFS Entries

Problem

You want to control a GPIO pin from the command line.

Solution

Controlling LEDs by Using SYSFS Entries introduces the sysfs. This recipe shows how to read and write a GPIO pin.

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. Or you can follow these Mapping Header Numbers to gpio Numbers.

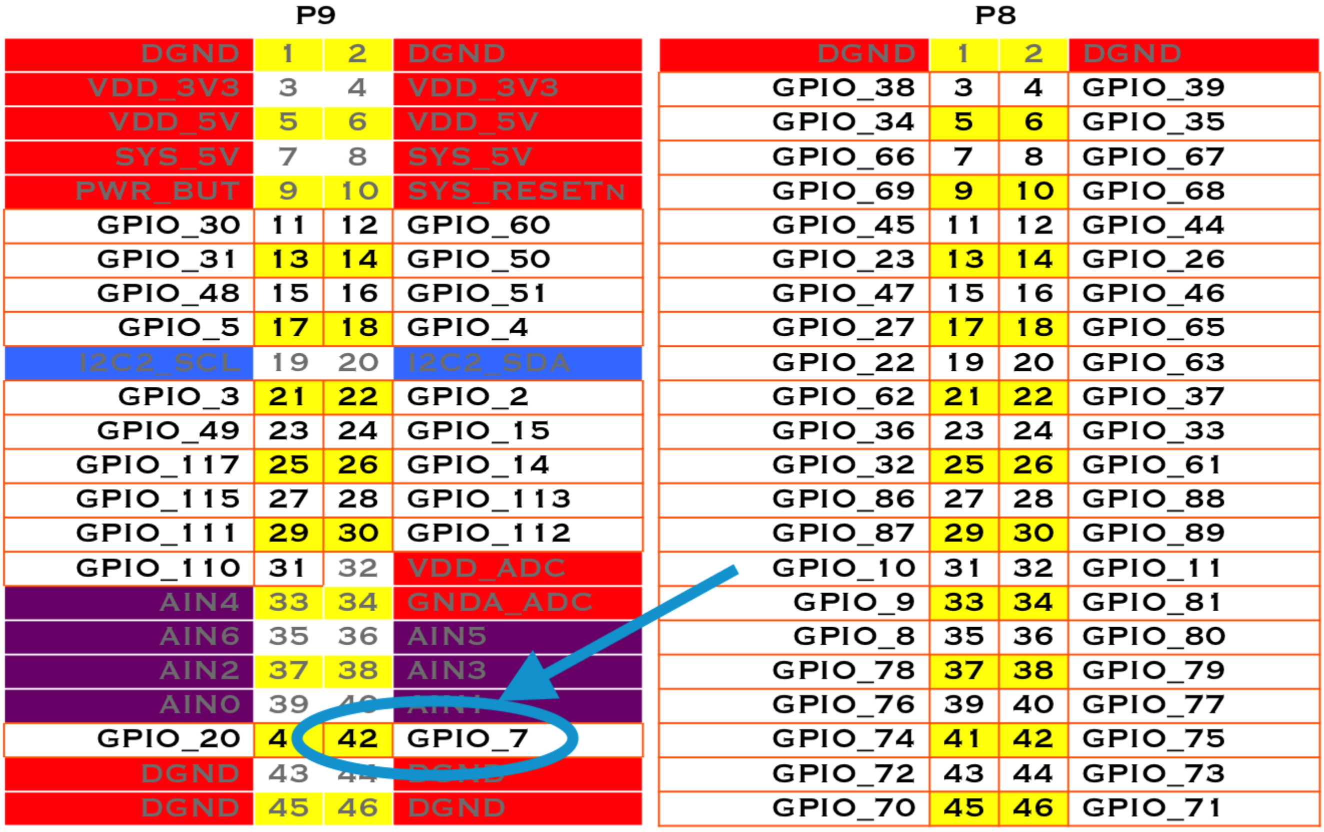


Figure 63. Mapping P9_42 header position to GPIO 7

Next, change to the GPIO sysfs directory:

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/class/gpio/</strong>
bone$ <strong>ls</strong>
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
</pre>
```

The ls command shows all the GPIO pins that have be exported. In this case, none have, so you see only the four GPIO controllers. Export using the export command:

```
<pre data-type="programlisting">
bone$ <strong>echo 7 > export</strong>
bone$ <strong>ls</strong>
export  gpio7  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
</pre>
```

Now you can see the *gpio7* directory. Change into the *gpio7* directory and look around:

```
<pre data-type="programlisting">
bone$ <strong>cd gpio7</strong>
bone$ <strong>ls</strong>
active_low  direction  edge  power  subsystem  uevent  value
bone$ <strong>cat direction</strong>
in
bone$ <strong>cat value</strong>
0
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>cat value</strong>
1
</pre>
```

The 1 informs you that the switch is pressed. When you are done with GPIO 7, you can always unexport it:

```
<pre data-type="programlisting">
bone$ <strong>cd ..</strong>
bone$ <strong>echo 7 > unexport</strong>
bone$ <strong>ls</strong>
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>cd /sys/class/gpio/</strong>
bone$ <strong>echo 50 > export</strong>
bone$ <strong>ls</strong>
gpio50  gpiochip0  gpiochip32  gpiochip64  gpiochip96
bone$ <strong>cd gpio50</strong>
bone$ <strong>ls</strong>
active_low  direction  edge  power  subsystem  uevent  value
bone$ <strong>cat direction</strong>
in
</pre>
```

By default, P9_14 is set as an input. Switch it to an output and turn it on:

```
<pre data-type="programlisting">
bone$ <strong>echo out > direction</strong>
bone$ <strong>echo 1 > value</strong>
bone$ <strong>echo 0 > value</strong>
</pre>
```

The LED turns on when a 1 is written to value and turns off when a 0 is written.

Compiling the Kernel

Problem

You need to download, patch, and compile the kernel from its source code.

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 resourses.

Downloading and Compiling the Kernel

To download and compile the kernel, follow these steps:

```
host$ git clone git@github.com:RobertCNelson/ti-linux-kernel-dev.git ①
host$ cd ti-linux-kernel-dev/
host$ git tag ②
host$ git checkout 5.10.120-ti-r48 -b 5.10.120-ti-r48 ③
host$ time ./build_deb.sh ④
```

① The first command clones a repository with the tools to build the kernel for the Bone.

② This command lists all the different versions of the kernel that you can build. You'll need to pick one of these. How do you know which one to pick? A good first step is to choose the one you are currently running. uname -a will reveal which one that is. When you are able to reproduce the current kernel, go to Linux Kernel Newbies to see what features are available in other kernels. LinuxChanges shows the features in the newest kernel and LinuxVersions links to features of pervious kernels.

③ When you know which kernel to try, use git checkout to check it out. This command checks out at tag 3.8.13-bone60 and creates a new branch, v3.8.13-bone60.

④ build_kernel is the master builder. If needed, it will download the cross compilers needed to compile the kernel (linaro [http://www.linaro.org/] 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

Here is another repo to use for non-ti kernels.

```
host$ git clone https://github.com/RobertCNelson/bb-kernel.git ①
host$ cd bb-kernel
host$ git tag ②
host$ git checkout 3.8.13-bone60 -b v3.8.13-bone60 ③
host$ ./build_kernel.sh ④
```

After the kernel is patched, you'll see a screen similar to Kernel configuration menu, on which you can configure the kernel.

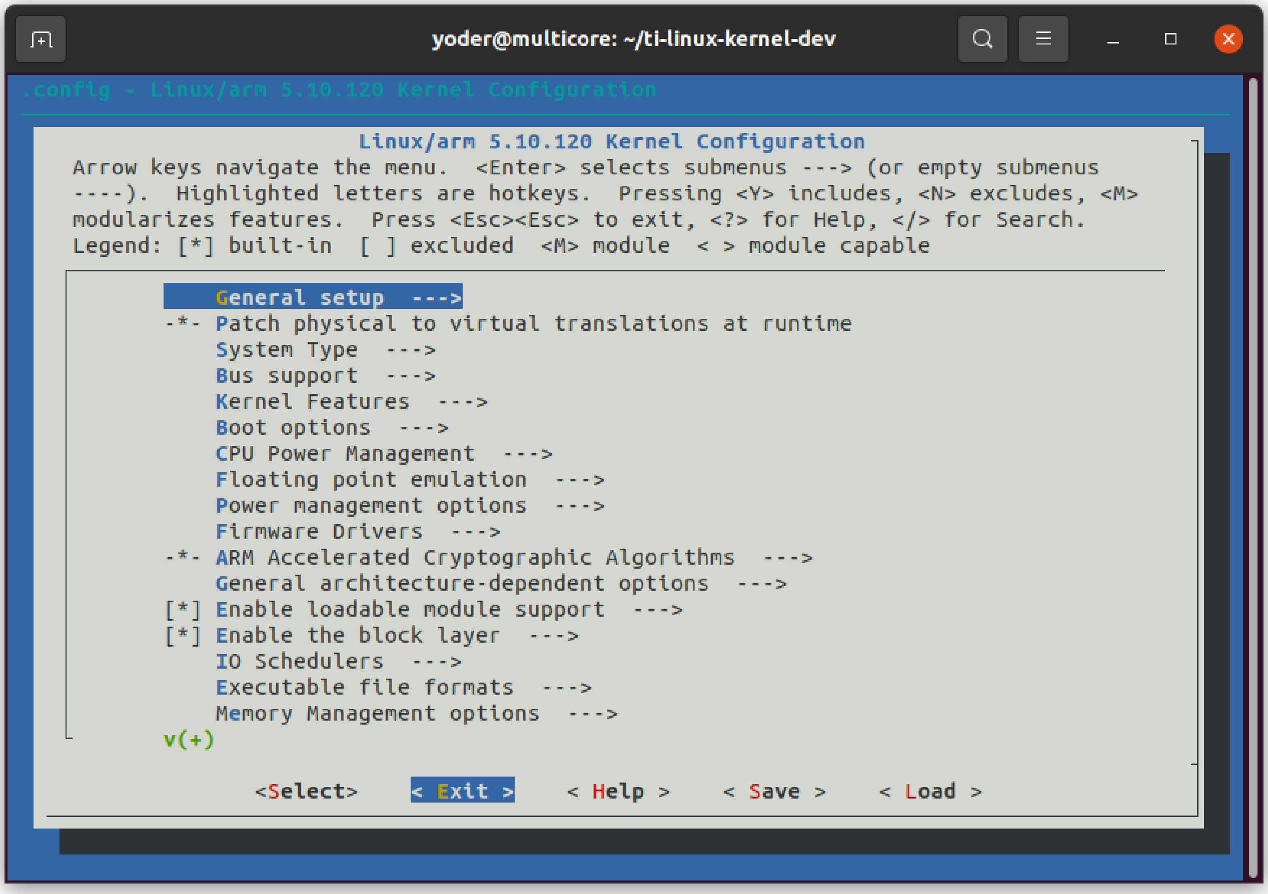


Figure 64. Kernel configuration menu

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.

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:

```
<pre data-type="programlisting">
host$ <strong>cd ti-linux-kernel-dev/deploy</strong>
host$ <strong>ls -sh</strong>
total 42M
7.7M linux-headers-5.10.120-ti-r48_1xross_armhf.deb  8.0K linux-
upstream_1xross_armhf.buildinfo
 33M linux-image-5.10.120-ti-r48_1xross_armhf.deb    4.0K linux-
upstream_1xross_armhf.changes
1.1M linux-libc-dev_1xross_armhf.deb
</pre>
```

The linux-image- file is the one we want. It contains over 3000 files.

```
<pre data-type="programlisting">
host$ <strong>dpkg -c linux-image-5.8.11-bone17_1xross_armhf.deb  | wc</strong>
   3177   19062  371016
</pre>
```

The dpkg -c command lists the contents (all the files) in the .deb file and the wc counts all the lines in the output. You can see those files with:

```
<pre data-type="programlisting">
bone$ <strong>dpkg -c linux-image-5.8.11-bone17_1xross_armhf.deb  | less</strong>
drwxr-xr-x root/root         0 2022-08-08 22:43 ./
drwxr-xr-x root/root         0 2022-08-08 22:43 ./boot/
-rw-r--r-- root/root   4749032 2022-08-08 22:43 ./boot/System.map-5.10.120-ti-r48
-rw-r--r-- root/root    190604 2022-08-08 22:43 ./boot/config-5.10.120-ti-r48
drwxr-xr-x root/root         0 2022-08-08 22:43 ./boot/dtbs/
drwxr-xr-x root/root         0 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/
-rwxr-xr-x root/root     90652 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/am335x-baltos-
ir2110.dtb
-rwxr-xr-x root/root     91370 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/am335x-baltos-
ir3220.dtb
-rwxr-xr-x root/root     91641 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/am335x-baltos-
ir5221.dtb
-rwxr-xr-x root/root     88692 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/am335x-
base0033.dtb
-rwxr-xr-x root/root    197471 2022-08-08 22:43 ./boot/dtbs/5.10.120-ti-r48/am335x-bone-
uboot-univ.dtb
</pre>
```

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.

```
<pre data-type="programlisting">
host$ <strong>scp linux-image-5.10.120-ti-r48_1xross_armhf.deb bone:.</strong>
</pre>
```

Hint: You might have to use debian@192.168.7.2 for bone if you haven't set everything up.

Now ssh to the bone.

```
<pre data-type="programlisting">
host$ <strong>ssh bone</strong>
bone$ <strong>ls -sh</strong>
bin  exercises linux-image-5.10.120-ti-r48_1xross_armhf.deb
</pre>
```

Now install it.

```
<pre data-type="programlisting">
bone$ <strong>sudo dpkg --install linux-image-5.10.120-ti-r48_1xross_armhf.deb</strong>
```

```
</pre>
```

Wait a while. Once done check /boot.

```
<pre data-type="programlisting">
bone$ <strong>ls -sh /boot</strong>
total 122M
188K config-5.10.120-ti-r46      7.6M initrd.img-5.10.120-ti-rt-r47  4.0K uEnv.txt
188K config-5.10.120-ti-r47       13M initrd.img-5.10.120-ti-rt-r48  4.0K uEnv.txt.orig
188K config-5.10.120-ti-r48      4.0K SOC.sh                         9.9M vmlinuz-
5.10.120-ti-r46
188K config-5.10.120-ti-rt-r47   4.5M System.map-5.10.120-ti-r46      11M vmlinuz-
5.10.120-ti-r47
188K config-5.10.120-ti-rt-r48   4.6M System.map-5.10.120-ti-r47      11M vmlinuz-
5.10.120-ti-r48
4.0K dtbs                        4.6M System.map-5.10.120-ti-r48      11M vmlinuz-
5.10.120-ti-rt-r47
7.5M initrd.img-5.10.120-ti-r46  4.6M System.map-5.10.120-ti-rt-r47   11M vmlinuz-
5.10.120-ti-rt-r48
7.6M initrd.img-5.10.120-ti-r47  4.6M System.map-5.10.120-ti-rt-r48
 13M initrd.img-5.10.120-ti-r48  4.0K uboot

</pre>
```

You see the new kernel files. Check uEnv.txt.

```
<pre data-type="programlisting">
bone$ <strong>head /boot/uEnv.txt</strong>
#Docs: http://elinux.org/Beagleboard:U-boot_partitioning_layout_2.0
uname_r=5.10.120-ti-r48
# uname_r=5.10.120-ti-rt-r47
</pre>
```

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.

```
<pre data-type="programlisting">
bone$ <strong>sudo reboot</strong>
bone$ <strong>uname -a</strong>
Linux breadboard-home 5.10.120-ti-r48 #1xross SMP PREEMPT Mon Aug 8 18:30:51 EDT 2022
armv7l GNU/Linux
</pre>
```

Using the Installed Cross Compiler

Problem

You have followed the instructions in Compiling the Kernel and want to use the cross compiler it has downloaded.

TIP

> You can cross-compile without installing the entire kernel source by running the following:
>
> ```
> <pre data-type="programlisting">
> host$ sudo apt install gcc-arm-linux-gnueabihf
> </pre>
> ```
>
> Then skip down to Setting Up Variables.

Solution

Compiling the Kernel installs a cross compiler, but you need to set up a couple of things so that it can be found. Compiling the Kernel installed the kernel and other tools in a directory called *ti-linux-kernel-dev*. Run the following commands to find the path to the cross compiler:

```
<pre data-type="programlisting">
host$ <strong>cd ti-linux-kernel-dev/dl</strong>
host$ <strong>ls</strong>
gcc-10.3.0-nolibc  x86_64-gcc-10.3.0-nolibc-arm-linux-gnueabi.tar.xz
</pre>
```

Here, the path to the cross compiler contains the version number of the compiler. Yours might be different from mine. cd into it:

```
<pre data-type="programlisting">
host$ <strong>cd gcc-10.3.0-nolibc/arm-linux-gnueabi</strong>
host$ <strong>ls</strong>
2020.10.3.0-arm-linux-gnueabi  arm-linux-gnueabi  bin  include  lib  libexec  share  x
</pre>
```

At this point, we are interested in what's in *bin*:

```
<pre data-type="programlisting">
host$ <strong>cd bin</strong>
host$ <strong>ls</strong>
arm-linux-gnueabi-addr2line   arm-linux-gnueabi-gcc-nm      arm-linux-gnueabi-nm
arm-linux-gnueabi-ar          arm-linux-gnueabi-gcc-ranlib  arm-linux-gnueabi-objcopy
arm-linux-gnueabi-as          arm-linux-gnueabi-gcov        arm-linux-gnueabi-objdump
arm-linux-gnueabi-c++filt     arm-linux-gnueabi-gcov-dump   arm-linux-gnueabi-ranlib
arm-linux-gnueabi-cpp         arm-linux-gnueabi-gcov-tool   arm-linux-gnueabi-readelf
arm-linux-gnueabi-elfedit     arm-linux-gnueabi-gprof       arm-linux-gnueabi-size
arm-linux-gnueabi-gcc         arm-linux-gnueabi-ld          arm-linux-gnueabi-strings
arm-linux-gnueabi-gcc-10.3.0  arm-linux-gnueabi-ld.bfd      arm-linux-gnueabi-strip
arm-linux-gnueabi-gcc-ar      arm-linux-gnueabi-lto-dump
</pre>
```

What you see are all the cross-development tools. You need to add this directory to the $PATH the shell uses to find the commands it runs:

```
<pre data-type="programlisting">
host$ <strong>pwd</strong>
```

```
/home/yoder/ti-linux-kernel-dev/dl/gcc-10.3.0-nolibc/arm-linux-gnueabi/bin

host$ <strong>echo $PATH</strong>
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/
snap/bin
</pre>
```

The first command displays the path to the directory where the cross-development tools are located. The second shows which directories are searched to find commands to be run. Currently, the cross-development tools are not in the $PATH. Let's add it:

```
<pre data-type="programlisting">
host$ <strong>export PATH=`pwd`:$PATH</strong>
host$ <strong>echo $PATH</strong>
/home/yoder/ti-linux-kernel-dev/dl/gcc-10.3.0-nolibc/arm-linux-
gnueabi/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/l
ocal/games:/snap/bin
</pre>
```

NOTE	Those are backtick characters (left of the "1" key on your keyboard) around pwd.

The second line shows the $PATH now contains the directory with the cross-development tools.

Setting Up Variables

Now, set up a couple of variables to know which compiler you are using:

```
<pre data-type="programlisting">
host$ <strong>export ARCH=arm</strong>
host$ <strong>export CROSS_COMPILE=arm-linux-gnueabi-</strong>
</pre>
```

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*.

Example 54. Simple helloWorld.c to test cross compiling (helloWorld.c)

```
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Hello, World! \n");
}
```

You can then cross-compile by using the following commands:

```
<pre data-type="programlisting">
host$ <strong>${CROSS_COMPILE}gcc helloWorld.c</strong>
host$ <strong>file a.out</strong>
```

```
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
</pre>
```

The file command shows that a.out was compiled for an ARM processor.

Applying Patches

Problem

You have a patch file that you need to apply to the kernel.

Solution

Simple kernel patch file (hello.patch) shows a patch file that you can use on the kernel.

Example 55. Simple kernel patch file (hello.patch)

```
From eaf4f7ea7d540bc8bb57283a8f68321ddb4401f4 Mon Sep 17 00:00:00 2001
From: Jason Kridner <jdk@ti.com>
Date: Tue, 12 Feb 2013 02:18:03 +0000
Subject: [PATCH] hello: example kernel modules

---
 hello/Makefile   |    7 +++++++
 hello/hello.c    |   18 ++++++++++++++++++
 2 files changed, 25 insertions(+), 0 deletions(-)
 create mode 100644 hello/Makefile
 create mode 100644 hello/hello.c

diff --git a/hello/Makefile b/hello/Makefile
new file mode 100644
index 0000000..4b23da7
--- /dev/null
+++ b/hello/Makefile
@@ -0,0 +1,7 @@
+obj-m := hello.o
+
+PWD   := $(shell pwd)
+KDIR  := ${PWD}/..
+
+default:
+   make -C $(KDIR) SUBDIRS=$(PWD) modules
diff --git a/hello/hello.c b/hello/hello.c
new file mode 100644
index 0000000..157d490
--- /dev/null
+++ b/hello/hello.c
```

```
@@ -0,0 +1,22 @@
+#include <linux/module.h>       /* Needed by all modules */
+#include <linux/kernel.h>       /* Needed for KERN_INFO */
+#include <linux/init.h>         /* Needed for the macros */
+
+static int __init hello_start(void)
+{
+    printk(KERN_INFO "Loading hello module...\n");
+    printk(KERN_INFO "Hello, World!\n");
+    return 0;
+}
+
+static void __exit hello_end(void)
+{
+    printk(KERN_INFO "Goodbye Boris\n");
+}
+
+module_init(hello_start);
+module_exit(hello_end);
+
+MODULE_AUTHOR("Boris Houndleroy");
+MODULE_DESCRIPTION("Hello World Example");
+MODULE_LICENSE("GPL");
```

Here's how to use it:

1. Install the kernel sources (Compiling the Kernel).
2. Change to the kernel directory (cd ti-linux-kernel-dev/KERNEL).
3. Add Simple kernel patch file (hello.patch) to a file named *hello.patch* in the *ti-linux-kernel-dev/KERNEL* directory.
4. Run the following commands:

```
<pre data-type="programlisting">
host$ <strong>cd ti-linux-kernel-dev/KERNEL</strong>
host$ <strong>patch -p1 &lt; hello.patch</strong>
patching file hello/Makefile
patching file hello/hello.c
</pre>
```

The output of the patch command apprises you of what it's doing. Look in the *hello* directory to see what was created:

```
<pre data-type="programlisting">
host$ <strong>cd hello</strong>
host$ <strong>ls</strong>
hello.c  Makefile
</pre>
```

Discussion

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.

Creating Your Own Patch File

Problem

You made a few changes to the kernel, and you want to share them with your friends.

Solution

Create a patch file that contains just the changes you have made. Before making your changes, check out a new branch:

```
<pre data-type="programlisting">
host$ <strong>cd ti-linux-kernel-dev/KERNEL</strong>
host$ <strong>git status</strong>
# On branch master
nothing to commit (working directory clean)
</pre>
```

Good, so far no changes have been made. Now, create a new branch:

```
<pre data-type="programlisting">
host$ <strong>git checkout -b hello1</strong>
host$ <strong>git status</strong>
# On branch hello1
nothing to commit (working directory clean)
</pre>
```

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:

```
<pre data-type="programlisting">
host$ <strong>cd ti-linux-kernel-dev/KERNEL/drivers/char/</strong>
host$ <strong>git status</strong>
# 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)
#
#   examples/
no changes added to commit (use "git add" and/or "git commit -a")
```

```
</pre>
```

Add the files that were created and commit them:

```
<pre data-type="programlisting">
host$ <strong>git add Kconfig Makefile examples</strong>
host$ <strong>git status</strong>
# 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$ <strong>git commit -m "Files for hello1 kernel module"</strong>
[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
 </pre>
```

Finally, create the patch file:

```
<pre data-type="programlisting">
host$ <strong>git format-patch master --stdout &gt; hello1.patch</strong>
 </pre>
```

Real-Time I/O

Introduction

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 Python 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.

I/O with Python

Problem

You want to read an input pin and write it to the output as quickly as possible with Python.

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
- 220 Ω 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.

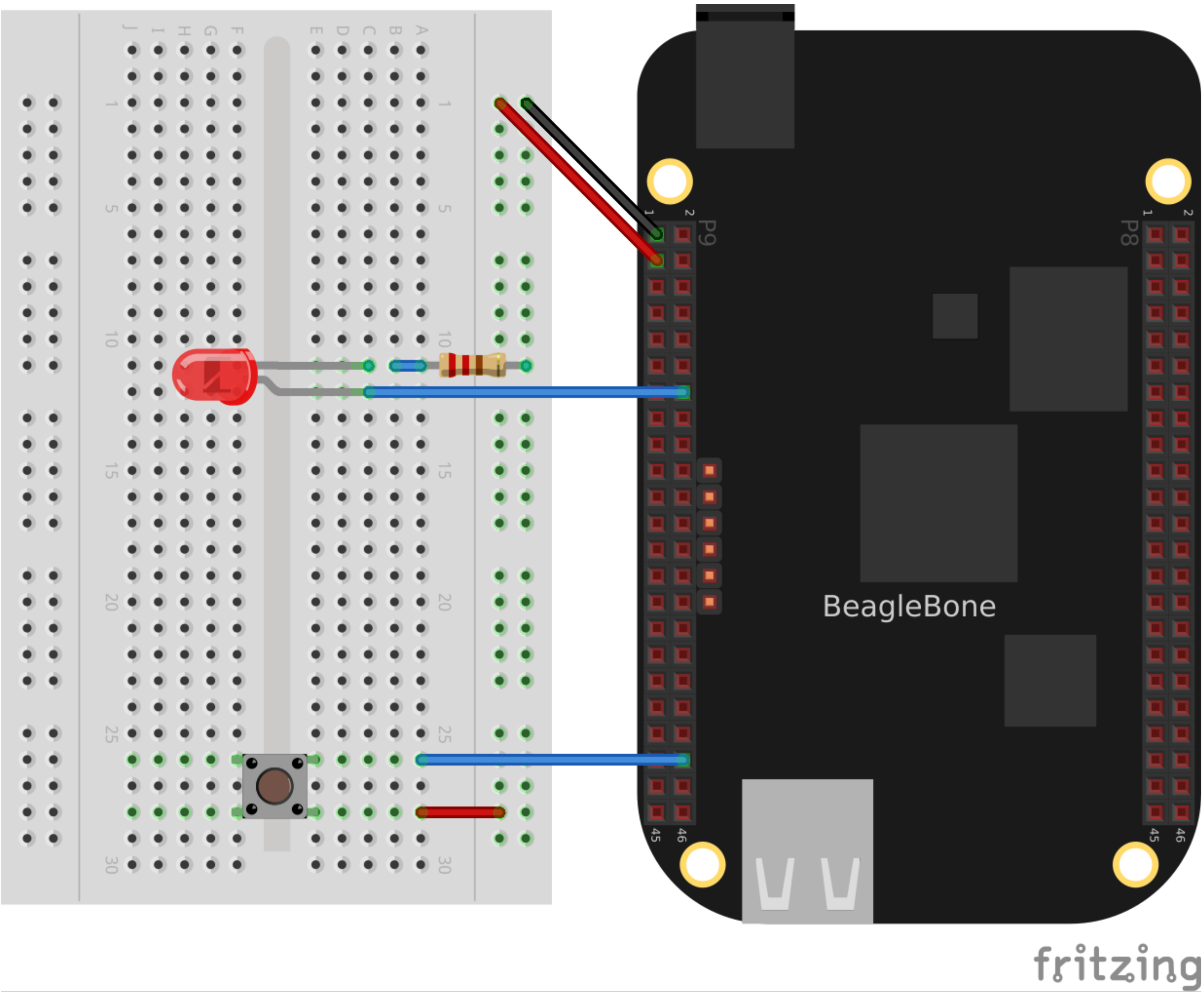


Figure 65. Diagram for wiring a pushbutton and LED with the LED attached to P9_14

The code in Monitoring a pushbutton (pushLED.js) 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.

Example 56. Monitoring a pushbutton (pushLED.py)

```
#!/usr/bin/env python
# ////////////////////////////////////////
# //    pushLED.py
# //    Blinks an LED attached to P9_12 when the button at P9_42 is pressed
# //    Wiring:
# //    Setup:
# //    See:
# ////////////////////////////////////////
import time
import os

ms = 50   # Read time in ms

LED="50"   # Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip 1, line
18 maps to 50
```

```
button="7" # P9_42 mapps to 7

GPIOPATH="/sys/class/gpio/"

# Make sure LED is exported
if (not os.path.exists(GPIOPATH+"gpio"+LED)):
    f = open(GPIOPATH+"export", "w")
    f.write(LED)
    f.close()

# Make it an output pin
f = open(GPIOPATH+"gpio"+LED+"/direction", "w")
f.write("out")
f.close()

# Make sure button is exported
if (not os.path.exists(GPIOPATH+"gpio"+button)):
    f = open(GPIOPATH+"export", "w")
    f.write(button)
    f.close()

# Make it an output pin
f = open(GPIOPATH+"gpio"+button+"/direction", "w")
f.write("in")
f.close()

# Read every ms
fin  = open(GPIOPATH+"gpio"+button+"/value", "r")
fout = open(GPIOPATH+"gpio"+LED+"/value", "w")

while True:
    fin.seek(0)
    fout.seek(0)
    fout.write(fin.read())
    time.sleep(ms/1000)
```

Example 57. Monitoring a pushbutton (pushLED.js)

```
#!/usr/bin/env node
////////////////////////////////////////
//  pushLED.js
//  Blinks an LED attached to P9_12 when the button at P9_42 is pressed
//  Wiring:
//  Setup:
//  See:
////////////////////////////////////////
const fs = require("fs");

const ms = 500   // Read time in ms
```

```
const LED="50";   // Look up P9.14 using gpioinfo | grep -e chip -e P9.14.  chip
1, line 18 maps to 50
const button="7"; // P9_42 mapps to 7

GPIOPATH="/sys/class/gpio/";

// Make sure LED is exported
if(!fs.existsSync(GPIOPATH+"gpio"+LED)) {
    fs.writeFileSync(GPIOPATH+"export", LED);
}
// Make it an output pin
fs.writeFileSync(GPIOPATH+"gpio"+LED+"/direction", "out");

// Make sure button is exported
if(!fs.existsSync(GPIOPATH+"gpio"+button)) {
    fs.writeFileSync(GPIOPATH+"export", button);
}
// Make it an input pin
fs.writeFileSync(GPIOPATH+"gpio"+button+"/direction", "in");

// Read every ms
setInterval(flashLED, ms);

function flashLED() {
    var data = fs.readFileSync(GPIOPATH+"gpio"+button+"/value").slice(0, -1);
    console.log('data = ' + data);
    fs.writeFileSync(GPIOPATH+"gpio"+LED+"/value", data);
 }
```

Run pushLED.js_ by using the following commands:

```
<pre data-type="programlisting">
bone$ <strong>./pushLED.js</strong>
data = 0
data = 0
data = 1
data = 1
^C
</pre>
```

Press ^C (Ctrl-C) to stop the code.

I/O with C

Problem

You want to use the C language to process inputs in real time, or Python/JavaScript isn't fast enough.

Solution

I/O with Python shows how to control an LED with a pushbutton using Python. This recipe accomplishes the same thing using C. It does it in the same way, opening the correct /sys/class/gpio files and reading an writing them.

Wire up the pushbutton and LED as shown in Diagram for wiring a pushbutton and LED with the LED attached to P9_14and open *pushLED.c*.

Example 58. Code for reading a switch and blinking an LED (pushLED.c)

```
////////////////////////////////////////
//  pushLED.c
//  Blinks the P9_14 pin based on the P9_42 pin
//  Wiring:
//  Setup:
//  See:
////////////////////////////////////////
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define MAXSTR 100

int main() {
  FILE *fpbutton, *fpLED;
  char LED[] = "50";   // Look up P9.14 using gpioinfo | grep -e chip -e P9.14.
chip 1, line 18 maps to 50
  char button[] = "7"; // Look up P9.42 using gpioinfo | grep -e chip -e P9.42.
chip 0, line 7 maps to 7
  char GPIOPATH[] = "/sys/class/gpio";
  char path[MAXSTR] = "";

  // Make sure LED is exported
  snprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", LED);
  if (!access(path, F_OK) == 0) {
    snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
    fpLED = fopen(path, "w");
    fprintf(fpLED, "%s", LED);
    fclose(fpLED);
  }

  // Make it an output LED
  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", LED, "/direction");
  fpLED = fopen(path, "w");
  fprintf(fpLED, "out");
  fclose(fpLED);

  // Make sure bbuttonutton is exported
  snprintf(path, MAXSTR, "%s%s%s", GPIOPATH, "/gpio", button);
  if (!access(path, F_OK) == 0) {
```

```
    snprintf(path, MAXSTR, "%s%s", GPIOPATH, "/export");
    fpbutton = fopen(path, "w");
    fprintf(fpbutton, "%s", button);
    fclose(fpbutton);
  }

  // Make it an input button
  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", button, "/direction");
  fpbutton = fopen(path, "w");
  fprintf(fpbutton, "in");
  fclose(fpbutton);

  // I don't know why I can open the LED outside the loop and use fseek before
  //  each read, but I can't do the same for the button.  It appears it needs
  //  to be opened every time.
  snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", LED,    "/value");
  fpLED    = fopen(path, "w");

  char state = '0';

  while (1) {
    snprintf(path, MAXSTR, "%s%s%s%s", GPIOPATH, "/gpio", button, "/value");
    fpbutton = fopen(path, "r");
    fseek(fpLED, 0L, SEEK_SET);
    fscanf(fpbutton, "%c", &state);
    printf("state: %c\n", state);
    fprintf(fpLED, "%c", state);
    fclose(fpbutton);
    usleep(250000);   // sleep time in microseconds
  }
}
```

Compile and run the code:

```
<pre data-type="programlisting">
bone$ <strong>gcc -o pushLED pushLED.c</strong>
bone$ <strong>./pushLED</strong>
state: 1
state: 1
state: 0
state: 0
state: 0
state: 1
^C
</pre>
```

The code responds quickly to the pushbutton. If you need more speed, comment-out the printf() and the sleep().

I/O with devmem2

Problem

Your C code isn't responding fast enough to the input signal. You want to read the GPIO registers directly.

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:

```
<pre data-type="programlisting">
bone$ <strong>wget https://bootlin.com/pub/mirror/devmem2.c</strong>
bone$ <strong>gcc -o devmem2 devmem2.c</strong>
bone$ <strong>sudo mv devmem2 /usr/bin</strong>
</pre>
```

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.

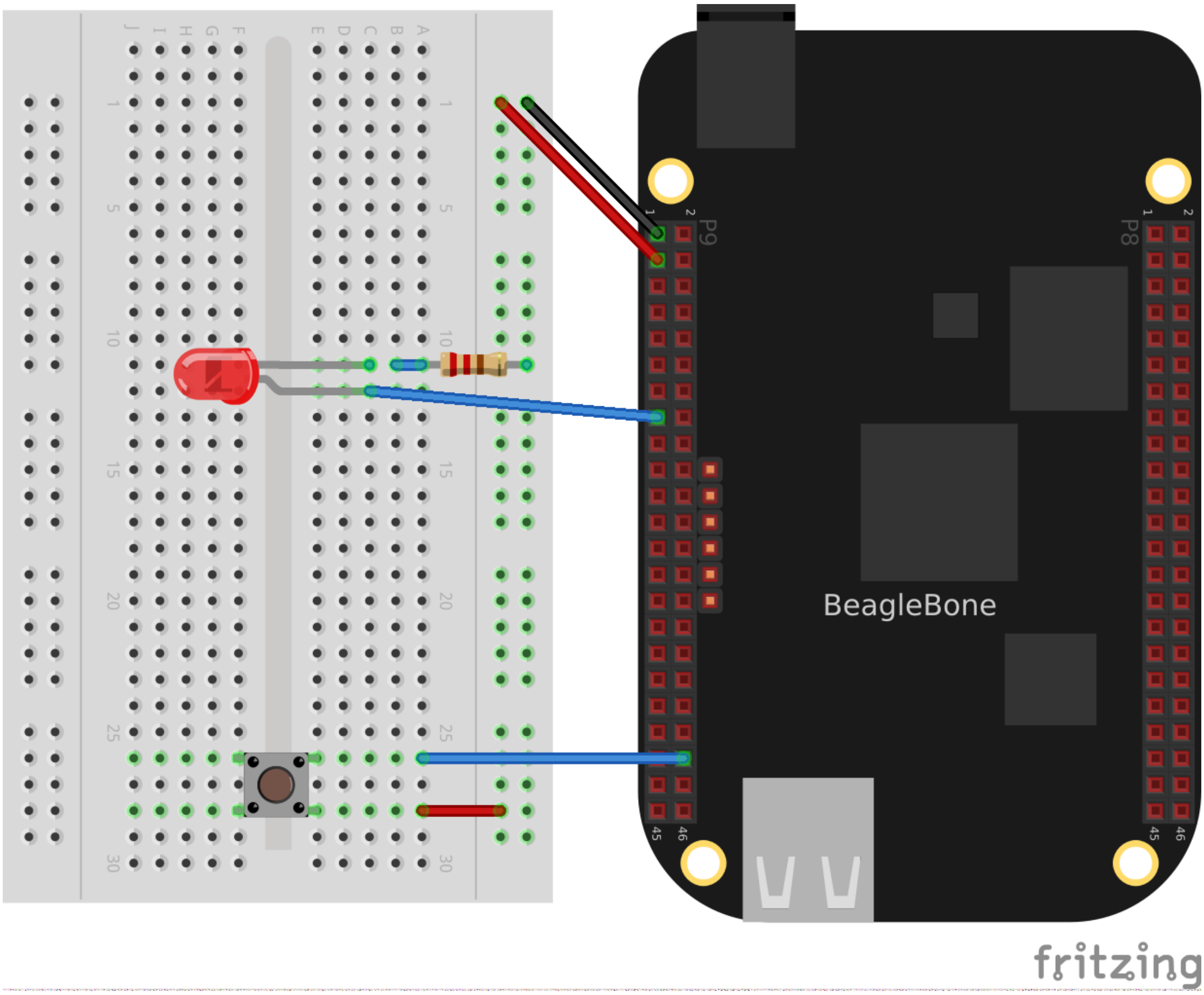


Figure 66. Diagram for wiring a pushbutton and LED with the LED attached to P9_13

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:

```
<pre data-type="programlisting">
bone$ <strong>cd cd /sys/class/gpio/</strong>
bone$ <strong>echo 31 > export</strong>
bone$ <strong>cd gpio31</strong>
bone$ <strong>echo out > direction</strong>
bone$ <strong>echo 1 > value</strong>
bone$ <strong>echo 0 > value</strong>
</pre>
```

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 GPIO0 in the Memory Map chapter (sensors). Table 2-2 indicates that GPIO0 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:

```
<pre data-type="programlisting">
bone$ <strong>sudo devmem2 0x44E07138</strong>
/dev/mem opened.
Memory mapped at address 0xb6f8e000.
Value at address 0x44E07138 (0xb6f8e138): 0xC000C404

</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo devmem2 0x44E07190 w 0x80000000</strong>
/dev/mem opened.
Memory mapped at address 0xb6fd7000.
Value at address 0x44E07190 (0xb6fd7190): 0x80000000
Written 0x80000000; readback 0x0
</pre>
```

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:

```
<pre data-type="programlisting">
bone$ <strong>sudo devmem2 0x44e07138</strong>
/dev/mem opened.
Memory mapped at address 0xb6fe2000.
Value at address 0x44E07138 (0xb6fe2138): 0x4000C484
</pre>
```

Here, bit 7 is set in 0x4000C484, showing the button is pushed.

Discussion

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.

I/O with C and mmap()

Problem

Your C code isn’t responding fast enough to the input signal.

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*.

Example 59. Memory address definitions (pushLEDmmap.h)

```
// From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
// -through-dev-mem
// user contributions licensed under cc by-sa 3.0 with attribution required
// http://creativecommons.org/licenses/by-sa/3.0/
// http://blog.stackoverflow.com/2009/06/attribution-required/
// Author: madscientist159
(http://stackoverflow.com/users/3000377/madscientist159)

#ifndef _BEAGLEBONE_GPIO_H_
#define _BEAGLEBONE_GPIO_H_

#define GPIO0_START_ADDR 0x44e07000
#define GPIO0_END_ADDR   0x44e08000
#define GPIO0_SIZE (GPIO0_END_ADDR - GPIO0_START_ADDR)

#define GPIO1_START_ADDR 0x4804C000
#define GPIO1_END_ADDR   0x4804D000
#define GPIO1_SIZE (GPIO1_END_ADDR - GPIO1_START_ADDR)

#define GPIO2_START_ADDR 0x41A4C000
#define GPIO2_END_ADDR   0x41A4D000
#define GPIO2_SIZE (GPIO2_END_ADDR - GPIO2_START_ADDR)

#define GPIO3_START_ADDR 0x41A4E000
#define GPIO3_END_ADDR   0x41A4F000
#define GPIO3_SIZE (GPIO3_END_ADDR - GPIO3_START_ADDR)
```

```
#define GPIO_DATAIN 0x138
#define GPIO_SETDATAOUT 0x194
#define GPIO_CLEARDATAOUT 0x190

#define GPIO_03  (1<<3)
#define GPIO_07  (1<<7)
#define GPIO_31  (1<<31)
#define GPIO_60  (1<<28)
#endif
```

Add the code in Code for directly reading memory addresses (pushLEDmmap.c) to a file named *pushLEDmmap.c*.

Example 60. Code for directly reading memory addresses (pushLEDmmap.c)

```
// From: http://stackoverflow.com/questions/13124271/driving-beaglebone-gpio
// -through-dev-mem
// user contributions licensed under cc by-sa 3.0 with attribution required
// http://creativecommons.org/licenses/by-sa/3.0/
// http://blog.stackoverflow.com/2009/06/attribution-required/
// Author: madscientist159
(http://stackoverflow.com/users/3000377/madscientist159)
//
// Read one gpio pin and write it out to another using mmap.
// Be sure to set -O3 when compiling.
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <signal.h>    // Defines signal-handling functions (i.e. trap Ctrl-C)
#include "pushLEDmmap.h"

// Global variables
int keepgoing = 1;    // Set to 0 when Ctrl-c is pressed

// Callback called when SIGINT is sent to the process (Ctrl-C)
void signal_handler(int sig) {
    printf( "\nCtrl-C pressed, cleaning up and exiting...\n" );
    keepgoing = 0;
}

int main(int argc, char *argv[]) {
    volatile void *gpio_addr;
    volatile unsigned int *gpio_datain;
    volatile unsigned int *gpio_setdataout_addr;
    volatile unsigned int *gpio_cleardataout_addr;

    // Set the signal callback for Ctrl-C
```

```
    signal(SIGINT, signal_handler);

    int fd = open("/dev/mem", O_RDWR);

    printf("Mapping %X - %X (size: %X)\n", GPIO0_START_ADDR, GPIO0_END_ADDR,
                                           GPIO0_SIZE);

    gpio_addr = mmap(0, GPIO0_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
                    GPIO0_START_ADDR);

    gpio_datain            = gpio_addr + GPIO_DATAIN;
    gpio_setdataout_addr   = gpio_addr + GPIO_SETDATAOUT;
    gpio_cleardataout_addr = gpio_addr + GPIO_CLEARDATAOUT;

    if(gpio_addr == MAP_FAILED) {
        printf("Unable to map GPIO\n");
        exit(1);
    }
    printf("GPIO mapped to %p\n", gpio_addr);
    printf("GPIO SETDATAOUTADDR mapped to %p\n", gpio_setdataout_addr);
    printf("GPIO CLEARDATAOUT mapped to %p\n", gpio_cleardataout_addr);

    printf("Start copying GPIO_07 to GPIO_31\n");
    while(keepgoing) {
        if(*gpio_datain & GPIO_07) {
            *gpio_setdataout_addr= GPIO_31;
        } else {
            *gpio_cleardataout_addr = GPIO_31;
        }
        //usleep(1);
    }

    munmap((void *)gpio_addr, GPIO0_SIZE);
    close(fd);
    return 0;
}
```

Now, compile and run the code:

```
<pre data-type="programlisting">
bone$ <strong>gcc -O3 pushLEDmmap.c -o pushLEDmmap</strong>
bone$ <strong>sudo ./pushLEDmmap</strong>
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...
</pre>
```

The code is in a tight while loop that checks the status of GPIO 7 and copies it to GPIO 31.

Tighter Delay Bounds with the PREEMPT_RT Kernel

Problem

You want to run real-time processes on the Beagle, but the OS is slowing things down.

Solution

The Kernel can be compiled with PREEMPT_RT enabled which reduces the delay from when a thread is scheduled to when it runs.

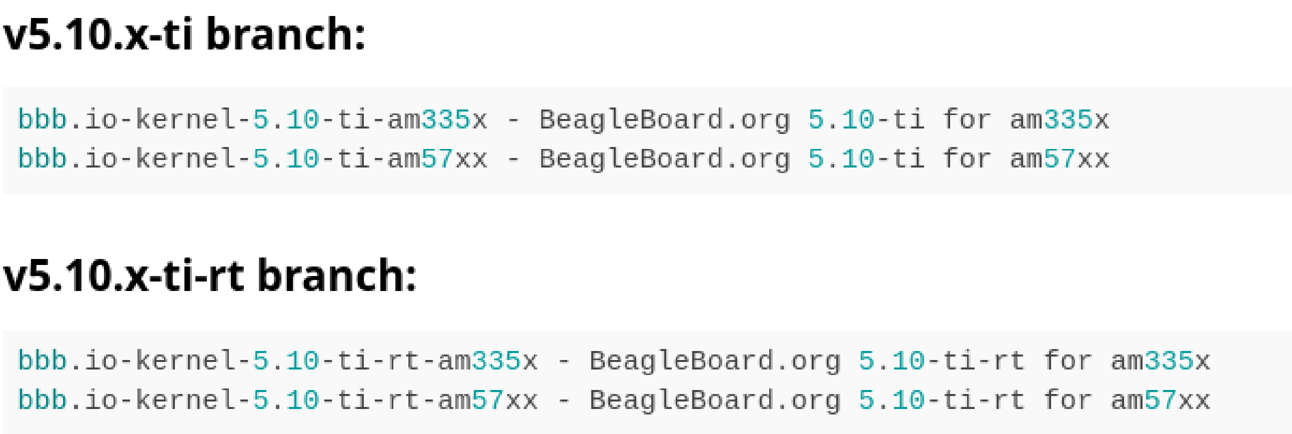
Switching to a PREEMPT_RT kernel is rather easy, but be sure to follow the steps in the Discussion to see how much the latencies are reduced.

- First see which kernel you are running:

```
<pre data-type="programlisting">
bone$ <strong>uname -a</strong>
Linux breadboard-home 5.10.120-ti-r47 #1bullseye SMP PREEMPT Tue Jul 12 18:59:38 UTC 2022
armv7l GNU/Linux
</pre>
```

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.



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
```

Figure 67. The regular and RT kernels

In The regular and RT kernels you see the regular kernel on top and the RT below.

- We want the RT one.

```
<pre data-type="programlisting">
bone$ <strong>sudo apt update</strong>
bone$ <strong>sudo apt install bbb.io-kernel-5.10-ti-rt-am335x</strong>
</pre>
```

NOTE	Use the **am57xx** if you are using the BeagleBoard AI or AI64.

- Before rebooting, edit /boot/uEnv.txt to start with:

```
<pre data-type="programlisting">
#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=
</pre>
```

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.

Discussion

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.

Cyclictest

cyclictest is one tool for measuring the latency from when a thread is schduled 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 install.sh to see what to install.

Example 61. install.sh

```
sudo apt install rt-tests
# You can run gnuplot on the host
sudo apt install gnuplot
```

- Open up another window and start something that will create a load on the Bone, then run the following:

```
<pre data-type="programlisting">
bone$ <strong>time sudo ./hist.gen > nort.hist</strong>
</pre>
```

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.

Example 62. hist.gen

```
#!/bin/sh
# This code is from Julia Cartwright julia@kernel.org

cyclictest -m -S -p 90 -h 400 -l "${1:-100000}"
```

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:

```
<pre data-type="programlisting">

bone$ sudo bash
bone# ulimit -r unlimited
bone# ./hist.gen > nort.hist
bone# exit
</pre>
```

- Now you are ready to reboot into the RT kernel and run the test again.

```
<pre data-type="programlisting">
bone$ <strong>reboot</strong>
</pre>
```

- After rebooting:

```
<pre data-type="programlisting">
bone$ <strong>uname -a</strong>
Linux breadboard-home 5.10.120-ti-rt-r47 #1bullseye SMP PREEMPT RT Tue Jul 12 18:59:38 UTC
2022 armv7l GNU/Linux
</pre>
```

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 scipt again (note it's being saved in rt.hist this time.)

```
<pre data-type="programlisting">
bone$ <strong>time sudo ./hist.gen > rt.hist</strong>
</pre>
```

NOTE At this point yoou can edit /boot/uEnt.txt to boot the non RT kernel and reboot.

Now it's time to plot the results.

```
<pre data-type="programlisting">
bone$ <strong>gnuplot hist.plt</strong>
</pre>
```

This will generate the file **cyclictest.png** which contains your plot. It should look like:

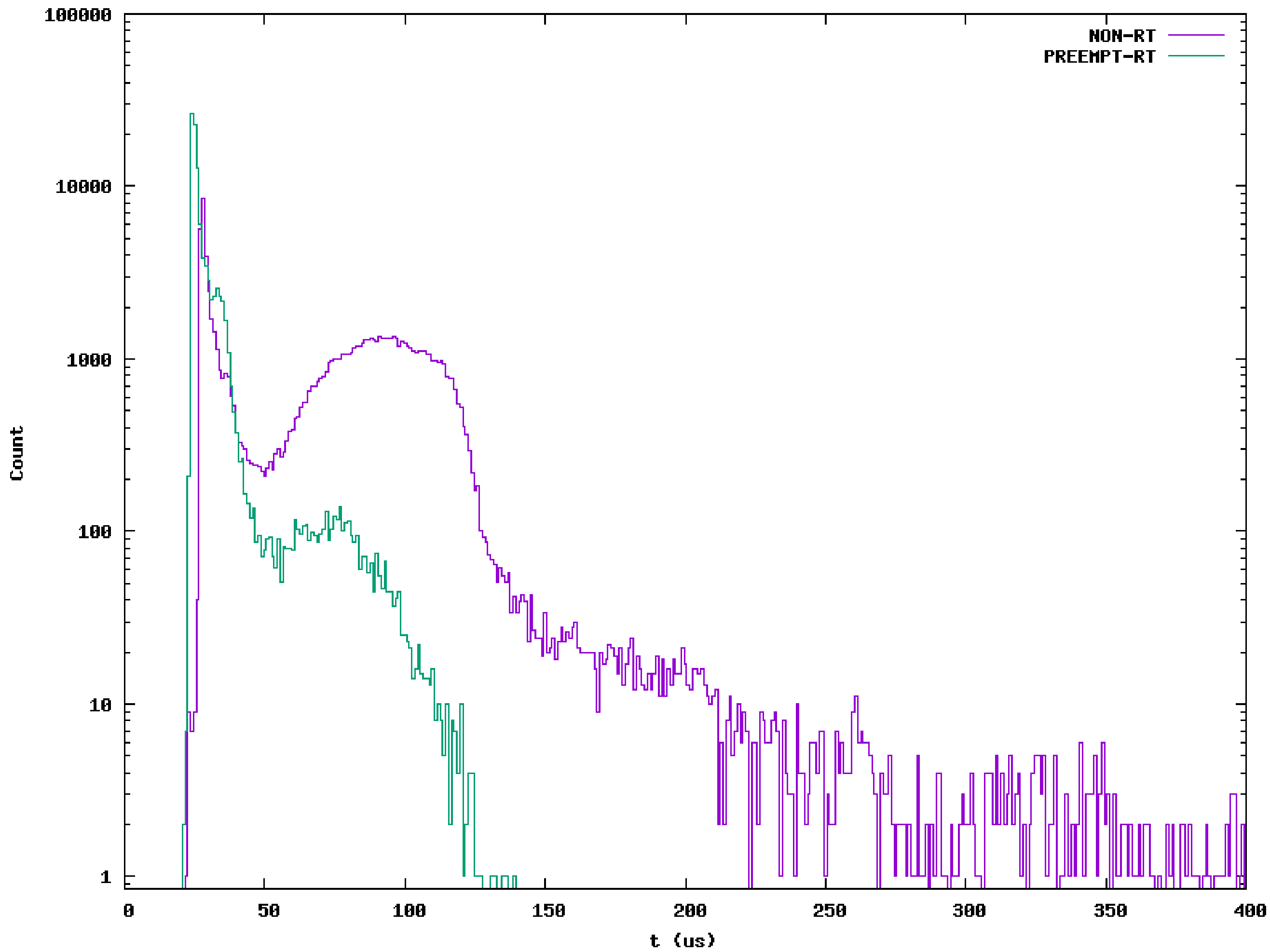


Figure 68. Histogram of Non-RT and RT kernels running cyclictest

Notice the NON-RT data have much longer latenices. They may not happen often (fewer than 10 times in each bin), but they are occuring and may be enough to miss a real-time deadline.

The PREEMPT-RT times are all under a 150 μs.

Capes

Introduction

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 from Bone-sized (Using a 128 x 128-Pixel LCD Cape) to much larger than the Bone (Using a Seven-Inch LCD Cape).

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.

Using a Seven-Inch LCD Cape

Problem

You want to display the Bone's desktop on a portable LCD.

Solution

A number of LCD capes are built for the Bone, ranging in size from three to seven inches. This recipe attaches a seven-inch BeagleBone LCD7 from CircuitCo (shown in Seven-inch LCD from CircuitCo [3]) to the Bone.

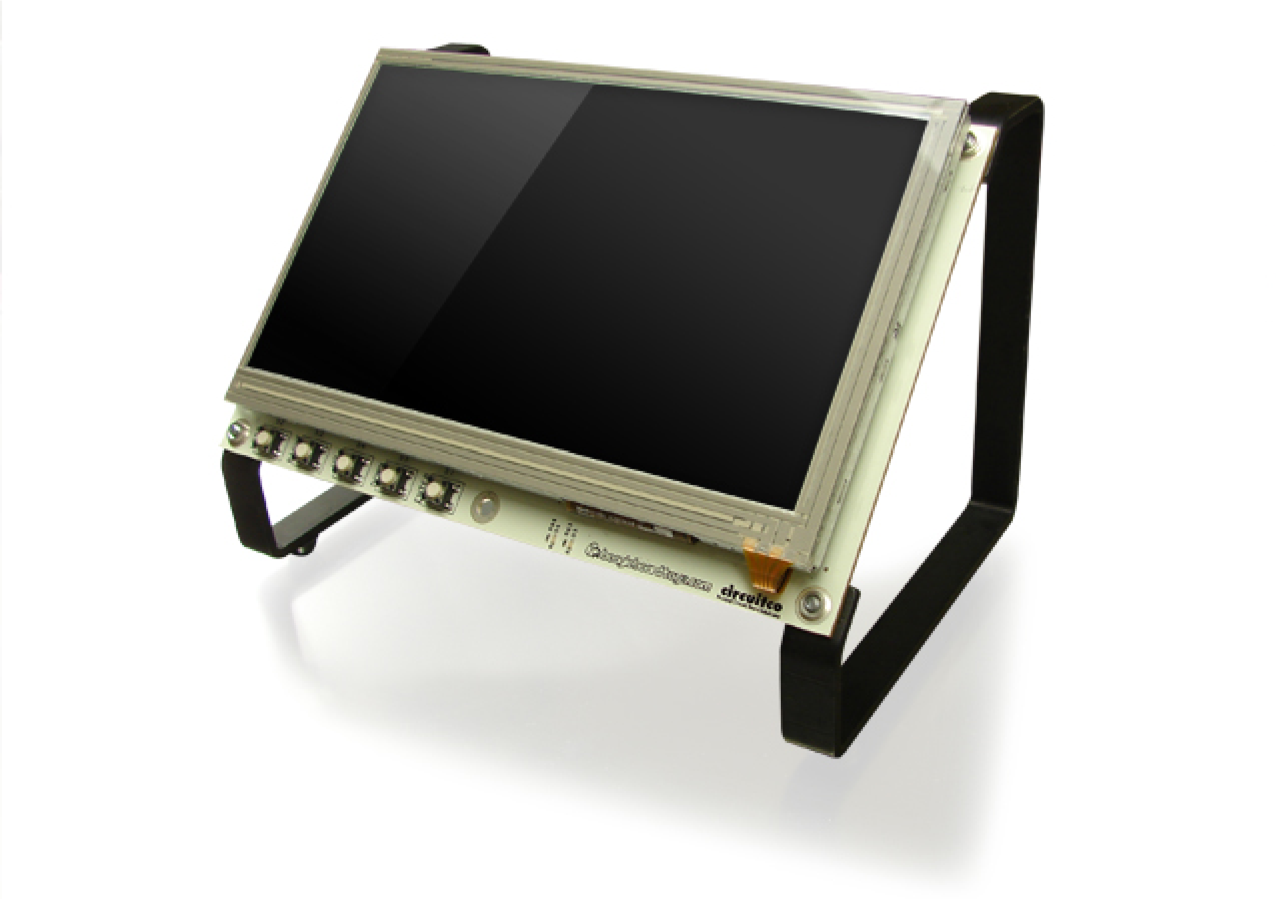


Figure 69. Seven-inch LCD from CircuitCo [3]

To make this recipe, you will need:

- Seven-inch LCD cape
- A 5 V power supply

Just attach the Bone to the back of the LCD, making sure pin 1 of P9 lines up with pin 1 of P9 on the LCD. Apply a 5 V power supply, and the desktop will appear on your LCD, as shown in Seven-inch LCD desktop.

Figure 70. Seven-inch LCD desktop

Attach a USB keyboard and mouse, and you have a portable Bone. Wireless keyboard and mouse combinations make a nice solution to avoid the need to add a USB hub.

Using a 128 x 128-Pixel LCD Cape

Problem

You want to use a small LCD to display things other than the desktop.

Solution

The MiniDisplay is a 128 x 128 full-color LCD cape that just fits on the Bone, as shown in MiniDisplay 128 x 128-pixel LCD from CircuitCo.

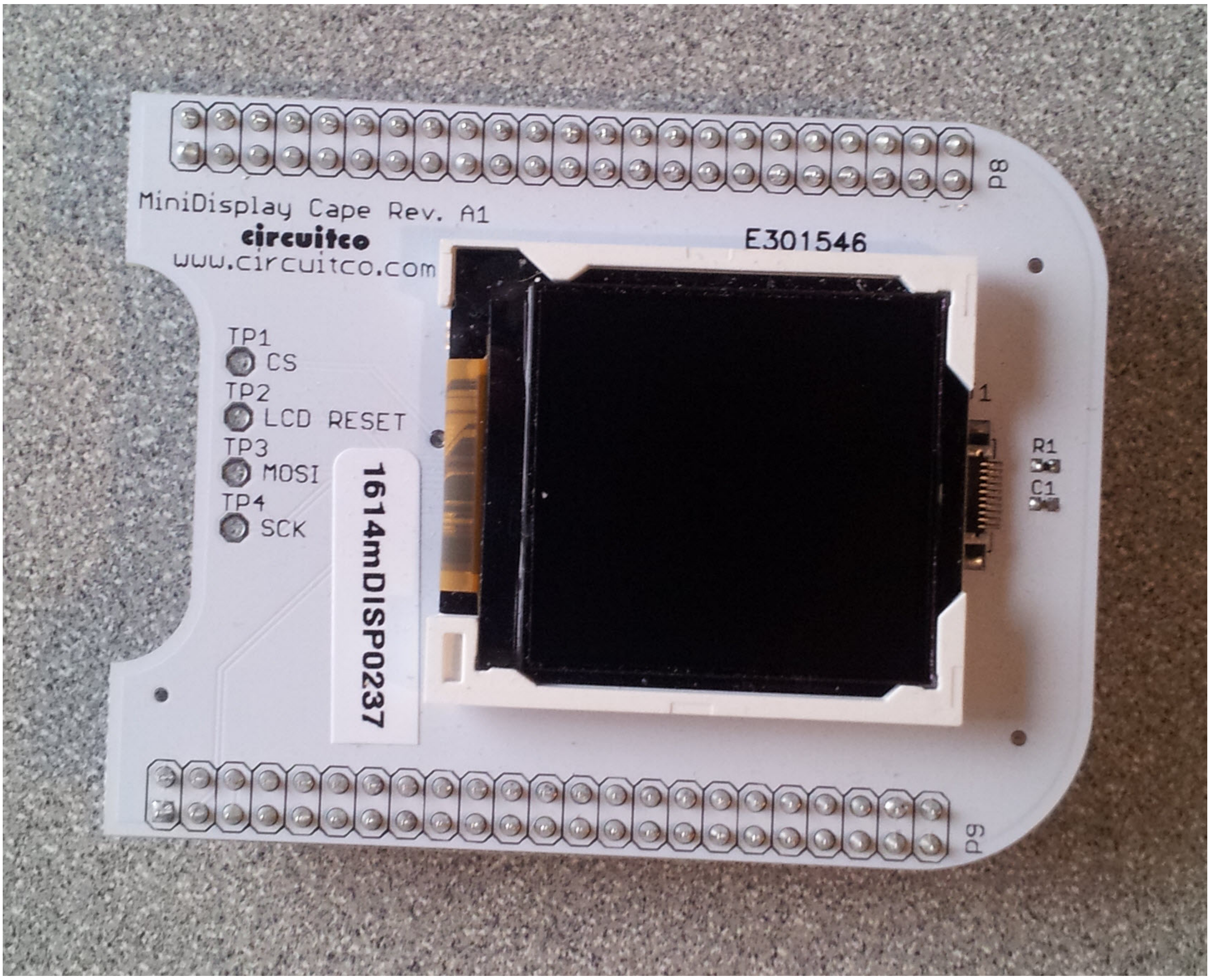


Figure 71. MiniDisplay 128 x 128-pixel LCD from CircuitCo

To make this recipe, you will need:

- MiniDisplay LCD cape

Attach to the Bone and apply power. Then run the following commands:

```
<pre data-type="programlisting">
# From http://elinux.org/CircuitCo:MiniDisplay_Cape
# Datasheet:
# https://www.crystalfontz.com/products/document/3277/ST7735_V2.1_20100505.pdf
bone$ <strong>wget http://elinux.org/images/e/e4/Minidisplay-example.tar.gz</strong>
bone$ <strong>tar zmxvf Minidisplay-example.tar.gz</strong>
bone$ <strong>cd minidisplay-example</strong>
bone$ <strong>make</strong>
bone$ <strong>./minidisplay-test</strong>
Unable to initialize SPI: No such file or directory
Aborted
</pre>
```

WARNING You might get a compiler warning, but the code should run fine.

The MiniDisplay uses the Serial Peripheral Interface (SPI) interface, and it's not initialized. The manufacturer's website suggests enabling SPI0 by using the following commands:

```
<pre data-type="programlisting">
bone$ <strong>export SLOTS=/sys/devices/bone_capemgr.*/slots</strong>
bone$ <strong>echo BB-SPIDEV0 &gt; $SLOTS</strong>
</pre>
```

Hmmm, something isn't working here. Here's how to see what happened:

```
<pre data-type="programlisting">
bone$ <strong>dmesg | tail</strong>
[  625.334497] bone_capemgr.9: part_number 'BB-SPIDEV0', version 'N/A'
[  625.334673] bone_capemgr.9: slot #11: generic override
[  625.334720] bone_capemgr.9: bone: Using override eeprom data at slot 11
[  625.334769] bone_capemgr.9: slot #11: 'Override Board Name,00A0,Override \
               Manuf,BB-SPIDEV0'
[  625.335026] bone_capemgr.9: slot #11: \Requesting part number/version based \
               'BB-SPIDEV0-00A0.dtbo
[  625.335076] bone_capemgr.9: slot #11: Requesting firmware \
               'BB-SPIDEV0-00A0.dtbo' \
               for board-name 'Override Board Name', version '00A0'
[  625.335144] bone_capemgr.9: slot #11: dtbo 'BB-SPIDEV0-00A0.dtbo' loaded; \
               converting to live tree
[  625.341842] bone_capemgr.9: slot #11: BB-SPIDEV0 conflict P9.21 \
               (#10:bspwm_P9_21_b) <a class="co" id="capemgr_conflict_co"
href="#capemgr_conflict" ><img src="callouts/1.png" alt="1"/></a>
[  625.351296] bone_capemgr.9: slot #11: Failed verification
</pre>
```

<dl class="calloutlist">
 <dt></dt>
  <dd>Shows there is a conflict for pin <code>P9_21</code>: it's already configured for pulse width modulation (PWM).</dd>

</dl>

Here's how to see what's already configured:

```
<pre data-type="programlisting">
bone$ <strong>cat $SLOTS</strong>
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
 7: ff:P-O-L Override Board Name,00A0,Override Manuf,bspm_P9_42_27
 8: ff:P-O-L Override Board Name,00A0,Override Manuf,bspm_P9_41_27
 9: ff:P-O-L Override Board Name,00A0,Override Manuf,am33xx_pwm
10: ff:P-O-L Override Board Name,00A0,Override Manuf,bspwm_P9_21_b <a class="co"
```

```
id="capemgr_load_co" href="#capemgr_load"><img src="callouts/1.png" alt="1"/></a>
</pre>
```

```
<dl class="calloutlist">
 <dt><a id="capemgr_load" href="#capemgr_load_co"><img src="callouts/1.png"
alt="1"/></a></dt>
  <dd>You can see the eMMC, HDMI, and three PWMs are already using some of the pins. Slot
10 shows <code>P9_21</code> is in use by a PWM.</dd>
</dl>
```

You can unconfigure it by using the following commands:

```
<pre data-type="programlisting">
bone$ <strong>echo -10 &gt; $SLOTS</strong>
bone$ <strong>cat $SLOTS</strong>
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
 7: ff:P-O-L Override Board Name,00A0,Override Manuf,bspm_P9_42_27
 8: ff:P-O-L Override Board Name,00A0,Override Manuf,bspm_P9_41_27
 9: ff:P-O-L Override Board Name,00A0,Override Manuf,am33xx_pwm
</pre>
```

Now P9_21 is free for the MiniDisplay to use.

NOTE	In future Bone images, all of the pins will already be allocated as part of the main device tree using runtime pinmux helpers and configured at runtime using the config-pin utility. This would eliminate the need for device tree overlays in most cases.

Now, configure it for the MiniDisplay and run a test:

```
<pre data-type="programlisting">
bone$ <strong>echo BB-SPIDEV0 &gt; $SLOTS</strong>
bone$ <strong>./minidisplay-test</strong>
</pre>
```

You then see Boris, as shown in MiniDisplay showing Boris [5].

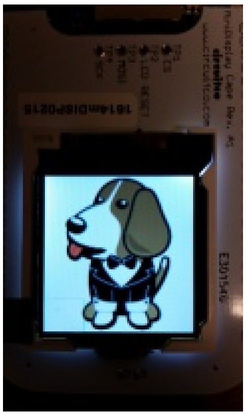


Figure 72. MiniDisplay showing Boris [5]

Connecting Multiple Capes

Problem

You want to use more than one cape at a time.

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.

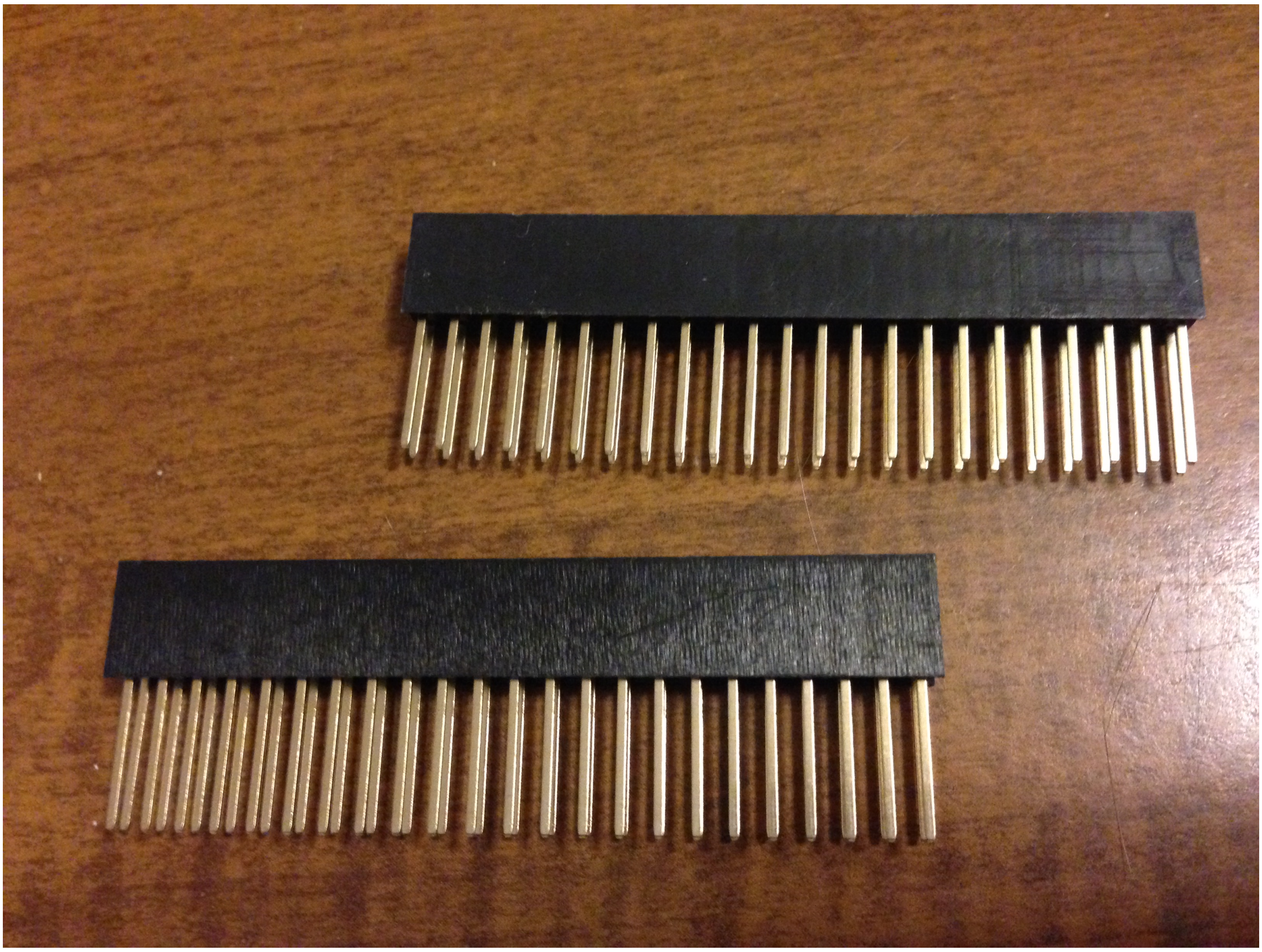


Figure 73. Stacking headers

Note that larger LCD panels might provide expansion headers, such as the ones shown in Back side of LCD7 cape [7], rather than the stacking headers, and that those can also be used for adding additional capes.

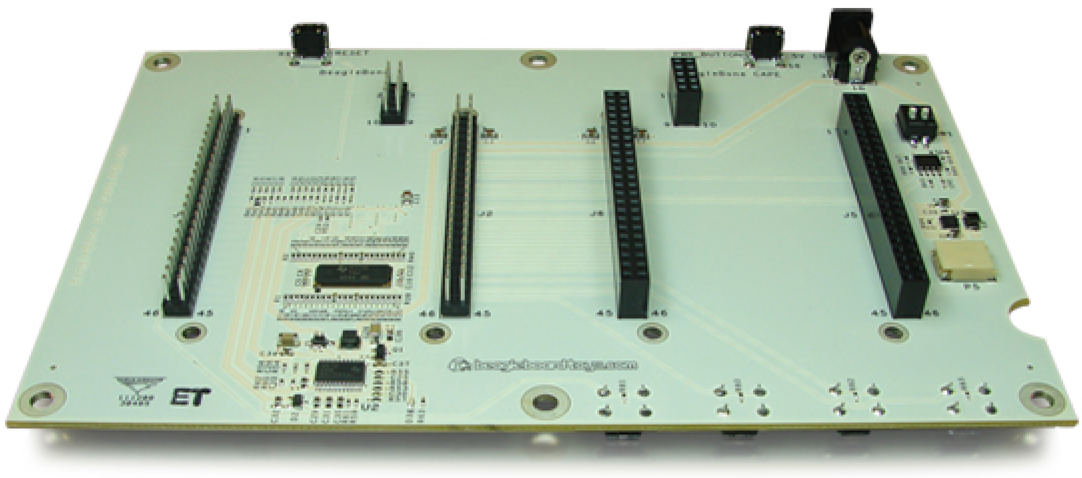


Figure 74. Back side of LCD7 cape [7]

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 Pins utilized by CircuitCo Audio Cape [9] for the Circuitco Audio Cape.

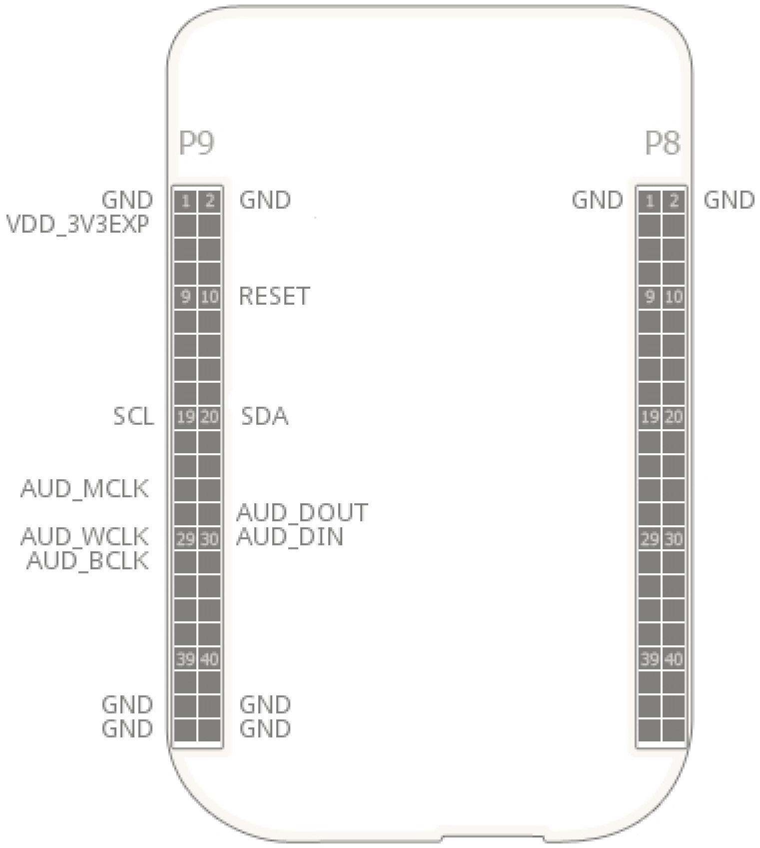


Figure 75. Pins utilized by CircuitCo Audio Cape [9]

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^2C 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.

Moving from a Breadboard to a Protoboard

Problem

You have your circuit working fine on the breadboard, but you want a more reliable solution.

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. BeagleBone breadboard [11] shows the BeagleBone Breadboard, which is just one protoboard option.

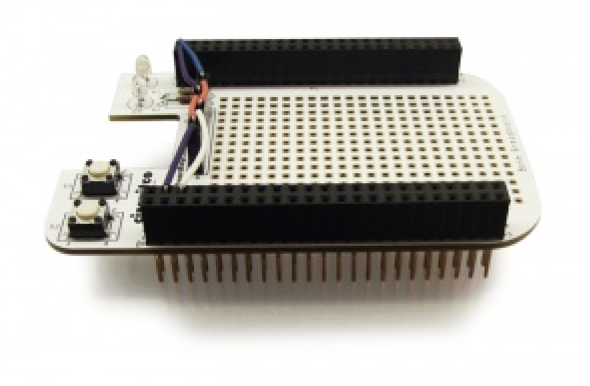


Figure 76. BeagleBone breadboard [11]

You just solder your parts on the protoboard as you had them on the breadboard.

Creating a Prototype Schematic

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?

Solution

In [tips_fritzing], 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).

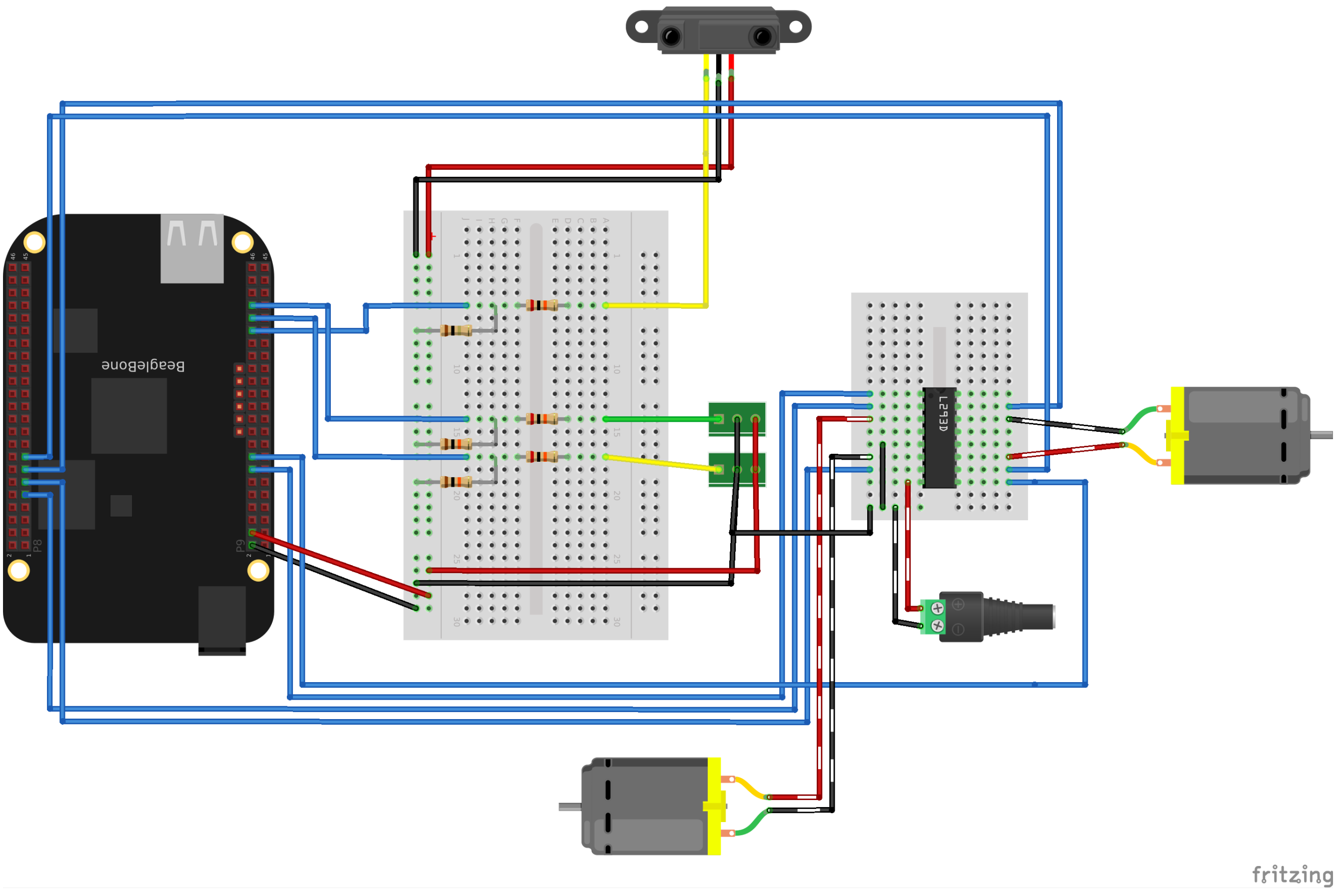


Figure 77. A simple robot controller diagram (quickBot.fzz)

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 [sensors_hc-sr04] 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.

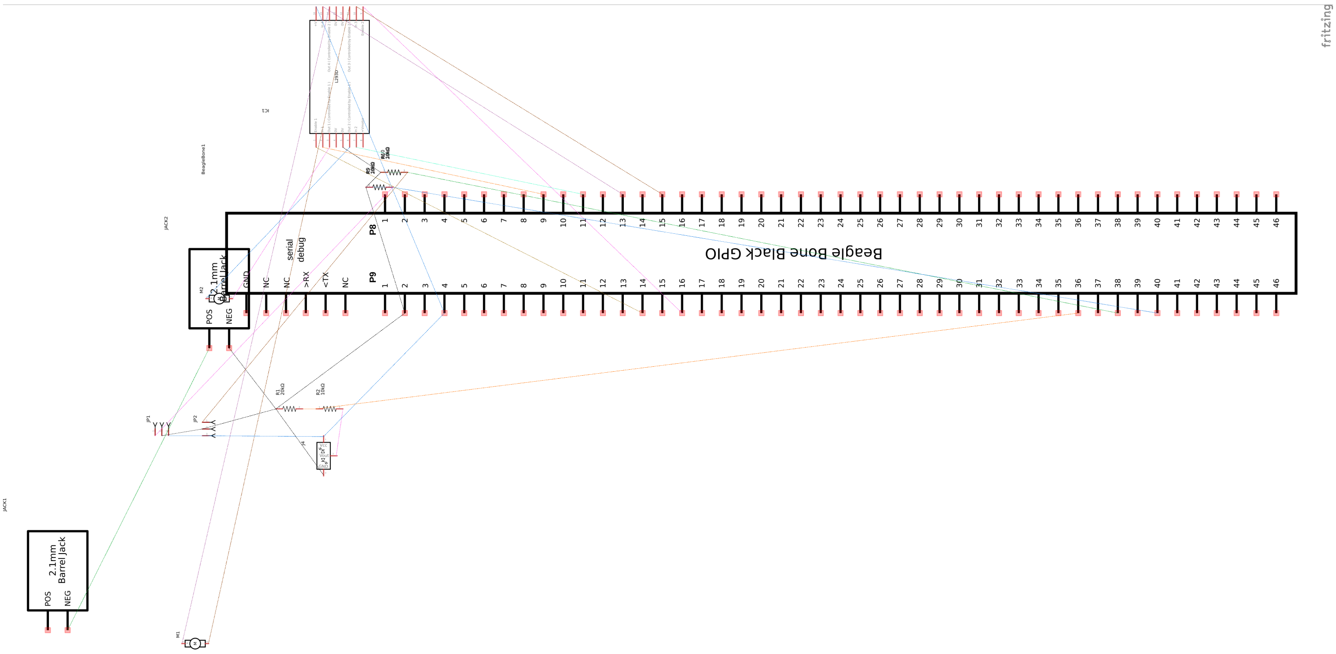


Figure 78. Automatically generated schematic

Cleaned-up schematic shows my cleaned-up schematic. I did it by moving the parts around until it looked better.

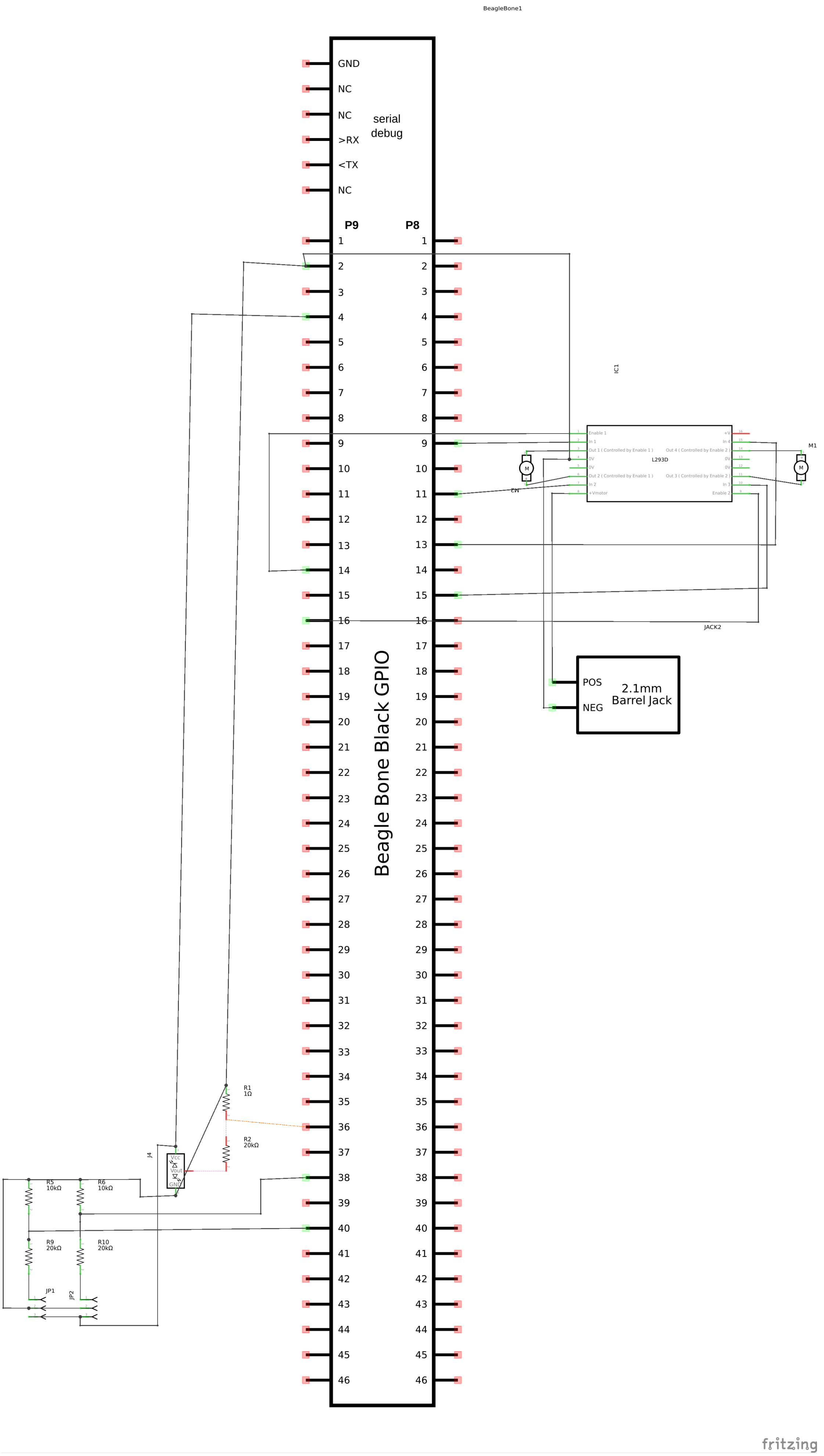


Figure 79. Cleaned-up schematic

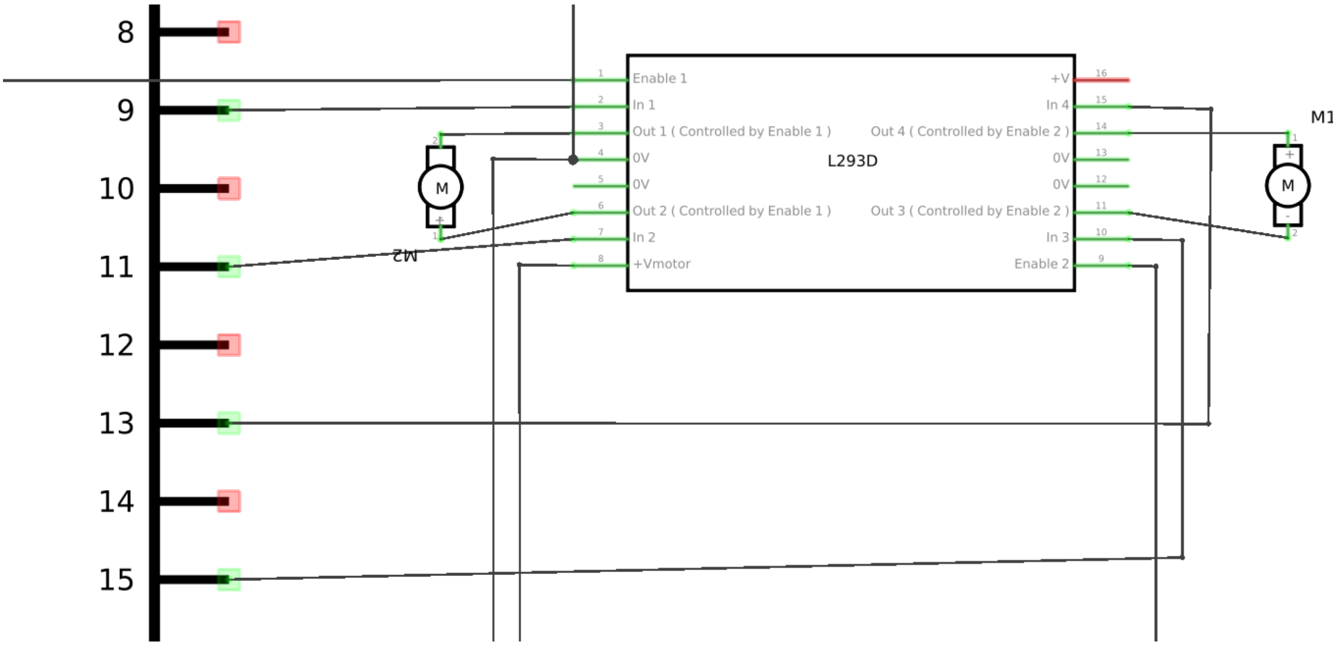


Figure 80. Zoomed-in schematic

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.

Verifying Your Cape Design

Problem

You've got a design. How do you quickly verify that it works?

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).

Example 63. Testing the quickBot motors interface (quickBot_motor_test.js)

```
<pre data-type="programlisting">#!/usr/bin/env node
var b = require('bonescript');
var M1_SPEED    = 'P9_16'; <a class="co" id="co_hello_CO1-1_unique"
href="#callout_hello_CO1-1_unique"><img src="callouts/1.png" alt="1"/></a>
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; <a class="co" id="co_hello_CO1-2_unique" href="#callout_hello_CO1-
```

```
2_unique"><img src="callouts/2.png" alt="2"/></a>
var fast = 0.95;
var slow = 0.7;
var state = 0;  <a class="co" id="co_hello_CO1-3_unique" href="#callout_hello_CO1-
3_unique"><img src="callouts/3.png" alt="3"/></a>

b.pinMode(M1_FORWARD, b.OUTPUT); <a class="co" id="co_hello_CO1-4_unique"
href="#callout_hello_CO1-4_unique"><img src="callouts/4.png" alt="4"/></a>
b.pinMode(M1_BACKWARD, b.OUTPUT);
b.pinMode(M2_FORWARD, b.OUTPUT);
b.pinMode(M2_BACKWARD, b.OUTPUT);
b.analogWrite(M1_SPEED, 0, freq); <a class="co" id="co_hello_CO1-5_unique"
href="#callout_hello_CO1-5_unique"><img src="callouts/5.png" alt="5"/></a>
b.analogWrite(M2_SPEED, 0, freq);

updateMotors(); <a class="co" id="co_hello_CO1-6_unique" href="#callout_hello_CO1-
6_unique"><img src="callouts/6.png" alt="6"/></a>

function updateMotors() { <img src="callouts/6.png" alt="6"/>
    //console.log("Setting state = " + state); <a class="co" id="co_hello_CO1-
7_unique" href="#callout_hello_CO1-7_unique"><img src="callouts/7.png" alt="7"/></a>
    updateLEDs(state); <img src="callouts/7.png" alt="7"/>
    switch(state) { <img src="callouts/3.png" alt="3"/>
        case 0:
        default:
            M1_set(0); <a class="co" id="co_hello_CO1-8_unique"
href="#callout_hello_CO1-8_unique"><img src="callouts/8.png" alt="8"/></a>
            M2_set(0);
            state = 1; <img src="callouts/3.png" alt="3"/>
            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); <img src="callouts/3.png" alt="3"/>
}

function updateLEDs(state) { <img src="callouts/7.png" alt="7"/>
    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;
    }
}

function M1_set(speed) { <img src="callouts/8.png" alt="8"/>
    speed = (speed &gt; 1) ? 1 : speed; <a class="co" id="co_hello_CO1-9_unique"
href="#callout_hello_CO1-9_unique"><img src="callouts/9.png" alt="9"/></a>
    speed = (speed &lt; -1) ? -1 : speed;
    b.digitalWrite(M1_FORWARD, b.LOW);
    b.digitalWrite(M1_BACKWARD, b.LOW);
    if(speed &gt; 0) {
        b.digitalWrite(M1_FORWARD, b.HIGH);
    } else if(speed &lt; 0) {
        b.digitalWrite(M1_BACKWARD, b.HIGH);
    }
    b.analogWrite(M1_SPEED, Math.abs(speed), freq); <a class="co" id="co_hello_CO1-
10_unique" href="#callout_hello_CO1-10_unique"><img src="callouts/10.png"
```

```
alt="10"/></a>
}

function M2_set(speed) {
    speed = (speed &gt; 1) ? 1 : speed;
    speed = (speed &lt; -1) ? -1 : speed;
    b.digitalWrite(M2_FORWARD, b.LOW);
    b.digitalWrite(M2_BACKWARD, b.LOW);
    if(speed &gt; 0) {
        b.digitalWrite(M2_FORWARD, b.HIGH);
    } else if(speed &lt; 0) {
        b.digitalWrite(M2_BACKWARD, b.HIGH);
    }
    b.analogWrite(M2_SPEED, Math.abs(speed), freq);
}</pre>

<dl class="calloutlist">
<dt><a class="co" id="callout_hello_CO1-1_unique" href="#co_hello_CO1-1_unique"><img
src="callouts/1.png" alt="1"/></a></dt>
<dd><p>Define each pin as a variable. This makes it easy to change to another pin if
you decide that is necessary.</p></dd>
<dt><a class="co" id="callout_hello_CO1-2_unique" href="#co_hello_CO1-2_unique"><img
src="callouts/2.png" alt="2"/></a></dt>
<dd><p>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 <a data-type="xref"
href="#quickBot_motor_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.</p></dd>
<dt><a class="co" id="callout_hello_CO1-3_unique" href="#co_hello_CO1-3_unique"><img
src="callouts/3.png" alt="3"/></a></dt>
<dd><p>Use a simple variable such as <code>state</code> to keep track of the test
phase. This is used in a <code>switch</code> 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&#8217;t entered until
after a two-second delay, as specified in the call to
<code>setTimeout()</code>.</p></dd>
<dt><a class="co" id="callout_hello_CO1-4_unique" href="#co_hello_CO1-4_unique"><img
src="callouts/4.png" alt="4"/></a></dt>
<dd><p>Perform the initial setup of all the pins.</p></dd>
<dt><a class="co" id="callout_hello_CO1-5_unique" href="#co_hello_CO1-5_unique"><img
src="callouts/5.png" alt="5"/></a></dt>
<dd><p>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 <code>pinMode()</code> function doesn&#8217;t have an argument for
providing the update frequency, so use the <code>analogWrite()</code> function,
instead. You can review using the PWM in <a data-type="xref"
href="#motors_servo"/>.</p></dd>
<dt><a class="co" id="callout_hello_CO1-6_unique" href="#co_hello_CO1-6_unique"><img
src="callouts/6.png" alt="6"/></a></dt>
<dd><p><code>updateMotors()</code> 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 <code>setTimeout()</code> JavaScript function. The first call is
used to prime the loop.</p></dd>
<dt><a class="co" id="callout_hello_CO1-7_unique" href="#co_hello_CO1-7_unique"><img
src="callouts/7.png" alt="7"/></a></dt>
<dd><p>The call to <code>console.log()</code> was initially here to observe the state
transitions in the debug console, but it was replaced with the
<code>updateLEDs()</code> call. Using the <code>USER</code> LEDs makes it possible to
note the state transitions without having visibility of the debug console.
<code>updateLEDs()</code> is defined later.</p></dd>
<dt><a class="co" id="callout_hello_CO1-8_unique" href="#co_hello_CO1-8_unique"><img
src="callouts/8.png" alt="8"/></a></dt>
<dd><p>The <code>M1_set()</code> and <code>M2_set()</code> 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 <code>speed</code>, as defined between <code>-1</code>
(maximum reverse), <code>0</code> (stop), and <code>1</code> (maximum
forward).</p></dd>
<dt><a class="co" id="callout_hello_CO1-9_unique" href="#co_hello_CO1-9_unique"><img
src="callouts/9.png" alt="9"/></a></dt>
<dd><p>Perform simple bounds checking to ensure that speed values are between <code>-
1</code> and <code>1</code>.</p></dd>
<dt><a class="co" id="callout_hello_CO1-10_unique" href="#co_hello_CO1-10_unique"><img
src="callouts/10.png" alt="10"/></a></dt>
<dd><p>The <code>analogWrite()</code> call uses the absolute value of
<code>speed</code>, making any negative numbers a positive magnitude.</p></dd>
</dl>
```

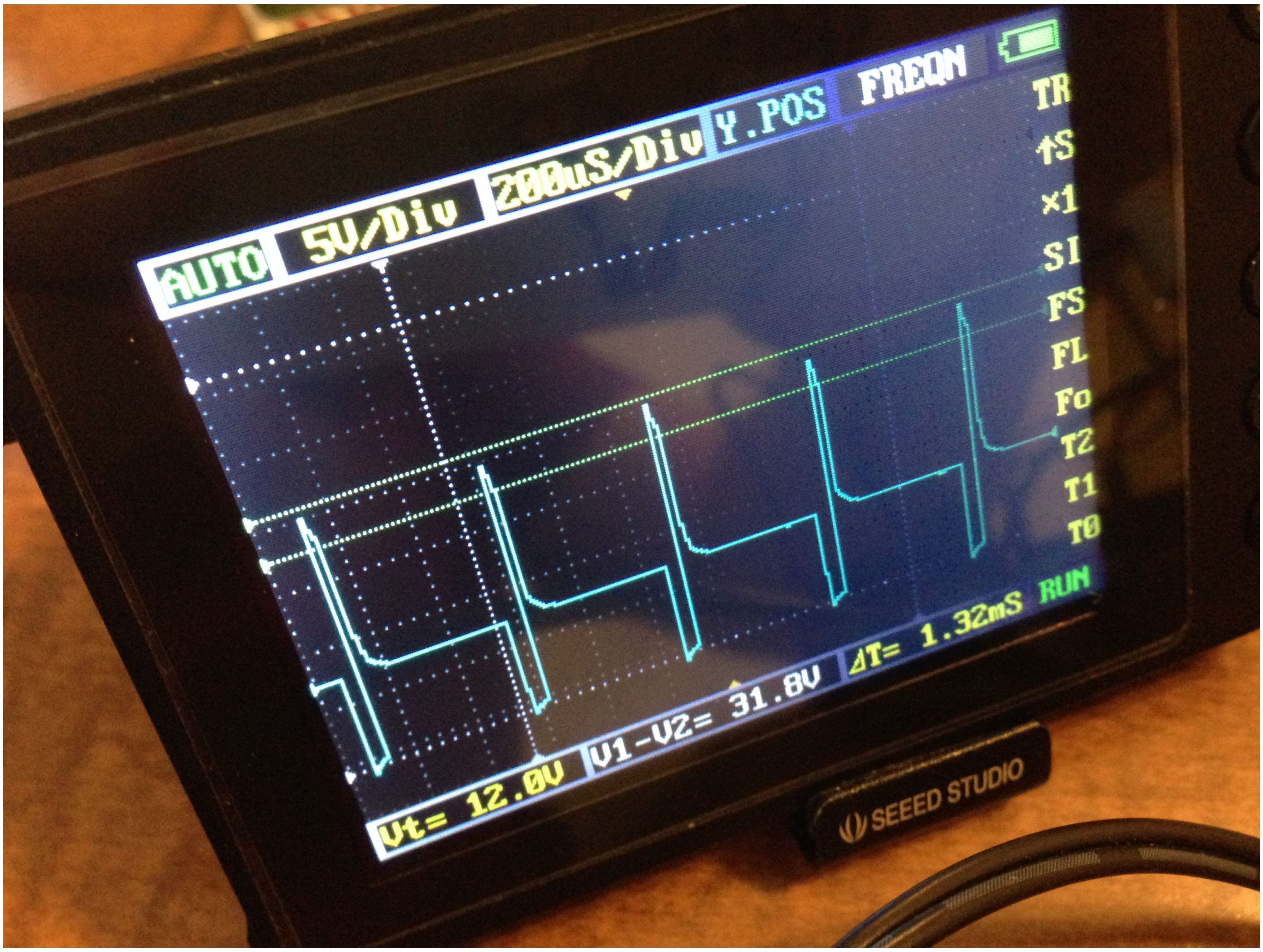


Figure 81. quickBot motor test showing kickback

Using the solution in [basics_autorun], you can untether from your coding station to test your design at your lab workbench, as shown in quickBot motor test code under scope.

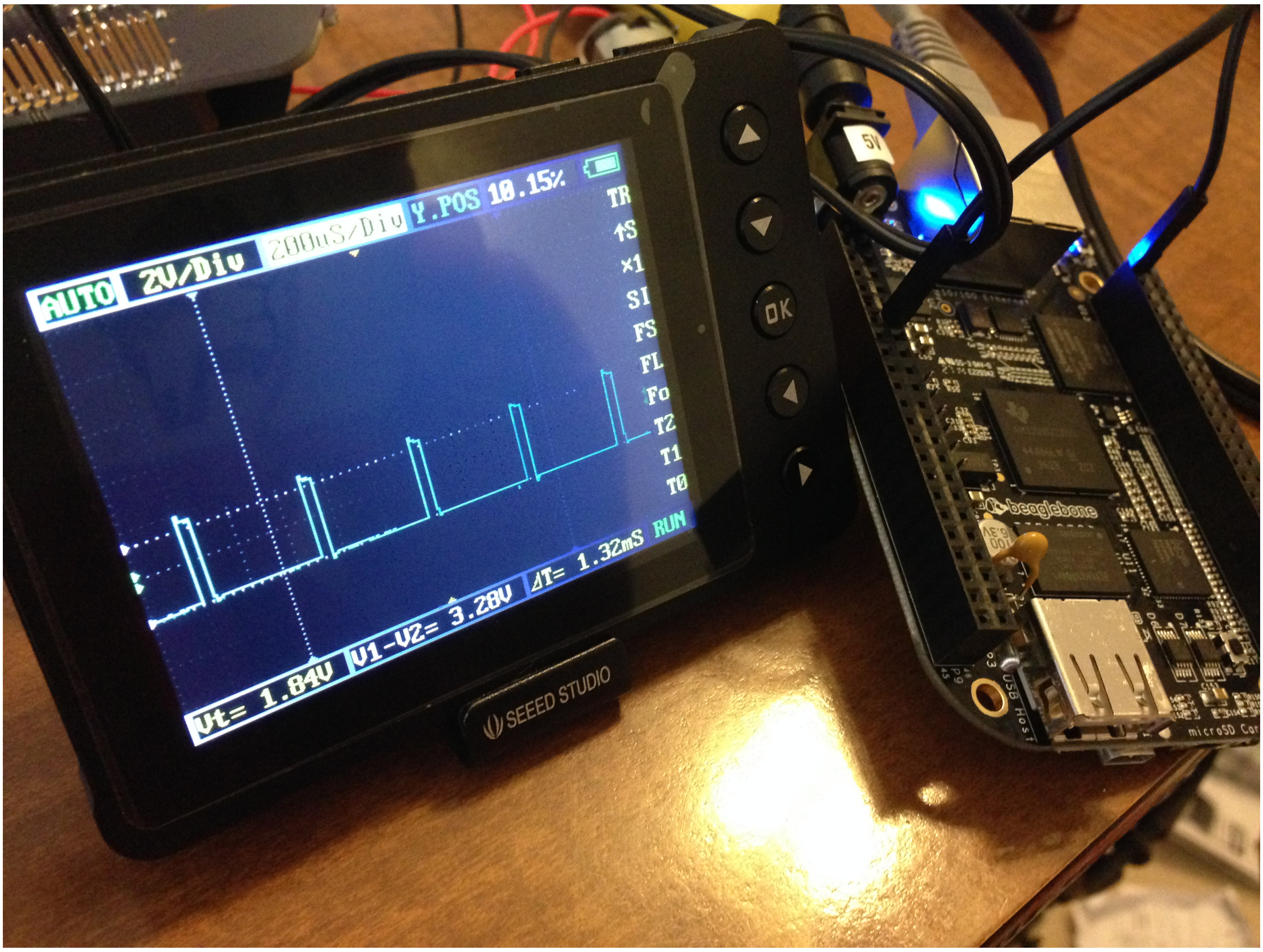


Figure 82. quickBot motor test code under scope

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.

Laying Out Your Cape PCB

Problem

You've generated a diagram and schematic for your circuit and verified that they are correct. How do you create a PCB?

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.

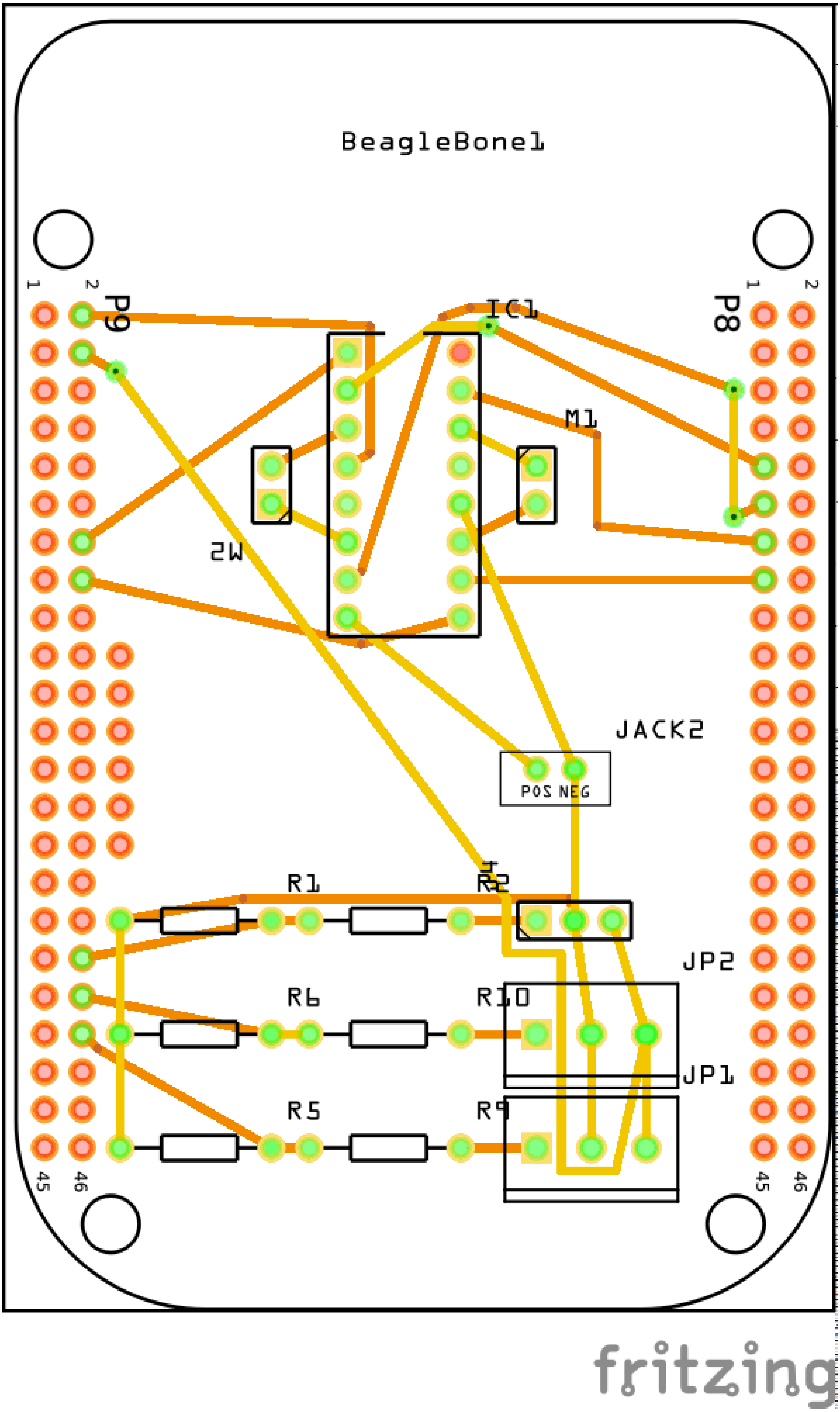


Figure 83. Simple robot PCB

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.

Discussion

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.

Customizing the Board Outline

One challenge that slipped my first pass review was the board outline. The part we installed in [tips_fritzing] 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).

Example 64. Outline SVG for BeagleBone cape (beaglebone_cape_boardoutline.svg)

```
<pre data-type="programlisting">&lt;?xml version='1.0' encoding='UTF-8'
standalone='no'?&gt;
&lt;svg xmlns="http://www.w3.org/2000/svg" version="1.1"
    width="306"  height="193.5"&gt;&lt;!--<a class="co" id="co_capes_bo_1_co"
href="#callout_capes_bo_1_co"><img src="callouts/1.png" alt="1"/></a>--&gt;
 &lt;g id="board"&gt;&lt;!--<a class="co" id="co_capes_bo_2_co"
href="#callout_capes_bo_2_co"><img src="callouts/2.png" alt="2"/></a>--&gt;
  &lt;path fill="#338040" id="boardoutline" d="M 22.5,0 l 0,56 L 72,56
      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
      l 238.5,0 c 24.85281,0 45,-20.14719 45,-45 L 306,45
      C 306,20.14719 285.85281,0 261,0 z"/&gt;&lt;!--<a class="co"
id="co_capes_bo_3_co" href="#callout_capes_bo_3_co"><img src="callouts/3.png"
alt="3"/></a>--&gt;
 &lt;/g&gt;
&lt;/svg&gt;
</pre>
```

```
<dl class="calloutlist">
<dt><a class="co" id="callout_capes_bo_1_co" href="#co_capes_bo_1_co"><img
src="callouts/1.png" alt="1"/></a></dt><dd><p>This is a standard SVG header. The width and
```

```
height are set based on the BeagleBone outline provided in the Adafruit library.</p></dd>
<dt><a class="co" id="callout_capes_bo_2_co" href="#co_capes_bo_2_co"><img
src="callouts/2.png" alt="2"/></a></dt><dd><p>Fritzing requires the element to be within a
layer called <code>board</code>.</p></dd>
<dt><a class="co" id="callout_capes_bo_3_co" href="#co_capes_bo_3_co"><img
src="callouts/3.png" alt="3"/></a></dt><dd><p>Fritzing requires the color to be
<code>#338040</code> and the layer to be called <code>boardoutline</code>. 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.</p></dd>
</dl>
```

The measurements are taken from the BeagleBone Black System Reference Manual, as shown in Cape dimensions.

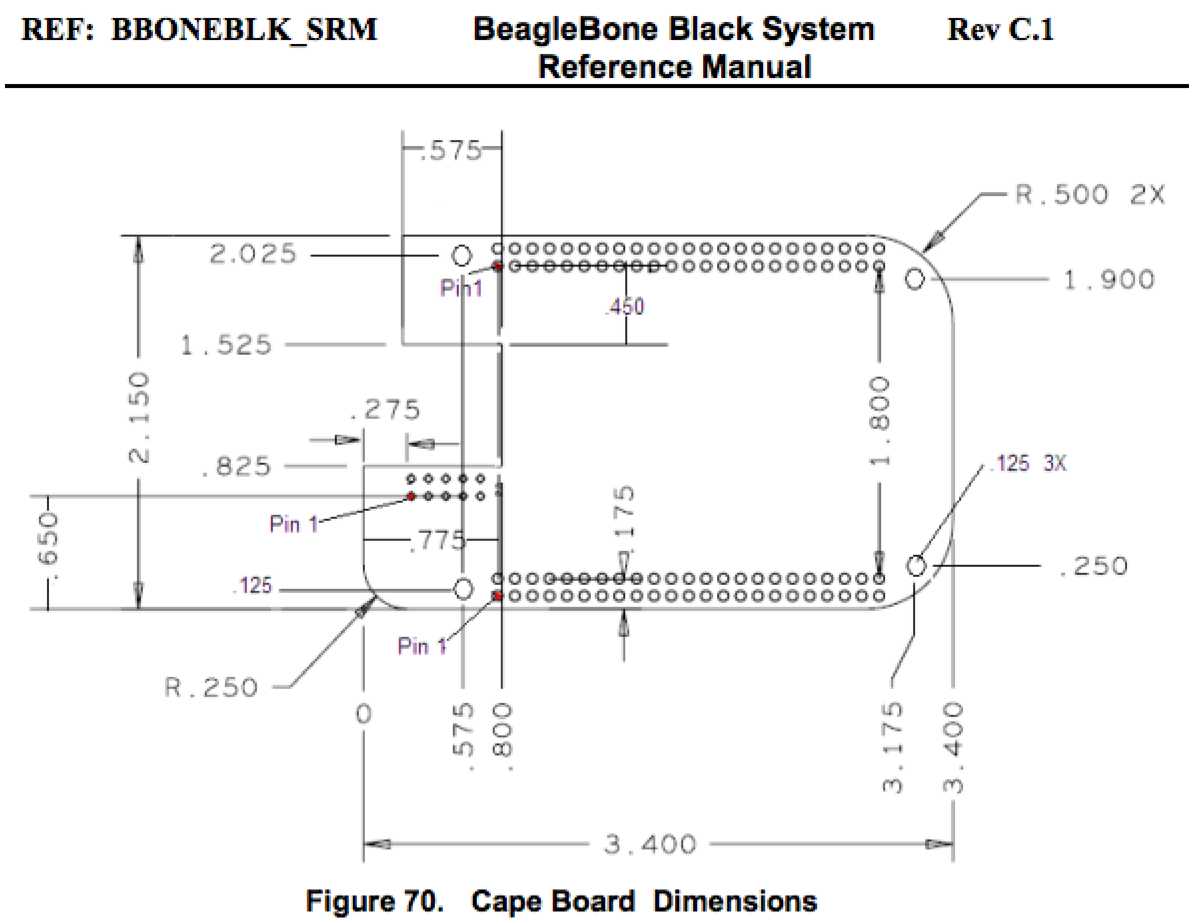


Figure 84. Cape dimensions

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.

Figure 85. Rendered cape outline in Chrome

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.

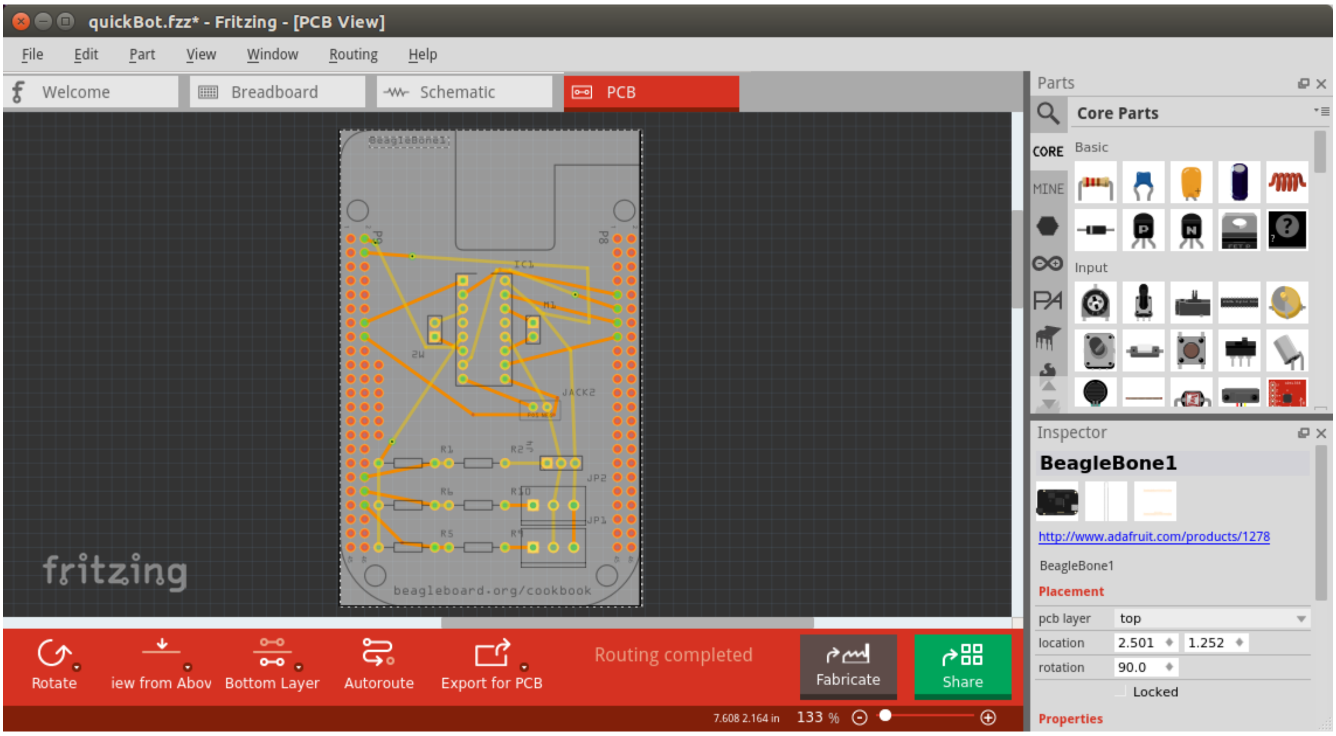


Figure 86. PCB with original board, without notch for Ethernet connector

Hide all but the Board Layer (PCB with all but the Board Layer hidden).

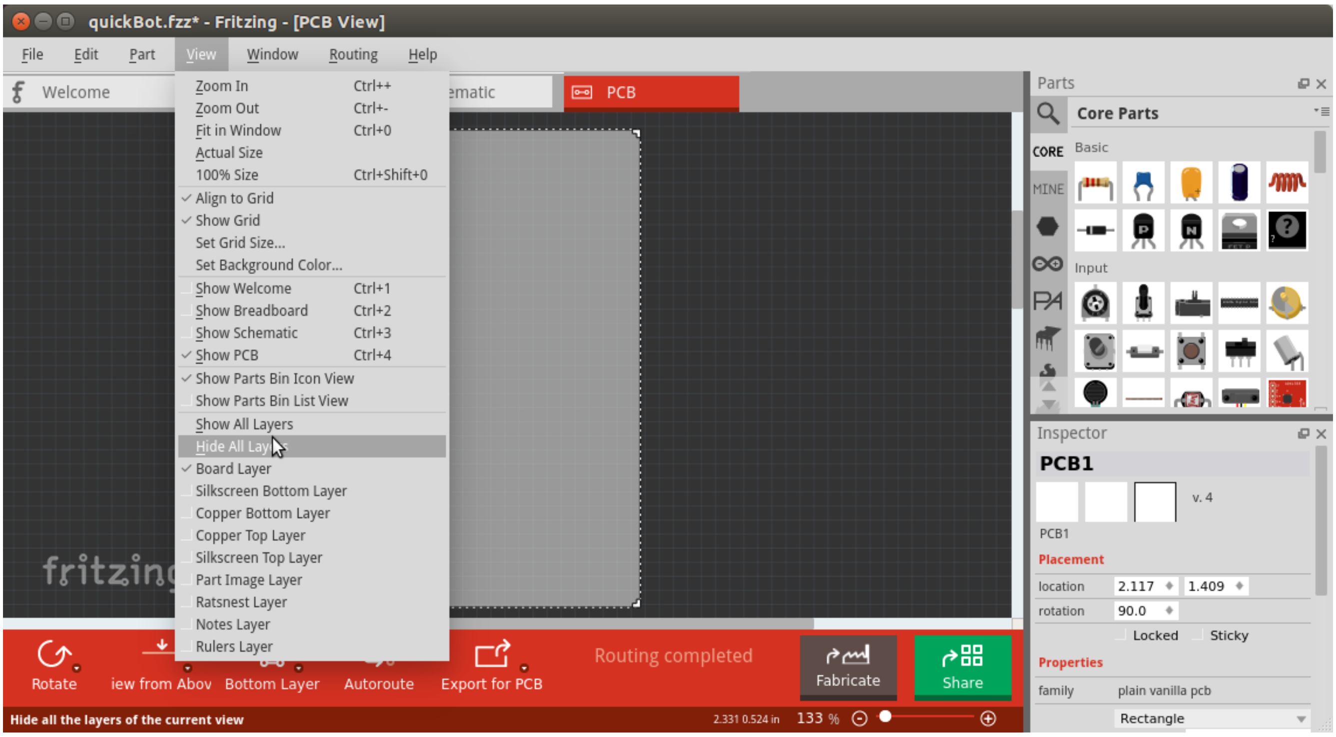


Figure 87. PCB with all but the Board Layer hidden

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).

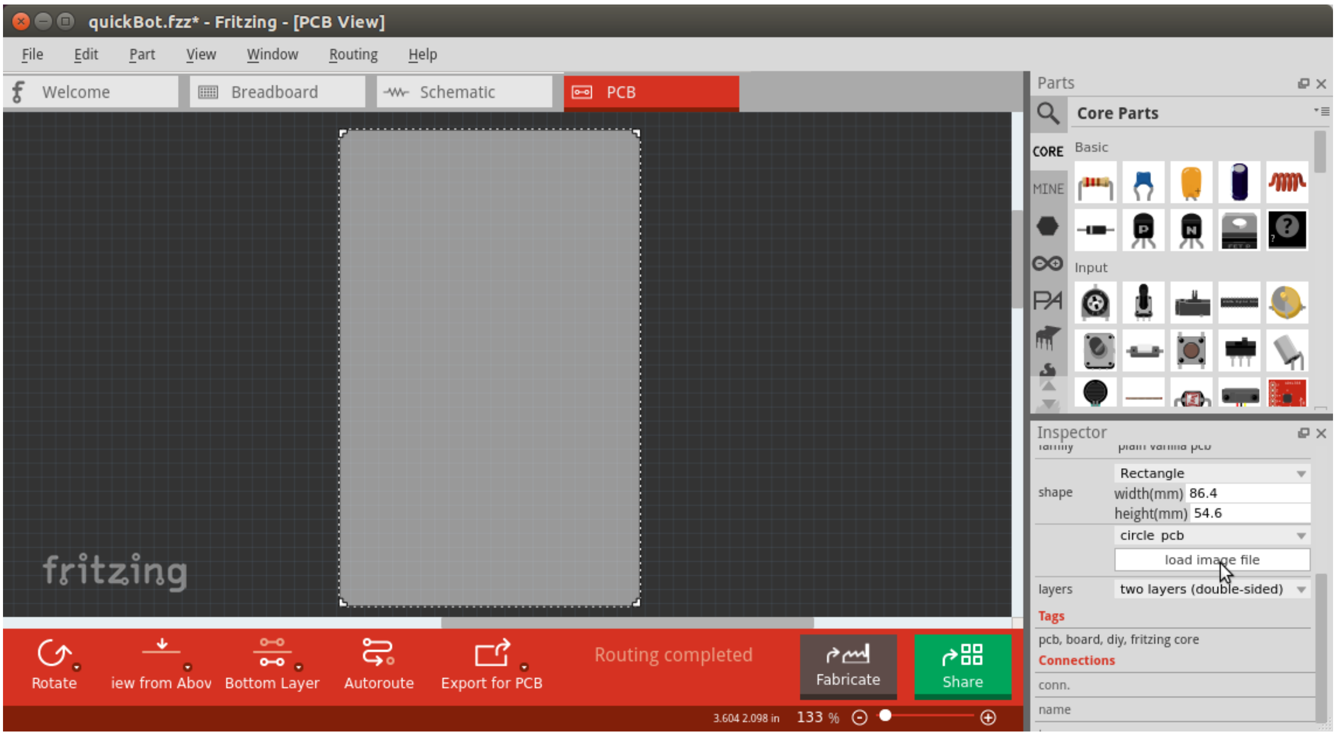


Figure 88. Clicking :load image file: with PCB1 selected

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.

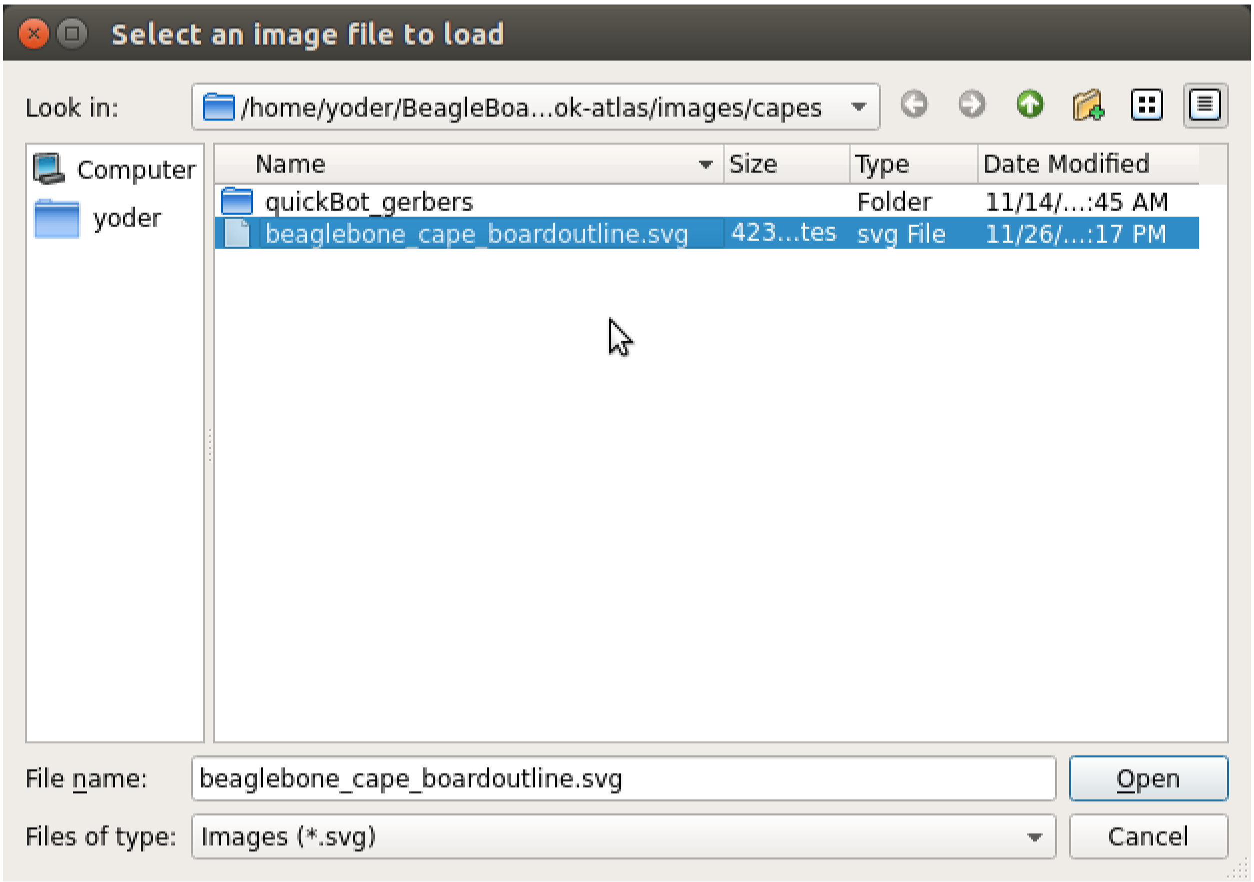


Figure 89. Selecting the .svg file

Turn on the other layers and line up the Board Layer with the rest of the PCB, as shown in PCB Inspector.

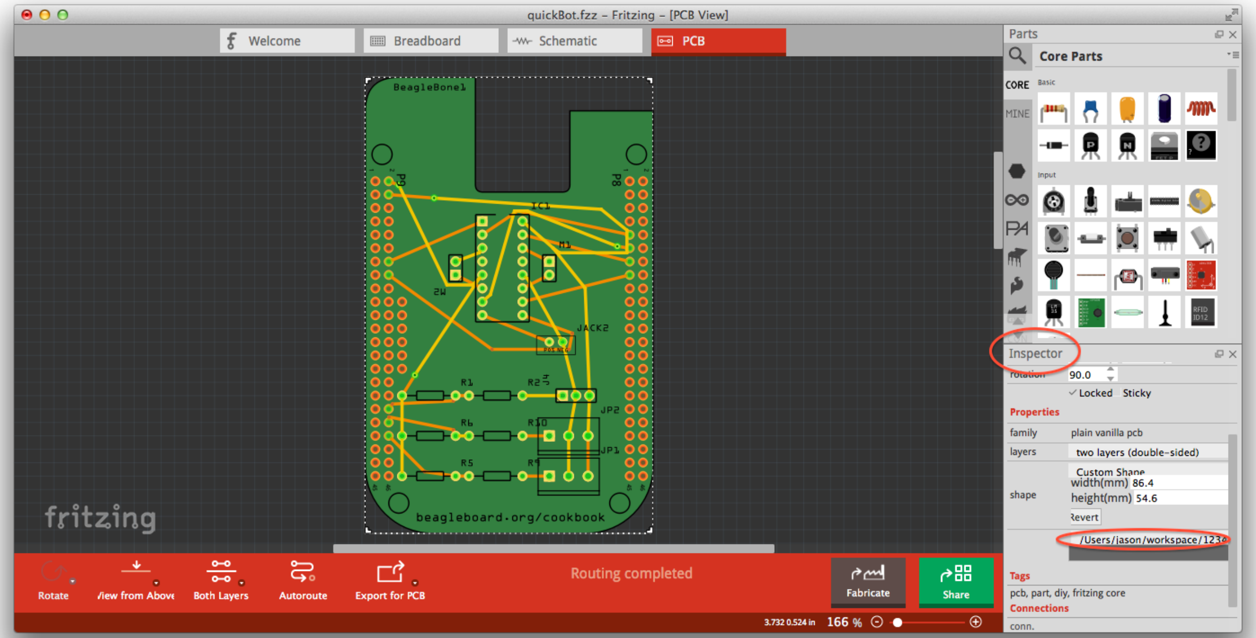


Figure 90. PCB Inspector

Now, you can save your file and send it off to be made, as described in Producing a Prototype.

PCB Design Alternatives

There are other free PCB design programs. Here are a few.

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 (Using a 128 x 128-Pixel LCD Cape) has the schematic shown in Schematic for the MiniDisplay cape and PCB shown in PCB for MiniDisplay cape.

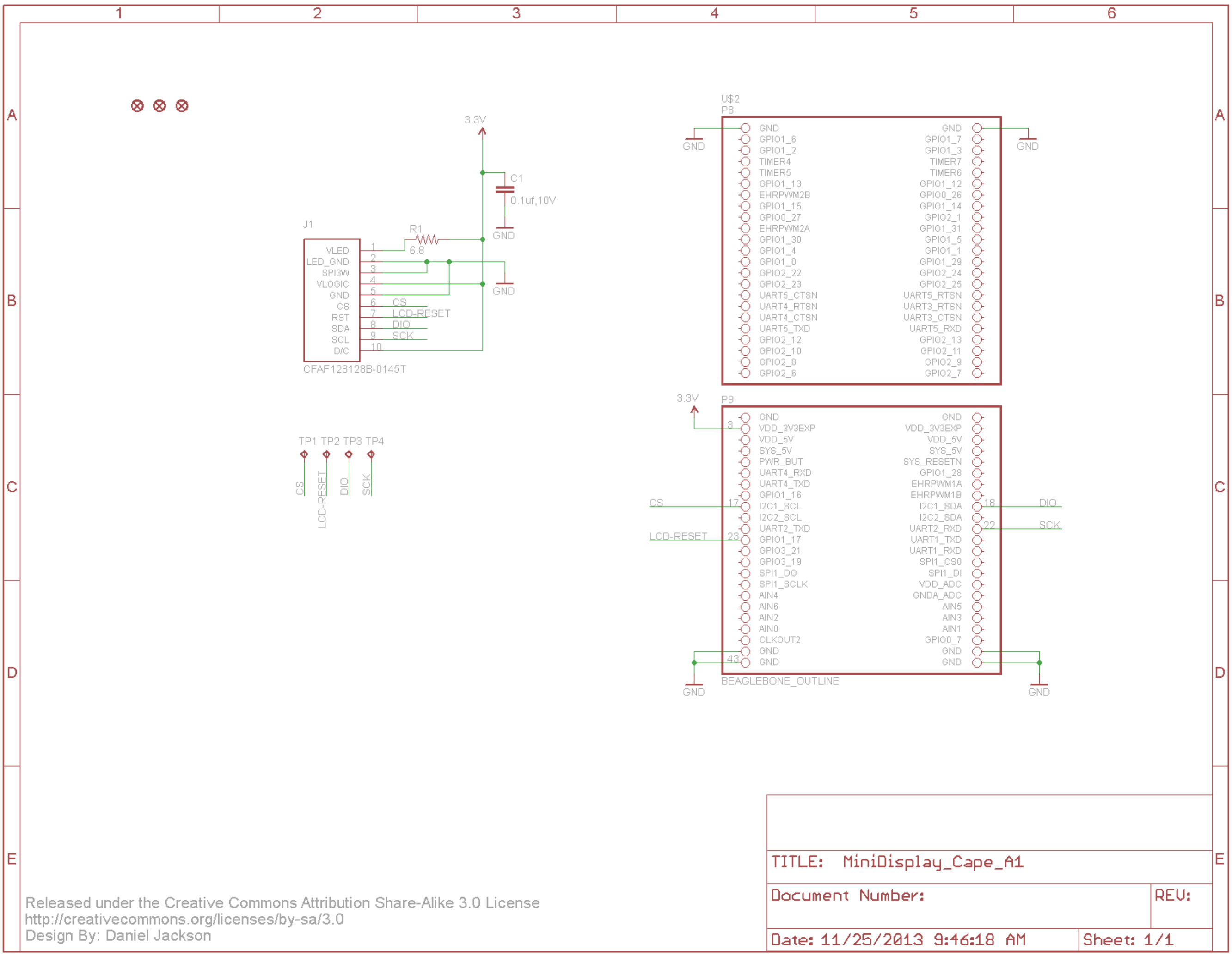


Figure 91. Schematic for the MiniDisplay cape

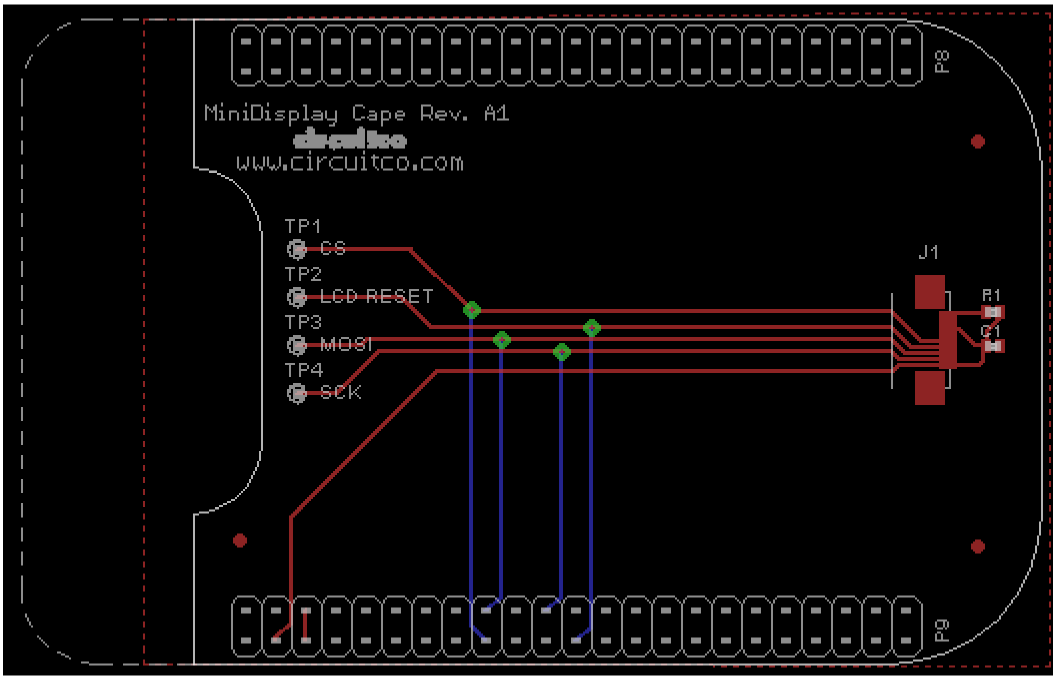


Figure 92. PCB for MiniDisplay cape

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 runs 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:

```
<pre data-type="programlisting">
host$ <strong>sudo apt install eagle</strong>
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$ <strong>eagle</strong>
</pre>
```

You'll see the startup screen shown in Eagle PCB startup screen.

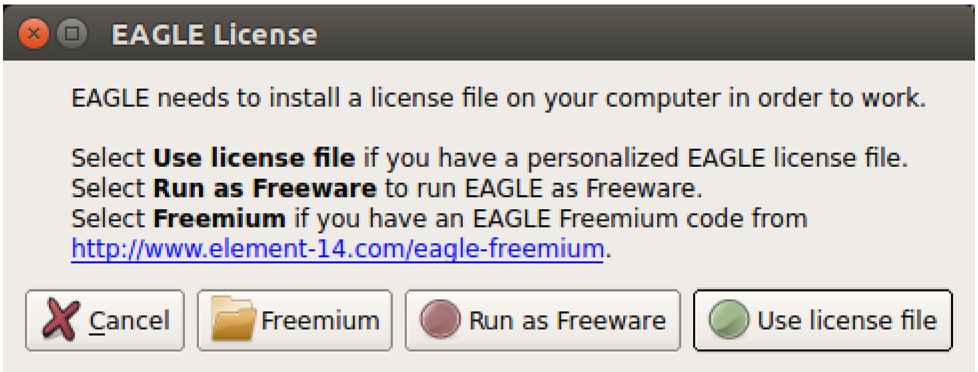


Figure 93. Eagle PCB startup screen

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:

```
<pre data-type="programlisting">
host$ <strong>chmod +x eagle-lin-7.2.0.run</strong>
host$ <strong>./eagle-lin-7.2.0.run</strong>
</pre>
```

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.

Figure 94. The Eagle installation 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):

```
<pre data-type="programlisting">
host$ <strong>cd /usr/bin</strong>
host$ <strong>sudo rm eagle</strong>
host$ <strong>sudo ln -s ~/eagle-7.2.0/bin/eagle .</strong>
host$ <strong>cd</strong>
host$ <strong>eagle</strong>
</pre>
```

The ls 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.

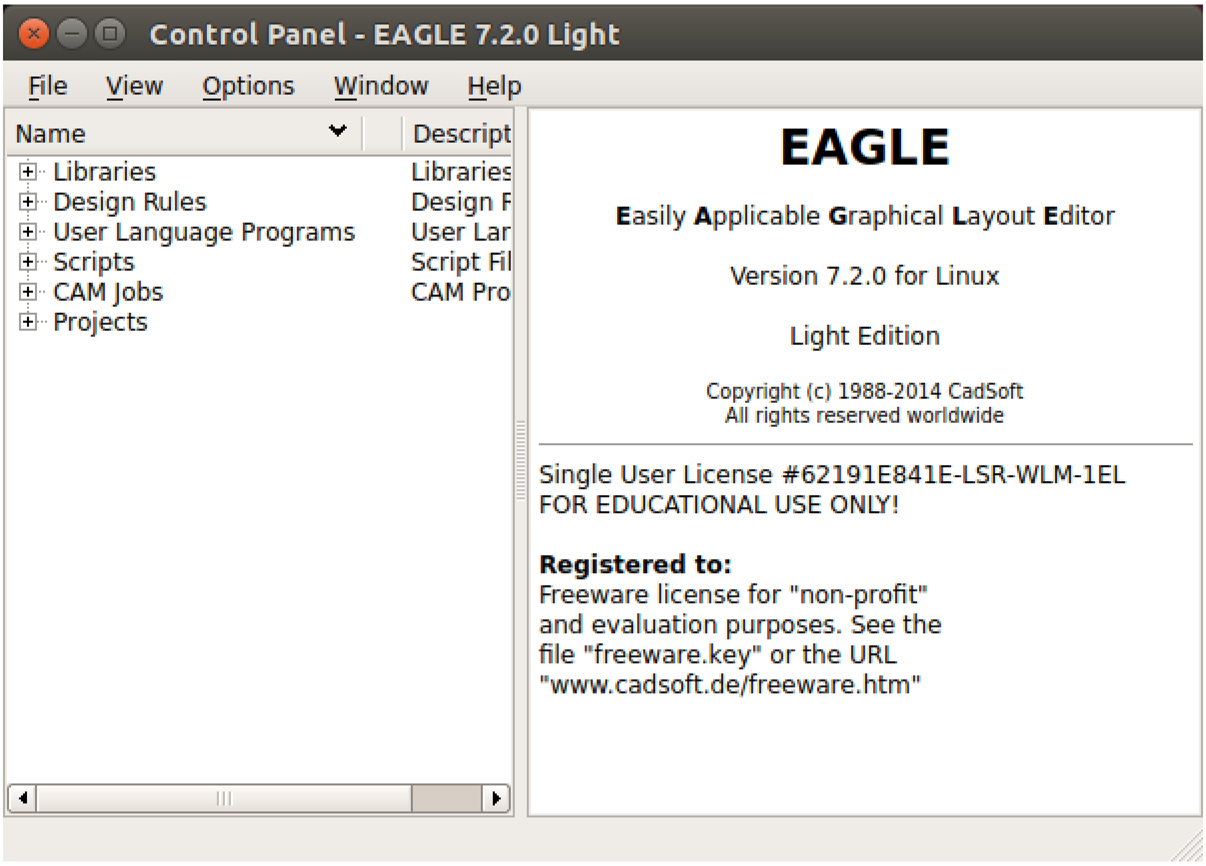


Figure 95. The Eagle start screen

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.

DesignSpark PCB

The free DesignSpark PCB 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.

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.

Kicad

Unlike the previously mentioned free (no-cost) solutions, Kicad is open source and provides some features beyond those of Fritzing. Notably, CircuitHub (discussed in Putting Your Cape Design into Production) provides support for uploading Kicad designs.

Migrating a Fritzing Schematic to Another Tool

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?

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:

```
<pre data-type="programlisting">
host$ <strong>sudo apt install libfreetype6</strong>
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$ <strong>sudo pip install freetype-py</strong>
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...
</pre>
```

> **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:

```
<pre data-type="programlisting">
host$ <strong>git clone git@github.com:upverter/schematic-file-converter.git</strong>
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.
host$ <strong>cd schematic-file-converter</strong>
host$ <strong>sudo python setup.py install</strong>
```

```
.
.
.
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$ <strong>cd ..</strong>
host$ <strong>python -m upconvert.upconverter -h</strong>
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
</pre>
```

At the time of this writing, Upverter suppports 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

File type	Support
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:

```
<pre data-type="programlisting">
host$ <strong>python -m upconvert.upconverter -i quickBot.fzz \
-f fritzing -o quickBot-eaglexml.sch -t eaglexml --unsupported</strong>
WARNING: RUNNING UNSUPPORTED VERSION OF PYTHON (2.7 > 2.6)
DEBUG:main:parsing quickBot.fzz in format fritzing
host$ <strong>ls -l</strong>
total 188
-rw-rw-r-- 1 ubuntu ubuntu  63914 Nov 25 19:47 quickBot-eaglexml.sch
-rw-r--r-- 1 ubuntu ubuntu 122193 Nov 25 19:43 quickBot.fzz
drwxrwxr-x 9 ubuntu ubuntu   4096 Nov 25 19:42 schematic-file-converter
</pre>
```

Output of Upverter conversion shows the output of the conversion.

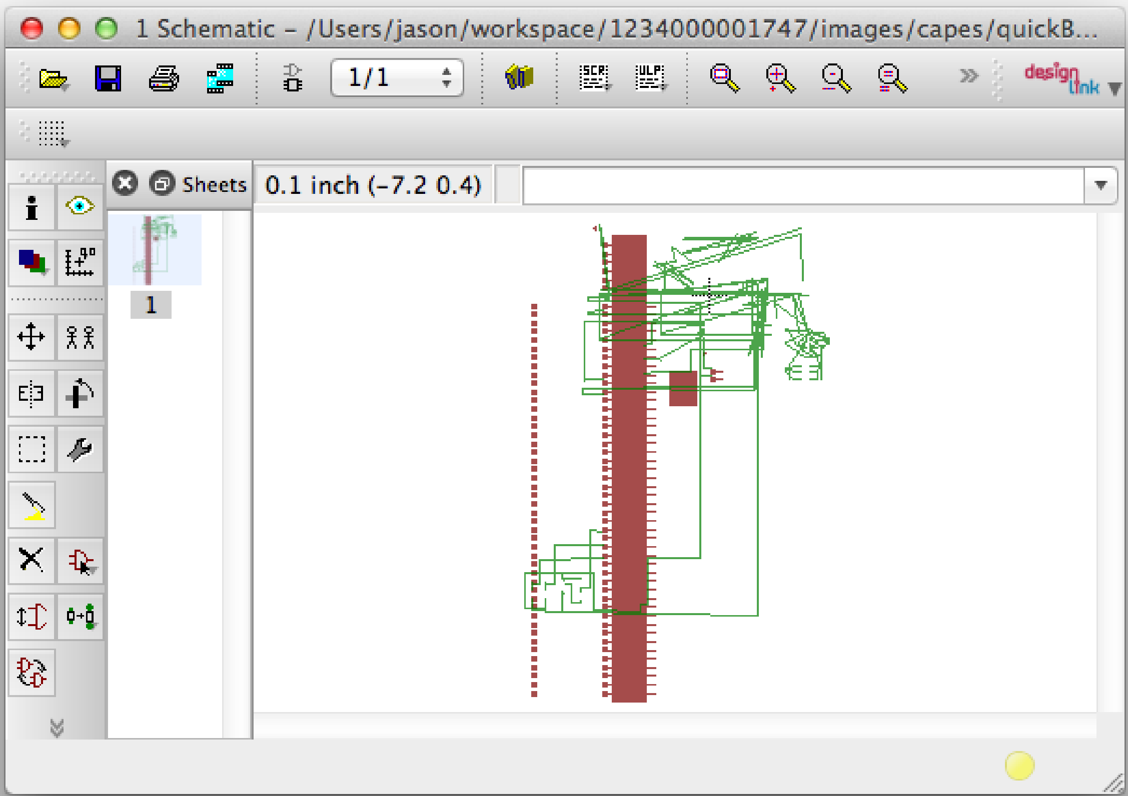


Figure 96. Output of Upverter conversion

No one said it would be pretty!

Discussion

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.

Producing a Prototype

Problem

You have your PCB all designed. How do you get it made?

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.

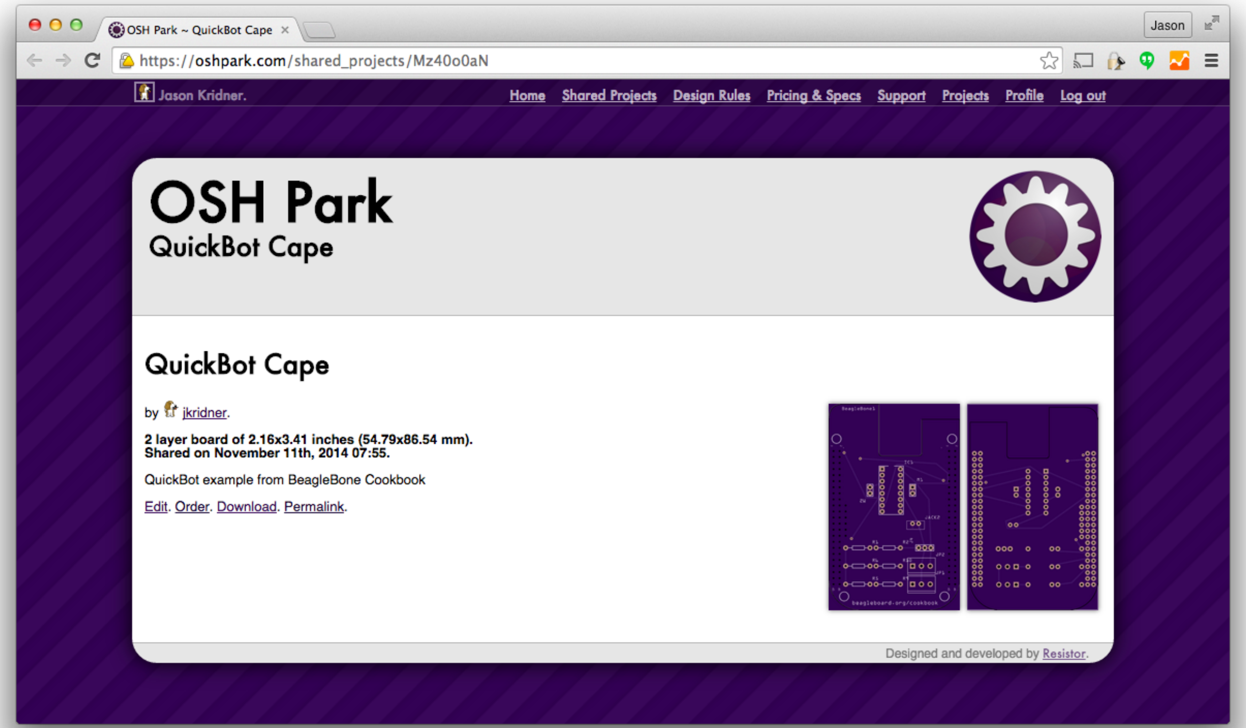


Figure 97. The OSH Park QuickBot Cape shared project page

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.

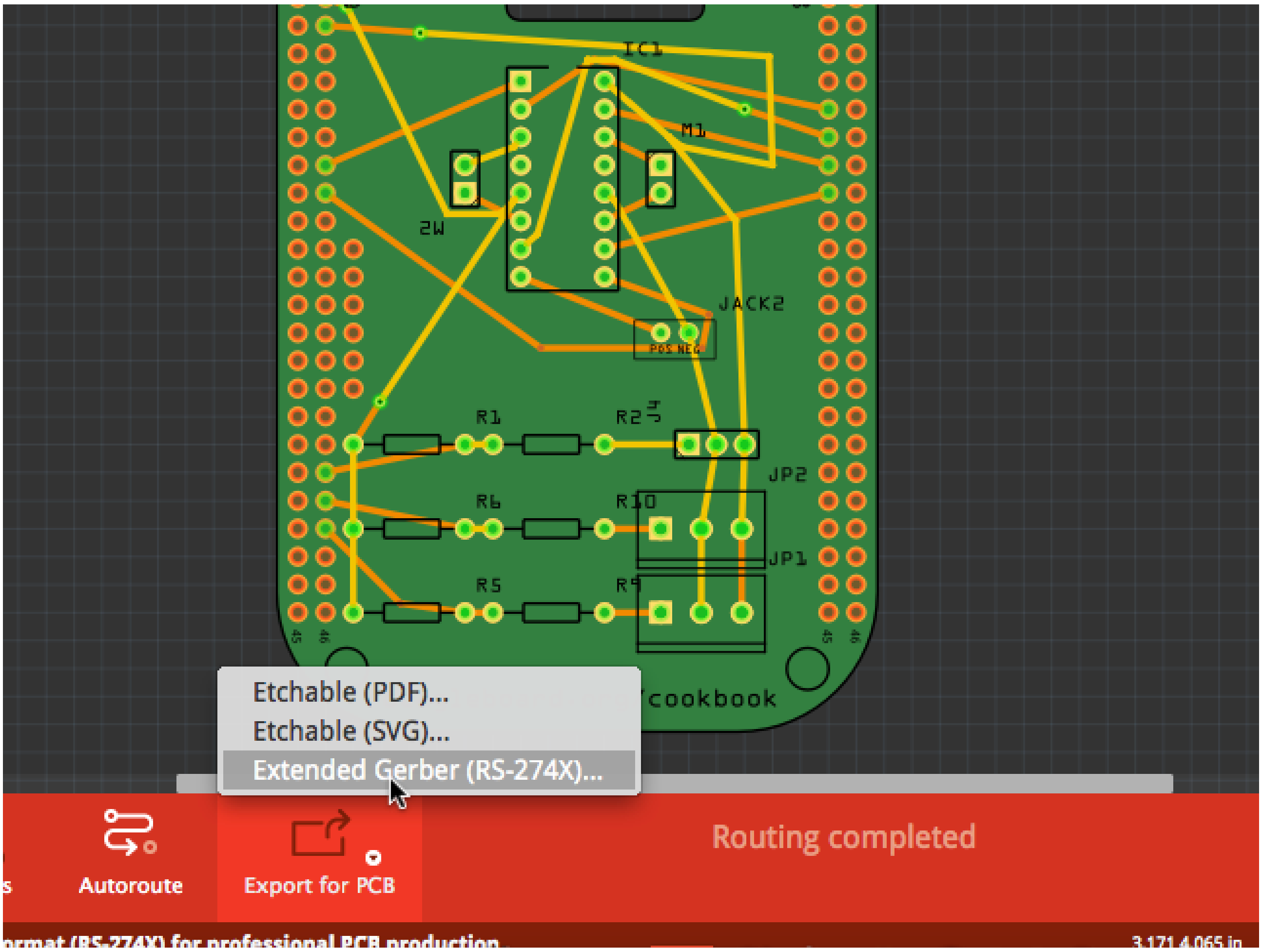


Figure 98. Choosing "Extended Gerber" in Fritzing

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.

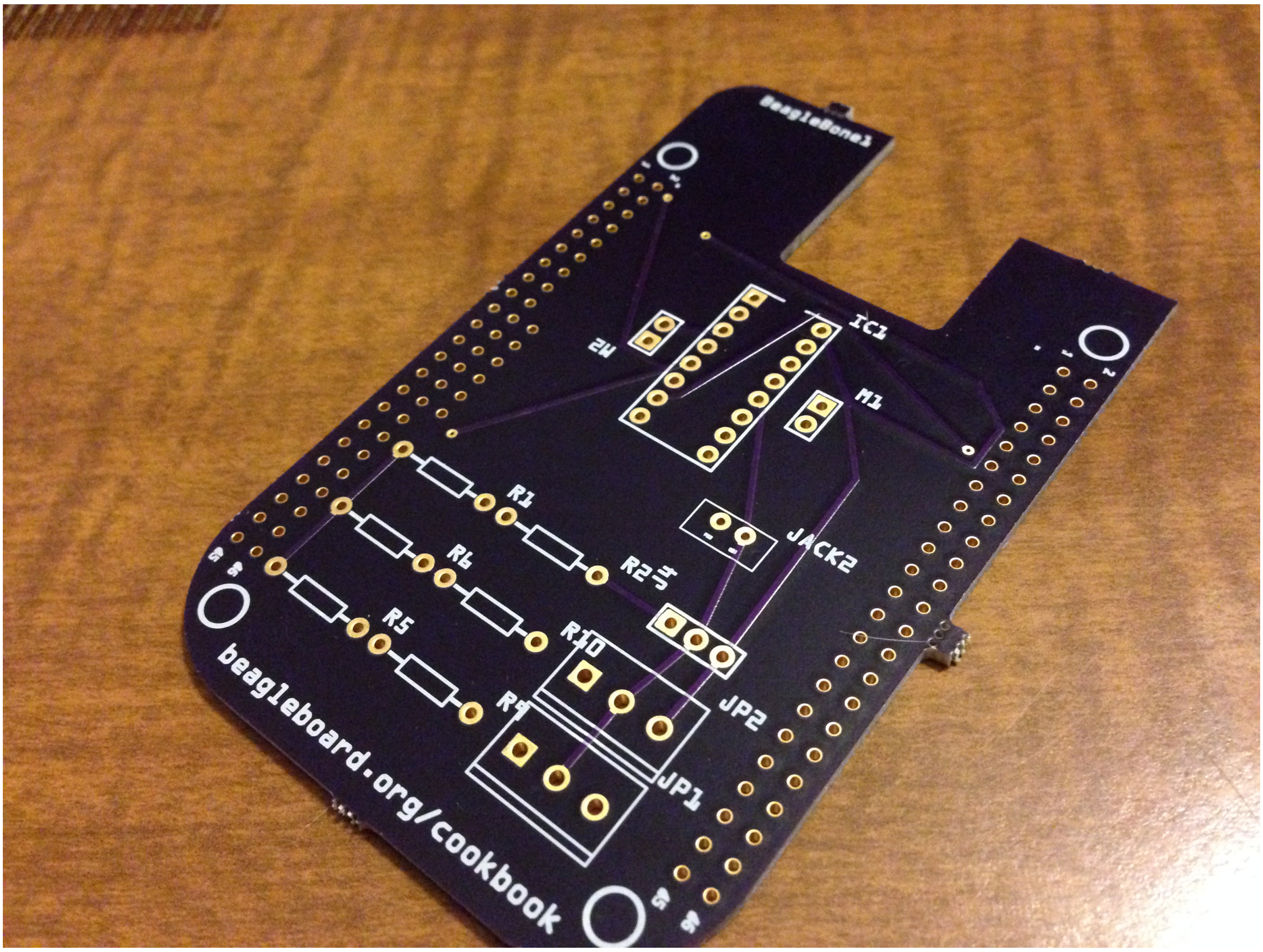


Figure 99. QuickBot PCB

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.

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).

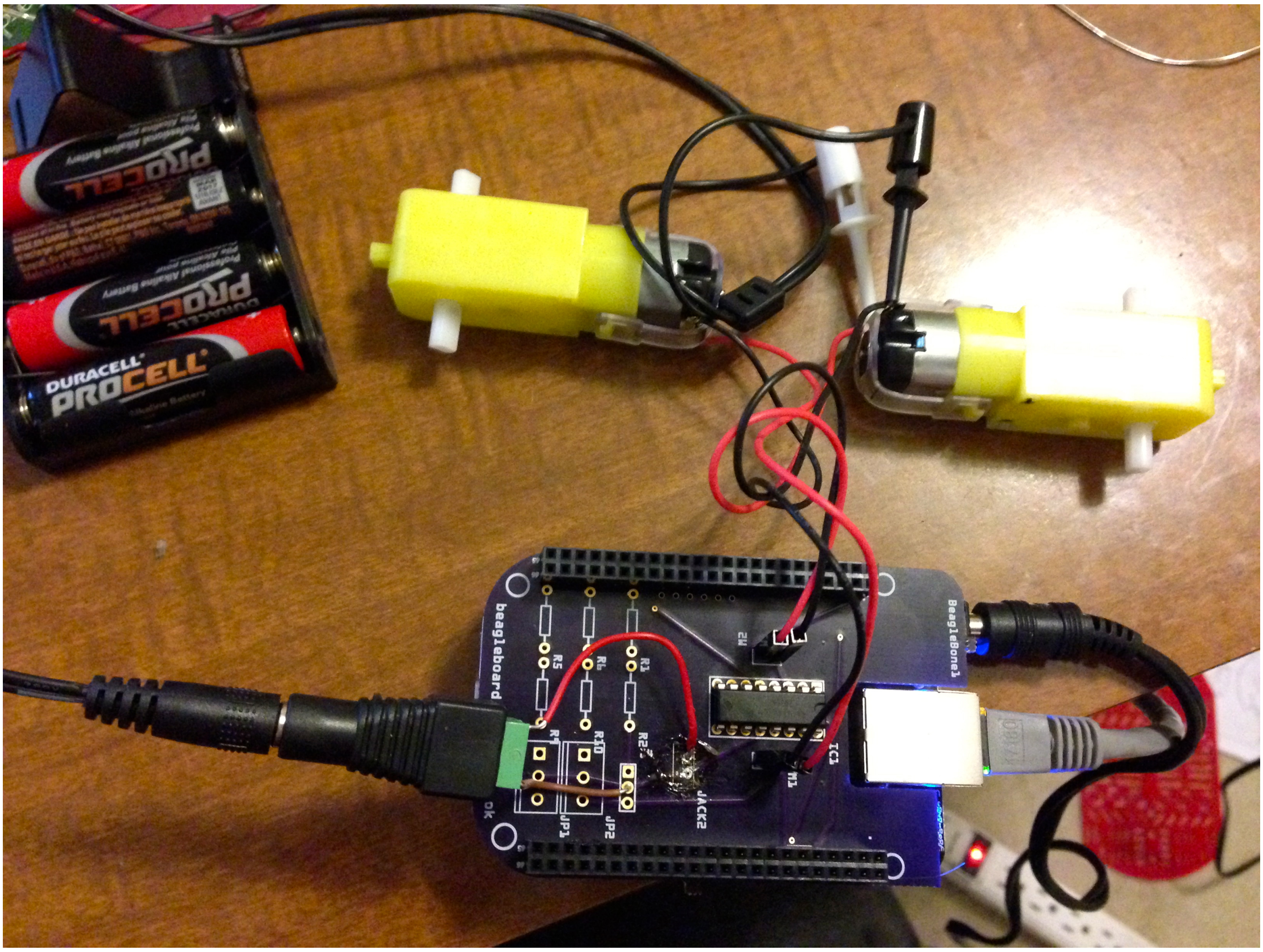


Figure 100. QuickBot motors under test

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!

Creating Contents for Your Cape Configuration EEPROM

Problem

Your cape is ready to go, and you want it to automatically initialize when the Bone boots up.

Solution

Complete capes have an I^{2}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.

Putting Your Cape Design into Production

Problem

You want to share your cape with others. How do you scale up?

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.

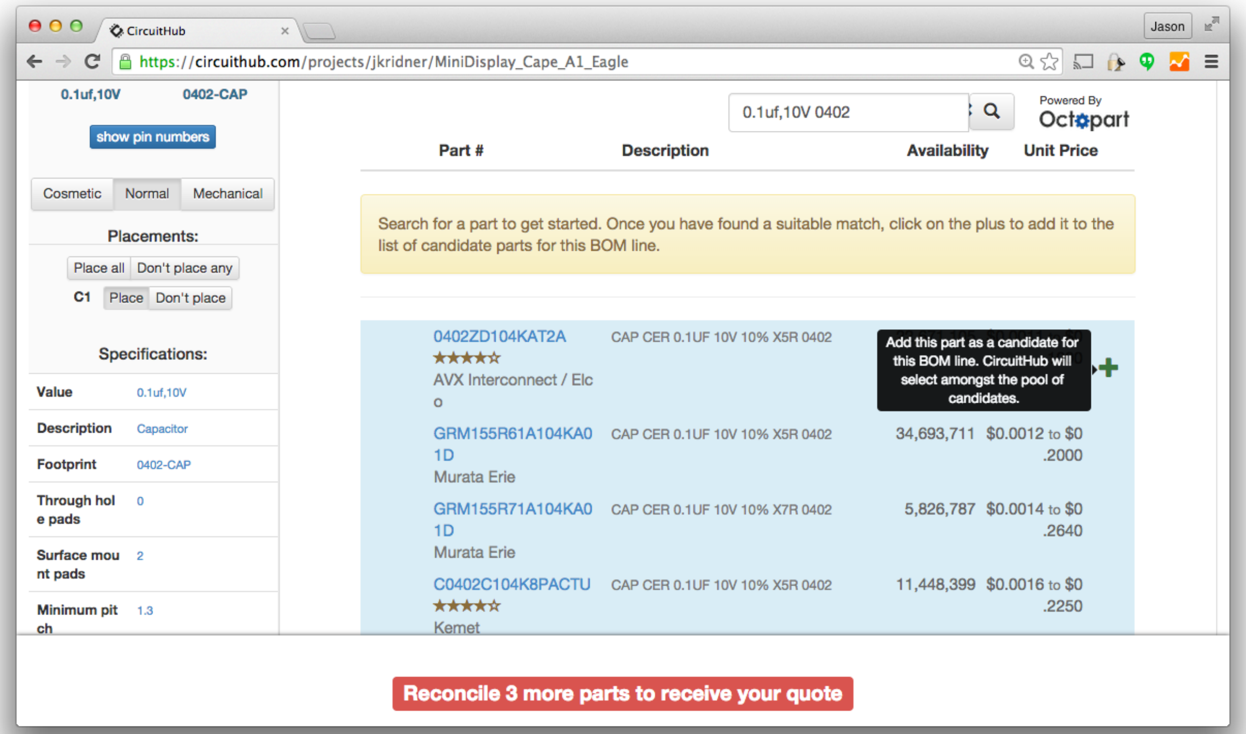


Figure 101. CircuitHub part matching

When you’ve selected all of your parts, a quote tool appears at the bottom of the page, as shown in CircuitHub quote generation.

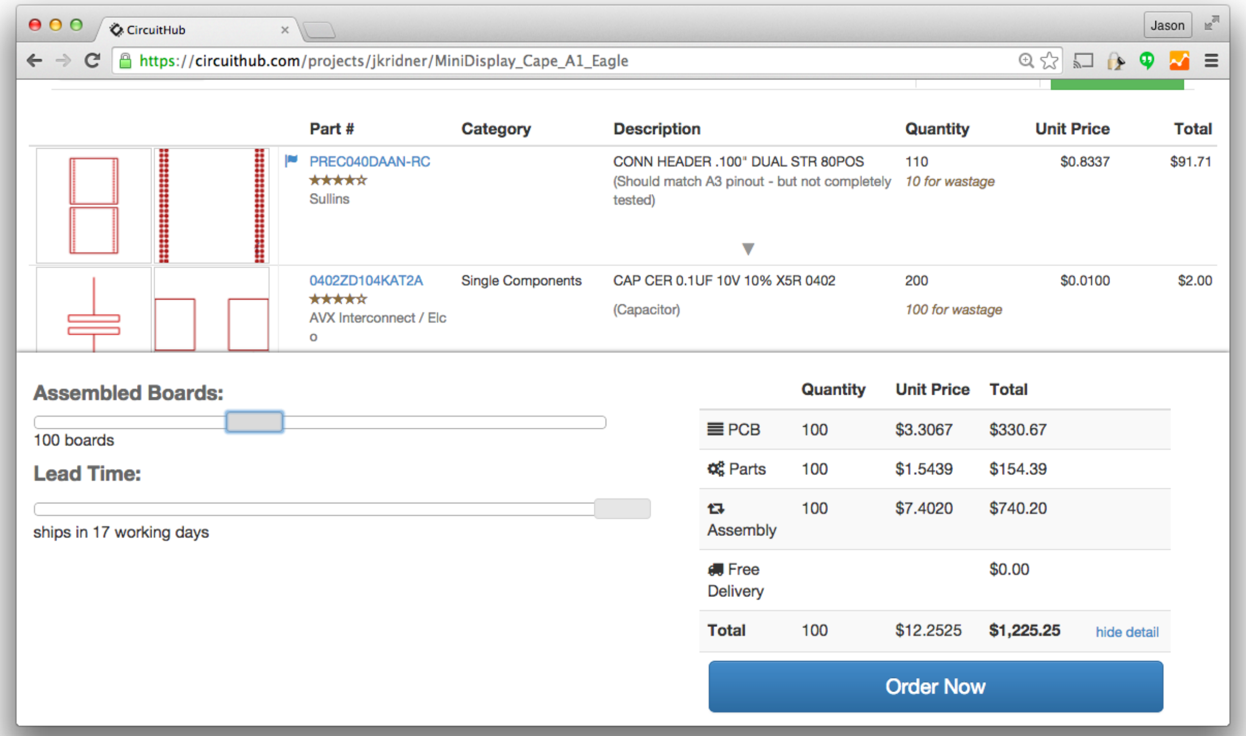


Figure 102. CircuitHub quote generation

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.

Table 3. 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 4. 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!

Discussion

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 the official contract manufacturer of BeagleBoard.org and not the same company as CircuitHub, the online contract manufacturing service. CircuitCo would be an excellent choice for you to consider to perform your contract manufacturing, but it doesn't offer an online quote service at this point, so it isn't as easy to include details on how to engage with it in this book.

[1] This solution, written by Elias Bakken (@AgentBrum), originally appeared on Hipstercircuits.

[2] [capes_7inLCD_fig] was originally posted by CircuitCo at http://elinux.org/File:BeagleBone-LCD7-Front.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[3] Seven-inch LCD from CircuitCo [2] was originally posted by CircuitCo at http://elinux.org/File:BeagleBone-LCD7-Front.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[4] [capes_miniDisplayBoris] was originally posted by David Anders at http://elinux.org/File:Minidisplay-boris.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[5] MiniDisplay showing Boris [4] was originally posted by David Anders at http://elinux.org/File:Minidisplay-boris.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[6] [capes_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.

[7] Back side of LCD7 cape [6] was originally posted by CircuitCo at http://elinux.org/File:BeagleBone-LCD-Backside.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[8] [Audio_cape_pins_fig] was originally posted by Djackson at http://elinux.org/File:Audio_pins_revb.png under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[9] Pins utilized by CircuitCo Audio Cape [8] was originally posted by Djackson at http://elinux.org/File:Audio_pins_revb.png under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[10] [capes_beaglebread_fig] was originally posted by William Traynor at http://elinux.org/File:BeagleBone-Breadboard.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

[11] BeagleBone breadboard [10] was originally posted by William Traynor at http://elinux.org/File:BeagleBone-Breadboard.jpg under a Creative Commons Attribution-ShareAlike 3.0 Unported License.